Comun, The Programming Language Specification
version 0.905, by Miloslav Čížˇ (drummyfish), 2023, released under CC0 1.0, public domain (https://creativecommons.org/publicdomain/zero/1.0/)
This is a specification of the comun programming language. Currently it is rather informal, a more formal one may be made once the language matures. Many things are yet left unspecified so as to keep this short, simple and allow evolution -- for example error behavior and many commonly used terms are handled without precise definitions, common meanings are assumed. Where something is unspecified, suppose freedom of implementation, try to implement a simple and logical solution that respects the spirit of the language.
Language naming and compliance: this is a recommendation for naming of implementations and modifications of the language. The language specified by this document is named *comun*; any implementation that calls itself a *comun* implementation should adhere to all mandatory requirements specified here. It is planned that future versions of this language will continue to be called *comun*, they will be distinguished just by the version number. A simplified version of comun that doesn't have to support preprocessing, file includes and user defined pointers and will typically only implement one type environment (environment 0) may be called *minicomun*. A name for versions further simplified to not even support user function definitions is *microcomun* -- such versions may still support calling of external functions. It is recommended different types of modifications choose yet a different name, which may or may not contain the word *comun* as its part.
Version numbering of this document has the traditional *MAJOR.MINOR* format. The first two digits of *MINOR* number increase with changes in the specification that make the language different, its further digits increase with minor changes not affecting the language itself (such as typo corrections, formatting etc). *MAJOR* number is incremented and *MINOR* number is set to zero after many significant changes since the start of the current *MAJOR* version. Note that the specification version number and version number of its various implementations are of course generally not the same.
Basics
The language is minimalist, statically typed, imperative, Turing complete, general purpose and low level, it mainly operates on stacks but also supports pointers for random memory access and implementing multiple stacks. It may be both interpreted and compiled. Postfix notation is used. English keywords are avoided; simple, short math-like symbols are preferred. The language aims for minimum of abstraction above and general similarity to traditional hardware architectures (i.e. easy mapping to assembly languages), it should enable good performance, being future proof, universal and as much as possible free (i.e. public domain, not requiring complex hardware etc.). The following do NOT belong among the language goals: high abstraction (non-imperative paradigms, ...), handholding, rapid programming ("productivity"), parallelism, mainstream popularity, making profit or being beginner friendly.
Character set: the language source code may only contain 7bit ASCII characters except for the character with value zero (this character is reserved for potentially terminating the source code string).
Blank character is any character with ASCII value smaller or equal to that of space (`␣`); except for the first preprocessing stage, the characters `[` and `]` are also considered blank (no matter where they appear). Platform specific newline sequences in the source code are to be considered a single **newline** character `\n` (ASCII value 10).
Comments: during processing of source code if the hash character (`#`) is read and this character is not part of a string literal, the processing considers all characters read blank characters until the end of the next nearest hash character (`#`) or newline (`\n`).
Tokens: source code of the language can be seen as a sequence of tokens. Token is a string appearing outside of comment, consisting of only non-blank characters, except for string literals which may contain blank characters but are considered a single token.
Identifiers: identifier is a string composed of at least one character and each character of an identifier has to be either a letter (`a` - `z`, `A` - `Z`), decimal digit (`0` - `9`) or underscore (`_`), however the first character cannot be a decimal digit.
Numeric literals: numeric literal may start with sign (`+` or `-`). If sign is present, base specifier may follow. Then one to many digits of given base follow. Possible bases are:
- **decimal**: Base specifier is `d`. This is the default base considered if no base specifier is present. Base digits are `0` to `9`. Examples: `0`, `-0`, `+d0123`, `732`.
- **hexadecimal**: Has to start with a sign followed by base specifier `x`. Base digits are `0` to `9` and `a` to `f`. Examples: `+x00ff`, `-x100`.
- **binary**: Has to start with a sign followed by base specifier `b`. Base digits are `0` and `1`. Examples: `+b0`, `-b1010110`.

Let a non-negative decimal literal without base specifiers and without any unnecessary leading zeros be called a **plain decimal numeric literal**.
String literals: a string literal is a string that starts and ends with the double quote character (`"`). Between these there is zero to many characters that are not double quote. There are no escape sequences for strings (values not representable in string literals can be pushed to stack as numeric literals).
Data types: only atomic integer types that are implicitly considered unsigned exist in the language. A data type is defined only by its bit width *N* which is a non-negative integer. 0 is a special value used to signify the platform's **native data type**, the data type identified by 0 has bit width equivalent to the platform's native integer type if possible, but at least 16 bits. Data type with bit width *N* can hold exactly the values 0 to 2^*N* - 1 (including both bounds). Some built-in operations may interpret memory values as signed; in that case two's complement representation is considered, i.e. results of operations are those that would be obtained in two's complement representation. For unsigned operations direct representation is considered.
Each data type has its own separate **type environment**. A type environment is defined in the same way as data type: by its integer non-negative bit-width *N*. This also holds for the data type identified by 0 (the platform's native data type). Note that type environment 0 is always separate, i.e. if for example given platform's native type has bit width 16, type environments 0 and 16 are still separate. Each type environment has its own pointer table and memory consisting of cells, each one holding a value of that environment's type. The cells have integer addresses, first one has address 0, the second one 1 and so on up until a limit imposed by the language processor. At any given point during source code processing exactly one type environment is active and the commands of the language operate on this environment's memory. I.e. the currently active type environment implies the **currently set data type**. The default type environment set at the start of language processing is the platform's native data type (0).
The language doesn't have to support all possible type environments, only type environment 0 is required, however it is recommended to also implement at least type environments 8, 16 and 32.
Program arguments: programs may receive external string arguments for example from the operating system. This passing of arguments can be seen just as if a string *S* was prepended to the source code where *S* is of form `0 Sn ... 0 S3 0 S2 0 S1 N` where *S1* ... *Sn* are string literals of the arguments and *N* is the number of arguments (if no arguments are passed only a single zero value will be present on the stack at the start of the program). If no parameters are passed or if parameter passing is not supported, a single value 0 still has to be pushed on the stack before program execution starts. If preprocessing is implemented then argument passing to the preprocessor itself may be allowed in the same way.
There must be at least 16 free (usable) cells in each supported type environment, starting from the initial main stack top address (however this should be much more if possible), before pushing program arguments.
There is no **standard library**.
Pointers
A pointer is a name-value pair stored in a pointer table. The pointer name is an identifier (with an exception described below) unique within the pointer table of given type environment and its value is a non-negative integer address referring to a specific cell in the type environment whose pointer table the pointer resides in. As each type environment has a separate pointer table, there can exist pointers with the same name in different type environments.
In each type environment special pointers named `0`, `1`, ... `9` always exist by default (these pointer names are an exception to pointer names having to be identifiers). Pointers `1`, ... `9` are read-only and can NOT be assigned addresses to directly with commands, their address is always derived from pointer `0`. Pointer `0` always contains the current address at which commands operate, pointer `1` contains address stored in pointer `0` minus 1 etc. As most commands exhibit a stack behavior, we will also refer to the pointer `0` as the **stack top** pointer. The initial addresses stored in pointers, including the pointer `0`, are decided by the language processor.
Commands
Here we describe the basic unstructured commands of the language, control structures are described in another section.
Built-in operations of the language usually work on the stack of the currently set type environment (see pointers and stack top).
Accessing (reading from or writing to) **memory outside bounds** (below 0 or above the maximum address) causes a run time error. Whether leaving memory bounds (without accessing it, e.g. merely moving stack top below address zero) causes an error is left to be decided by each implementation.
Implementation of operations as described below must ensure they do NOT affect values above the final stack top address (because the programmer may have temporarily moved downwards the stack with the intention to return back up later). For example it might be tempting to implement the increment operation (`++`) as a shorthand for pushing the value one and invoking the add command (`1 +`), however this cannot be done for the above mentioned reason.
A **pop** means decrementing stack top pointer. A pop itself mustn't modify any values in memory cells. A **push** of a value means incrementing stack top pointer and then writing the value to the stack top memory cell.
Type conversions: the result of an operation that's to be of data type of width *N* is the same as if first all the input operands of were converted to a binary number of infinite bit width (with sign extension if we are considering two's complement), then the operation was performed and lowest *N* bits of the result were taken.
In the following description of commands let *x*, *y* and *z* be the value stored in the cell at the address of pointer `0`, pointer `1` and pointer `2`, respectively, before performing the command, interpreted as an unsigned integer in direct representation unless mentioned otherwise. Operations explicitly mentioned as signed interpret operands as signed integers in two's complement representation.
Pointers and addresses: if a command works with an address of a pointer, the address just before invocation of the command is considered (this is significant e.g. with commands that modify stack top address and pop at the same time). If a command that directly modifies a pointer address is applied to one of the special pointers `1`, ... `9`, nothing happens, as these pointers are considered just virtual read-only pointers relative to pointer `0`.
The language commands are
- **numeric literal**: Pushes the value represented by the literal.
- **string literal**: Pushes ASCII values corresponding to the characters in the string literal from back to front, minus the double quote characters. E.g. `"abc"` pushes 99, then 98, then 97.
- identifier (**function call**): Pushes return address on call stack and jumps to executing the corresponding function. For details see the section on language structure.
- `+` (**addition**): Pops 2 values, pushes the value *y* + *x*.
- `++` (**increment**): Pops 1 value, pushes the value *x* + *1*.
- `-` (**subtraction**): Pops 2 values, pushes the value *y* - *x*.
- `--` (**decrement**): Pops 1 value, pushes the value *x* - *1*.
- `*` (**multiplication**): Pops 2 values, pushes the value *y* * *x*.
- `/` (**division**): Pops 2 values, pushes the value *y* / *x* (integer division). If *x* is 0, error occurs.
- `//` (**signed division**): Signed operation, pops 2 values, pushes the value *y* // *x* where // signifies signed integer division. If *x* is 0, error occurs.
- `%` (**remainder**): Pops 2 values, pushes the values *y* *mod* *x* (remainder after integer division). If *x* is 0, error occurs.
- `%%` (**signed remainder**): Signed operation, pops 2 values, pushes the value *y* - (*y* // *x*) * *x* where // is signed integer division. If *x* is 0, error occurs.
- `><` (**swap**): Pops 2 values, pushes the values *x*, then *y*.
- `^` (**pop**): Pops 1 value.
- `=` (**equality**): Pops 2 values. If *x* is equal to *y*, pushes the value 1, otherwise pushes the value 0.
- `!=` (**inequality**): Pops 2 values. If *x* is equal to *y*, pushes the value 0, otherwise pushes the value 1.
- `<` (**smaller inequality**): Pops 2 values. If *y* < *x*, pushes the value 1, otherwise pushes the value 0.
- `<<` (**smaller signed inequality**): Signed operation, pops 2 values. If *y* < *x*, pushes the value 1, otherwise pushes the value 0.
- `<=` (**smaller or same inequality**): Same as *smaller inequality*, but the comparison performed is *y* <= *x*.
- `<<=` (**smaller or same signed inequality**): Same as *smaller signed inequality*, but the comparison performed is *y* <= *x*.
- `>` (**greater inequality**): Same as *smaller inequality*, but the comparison performed is *y* > *x*.
- `>>` (**greater signed inequality**): Same as *smaller signed inequality*, but the comparison performed is *y* > *x*.
- `>=` (**greater or same inequality**): Same as *smaller inequality*, but the comparison performed is *y* >= *x*.
- `>>=` (**greater or same signed inequality**): Same as *smaller signed inequality*, but the comparison performed is *y* >= *x*.
- `||` (**logical or**): Pops 2 values. If at least one of the values *y* and *x* is non-zero, pushed the value 1, otherwise pushes the value 0.
- `&&` (**logical and**): Pops 2 values. If both *y* and *x* are non-zero, pushes the value 1, otherwise pushes the value 0.
- `!` (**logical not**): Pops 1 value. If *x* is non-zero, pushed the value 0, otherwise pushes the value 1.
- `|!|` (**logical xor**): Pops 2 values. If exactly one of the values *y* and *x* is non-zero, pushes the value 1, otherwise pushes the value 0.
- `|` (**bitwise or**): Pops 2 values, pushes the value *y* | *x* where | is a bitwise or operation.
- `&` (**bitwise and**): Pops 2 values, pushes the value *y* & *x* where & is a bitwise and operation.
- `!` (**bitwise not**): Pops 1 value, pushes the value *!x* where *!* is a bitwise negation operation.
- `|!` (**bitwise xor**): Pops 2 values, pushes the value *y* |*!* *x* where |*!* is a bitwise xor operation.
- `|>` (**shift right**): Pops 2 values, pushes the value *y* bit-shifted by *x* to the right (towards least significant bit).
- `|<` (**shift left**): Same as *shift right* but the shift performed is to the left.
- `<-` (**input**): Reads a single value from the input and pushes it. Typically this can mean reading a single text character from input and pushing its ASCII value, but this interpretation is not required. The operation is blocking and doesn't check for end of input. If end of input has been reached and this command is performed, 0 is pushed.
- `->` (**output**): Pops 1 value and outputs it. Typically this can mean printing out a character with ASCII value *x*, but this interpretation is not required.
- `-->` (**string output**): Equivalent to the sequence of commands `@' -> . ^`.
- `<?` (**input unfinished**): Pushes 0 if the previous `<-` command attempted to read from a finished input stream, otherwise 1.
- `??` (**conditional**): Pops 3 values. If *z* is not zero, pushes *y*, otherwise pushes *x*.
- `>N` where *N* is a plain decimal numeric literal (**type environment transfer**): Pops 1 value, then writes this value to the stack top in type environment *N* (without shifting the stack top pointer in environment *N*). Currently set data type isn't changed by this command.
- **pointer commands**: Let *N* and *M* be pointer names already declared before the command. Then the following are possible pointer commands:
  - `$N` (**pointer dereference**): Pushes the value stored at the address pointed to by pointer named *N*.
  - `$:N` (**pointer assignment**): Pops 1 value and stores it to the address pointed to by pointer named *N*.

    - `$N>M` (**pointer copy**): Copies the address stored in pointer named *N* to pointer named *M*.
    - `$N=M` (**pointer comparison**): Pushes value 0 if addresses in pointer *N* and *M* are the same, otherwise pushes 1 if address in pointer *N* is greater than address in pointer *M*, otherwise pushes 2.
    - `$>N` (**pointer increment**): Increments the address stored in pointer named *N* by 1.
    - `$<N` (**pointer decrement**): Decrements the address stored in pointer named *N* by 1.
    - `$+N` (**pointer add**): Pops 1 value, interprets it as a signed value and adds it to the address stored in pointer named *N*. If *N* is 0, then the final address in pointer `0` will be that which there was just before invoking this command plus *N* (i.e. pop is ignored here).
    - `$` (**push**): Pops 1 value, pushes the value *x* places below stack top.
    - `$$` (**address**): Pushes the address stored in pointer 0 (stack top).
- **directives**: Directives are compile-time commands that modify the language processor's state or behavior when it is analyzing the code, they don't represent a run time action (for example it is NOT possible to dynamically allocate a pointer with the pointer definition directive). Possible directives are:
    - `~N` where *N* is a plain decimal numeric literal corresponding to one of the supported type environments (**type environment switch**): Makes the language processor immediately change its current type environment to *type N*.
    - `~I` or `~I:N` where *I* is identifier and *N* is a plain decimal numeric literal (**pointer definition**): Creates a new pointer within the current type environment with the name *I* and size equal to the number represented by *N*. Pointer name must be unique among all pointer names within given type environment. The first variant of the command (without explicitly specified *N*) supposes *N* equal to 1. Size *N* here means setting the pointer value so that it points to an address which is followed by at least *N* cells of which none is pointed to by any other pointer. Note that *N* = 0 is a valid value.
    - `~:I` where *I* is identifier (**label**): Creates a new label with the name *I*. The label name must be unique among all label names in the source code. If the goto command isn't supported, this directive doesn't have to be supported.
    - `~S` where *S* is string literal (**file include**): Allows inclusion of source code from another file *F* identified by the string literal *S* which is a file name in given platform's format. If this command is encountered during compilation, it is replaced by the content of file *F*. Inclusion of a file that has already been opened is to be ignored (while issuing possible warning), i.e. any source code file can be processed at most once.

Any above mentioned command that performs at least one pop, except for the `-->` command (whose number of pops may be unknown at compile time), has also a **non-popping variant** whose name is formed by appending `'` to the name of the command. E.g. the command `+'` performs the same operation as command `+` but doesn't perform any pops.

Structure

Simple **branch** is of the following form

```
? COMMANDS .
```

Optionally there can also be an *else* branch, i.e.

```
? COMMANDS ; COMMANDS2 .
```

When at run time branch is encountered, a value is popped in the current type environment and it is checked whether this value was non-zero -- if so, `COMMANDS` are performed; if not and if an else branch is present, `COMMANDS2` are performed.

`?'` can be used instead of `?` in which case no pop is performed by the branch condition check.

Loop is of form

```
@ COMMANDS .
```

When at run time loop is encountered, a value is popped in the current type environment and it is checked whether the popped value was non-zero -- if so, `COMMANDS` are processed and the control jumps right before the loop start.

`!@` is the **loop break** command. When this command is encountered at run time, program control jumps right after the end of the current innermost loop. If this command is encountered outside any loop, an error occurs (it is recommended to be a compile time error but interpreters may also resort to a run time error only upon attempted execution of this command).

`@'` can be used instead of `@` in which case stack top is not popped when checking the loop condition. `@@` can be used instead of `@` to create an infinite loop, i.e. a loop that doesn't perform any condition checks (and no pops) and repeats until exited with the break command.

Function definition has the following format

```
functionName: COMMANDS .
```

Here `functionName` is the function name, an identifier unique among all other defined functions, and `COMMANDS` is the function body that is performed when the function is called. Function definition must appear on the outermost level of global scope, i.e. it cannot appear inside another function definition, inside a branch, inside a loop etc.

Function call is performed simply with a command equating the function name (argument passing and return values have to be performed via memory cells). Function calls work with a separate return address stack (different from the general stacks of data type environments), i.e. recursion is possible. It is possible to call a function that will be defined after the function call appears (forward declarations aren't needed).

`!.` is the **exit** command. If this command is encountered inside a function, the function is immediately left (just as if its end was reached). If this command is encountered outside a function, it immediately halts the program.

External functions can be called; once processing of source code is finished, any call to a function not defined in the source code is considered a call of an external function. **Exposing functions** from the language is also possible; the language processor can make all defined functions callable from external environments. In both cases the conventions for parameter passing and returning values are left to be decided by every implementation. Calling external functions and exposing functions is to serve the compatibility with other languages and computing environments, libraries made within the language itself are to simply use the file include command.

`>S` where *S* is identifier is the **goto** command. Support of this command is optional. When this command is encountered at run time, the program control jumps to the location of label with name *S*. Jumping to a label which will be defined after the goto command is possible. Jumping into functions from outside is possible with behavior matching that of the traditional call stack implementation of function calls; when end of a function that's been jumped into is reached, control jumps to the address on top of call stack (and call stack is popped), unless call stack is empty in which case a run time error occurs.

Preprocessor

The language may optionally include a preprocessor that allows for automatically modifying the underlying source code before compilation. I.e. with preprocessor activated the source code is first preprocessed to form a new source code which is then compiled. Preprocessor uses the same language as has been defined so far.

Blocks of preprocessing code start with the `[` character and end with the `]` character. Note that these characters are considered preprocessing delimiters everywhere, i.e. even if they appear inside a comment or string literal etc. These characters don't have to be separated by blank characters as language tokens do. Preprocessing blocks cannot be nested.

If preprocessing is active, any source code file's content is to be prepended with the `]` character and appended with the `[` character (this is important for correct behavior during file includes).

Suppose we have source code *S* (with `]` and `[` already prepended/appended and with all include commands already replaced with the corresponding file contents which have also been prepended/appended with the same characters) which is to be preprocessed to form the final preprocessed source code *S2*. *S2* can be seen as obtained from *S* like this:

1. Replace all strings that start with `]` and end with next nearest `[` with language code that prints the string minus the bounding `]` and `[` characters. This code may only modify memory cells above the current stack top in each type environment and after execution has to leave addresses in all pointers unchanged. Let the code at this stage be called *S1*. *S1* is the results of the **first preprocessing stage**.
2. Execute *S1*; the output it gives is *S2*. *S2* is the result of the **second preprocessing stage**. This is the final preprocessed code.

Note that this is only a description of semantics, the language preprocessor doesn't actually have to create the intermediate code *S1* if it can generate *S2* in another way.

cmn_all.cmn

```
# comun self hosted implementation: whole comun library
#
# This module just includes all the submodules of the library. It is to be used
# if you want to include the whole comun library easily.

~"cmn_general.cmn"
~"cmn_tokenizer.cmn"
~"cmn_bytecode.cmn"
~"cmn_bytecode_extra.cmn"
~"cmn_compiler.cmn"
~"cmn_optimizer.cmn"
~"cmn_interpreter.cmn"
```

cmn_bytecode.cmn

```
# comun self hosted implementation: bytecode library
# module prefix: b
#
# This module contains basic resource for handling the implementation specific
# comun BC. More advanced BC opeartions, such as its optimization and
# interpretation, are to be implemented elsewhere.
#
# Before including this library, define the following:
#
# - CMN_bBYTECODE_SIZE: Func pushing the num. of CMN_bBytecode cells, to TE 32.
# - CMN_bBytecode:      Ptr to mem in TE 16 where BC will be stored, must point
#                       to a block of at least CMN_bBYTECODE_SIZE free cells.

~"cmn_general.cmn"

~16 ~CMN_bCurrentInstr:0 ~0   # ptr to current instr within the BC
~32 ~CMN_bCurrentInstrAddr ~0 # BC addr of current instr

# Size of BC header in bytes.
CMN_bHEADER_SIZE: 8 .

# instr opcode modes:
CMN_bMODE21:  +b00000000 . # pop 2, push 1
CMN_bMODE1C1: +b00000001 . # pop 1, use C, push 1
CMN_bMODE11:  +b00000010 . # pop 1, push 1
CMN_bMODE01:  +b00000011 . # push 1

# instr opcode groups:
CMN_bOPCODE_AD: +b00100000 .
CMN_bOPCODE_SU: +b00100100 .
CMN_bOPCODE_MU: +b00101000 .
CMN_bOPCODE_DI: +b00101100 .
CMN_bOPCODE_DS: +b00110000 .
CMN_bOPCODE_MO: +b00110100 .
CMN_bOPCODE_MS: +b00111000 .
CMN_bOPCODE_GR: +b01000000 .
CMN_bOPCODE_GE: +b01000100 .
CMN_bOPCODE_SM: +b01001000 .
CMN_bOPCODE_SE: +b01001100 .
CMN_bOPCODE_GS: +b01010000 .
CMN_bOPCODE_BS: +b01010100 .
CMN_bOPCODE_SS: +b01011000 .
CMN_bOPCODE_LS: +b01011100 .
CMN_bOPCODE_EQ: +b01100000 .
CMN_bOPCODE_NE: +b01100100 .
CMN_bOPCODE_BA: +b01101000 .
CMN_bOPCODE_BO: +b01101100 .
CMN_bOPCODE_BX: +b01110000 .
CMN_bOPCODE_LA: +b01110100 .
CMN_bOPCODE_LO: +b01111000 .
CMN_bOPCODE_LX: +b01111100 .
CMN_bOPCODE_SR: +b10000100 .
CMN_bOPCODE_SL: +b10001000 .

# special instr opcodes:
CMN_bOPCODE_END: +b00000000 .
CMN_bOPCODE_NOP: +b00000001 .
CMN_bOPCODE_DES: +b00000010 .
CMN_bOPCODE_COC: +b00000011 .
CMN_bOPCODE_ERR: +b00000100 .
CMN_bOPCODE_CAL: +b00000111 .
CMN_bOPCODE_CAE: +b00001000 .
CMN_bOPCODE_RET: +b00001001 .
CMN_bOPCODE_JIA: +b00001010 .
CMN_bOPCODE_JNA: +b00001011 .
CMN_bOPCODE_JMA: +b00001100 .
CMN_bOPCODE_INI: +b00001111 .
CMN_bOPCODE_PSC: +b00010000 .
CMN_bOPCODE_PAC: +b00010001 .
CMN_bOPCODE_PAX: +b00010010 .
CMN_bOPCODE_PCO: +b00010011 .
CMN_bOPCODE_MEX: +b00010100 .
CMN_bOPCODE_MGE: +b00010101 .
CMN_bOPCODE_PCM: +b00010110 .
CMN_bOPCODE_PUX: +b00010111 .
CMN_bOPCODE_CON: +b00011010 .
CMN_bOPCODE_CND: +b00011011 .
CMN_bOPCODE_SWP: +b00011100 .
CMN_bOPCODE_TRA: +b00011101 .
CMN_bOPCODE_POP: +b00011110 .
CMN_bOPCODE_OUT: +b00011111 .

# typical instr opcodes:
CMN_bOPCODE_ADX: CMN_bOPCODE_AD CMN_bMODE21  | .
CMN_bOPCODE_ADC: CMN_bOPCODE_AD CMN_bMODE1C1 | .
CMN_bOPCODE_SUX: CMN_bOPCODE_SU CMN_bMODE21  | .
CMN_bOPCODE_SUC: CMN_bOPCODE_SU CMN_bMODE1C1 | .
CMN_bOPCODE_MUX: CMN_bOPCODE_MU CMN_bMODE21  | .
CMN_bOPCODE_MUC: CMN_bOPCODE_MU CMN_bMODE1C1 | .
CMN_bOPCODE_DIX: CMN_bOPCODE_DI CMN_bMODE21  | .
```

```
CMN_bOPCODE_DIC: CMN_bOPCODE_DI CMN_bMODE1C1 | .
CMN_bOPCODE_DSX: CMN_bOPCODE_DS CMN_bMODE21  | .
CMN_bOPCODE_DSC: CMN_bOPCODE_DS CMN_bMODE1C1 | .
CMN_bOPCODE_MOX: CMN_bOPCODE_MO CMN_bMODE21  | .
CMN_bOPCODE_MOC: CMN_bOPCODE_MO CMN_bMODE1C1 | .
CMN_bOPCODE_MSX: CMN_bOPCODE_MS CMN_bMODE21  | .
CMN_bOPCODE_MSC: CMN_bOPCODE_MS CMN_bMODE1C1 | .
CMN_bOPCODE_GRX: CMN_bOPCODE_GR CMN_bMODE21  | .
CMN_bOPCODE_GRC: CMN_bOPCODE_GR CMN_bMODE1C1 | .
CMN_bOPCODE_GEX: CMN_bOPCODE_GE CMN_bMODE21  | .
CMN_bOPCODE_GEC: CMN_bOPCODE_GE CMN_bMODE1C1 | .
CMN_bOPCODE_SMX: CMN_bOPCODE_SM CMN_bMODE21  | .
CMN_bOPCODE_SMC: CMN_bOPCODE_SM CMN_bMODE1C1 | .
CMN_bOPCODE_SEX: CMN_bOPCODE_SE CMN_bMODE21  | .
CMN_bOPCODE_SEC: CMN_bOPCODE_SE CMN_bMODE1C1 | .
CMN_bOPCODE_GSX: CMN_bOPCODE_GS CMN_bMODE21  | .
CMN_bOPCODE_GSC: CMN_bOPCODE_GS CMN_bMODE1C1 | .
CMN_bOPCODE_BSX: CMN_bOPCODE_BS CMN_bMODE21  | .
CMN_bOPCODE_BSC: CMN_bOPCODE_BS CMN_bMODE1C1 | .
CMN_bOPCODE_SSX: CMN_bOPCODE_SS CMN_bMODE21  | .
CMN_bOPCODE_SSC: CMN_bOPCODE_SS CMN_bMODE1C1 | .
CMN_bOPCODE_LSX: CMN_bOPCODE_LS CMN_bMODE21  | .
CMN_bOPCODE_LSC: CMN_bOPCODE_LS CMN_bMODE1C1 | .
CMN_bOPCODE_EQX: CMN_bOPCODE_EQ CMN_bMODE21  | .
CMN_bOPCODE_EQC: CMN_bOPCODE_EQ CMN_bMODE1C1 | .
CMN_bOPCODE_NEX: CMN_bOPCODE_NE CMN_bMODE21  | .
CMN_bOPCODE_NEC: CMN_bOPCODE_NE CMN_bMODE1C1 | .
CMN_bOPCODE_BAX: CMN_bOPCODE_BA CMN_bMODE21  | .
CMN_bOPCODE_BAC: CMN_bOPCODE_BA CMN_bMODE1C1 | .
CMN_bOPCODE_BOX: CMN_bOPCODE_BO CMN_bMODE21  | .
CMN_bOPCODE_BOC: CMN_bOPCODE_BO CMN_bMODE1C1 | .
CMN_bOPCODE_BXX: CMN_bOPCODE_BX CMN_bMODE21  | .
CMN_bOPCODE_BXC: CMN_bOPCODE_BX CMN_bMODE1C1 | .
CMN_bOPCODE_LAX: CMN_bOPCODE_LA CMN_bMODE21  | .
CMN_bOPCODE_LAC: CMN_bOPCODE_LA CMN_bMODE1C1 | .
CMN_bOPCODE_LOX: CMN_bOPCODE_LO CMN_bMODE21  | .
CMN_bOPCODE_LOC: CMN_bOPCODE_LO CMN_bMODE1C1 | .
CMN_bOPCODE_LXX: CMN_bOPCODE_LX CMN_bMODE21  | .
CMN_bOPCODE_LXC: CMN_bOPCODE_LX CMN_bMODE1C1 | .
CMN_bOPCODE_BNO: +b10000000 CMN_bMODE11      | .
CMN_bOPCODE_SRX: CMN_bOPCODE_SR CMN_bMODE21  | .
CMN_bOPCODE_SRC: CMN_bOPCODE_SR CMN_bMODE1C1 | .
CMN_bOPCODE_SLX: CMN_bOPCODE_SL CMN_bMODE21  | .
CMN_bOPCODE_SLC: CMN_bOPCODE_SL CMN_bMODE1C1 | .
CMN_bOPCODE_ADR: +b11110000 CMN_bMODE01      | .
CMN_bOPCODE_INU: +b11111000 CMN_bMODE01      | .
CMN_bOPCODE_INP: +b11111100 CMN_bMODE01      | .

CMN_bDES_IF:      1  .
CMN_bDES_ELSE:    2  .
CMN_bDES_ENDIF:   3  .
CMN_bDES_LOOP:    4  .
CMN_bDES_BREAK:   5  .
CMN_bDES_ENDLOOP: 6  .
CMN_bDES_FUNDEF:  7  .
CMN_bDES_EXIT:    8  .
CMN_bDES_GOTO:    9  .
CMN_bDES_LABEL:   10 .

# arg: 3addr
# Sets the current instr pointer within BC to 3addr.
CMN_bCurrentInstrSetAddr:
  ~32 $:CMN_bCurrentInstrAddr ~0
  ~16 0 ~0 CMN_bHEADER_SIZE 1 |> >16

  ~16
    $CMN_bBytecode>CMN_bCurrentInstr
    $+CMN_bCurrentInstr
  ~0

  ~32 $CMN_bCurrentInstrAddr +xffff <= ? ~0
    ~16 0 ~0
    ~32 $CMN_bCurrentInstrAddr >16 ~0
    ~16 $+CMN_bCurrentInstr ~0
  ;
    # addr bigger than 16 bits, use loop

    ~32
      $CMN_bCurrentInstrAddr @'
        --
        ~16 $>CMN_bCurrentInstr ~0
      .
      ^
    ~0
  .
.

# ret: instr
# Gets the currently pointed 16 bit instr.
CMN_bCurrentInstrGet: 0 ~16 $CMN_bCurrentInstr >0 ~0 .

# arg: 3adr
# ret: instr
# Gets instr at given addr in BC.
CMN_bGetInstrAt:
  ~32
    $CMN_bCurrentInstrAddr >< CMN_bCurrentInstrSetAddr
    CMN_bCurrentInstrGet
    CMN_bCurrentInstrSetAddr
  ~0
.

# Shifts to the next instr.
CMN_bCurrentInstrShift:
  ~16 $>CMN_bCurrentInstr ~0
  ~32 $CMN_bCurrentInstrAddr ++ $:CMN_bCurrentInstrAddr ~0
.

# Shifts to previous instr.
CMN_bCurrentInstrShiftBack:
  ~16 $<CMN_bCurrentInstr ~0
  ~32 $CMN_bCurrentInstrAddr -- $:CMN_bCurrentInstrAddr ~0
.

# ret: 2cs
# Computes the BC checksum, i.e. sum of all bytes in BC section mod 256.
CMN_bChecksum:
  # TODO: untested
  ~16 0 ~0 CMN_bHEADER_SIZE >16

  ~16
    $CMN_bBytecode>_CMN_g2P1
    1 |> $+_CMN_g2P1

    0 # result

    $_CMN_g2P1 @
      $_CMN_g2P1 $0 8 |> + +
      $>_CMN_g2P1
      $_CMN_g2P1
    .
    +xff &
  ~0
.

# arg: opcode typeEnv noPop const constCont
# ret: instr
# Creates a 16 bit instr code, typeEnv is 0, 1, 2 or 3, noPop is 0 (may pop) or
# 1 (won't pop), const is 0 to 16 (IC), constCont is 0 or 1
# (IC will continue).
CMN_bMakeInstr: 4 |< | >< 5 |< | >< 6 |< | 8 |< | .

CMN_bInstrOpcode: +x00ff & .
CMN_bInstrTypeEnv: 14 |> +b11 & .
CMN_bInstrWontPop: 13 |> 1 & .
CMN_bInstrConstContinues: 12 |> 1 & .
CMN_bInstrIsTypical: 4 |> +x0f & 2 >= .

# arg: opcode
# ret: bool
# Says if given opcode represents an instr that makes use of the NoPop bit.
CMN_bOpcodeUsesNoPop:
  $0 CMN_bOPCODE_RET <=
  $1 CMN_bOPCODE_JMA = |
  $1 CMN_bOPCODE_INI = |
  $1 CMN_bOPCODE_PSC = |
  $1 CMN_bOPCODE_PAC = |
  $1 CMN_bOPCODE_PCM = |
  $1 CMN_bOPCODE_ADR = |
  >< CMN_bOPCODE_INP = |
  !!
.

# arg: opcode
# ret: bool
# Says if given opcode makes use of TE bits.
CMN_bOpcodeUsesTypeEnv:
  $0 CMN_bOPCODE_RET <= $1 CMN_bOPCODE_COC != &
  $1 CMN_bOPCODE_JMA = |
  >< CMN_bOPCODE_INI = |
  !!
.

CMN_bCurrentInstrTypeEnv: CMN_bCurrentInstrGet CMN_bInstrTypeEnv .

# arg: instr
# ret: bool
# Checks if instr may touch the ptr table (only using ST addr doesn't count).
CMN_bInstrUsesPtrs:
  CMN_bInstrOpcode $0 CMN_bOPCODE_PSC >= >< CMN_bOPCODE_PCM <= &
.

# ret: p2 p1
# Analyzes currently pointed instr, pushes p2 and p1, each one being the num of
# ptrs the instr uses. If given pointer is not used, the val will be -1 (i.e. if
# P1 is -1, the instr doesn't work with ptrs).
CMN_bCurrentInstrExtractPtrs:
  CMN_bCurrentInstrGet $0 CMN_bInstrUsesPtrs ?
    CMN_bInstrOpcode

    $0 CMN_bOPCODE_PAX = $1 CMN_bOPCODE_MEX = || $1 CMN_bOPCODE_MGE = || ?
      # pointer in C
      ^ -1 CMN_bCurrentInstrGetConst
    ;
      # pointer in C1 and maybe also C2

      CMN_bCurrentInstrGetConsts32

      0 ~32 >0 ~0
      0 ~32 >0 ~0
      ><

      $2 CMN_bOPCODE_PCO != $3 CMN_bOPCODE_PCM != && ?
        # C2 pointer not used, set to -1

        -1 $:2
      .

      >< $:2
    .
  ;
    ^ -1 -1
  .
.

# ret: mems
# Pushes a num that in its lowest 4 bits says which TE mems get affected by it.
CMN_bInstrAffectsMems:
  $0 CMN_bInstrIsTypical ?
    CMN_bInstrTypeEnv 1 >< |<
  ;
    $0 CMN_bInstrOpcode CMN_bOPCODE_TRA = ?
      $0 8 |> +x0f & 1 >< |<
      >< CMN_bInstrTypeEnv 1 >< |<  |
    ;
      $0 CMN_bInstrOpcode
      $0 CMN_bOPCODE_RET > ><
      $0 CMN_bOPCODE_PCO != ><
      $0 CMN_bOPCODE_PAC != ><
      $0 CMN_bOPCODE_PSC != ><
      CMN_bOPCODE_JMA != & & & & ?
        CMN_bInstrTypeEnv 1 >< |<
      ;
        ^ 0
      .
    .
  .
.

# ret: 3const
# Gets currently pointed instr's IC (considering also potential continuation in
# following COC instrs).
CMN_bCurrentInstrGetConst32:
  ~16 $CMN_bCurrentInstr>_CMN_g2P1 ~0
  ~32
    0 # shift
    0 # result (r)

    1 @
      0
      ~16 $_CMN_g2P1 >32 $>_CMN_g2P1 ~32
      $0 8 |> +x0f & $3 |< # constAdd
      $3 4 + $:4           # shift += 4
      >< 12 |> 1 &         # contCont
      ^
      +                    # result += constAdd
      $>0 $>0
      $:1
    .

    >< ^
```

```
  ~0
.

# ret: const
CMN_bCurrentInstrGetConst: 0 ~32 CMN_bCurrentInstrGetConst32 >0 ~0 .

# ret: bitCount
# Pushes the bit count of currently pointed instr's IC (will be a multip. of 4).
CMN_bCurrentInstrGetConstBits:
  0 # prepare space

  ~16
    $CMN_bCurrentInstr>_CMN_g2P1

    0
    1 @
      ++
      $_CMN_g2P1 12 |> 1 & # const. continues?
      $>_CMN_g2P1
    .

    2 |< # times 4
    >0
  ~0
.

# ret: 3c2 3c1
# Considers currently pointed instr as having two ICs, pushes them.
CMN_bCurrentInstrGetConsts32:
  CMN_bCurrentInstrGetConstBits 1 |> $0
  ~32
    CMN_bCurrentInstrGetConst32 $0

    0 ~0 >32 ~32
    |>
    ><

    +xffffffff
    0 ~0 >32 ~32
    |< ! &
  ~0
.

# ret: C2 C1
CMN_bCurrentInstrGetConsts:
  CMN_bCurrentInstrGetConsts32
  0 ~32 >0 ~0 0 ~32 >0 ~0 ><
.

# Outputs BC in binary form.
CMN_bOutput:
  ~16 $CMN_bBytecode>_CMN_g2P1 ~0

  CMN_bHEADER_SIZE 1 |> @'
    ~16 $_CMN_g2P1 +xff & -> $_CMN_g2P1 8 |> -> $>_CMN_g2P1 ~0
    --
  .
  ^

  ~32 0 CMN_bCurrentInstrSetAddr ~0

  1 @
    CMN_bCurrentInstrGet
    $0 +xff & ->
    $0 8 |> ->
    CMN_bCurrentInstrShift
  .
.

# TODO: sanity check func?
```

---

cmn_bytecode_extra.cmn

```
# comun self hosted implementation: bytecode extra library
# module prefix: e
#
# This module contains additional resources for handling the implementation
# specific comun BC. This module is separated from the basic BC module so that
# programs not needing advanced BC manipulation aren't burdened with compiling
# this extra code, which might increase RAM requirements (even if this code is
# later optimized out).

~"cmn_bytecode.cmn"

# arg: opcode
# ret: char3 char2 char1
# Converts instr opcode to three letter mnemonic.
CMN_eOpcodeToMnemo:
  # TODO: this all could be further minimized, e.g. make a helper func for
  # pushing a series of '?'s

  # TODO: move this whole func elsewhere to make this module smaller?

  $0>_CMN_gP1

  $0 16 % 3 * ++ ++
  >< 16 /
          #   f  e  d  c  b  a  9  8  7  6  5  4  3  2  1  0
    $0 0 = ?  "INI??????JMAJNAJIARETCAECAL??????ERRCOCDESNOPEND"
  ; $0 1 = ?  "OUTPOPTRASWPCNDCON??????PUXPCMMGEMEXPCOPAXPACPSC"
  ; $0 2 = ?  "??????DICDIX??????MUCMUX??????SUCSUX??????ADCADX"
  ; $0 3 = ?  "????????????????????MSCMSX??????MOCMOX??????DSCDSX"
  ; $0 4 = ?  "??????SECSEX??????SMCSMX??????GECGEX??????GRCGRX"
  ; $0 5 = ?  "??????LSCLSX??????SSCSSX??????BSCBSX??????GSCGSX"
  ; $0 6 = ?  "??????BOCBOX??????BACBAX??????NECNEX??????EQCEQX"
  ; $0 7 = ?  "??????LXCLXX??????LOCLOX??????LACLAX??????BXCBXX"
  ; $0 8 = ?  "????????????????????SLCSLX??????SRCSRX???BNO???????"
  ; $0 15 = ? "INP??????????INU?????????????????????ADR?????????"
  ;
    $_CMN_gP1>0
    ^
    "???"
    !.
  . . . . . . . . . .

  $_CMN_gP1>0

  $+_CMN_gP1

  $_CMN_gP1 $>_CMN_gP1
  $_CMN_gP1 $>_CMN_gP1
  $_CMN_gP1
.

# arg: 3ptrI
# ret: 3ptrTableI
# Converts ptr index as encoded in an instr to an index in PT, doesn't check
# index validity.
CMN_ePtrIToPtrTableI: ~32 $0 15 > >< 15 - 0 ?? ~0 .

# arg: pc0 ss0 pc1 ss1 pc2 ss2 pc3 ss3
# Analyzes loaded BC, pcN is exact num of ptrs used in TE N (num of user ptrs
# plus ptr 0, excluding ptrs 1, 2, ...) and ssN is estimated min. stack size (in
# cells) in TE N. If TE N isn't used by the BC, pcN and ssN will be both 0.
CMN_eAnalyze:
  ~32 0 CMN_bCurrentInstrSetAddr ~0
  0 0 0 0 0 0 0 0 # results

  1 @
    CMN_bCurrentInstrExtractPtrs

    $0 -1 != ?
      CMN_gMaxS
      CMN_ePtrIToPtrTableI # TODO: WTF THIS CAN'T BE HERE

      $0>_CMN_gP1
      3 CMN_bCurrentInstrTypeEnv - -2 * -2 + $+_CMN_gP1
      $_CMN_gP1 CMN_gMax $:_CMN_gP1
    ;
      ^ ^
    .

    CMN_bCurrentInstrGet

    # setting pointer to high addr, we simply add the current esitmate to this
    # addr -- this is not accurate and can overestimate mem if many ptrs are set
    # a high addr
    $0 CMN_bInstrOpcode CMN_bOPCODE_PSC = ?
      CMN_bCurrentInstrGetConsts ^
      $0>_CMN_gP1
      3 CMN_bCurrentInstrTypeEnv - -2 * 2 - $+_CMN_gP1
      $_CMN_gP1 + $:_CMN_gP1
    .

    $0 CMN_bInstrAffectsMems

    $0 +b1000 & ? $2 ++ $:3 .
    $0 +b0100 & ? $4 ++ $:5 .
    $0 +b0010 & ? $6 ++ $:7 .
       +b0001 & ? $7 ++ $:8 .

    ~16
      $>CMN_bCurrentInstr
    ~0
  .

  # add 1 ptr (ST) to ptr count in each TE actually used
  $7 $7 | 0 != $8 + $:8
  $5 $5 | 0 != $6 + $:6
  $3 $3 | 0 != $4 + $:4
  $1 $1 | 0 != $2 + $:2
.

# arg: instr
# ret: 0 sn ... s2 s1 s0
# Converts a 16 bit instr to a zero terminated human readable string.
CMN_eInstrToStr:
  $:_CMN_gV1

  0 # string terminator

  $_CMN_gV1 CMN_bInstrConstContinues ">" " " ??
  $_CMN_gV1 CMN_bInstrWontPop "'" " " ??
  $_CMN_gV1 +xff & CMN_eOpcodeToMnemo " "

  $_CMN_gV1 CMN_bInstrTypeEnv # TE string
  $0 3 = + 8 *
  $0 10 % "0" + >< 10 / "0" +

  "   "

  16  # to binary string
  @'
    --

    $0 7 = ?
      " " ><
    .

    $_CMN_gV1 15 $2 - |> 1 & "0" + ><
  .
  ^
.

# ret: succ
# Loads BC from input to CMN_bBytecode, returns 1 (OK) or 0 (error: BC too big).
CMN_eLoadBC:
  ~32 CMN_bBYTECODE_SIZE ~0
  ~16 $CMN_bBytecode>_CMN_g2P2 ~0
  ~32
    @' # set _CMN_g2P2 to the end of BC mem
      ~16 $>_CMN_g2P2 ~32
      --
    .
    ^
  ~0

  ~16
    $CMN_bBytecode>_CMN_g2P1

    0 ~0 CMN_bHEADER_SIZE 2 / >16 ~16 $:_CMN_g2V1

    @@
      <? !! ?
        ~0 1 ~16
        !@
      .

      <- <- 8 |< |

      $:_CMN_g2P1

      $_CMN_g2V1 ?
        $_CMN_g2V1 -- $:_CMN_g2V1
      ;
        $_CMN_g2P1 !! ? # END instr => stop reading
          ~0 1 ~16
          !@
        .
      .

      $_CMN_g2P1=_CMN_g2P2 !! ?
        ~0 0 ~16 # error, BC too big
        !@
      .

      $>_CMN_g2P1
    .
  ~0
.
```

cmn_compiler.cmn

```
# comun self hosted implementation: compiler library
# module prefix: c
#
# This module contains resources that can be used to compile comun source code
# to the implementation specific comun BC. Doesn't do nontrivial optimizations.
#
# Before including this library, define the following:
#
# - CMN_cMEM_SIZE: Func returning a const, size of CMN_cMem in cells, in TE 32.
# - CMN_cMem:      Ptr to mem in TE 32 that will be used by com as a working
#                  mem, this affects symbol table size.
# - CMN_cInclude:  Func that will be called by the compiler when file include is
#                  encountered, with filename as traditional zero terminated str
#                  on stack in TE 0, which has to be popped by the func.
#
# The following are details concerning internal working of the com.
#
# The com mem is organized like so:
#
#   S0C S0B S0A S0N  S1C S1B S1A S1N ...
#
# SxA/B/C encodes xth symbol in symbol tab., SxN directly stores the symbol val.
#
# The com DOESN'T USE A PARSE STACK, which makes the code simpler and reduces
# mem usage (but is potentially a bit slower). Instead it work right in the BC
# it is generating (i.e. it effectively uses the BC as stack). A small HACK is
# used here: we use the no-pop bit to mark unhandled instrs (the instrs we deal
# with here never make use of no-pop bit themselves, so we are free to use it
# temporarily). Once such instr is handled, we reset the no-pop bit (equivalent
# to popping it from a stack). The portions of code making use of this are
# commented with "HACK".
#
# Symbol table: array of pairs (pseudohash, N) where pseudohash is a 16 char
# value associated with identifier whose first symbol indicates the type of
# symbol, and N is the symbol value. A symbol can also be seen as having a
# value M, which is its order among all records of the same type (starting with
# same letter). The following are possibly symbol types:
#
#  f: func location, N = BC address of the func
#  0: ptr in TE 0, N = size (in cells), M = ptr ID
#  1: ptr in TE 8, ...
#  2: ptr in TE 16, ...
#  3: ptr in TE 32, ...
#  i: included file name (to prevent double includes), N is unused
#  c: yet unknown func call, N is call ID; this record is used when a yet
#     undefined func is called; in such case CAE (instead of CAL) is produced,
#     storing N, which can later be used to resolve previously unknown calls (or
#     determine the call to really be of external func).
#  g: goto ID, N is ID
#  l: label, N is addr

~"cmn_general.cmn"
~"cmn_bytecode.cmn"
~"cmn_tokenizer.cmn"

~CMN_cState         # compiler state

~32
  ~_CMN_cMemEnd:0   # last cell of CMN_cpMem
  ~_CMN_cInterState # internal state of com, structure is this:
                    # - 2 highest bits: current TE
                    # - 1 next highest bit: no_pop bit
                    # - 1 next highest bit: default addr size
                    #   (0 -> 16bit, 1 -> 32bit)
                    # - 16 lowest bits: symbol table item count
                    # - rest of the bits: unknown jump count
~0

~16 ~_CMN_cBytecodeEnd:0 ~0    # last cell of CMN_bBytecode

# compiler states:
CMN_cSTATE_OK:                0  . # all fine, continuing
CMN_cSTATE_DONE:              1  . # finished without error
CMN_cSTATE_ERROR:             2  . # further unspecified com error
CMN_cSTATE_ERROR_TOKEN:       3  . # error: badly formed token
CMN_cSTATE_ERROR_UNEXPECTED:  4  . # error: unexpected token or end
CMN_cSTATE_ERROR_NAME_UNK:    5  . # error: unknown name
CMN_cSTATE_ERROR_NAME_EXISTS: 6  . # error: name redefined
CMN_cSTATE_ERROR_SYMBOL_TAB:  7  . # error: symbol table ran out of mem
CMN_cSTATE_ERROR_BC_TOO_BIG:  8  . # error: ran out of mem for BC
CMN_cSTATE_ERROR_ADDR_TOO_BIG: 9  . # error: addr. too big (increase addr. size)
CMN_cSTATE_ERROR_UNSUPPORTED: 10 . # error: unsupported feature

# arg: n
# ret: c
# Converts lower 6 bits of n to ASCII char, for use by string pseudohash func
CMN_cNumToChar:
  64 %
  $0 26 < ?  "a" +       ;
  $0 52 < ?  26 - "A" +  ;
  $0 62 < ?  52 - "0" +  ;
             ^ "_" # stands for both 62 and 63
  . . .
.

# arg: c
# ret: n
# Performs the opposite of CMN_cNumToChar.
CMN_cCharToNum:
  $0 "a" >= $1 "z" <= & ?  "a" -       ;
  $0 "A" >= $1 "Z" <= & ?  "A" - 26 +  ;
  $0 "0" >= $1 "9" <= & ?  "0" - 52 +  ;
                           ^ 63
  . . .
.

# ret: te
# Gets currently set TE
_CMN_cCurrentTypeEnvGet: 0 ~32 $_CMN_cInterState 30 |> >0 ~0 .

# arg: te
_CMN_cCurrentTypeEnvSet:
  ~32 $_CMN_cInterState +x3fffffff & 0 ~0 >32 ~32 30 |< | $:_CMN_cInterState ~0
.

# ret: np
_CMN_cNoPopGet: 0 ~32 $_CMN_cInterState 29 |> 1 & >0 ~0 .

# arg: np
_CMN_cNoPopSet:
  ~32 $_CMN_cInterState +xdfffffff & 0 ~0 >32
  ~32 29 |< | $:_CMN_cInterState ~0
.

# arg: opcode c4
# If com state is OK, appends new instr with said opcode to BC, if needed then
# instr TE and no-pop bits are set according to _CMN_cInterState, IC is set to
# c4, which must be < 16 may (use another func for larger IC). On error (no more
# space) com state is set.
CMN_cEmitInstr:
  CMN_bCurrentInstrShift

  ~16 $CMN_bCurrentInstr=_CMN_cBytecodeEnd 1 != ? ~0
    $1 CMN_bOpcodeUsesTypeEnv _CMN_cCurrentTypeEnvGet 0 ??
    ><
    $2 CMN_bOpcodeUsesNoPop _CMN_cNoPopGet 0 ??
    >< 0
    CMN_bMakeInstr ~16 0 ~0 >16 ~16 $:CMN_bCurrentInstr ~0
  ;
    CMN_bCurrentInstrShiftBack
    CMN_cSTATE_ERROR_BC_TOO_BIG $:CMN_cState
  .
.

# arg: 3c
# ret: n
# Returns size (in instrs) a constant will take to store.
CMN_cConstInstrSize: 0 ~32 1 @ ~0 ++ ~32 4 |> $0 . ^ ~0 .

# arg: 3c n
# Fills current instr and n - 1 following COCs with given IC.
_CMN_cFillConstN:
  0 _CMN_cNoPopSet

  @@
    ~16 0 ~0
    ~32 $0 >16 4 |> ~0
    ~16 +b1111 & 8 |< $CMN_bCurrentInstr +xe0ff & | $:CMN_bCurrentInstr ~0

    --
    ?'
      ~16 $CMN_bCurrentInstr +x1000 | $:CMN_bCurrentInstr ~0
      CMN_bOPCODE_COC 0 CMN_cEmitInstr
    ;
      ~32 ^ ~0
      !@
    .
  .
  ^
.

# arg: 3c
# Fills current instr with given IC, may append COC instrs to make it fit.
CMN_cFillConst: ~32 $0 ~0 CMN_cConstInstrSize _CMN_cFillConstN .

# arg: 3c1 3c2
# Fills current instr with given two ICs, may append COCs to make them fit.
CMN_cFillConsts:
  ~32 $1 $1 ~0 CMN_cConstInstrSize CMN_cConstInstrSize CMN_gMax
  $0
  ~32 >< ~0
  _CMN_cFillConstN
  ~16 $CMN_bCurrentInstr +x1000 | $:CMN_bCurrentInstr ~0
  CMN_bOPCODE_COC 0 CMN_cEmitInstr
  _CMN_cFillConstN
.

# ret: N
# Returns default size (in instrs) that should be allocated for unknown addr.
_CMN_cDefaultAddrSize: 0 ~32 $_CMN_cInterState 28 |> 1 & >0 ~0 8 4 ?? .

# arg: 3addr
# Fills back an addr (3addr) to instr pointed to by _CMN_g2P1, may overwrite
# some following instrs with COCs (needs prealloated unknown addr), if 3addr is
# too big for the space to fit, args are popped and error state of com is set.
_CMN_cFillBackAddr:
  ~32
    $0 0 ~0 _CMN_cDefaultAddrSize >32 ~32

    2 |< |> ? # any non-zero bits above maximum?
      ^ ~0 CMN_cSTATE_ERROR_ADDR_TOO_BIG $:CMN_cState ~32
    ;
      $CMN_bCurrentInstrAddr # back up
      ><
      ~16 $_CMN_g2P1>CMN_bCurrentInstr ~32 # we ignore CMN_bCurrentInstrAddr
      CMN_cFillConst
      CMN_bCurrentInstrSetAddr
    .
  ~0
.

# Emits additional NOP instrs to hold place for later filling yet unknown addr.
_CMN_cPrepareUnknownAddr:
  _CMN_cDefaultAddrSize --
  @' CMN_bOPCODE_NOP 0 CMN_cEmitInstr -- .
  ^
.

# ret: 3n
# Pushes number of unknown jump encountered so far.
_CMN_cUnknownJumpCount: ~32 $_CMN_cInterState 16 |> +x0fff & ~0 .

_CMN_cUnknownJumpCountInc:
  ~32
    _CMN_cUnknownJumpCount +x0fff != ?
      $_CMN_cInterState +x00010000 + $:_CMN_cInterState
    ;
      ~0 CMN_cSTATE_ERROR $:CMN_cState ~32
    .
  ~0
.

# ret: succ
# Helper func, expects _CMN_g2P1 to point to an instr within BC, will seek this
# ptr to the nearest previous DES instr that has no-pop bit set (see HACK). Also
# increments num in _CMN_gV1 by steps performed. Returns 1 (OK) or 0 (error).
_CMN_cSeekNextInstrMatch:
  @@ # WARNING: relying on reserved BC bits to be 0 to potentially stop the loop
    ~16
      $_CMN_g2P1 ?'
        ~0 0 ~16 >0
        ~0
          $0 CMN_bInstrOpcode CMN_bOPCODE_DES = >< +x2000 & && ?
            1 # succ
            !@
          .
        ~16
      ;
        ~0 0 ~16 # fail
        ^ !@
      .

      $<_CMN_g2P1
      ~0 $_CMN_gV1 ++ $:_CMN_gV1 ~16
    ~0
  .
.
```

```
# ret: 2bool
# Helper func, moves code from CMN_cFeedChar, returns 1 (matched) or 0 in TE 16.
_CMN_cHandleSimpleCommand:
  ~16 1 ~0 # result

  # first check control structures, push 0/1/2/3/4/5 (none/?/@/@@/!@/!.)

  0 "?"  CMN_tTokenMatch ? 1 ;
  0 "@"  CMN_tTokenMatch ? 2 ;
  0 "@@" CMN_tTokenMatch ? 3 ;
  0 "!@" CMN_tTokenMatch ? 4 ;
  0 "!." CMN_tTokenMatch ? 5 ;
    0
  . . . . .

  ?'
    CMN_bOPCODE_DES

    $1 1 = ? CMN_bDES_IF ;
    $1 4 = ? CMN_bDES_BREAK ;
    $1 5 = ? CMN_bDES_EXIT ;
           CMN_bDES_LOOP
    . . .

    CMN_cEmitInstr

    ~16 $CMN_bCurrentInstr +x2000 | $:CMN_bCurrentInstr ~0 # HACK

    $0 3 != ?
      $0 4 >= CMN_bOPCODE_JMA CMN_bOPCODE_JNA ?? 0 CMN_cEmitInstr
     _CMN_cPrepareUnknownAddr
    .
    ^

    !.
  .

  ^ # control structure type

  0 ";" CMN_tTokenMatch ?
    CMN_bOPCODE_DES CMN_bDES_ELSE CMN_cEmitInstr

    ~16 $CMN_bCurrentInstr>_CMN_g2P1 ~0

    @@ # find what the end matches
      _CMN_cSeekNextInstrMatch

      ? # succ?
        0 ~16 $_CMN_g2P1 >0 ~0 8 |> +x0f &

        $0 CMN_bDES_IF = ? # else matched if, nice
          ~16
            $_CMN_g2P1 +xdfff & $:_CMN_g2P1                # unmark the if
            $CMN_bCurrentInstr +x2000 | $:CMN_bCurrentInstr # mark the else
            $>_CMN_g2P1                                    # move to jump ins.
          ~0

          ~32 0 ~0 _CMN_cDefaultAddrSize >32
          ~32 $CMN_bCurrentInstrAddr + ++ _CMN_cFillBackAddr ~0

           CMN_bOPCODE_JMA 0 CMN_cEmitInstr
           _CMN_cPrepareUnknownAddr
          !@
        ;
        $0 CMN_bDES_BREAK = $1 CMN_bDES_EXIT = | $1 CMN_bDES_GOTO = | ?
          # ignore breaks etc., just move to next instr.
          ~16 $<_CMN_g2P1 ~0
        ;
          CMN_cSTATE_ERROR_UNEXPECTED $:CMN_cState
          !@
        . .

        ^
      ;
        CMN_cSTATE_ERROR_UNEXPECTED $:CMN_cState
        !@
      .
  .

  !.
.

  0 "." CMN_tTokenMatch ?
    ~16 $CMN_bCurrentInstr>_CMN_g2P1 ~0 # for match search
    0 $:_CMN_gV1                        # will hold addr decrement

    @@
      _CMN_cSeekNextInstrMatch

      ? # success?
        0 ~16 $_CMN_g2P1 >0 ~0 8 |> +x0f &

        $0 CMN_bDES_IF = $1 CMN_bDES_ELSE = | ? # end matched if or else
          ~16
            $_CMN_g2P1 +xdfff & $:_CMN_g2P1     # unmark the if
            $>_CMN_g2P1                         # move to jump instr
          ~0

          ~32 $CMN_bCurrentInstrAddr ++ ++ _CMN_cFillBackAddr ~0

          CMN_bOPCODE_DES CMN_bDES_ENDIF CMN_cEmitInstr
          !@
        ;
        $0 CMN_bDES_LOOP = ? # end matched loop
          ~32 $CMN_bCurrentInstrAddr ++ 0 ~0
          $_CMN_gV1 >32 # addr decrement
          ~32 - ~0 # now we have the jump back addr on stack 32 for later

          0 # make room for instr type

          ~16
            $_CMN_g2P1 +xdfff & $:_CMN_g2P1 # unmark the DES
            $>_CMN_g2P1                     # move to next instr (jump)
            $_CMN_g2P1 >0                   # move instr type to stack 0
          ~0

          # compute the final (after loop) jump addr. on stack 32:
          ~32 $0 0 ~0 CMN_cConstInstrSize >32
          ~32 $CMN_bCurrentInstrAddr ++ ++ + ~0

          # now handle loop breaks, go instr by instr:
          CMN_bOPCODE_DES 0 1 CMN_bDES_BREAK 0 CMN_bMakeInstr # for comparison

          ~16
            $_CMN_g2P1>_CMN_g2P2

            1 @
              ~0 $0 0 ~16 $_CMN_g2P1 >0

              ~0 = ? ~16
                $_CMN_g2P1 +xdfff & $:_CMN_g2P1
                $>_CMN_g2P1
                ~32 $0 _CMN_cFillBackAddr ~16 # fill in break addr.
              .

              $>_CMN_g2P1
              $_CMN_g2P1=CMN_bCurrentInstr 2 =
            .

            $_CMN_g2P2>_CMN_g2P1
          ~0

          ^ # the comparison instr

          CMN_bInstrOpcode CMN_bOPCODE_JNA = ? # non-infinite loop?
            ~32 $0 _CMN_cFillBackAddr ~0       # then fill back jump addr.
          .

          ~32 ^ ~0 # after-loop jump addr.

          # now just append the final instrs
          CMN_bOPCODE_DES CMN_bDES_ENDLOOP CMN_cEmitInstr
          CMN_bOPCODE_JMA 0 CMN_cEmitInstr
          CMN_cFillConst # fill the above jump addr
          !@
        ;
        $0 CMN_bDES_FUNDEF = ? # end matched func definition
          ~16
             $_CMN_g2P1 +xdfff & $:_CMN_g2P1 # unmark, HACK
             $>_CMN_g2P1
          ~0

          ~32 $CMN_bCurrentInstrAddr ++ ++ _CMN_cFillBackAddr ~0

          CMN_bOPCODE_RET 0 CMN_cEmitInstr

          ~16 # now go backwards from RET and fill back addr of every exit
            $CMN_bCurrentInstr>_CMN_g2P1

            @@
              $<_CMN_g2P1
              ~0 0 ~16 $_CMN_g2P1 >0

              ~0
                $0 CMN_bInstrOpcode CMN_bOPCODE_DES = ?
                  $0 8 |> +x0f &

                  $0 CMN_bDES_FUNDEF = ?
                    !@
                  ;
                  $0 CMN_bDES_EXIT = $2 +x2000 & && ?
                    ~16
                      $_CMN_g2P1 +xdfff & $:_CMN_g2P1 # unmark
                      $>_CMN_g2P1
                    ~0

                    ~32 $CMN_bCurrentInstrAddr _CMN_cFillBackAddr ~0
                    ~16 $<_CMN_g2P1 ~0
                  . .

                  ^
                .
                ^
              ~16
            .
          ~0

          !@
        ;
          # breaks, exits, ... just ignore here
          ~16 $<_CMN_g2P1 ~0
          $_CMN_gV1 ++ $:_CMN_gV1
        . . .

        ^ # DES
      ;
        CMN_cSTATE_ERROR_UNEXPECTED $:CMN_cState
        !@
      . # match succ if
    . # match-search loop

    !.
  . # "." token

  0 "+"   CMN_tTokenMatch ? CMN_bOPCODE_ADX 0 CMN_cEmitInstr !. .
  0 "++"  CMN_tTokenMatch ? CMN_bOPCODE_ADC 1 CMN_cEmitInstr !. .
  0 "-"   CMN_tTokenMatch ? CMN_bOPCODE_SUX 0 CMN_cEmitInstr !. .
  0 "--"  CMN_tTokenMatch ? CMN_bOPCODE_SUC 1 CMN_cEmitInstr !. .
  0 "*"   CMN_tTokenMatch ? CMN_bOPCODE_MUX 0 CMN_cEmitInstr !. .
  0 "/"   CMN_tTokenMatch ? CMN_bOPCODE_DIX 0 CMN_cEmitInstr !. .
  0 "//"  CMN_tTokenMatch ? CMN_bOPCODE_DSX 0 CMN_cEmitInstr !. .
  0 "%"   CMN_tTokenMatch ? CMN_bOPCODE_MOX 0 CMN_cEmitInstr !. .
  0 "%%"  CMN_tTokenMatch ? CMN_bOPCODE_MSX 0 CMN_cEmitInstr !. .
  0 "><"  CMN_tTokenMatch ? CMN_bOPCODE_SWP 0 CMN_cEmitInstr !. .
  0 "^"   CMN_tTokenMatch ? CMN_bOPCODE_POP 0 CMN_cEmitInstr !. .
  0 "="   CMN_tTokenMatch ? CMN_bOPCODE_EQX 0 CMN_cEmitInstr !. .
  0 "!="  CMN_tTokenMatch ? CMN_bOPCODE_NEX 0 CMN_cEmitInstr !. .
  0 "<"   CMN_tTokenMatch ? CMN_bOPCODE_SMX 0 CMN_cEmitInstr !. .
  0 "<<"  CMN_tTokenMatch ? CMN_bOPCODE_SSX 0 CMN_cEmitInstr !. .
  0 "<="  CMN_tTokenMatch ? CMN_bOPCODE_SEX 0 CMN_cEmitInstr !. .
  0 "<<=" CMN_tTokenMatch ? CMN_bOPCODE_LSX 0 CMN_cEmitInstr !. .
  0 ">"   CMN_tTokenMatch ? CMN_bOPCODE_GRX 0 CMN_cEmitInstr !. .
  0 ">>"  CMN_tTokenMatch ? CMN_bOPCODE_GSX 0 CMN_cEmitInstr !. .
  0 ">="  CMN_tTokenMatch ? CMN_bOPCODE_GEX 0 CMN_cEmitInstr !. .
  0 ">>=" CMN_tTokenMatch ? CMN_bOPCODE_BSX 0 CMN_cEmitInstr !. .
  0 "||"  CMN_tTokenMatch ? CMN_bOPCODE_LOX 0 CMN_cEmitInstr !. .
  0 "&&"  CMN_tTokenMatch ? CMN_bOPCODE_LAX 0 CMN_cEmitInstr !. .
  0 "??"  CMN_tTokenMatch ? CMN_bOPCODE_CND 0 CMN_cEmitInstr !. .
  0 "!!"  CMN_tTokenMatch ? CMN_bOPCODE_EQC 0 CMN_cEmitInstr !. .
  0 "|!!" CMN_tTokenMatch ? CMN_bOPCODE_LXX 0 CMN_cEmitInstr !. .
  0 "|"   CMN_tTokenMatch ? CMN_bOPCODE_BOX 0 CMN_cEmitInstr !. .
  0 "&"   CMN_tTokenMatch ? CMN_bOPCODE_BAX 0 CMN_cEmitInstr !. .
  0 "!"   CMN_tTokenMatch ? CMN_bOPCODE_BNO 0 CMN_cEmitInstr !. .
  0 "|!"  CMN_tTokenMatch ? CMN_bOPCODE_BXX 0 CMN_cEmitInstr !. .
  0 "|<"  CMN_tTokenMatch ? CMN_bOPCODE_SLX 0 CMN_cEmitInstr !. .
  0 "|>"  CMN_tTokenMatch ? CMN_bOPCODE_SRX 0 CMN_cEmitInstr !. .
  0 "<-"  CMN_tTokenMatch ? CMN_bOPCODE_INP 0 CMN_cEmitInstr !. .
  0 "->"  CMN_tTokenMatch ? CMN_bOPCODE_OUT 0 CMN_cEmitInstr !. .
  0 "<?"  CMN_tTokenMatch ? CMN_bOPCODE_INU 0 CMN_cEmitInstr !. .
  0 ">0"  CMN_tTokenMatch ? CMN_bOPCODE_TRA 0 CMN_cEmitInstr !. .
  0 ">8"  CMN_tTokenMatch ? CMN_bOPCODE_TRA 1 CMN_cEmitInstr !. .
  0 ">16" CMN_tTokenMatch ? CMN_bOPCODE_TRA 2 CMN_cEmitInstr !. .
  0 ">32" CMN_tTokenMatch ? CMN_bOPCODE_TRA 3 CMN_cEmitInstr !. .

  0 "-->" CMN_tTokenMatch ?
    CMN_bOPCODE_DES CMN_bDES_LOOP CMN_cEmitInstr
    1 _CMN_cNoPopSet CMN_bOPCODE_JNA 0 CMN_cEmitInstr 0 _CMN_cNoPopSet
    ~16 $CMN_bCurrentInstr>_CMN_g2P2 ~0
    ~32 $CMN_bCurrentInstrAddr ~0
    _CMN_cPrepareUnknownAddr
    CMN_bOPCODE_OUT 0 CMN_cEmitInstr
    CMN_bOPCODE_DES CMN_bDES_ENDLOOP CMN_cEmitInstr
```

```
    CMN_bOPCODE_JMA 0 CMN_cEmitInstr
    CMN_cFillConst
    ~16 $_CMN_g2P2>_CMN_g2P1 ~0
    ~32 $CMN_bCurrentInstrAddr ++ ~0
    _CMN_cFillBackAddr
    CMN_bOPCODE_POP 0 CMN_cEmitInstr
    !.
  .

  ~16 ^ 0 ~0 # nothing matched
.

# arg: 0 sn ... s1 s0
# ret: 3n
# Converts ptr name to BC ptr index, for user ptrs (those not named "0", "1"
# etc.) currently set TE plays a role. -1 is returned on error.
_CMN_ptrNameToIndex:
  $1 0 = $1 "0" >= $1 & "9" <= & ?
    ~32 0 ~0 "0" - >32 ^
  ;
    CMN_cStrPseudohash "0" _CMN_cCurrentTypeEnvGet +
    CMN_cSymbolEncode ~32 $2 $2 $2 ~0 _CMN_cSymbolTableFind ?
      ~32 ^ _CMN_cSymbolGetIndex 16 + ~0
    ;
      ~32 ^ ^ ^ ^ -1 ~0
    .
  .
.

_CMN_cHandlePtrCommand:
  0 "$"  CMN_tTokenMatch ? CMN_bOPCODE_PUX 0 CMN_cEmitInstr !. .
  0 "$$" CMN_tTokenMatch ? CMN_bOPCODE_ADR 0 CMN_cEmitInstr !. .

  _CMN_tTokenToStr ^ # convert to str and drop "$"

  $0 ":" = $1 "+" = | ?
    ":" = CMN_bOPCODE_MEX CMN_bOPCODE_PAX ??  0 CMN_cEmitInstr

    _CMN_ptrNameToIndex

    ~32
      $0 -1 != ?
        CMN_cFillConst
      ;
        ^
        ~0 CMN_cSTATE_ERROR_NAME_UNK $:CMN_cState ~32
      .
    ~0

    !.
  .

  $0 ">" = $1 "<" = | ?
    ~32 0 ~0 ">" = 1 15 ?? >32 # const for PAC, 15 is -1 in 4b two's compl.

    CMN_bOPCODE_PAC 0 CMN_cEmitInstr
    _CMN_ptrNameToIndex

    ~32
      $0 -1 != ?
        >< CMN_cFillConsts
      ;
        ^ ^
        ~0 CMN_cSTATE_ERROR_NAME_UNK $:CMN_cState ~32
      .
    ~0

    !.
  .

  $0>_CMN_gP1

  @@
    $_CMN_gP1 0 = ?   0 !@ .
    $_CMN_gP1 ">" = ? 0 $:_CMN_gP1 1 !@ . # we insert 0 to separate the strs
    $_CMN_gP1 "=" = ? 0 $:_CMN_gP1 2 !@ .
    $<_CMN_gP1
  .

  ?'
    # $ptr>ptr2 or $ptr=ptr2
    1 = CMN_bOPCODE_PCO CMN_bOPCODE_PCM ??  0 CMN_cEmitInstr

    _CMN_ptrNameToIndex _CMN_ptrNameToIndex

    ~32
      ><
      $1 -1 != $1 -1 != & ?
        CMN_cFillConsts
      ;
        ^ ^ ~0 CMN_cSTATE_ERROR_NAME_UNK $:CMN_cState ~32
      .
    ~0
  ;
    ^
    # $ptr
    CMN_bOPCODE_MGE 0 CMN_cEmitInstr
    _CMN_ptrNameToIndex

    ~32
      $0 -1 != ?
        CMN_cFillConst
      ;
        ^ ~0 CMN_cSTATE_ERROR_NAME_UNK $:CMN_cState ~32
      .
    ~0
  .
.

# arg: 0 sn ... s1 s0
# ret: c14 c13 ... c0
# Converts a zero term. ASCII str. of any length into "pseudohash" usable in
# symbol table. The pseudohash consists of exactly 15 ASCII characters (16th not
# used to leave space for one type char.) and will be valid identifier (that can
# be used e.g. by a transpiler) that will try to resemble the original str. (for
# human readability). The result will NOT be zero term. (do not print it!).
CMN_cStrPseudohash:
  $0>_CMN_gP1

  0 # sum of all chars, used in hash
  0 # incrementing product, used in hash, depends on char order
  0 # length of string, used in hash

  @@ # go down and compute all the vals
    $_CMN_gP1

    $0 0 = ?
      ^
      !@
    .
    $0 $4 + $:4
    $2 ++ * $:2
    ++

    $<_CMN_gP1
  .

  $0 15 <= ?
    # extend from left
    >< ^ >< ^
    15 >< -

    @'
      $0>_CMN_gP1
      $0

      1 @
        $<_CMN_gP1 $_CMN_gP1 $>_CMN_gP1 $:_CMN_gP1 $<_CMN_gP1
        $_CMN_gP1
      .

      "_" $>_CMN_gP1 $:_CMN_gP1

      --
    .

    ^
  ;
    # shorten and insert three char hash
    $0>_CMN_gP1 $0 $:_CMN_gV1

    CMN_cNumToChar
    >< CMN_cNumToChar
    $2 CMN_cNumToChar $:3

    $_CMN_gV1 7 - -1 * $+0
    $_CMN_gP1 $<_CMN_gP1
    $_CMN_gP1 $<_CMN_gP1
    $_CMN_gP1 $<_CMN_gP1
    $_CMN_gP1
  .

  # shift everything one down (overwrite the bottom 0)

  $0>_CMN_gP1

  1 @
    $<_CMN_gP1
    $_CMN_gP1
  .

  1 @
    $>_CMN_gP1 $_CMN_gP1 $<_CMN_gP1 $:_CMN_gP1
    $>_CMN_gP1
    $0=_CMN_gP1
  .

  ^
.

# arg: c14 c13 ... c0 c
# ret: 3v1 3v2 3v3
# Encodes a symbol made of string chars (pseudohash) and one type char (c) to 3
# 32bit vals that may be directly stored in symbol table.
CMN_cSymbolEncode:
  ~32
    0 ~0 CMN_cCharToNum >32 ~32 $:_CMN_g3V1

    3 @'
      0

      5 @'
        ><
        6 |< 0 ~0 CMN_cCharToNum >32 ~32 |
        >< --
      .
      ^

      2 |< $_CMN_g3V1 +b11 & |
      $_CMN_g3V1 2 |> $:_CMN_g3V1

      >< --
    .
    ^
  ~0
.

# arg: 3v1 3v2 3v3
# ret: c14 c13 ... c0 c
# Performs the opposite of CMN_cSymbolEncode.
CMN_cSymbolDecode:
  ~32
    0 $:_CMN_g3V1

    3 @'
      ><

      $0 +b11 &
      $_CMN_g3V1 2 |< |
      $:_CMN_g3V1

      2 |>

      5 @'
        ><
        ~0 0 ~32 $0 >0 ~0 +b111111 & CMN_cNumToChar ~32
        6 |>
        >< --
      .
      ^

      ^
      --
    .
    ^

    ~0 0 ~32
    $_CMN_g3V1 >0 CMN_cNumToChar
  ~0
.

# arg: addrSize
# Initializes com, addrSize says the default addr size, currently either 0
# (16bit) or 1 (32bit), higher val will generate bigger BC (more space will be
# allocated for unknown addres) but also allow compilation of larger programs
# (note that bigger generated BC can still be later optimized to smaller size,
# but initially needs more RAM).
CMN_cInit:
  ~32 0 ~0 >32

  CMN_cSTATE_OK $:CMN_cState
```

```
  ~32
    CMN_bBYTECODE_SIZE
    0 ~0 CMN_bHEADER_SIZE 1 |> >32 ~32
    < ?
      ~0 CMN_cSTATE_ERROR_BC_TOO_BIG $:CMN_cState ~32
      !.
    .

    1 & 28 |< $:_CMN_cInterState

    $CMN_cMem>_CMN_cMemEnd
    CMN_cMEM_SIZE $+_CMN_cMemEnd

    ~16 $CMN_bBytecode>CMN_bCurrentInstr ~32

    CMN_bBYTECODE_SIZE @' # zero whole BC
      ~16 0 $:CMN_bCurrentInstr $>CMN_bCurrentInstr ~32
      --
    .
    ^

    0 CMN_bCurrentInstrSetAddr

    ~0
      CMN_bOPCODE_NOP 0 0 0 0 CMN_bMakeInstr ~16 0 ~0 >16
      ~16 $:CMN_bCurrentInstr ~0
    ~32
  ~0

  # set the BC end ptr:
  ~16 $CMN_bBytecode>_CMN_cBytecodeEnd ~0
  ~32
    CMN_bBYTECODE_SIZE # shift with loop, in case BC is bigger than 2^16
    @' ~16 $>_CMN_cBytecodeEnd ~32 -- .
    ^
  ~0

  CMN_tInit

  ~16 0 ~0 CMN_bOPCODE_INI 0 0 0 0 CMN_bMakeInstr >16 ~16 $:CMN_bCurrentInstr ~0
  _CMN_cPrepareUnknownAddr # here maybe we'll later insert a jump to init func
.

# ret: 3size
# Gets current size of symbol table (in items).
_CMN_cSymbolTableSize: ~32 $_CMN_cInterState +xffff & ~0 .

# arg: 3size
_CMN_cSymbolTableSizeSet:
  ~32 $_CMN_cInterState +xffff0000 & | $:_CMN_cInterState ~0
.

# arg: 3v1 3v2 3v3
# ret: succ 3val
# Finds symbol in symbol table, returns 1 (in TE 0) and the symbol val (in TE
# 32) on success, otherwise 0 (in both TE 0 and 32).
_CMN_cSymbolTableFind:
  ~32
    $CMN_cMem>_CMN_g3P1

    _CMN_cSymbolTableSize @'
      $_CMN_g3P1 $2 = $>_CMN_g3P1
      $_CMN_g3P1 $4 = $>_CMN_g3P1
      $_CMN_g3P1 $6 = $>_CMN_g3P1

      & & ?
        ^ ^ ^ ^ $_CMN_g3P1 ~0 1 ~32 # found
        !.
      .

      $>_CMN_g3P1 # skip the symbol value
      --
    .
    ^
  ~0

  ~32 ^ ^ ^ 0 ~0 0 # not found
.

# arg: 3val 3v1 3v2 3v3
# Appends an encoded symbol with given val to the symbol table, handles errors.
_CMN_cSymbolTableAdd:
  ~32
    $CMN_cMem>_CMN_g3P1
    _CMN_cSymbolTableSize $0 2 |<

    3 + # this is here for the if check below
    $+_CMN_g3P1

    $_CMN_g3P1=_CMN_cMemEnd 2 = ?
      -3 $+_CMN_g3P1 # check done, shift back

      ++ _CMN_cSymbolTableSizeSet

      $:_CMN_g3P1 $>_CMN_g3P1
      $:_CMN_g3P1 $>_CMN_g3P1
      $:_CMN_g3P1 $>_CMN_g3P1
      $:_CMN_g3P1
    ;
      ^ ~0 CMN_cSTATE_ERROR_SYMBOL_TAB $:CMN_cState ~32
    .
  ~0
.

# arg: n
# ret: c15 c14 ... c0 c 3val
# Gets and decodes Nth symbol and its val, N must be smaller than the num of
# items in the table (this isn't checked).
CMN_cSymbolGet:
  ~32 0 ~0 >32

  ~32
    $CMN_cMem>_CMN_g3P1
    ++ 2 |<
    $+_CMN_g3P1

    $<_CMN_g3P1 $_CMN_g3P1
    $<_CMN_g3P1 $_CMN_g3P1
    $<_CMN_g3P1 $_CMN_g3P1
    $<_CMN_g3P1 $_CMN_g3P1

    CMN_cSymbolDecode
  ~0
.

# arg: 3v1 3v2 3v3
# ret: 3index
# Returns index of given symbol among the symbols of the same type (starting
# with the same char). Doesn't check if the symbol exists.
_CMN_cSymbolGetIndex:
  ~32 $2 $2 $2 ~0 CMN_cSymbolDecode
  $0 $:_CMN_gV1
  _CMN_cPseudohashPop
  $_CMN_gV1 # type char

  ~32
    0 # result
    $CMN_cMem>_CMN_g3P1
    $>_CMN_g3P1
    $>_CMN_g3P1

    _CMN_cSymbolTableSize @'
      $_CMN_g3P1 $<_CMN_g3P1
      $_CMN_g3P1 $<_CMN_g3P1
      $_CMN_g3P1
      6 $+_CMN_g3P1

      # stack 32 now: v1 v2 v3 r i a b c

      $7 $3 = $7 $3 = $7 $3 = & & ?
        ^ ^ ^
        !@
      .

      ~0
        CMN_cSymbolDecode
        $0 $:_CMN_gV1
        _CMN_cPseudohashPop

        $0 $_CMN_gV1 = ?
          ~32 $1 ++ $:2 ~0
        .
      ~32

      --
    .
    ^

    >< ^ >< ^ >< ^
  ~0

  ^ # type char
.

# arg: c14 c13 ... c0 c
# ret:
_CMN_cPseudohashPop: ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ . # can be just one instr

# Helper func for the end iteration on symbol table. Expects _CMN_g3P2 to
# point to a symbol. If the symbol's val is not 0, pushes 1 to TE 0 (else 0) and
# decodes the symbol (else not). Shifts the pointer to following symbol.
_CMN_cEndSymbolIterate:
  ~32
    3 $+_CMN_g3P2 $_CMN_g3P2 ?
      $<_CMN_g3P2 $_CMN_g3P2
      $<_CMN_g3P2 $_CMN_g3P2
      $<_CMN_g3P2 $_CMN_g3P2
      CMN_cSymbolDecode
      4 $+_CMN_g3P2
      ~0 1 ~32
    ;
      # symbol value = 0, not interested (such ptr doesn't need addr)
      ~0 0 ~32
      $>_CMN_g3P2
    .
  ~0
.

# arg: 3v c
# ret: s14 ... s1 s0
# Helper for end pass, find symbol of given type (c) with given value, returns
# its pseudohash, relies on the symbol existing.
_CMN_cFindSymWithVal:
  ~32 0 ~0 >32 ~32 >< ~0 # let's store the char in TE 32

  ~32
    $CMN_cMem>_CMN_g3P2
    3 $+_CMN_g3P2

    _CMN_cSymbolTableSize @' # search the symbol table
      $1 $_CMN_g3P2 = ?     # val of symbol = instr const?
        $<_CMN_g3P2 $_CMN_g3P2
        $<_CMN_g3P2 $_CMN_g3P2
        $<_CMN_g3P2 $_CMN_g3P2
        3 $+_CMN_g3P2

        CMN_cSymbolDecode

        ~0
          0 ~32 $2 >0 ~0 = ?
            !@
          ;
            "a" # add back one popped char
            _CMN_cPseudohashPop
          .
        ~32

      . # symbol const comparison

      4 $+_CMN_g3P2
      --
    . # for each symbol
    ^

    ^ ^ # val, c
  ~0
.



# Helper, performs the final BC pass after compiler received last input char.
_CMN_cEndPass:
  ~32 $CMN_bCurrentInstrAddr ++ ~0 # BC end addr (for exits)

  # resolve unknown jumps AND check for unhandled DES instrs (HACK):
  ~16
    $CMN_bCurrentInstr @' # for each instr (relying on header ending in 0 0)
      ~0 0 ~16 >0
      ~0
        $0 CMN_bInstrOpcode CMN_bOPCODE_DES = ?
          $0 CMN_bInstrWontPop ?
            # DES with no-pop bit set (see HACK)
            $0 8 |> +x0f & CMN_bDES_EXIT = ?
              ~16
                $CMN_bCurrentInstr +xdfff & $:CMN_bCurrentInstr # unmark
                $>CMN_bCurrentInstr
              ~0

              ~32 $0 CMN_cFillConst ~0 # fill the BC end addr
            ; # other than exit means unhandled control struct (error)
              ^
```

```
            CMN_cSTATE_ERROR_UNEXPECTED $:CMN_cState
            !@
          .
        ;
        $0 8 |> +b1111 & CMN_bDES_GOTO = ?
          ~16 $>CMN_bCurrentInstr ~0
          CMN_bCurrentInstrGetConst32 "g" _CMN_cFindSymWithVal
          "l" CMN_cSymbolEncode _CMN_cSymbolTableFind ?
            ~16
              $CMN_bCurrentInstr>_CMN_g2P2
              _CMN_cPrepareUnknownAddr # delete the current const
              $_CMN_g2P2>CMN_bCurrentInstr
              CMN_cFillConst
            ~0
          ;
            ~32 ^ ~0
            CMN_cSTATE_ERROR_NAME_UNK $:CMN_cState # goto to unknown label
          .

          ~16 0 ~0
          CMN_bOPCODE_DES 0 0 CMN_bDES_GOTO 0 CMN_bMakeInstr >16

          ~16
            1 @
              $<CMN_bCurrentInstr
              $0 $CMN_bCurrentInstr !=
            .

            $<CMN_bCurrentInstr
            ^
          ~0
        . .
      .

      CMN_bInstrOpcode CMN_bOPCODE_CAE = ? # unknown call?
        # find matching "f" symbol for the "c" symbol
        CMN_bCurrentInstrGetConst32 "c" _CMN_cFindSymWithVal

        "f" CMN_cSymbolEncode _CMN_cSymbolTableFind ?
          ~16 0 ~0 CMN_bOPCODE_CAL 0 0 0 0 CMN_bMakeInstr >16
          ~16
            $:CMN_bCurrentInstr
            $CMN_bCurrentInstr>_CMN_g2P2
            _CMN_cPrepareUnknownAddr
            $_CMN_g2P2>CMN_bCurrentInstr
            CMN_cFillConst
          ~0
        ;
          ~32 ^ ~0
        .
      . # instr opcode = CAE?
    ~16

    $<CMN_bCurrentInstr
    $CMN_bCurrentInstr
  . # for all instrs
  ^
~0

~32 -- ~0 CMN_bCurrentInstrSetAddr # we use the above pushed addr to get back

# now potentially make an init func and set ptr addresses
0 # make separate init func?

~32
  $CMN_cMem>_CMN_g3P2

  _CMN_cSymbolTableSize  # counter
  @' # iterate on symbol table
    ~0 _CMN_cEndSymbolIterate ?
      $0 $:_CMN_gV1 _CMN_cPseudohashPop

      $_CMN_gV1 $0 "0" >= >< "3" <= & ?
        ~32 -4 $+_CMN_g3P2 ~0 # we'll want this symbol again
        ^ 1
        !@
      .
    .
    ~32

    --
  .
  # not popping the counter yet
~0

? # Make init func? We still have symbol ptr in _CMN_g3P2 and count. in TE 32.
  ~32 0 >< 0 >< 0 >< 0 >< ~0 # a3, a2, a1, a0: last addr in each TE

  0 _CMN_cCurrentTypeEnvSet
  CMN_bOPCODE_DES CMN_bDES_FUNDEF CMN_cEmitInstr
  CMN_bOPCODE_JMA 0 CMN_cEmitInstr _CMN_cPrepareUnknownAddr

  ~16 0 ~0 CMN_bOPCODE_CAL 0 0 0 0 CMN_bMakeInstr >16
  ~16 0 ~0 CMN_bHEADER_SIZE 1 |> >16
  ~16 $CMN_bBytecode>_CMN_g2P1 $+_CMN_g2P1 $:_CMN_g2P1 ~0
  ~32 $CMN_bCurrentInstrAddr ++ ~0 _CMN_cFillBackAddr

  ~32
  @' # continue iterating on symbols, we stopped at first we shall handle
    ~0 _CMN_cEndSymbolIterate ?
      $0 $0 "0" >= >< "3" <= & ?
        $0 "0" - _CMN_cCurrentTypeEnvSet
        ~32
          $<_CMN_g3P2 $_CMN_g3P2
          $<_CMN_g3P2 $_CMN_g3P2
          $<_CMN_g3P2 $_CMN_g3P2
          $<_CMN_g3P2 $_CMN_g3P2
          4 $+_CMN_g3P2

          _CMN_cSymbolGetIndex

          # now we have on stack 32:
          # a3 a2 a1 a0 i symbolVal ptrID

          16 + # user pointers start at ID 16
          ><

          $0>_CMN_g3P1
          0 ~0 $0 >32 ~32 "0" - -1 * 3 -
          $+_CMN_g3P1 # set the ptr to aN

          $_CMN_g3P1 +
          $_CMN_g3P1
          >< $:_CMN_g3P1
        ~0

        ~32 ?' ~0 # only set ptr whose value is not 0 (0 is default)
          CMN_bOPCODE_PSC 0 CMN_cEmitInstr
          CMN_cFillConsts
        ;
          ~32 ^ ^ ~0
            .
          .

          _CMN_cPseudohashPop
        .
        ~32

        --
      . # symbol iterate loop

      ^ # counter

      4 @' # pop a0, a1, a2, a3, potentially setting addr of ptr 0
        ><

        ?'
          ~0 0 ~32 4 $2 - >0 _CMN_cCurrentTypeEnvSet
          ~0 CMN_bOPCODE_PSC 0 CMN_cEmitInstr ~32
          0 >< CMN_cFillConsts
        ;
          ^
        .

        --
      .
      ^
      ~0

      0 _CMN_cCurrentTypeEnvSet

      CMN_bOPCODE_INI 0 CMN_cEmitInstr
      CMN_bOPCODE_RET 0 CMN_cEmitInstr

      # now fill back the JMA jump addr
      ~16 0 ~0 CMN_bOPCODE_JMA 0 0 0 0 CMN_bMakeInstr >16

      ~16
        $CMN_bCurrentInstr>_CMN_g2P1

        1 @
          $<_CMN_g2P1
          $0 $_CMN_g2P1 !=
        .

        ^ # comparison instr
      ~0

      ~32 $CMN_bCurrentInstrAddr ++ ~0 _CMN_cFillBackAddr
    ;
      ~32 ^ ~0 # counter
    . # init func?

    ~16
      # fill the header:
      CMN_bChecksum

      $CMN_bBytecode>_CMN_g2P1
      +x4243 $:_CMN_g2P1 # magic number "CB"

      $>_CMN_g2P1 $>_CMN_g2P1 $:_CMN_g2P1
    ~0

    $CMN_cState CMN_cSTATE_OK = ?
      CMN_cSTATE_DONE $:CMN_cState
    .
  .

  # arg: s14 ... s1 s0 c
  # ret:
  # Helper for handling encounters of unknown jumps (calls, gotos, ...), checks
  # prepares unknown addr. and fills in a temporary ID that given pseudohash has
  # in sym. table (will be added if it doesn't exist).
  _CMN_cHandleUnknownJump:
    _CMN_cPrepareUnknownAddr
    _CMN_cUnknownJumpCount # unknown jump ID
    CMN_cSymbolEncode      # 32 stack now: callID v3 v2 v1

    ~32 $2 $2 $2 ~0 _CMN_cSymbolTableFind ?
      ~32 $:4 ^ ^ ^ ~0 # rewrite the ID by the one we got, pop the rest
    ;
      ~32 ^ ~0 # pop the zero returned by _CMN_cSymbolTableFind
      _CMN_cSymbolTableAdd
      _CMN_cUnknownJumpCount # we consumed this, so push again
      _CMN_cUnknownJumpCountInc
    .

    ~16
      0 ~0 _CMN_cDefaultAddrSize >16 ~16 -1 *
      $CMN_bCurrentInstr>_CMN_g2P1
      $+_CMN_g2P1
      $>_CMN_g2P1
    ~0

    _CMN_cFillBackAddr # temporarily fill the ID to CAE
  .

  # arg: char
  # Feeds the next character to com, which may or may not be popped by this func,
  # which must be left so -- the behavior is similar to CMN_tFeedChar. Nothing is
  # returned, state will be set in CMN_cState. Compiler will be using parser and
  # generating instrs into CMN_bBytecode using CMN_bCurrentInstr which therefore
  # mustn't be modified while com is working. When com finishes (either with succ
  # or error), it will clean all trash on main stack (e.g. partially parsed tok.).
  CMN_cFeedChar:
    # WARNING: do NOT exit (!.) inside this function, it would skip the end code

    $CMN_cState CMN_cSTATE_OK = ?
      ~16 0 ~0 $0 0 = >16 # end? we temporarily store this in TE 16 (can't use
                          # TE 0 because we have token there
      CMN_tFeedChar

      $0 CMN_tTOKEN_NONE = ?
        ^ # token type
      ;
        $1 "'" = ? # no pop?
          >< ^     # remove the last "'"
          1
        ;
          0
        .

        _CMN_cNoPopSet

        $0 CMN_tTOKEN_COMMAND = ?
          ^ # token type

          $_CMN_tSTBackup>_CMN_gP1
          $>_CMN_gP1

          $_CMN_gP1 "~" = ?
            $>_CMN_gP1
```

```
    $_CMN_gP1 "0" = ?  0 _CMN_cCurrentTypeEnvSet  ;
    $_CMN_gP1 "8" = ?  1 _CMN_cCurrentTypeEnvSet  ;
    $_CMN_gP1 "1" = ?  2 _CMN_cCurrentTypeEnvSet  ;
    $_CMN_gP1 "3" = ?  3 _CMN_cCurrentTypeEnvSet  ;
    $_CMN_gP1 34 = ?
      # ~"...", file include

      ^ # remove "
      _CMN_tTokenToStr ^ ^ # remove ~"

      ~32 0 ~0 # symbol value for later

      CMN_cStrPseudohash "i" CMN_cSymbolEncode

      ~32 $2 $2 $2 _CMN_cSymbolTableFind ^ ~0 ?
        ~32 ^ ^ ^ ^ ~0 # pop encoded and symbol val
        # repeated include, ignore
      ;
        _CMN_cSymbolTableAdd
        _CMN_tTokenToStr ^ ^ # remove ~"
        CMN_cInclude
      .
    ;
    $_CMN_gP1 ":" = ? # label?
      _CMN_tTokenToStr ^ ^ # remove ~:
      CMN_cStrPseudohash "l"
      ~32 $CMN_bCurrentInstrAddr ++ ~0
      CMN_cSymbolEncode

      ~32 $2 $2 $2 _CMN_cSymbolTableFind ^ ~0 ?
        ~32 ^ ^ ^ ^ ~0
        CMN_cSTATE_ERROR_NAME_EXISTS $:CMN_cState
      ;
        _CMN_cSymbolTableAdd
      .
    ; # ptr decl.
      $0>_CMN_gP1
      0 # does token contain ':'?

      1 @
        $_CMN_gP1 ":" = ?
          ^ 1 !@
        .

        $<_CMN_gP1
        $_CMN_gP1=_CMN_tSTBackup
      .

      ? # size present
        CMN_tLiteralVal 1 = ?
          1 @ ":" != . # delete the part after ':'
        ;
          ~32 ^ ~0
          CMN_cSTATE_ERROR_TOKEN $:CMN_cState
        .
      ;
        ~32 1 ~0 # no size specified implies 1
      .

      _CMN_tTokenToStr ^ # pop removes '~'
      CMN_cStrPseudohash _CMN_cCurrentTypeEnvGet "0" +
      CMN_cSymbolEncode

      ~32 $2 $2 $2 _CMN_cSymbolTableFind ^ ~0 ?
        ~32 ^ ^ ^ ~0
        CMN_cSTATE_ERROR_NAME_EXISTS $:CMN_cState
      ;
        _CMN_cSymbolTableAdd
      .
    . . . . . .
  ;
  $_CMN_gP1 "$" = ?
    _CMN_cHandlePtrCommand
  ;
    _CMN_cHandleSimpleCommand

    ~16 !! ? ~0 # no command matched?
      # now we suppose it's goto (only thing left)
      CMN_bOPCODE_DES CMN_bDES_GOTO CMN_cEmitInstr
      CMN_bOPCODE_JMA 0 CMN_cEmitInstr
      _CMN_tTokenToStr ^ CMN_cStrPseudohash "g"
      _CMN_cHandleUnknownJump
    .
  . .
;
$0 CMN_tTOKEN_NAME = ?
  ^ # token type

  _CMN_tTokenToStr CMN_cStrPseudohash "f" CMN_cSymbolEncode

  ~32 $2 $2 $2 ~0 # keep these for the later else branch

  _CMN_cSymbolTableFind ?
    CMN_bOPCODE_CAL 0 CMN_cEmitInstr
    CMN_cFillConst # fill the found addr
    ~32 ^ ^ ^ ~0
  ;
    # symbol not found, this is NOT an error, func may be defined later
    ~32 ^ ~0 # pop the zero returned by _CMN_cSymbolTableFind

    CMN_bOPCODE_CAE 0 CMN_cEmitInstr # for now emit CAE
    CMN_cSymbolDecode ^ "c" # change type from "f" to "c"
    _CMN_cHandleUnknownJump
  .
;
$0 CMN_tTOKEN_FUNC_DEF = ?
  ^ # token type

  CMN_bOPCODE_DES CMN_bDES_FUNDEF CMN_cEmitInstr
  ~16 $CMN_bCurrentInstr +x2000 | $:CMN_bCurrentInstr ~0 # mark, HACK
  CMN_bOPCODE_JMA 0 CMN_cEmitInstr
  _CMN_cPrepareUnknownAddr

  # add to symbol table:
  ^ # remove the ":"
  _CMN_tTokenToStr CMN_cStrPseudohash "f"

  ~32 $CMN_bCurrentInstrAddr ++ ~0
  CMN_cSymbolEncode

  ~32 $2 $2 $2 _CMN_cSymbolTableFind ^ ~0 # already exists?
  ?
    ~32 ^ ^ ^ ^ ~0
    CMN_cSTATE_ERROR_NAME_EXISTS $:CMN_cState
  ;
    _CMN_cSymbolTableAdd
  .
;
$0 CMN_tTOKEN_NUMBER = ?
  ^ # token type

    CMN_tLiteralVal

    $0 !! ?
      CMN_cSTATE_ERROR_TOKEN $:CMN_cState
      ~32 ^ ~0
    ;
      # mask out only needed bits, to not have unnecessary COCs:
      _CMN_cCurrentTypeEnvGet 1 = ?  ~32 +xff & ~0   ;
      _CMN_cCurrentTypeEnvGet 2 = ?  ~32 +xffff & ~0
      . .

      1 _CMN_cNoPopSet
      CMN_bOPCODE_CON 0 CMN_cEmitInstr
      CMN_cFillConst
    .

    ^
  ;
  $0 CMN_tTOKEN_STRING = ?
    ^ # token type

    $0>_CMN_gP1

    @@
      $<_CMN_gP1

      $_CMN_gP1 34 = ? # double quotes?
        !@
      .

      1 _CMN_cNoPopSet
      CMN_bOPCODE_CON 0 CMN_cEmitInstr

      ~32 0 ~0 $_CMN_gP1 >32 CMN_cFillConst
    .
  ;
    # CMN_tTOKEN_ERROR
    ^ # token type
  . . . . .

  CMN_tPopToken
. # token detected?

 ~16 ? ~0 # end? this was pushed at very top
   _CMN_cEndPass
 .
;
  ^ # input char
. # state OK?
.
```

cmn_general.cmn

```
# comun self hosted implementation: general library
# module prefix: g
#
# This module contains resources that will generally be needed by any other
# program making use of this comun implementation.
#
# NOTES:
#
# === ABBREVIATIONS ===
#
# - addr: address
# - arg: argument
# - BC: bytecode
# - com: compiler
# - const: constant
# - func: function
# - IC: immediate constant
# - instr(s): instruction(s)
# - interp: interpreter
# - mem: memory
# - num: number
# - PT: ptr table
# - ptr: pointer
# - ST: stack top
# - succ: success
# - TE: type environment
# - tmp: temporary
# - val: value
# - var: variable
#
# === NAMING ===
#
# - each library file starts with "cmn_" prefix
# - each public symbol the library starts with "CMN_" prefix
# - each private (not meant for external use) starts with "_CMN_" prefix
# - after CMN_/_CMN_/ a file (module) specific prefix follows
#
# === FUNCTION DOCUMENTATION ===
#
# Arguments are documented on line staring with "arg: ". Arguments outside TE
# 0 will be prefixed with their TE (1, 2 or 3), in order from lowest stack addr
# to stack top, i.e. e.g. "paramX 2paramY paramW" means func takes 2
# parameters in TE 0 (paramX below paramW) and 1 in TE 16. Func return
#
# Return vals are documented on line starting with "ret: ". The notation is
# same as with args.

# temporary helper variables and ptrs, WATCH OUT: their val may change
# after calling a func, if you need them to retain val, check if the
# func you are calling modifies given variable. Minimize relying on this.

CMN_gLANG_V_STR: "0.95" . # version of implemented language

~_CMN_gV1      # tmp var
~_CMN_gP1:0    # tmp ptr

~16
  ~_CMN_g2V1   # tmp var in TE 16
  ~_CMN_g2P1:0 # tmp ptr in TE 16
  ~_CMN_g2P2:0 # tmp ptr 2 in TE 16
~0

~32
  ~_CMN_g3V1   # tmp var in TE 32
  ~_CMN_g3P1:0 # tmp ptr in TE 32
  ~_CMN_g3P2:0 # tmp ptr 2 in TE 32
~0

CMN_gMax:  <' ? >< . ^ .
CMN_gMaxS: <<' ? >< . ^ .
```

cmn_interpreter.cmn

```
# comun self hosted implementation: interpreter library
# module prefix: i
#
# This module implementas the comun BC interpreter.
#
# Before including this library, define the following:
#
# - CMN_iMEM_SIZE: Func returning a const, size of CMN_iMem in cells, in TE 32.
# - CMN_iMem:      Ptr to mem in TE 32 that will be used by the interp, must
#                  point to at least CMN_iMEM_SIZE free cells.
# - CMN_iIn        Func that will be called to perform comun input, after being
#                  called it has to leave exactly one input val on ST (in TE 0).
#                  Returning -1 will be interpreted as end of input.
# - CMN_iOut       Func that will be called to perform comun output, it has to
#                  pop one ST item (in TE 0).
# - CMN_iGetArgs   Func that will be called on encountering the INI instr, it
#                  has to push (in TE 0) N (even 0) zero terminated strs
#                  followed by val N.
# - CMN_iCallExt   Func that will be called when BC calls external func N. This
#                  func will get N on TE 0 stack, the val has to be popped.
#
# The following are details concerning internal working of the interp.
#
# Cell bit width of TE 0 is set to 32.
#
# For simplicity interp mostly works in TE 32, using CMN_iMem. The structure of
# this mem is following:
#
#   ptrTable0 ptrTable1 ptrTable2 ptrTable3 callStack stackCell0 stackCell1 ...
#
# ptrTableN is PT of environment N, its total size depends on and is allocated
# based on BC initial analysis, index 0 is ptr 0, index 1 is first user ptr,
# index 2 is second user pointer etc.
#
# callStack is mem used for returning from called funcs, its size in cells is
# set inside the interp initialization func.
#
# stackCell is a cell of the simulated computation stack, each cell here holds
# one val for each TE that is used, the size and structure of this cell depends
# on BC initial analysis and is set as follows depending on which TEs are used
# by the BC:
#
# - only 1 TE used: Cell's size is 1, holding as many of the environment's cells
#   as possible, e.g. if TE 0 is used, each cell holds one val {0}, if TE 1 is
#   used each cell holds 4 (8 bit) vals {1,1,1,1} etc.
# - 3 or 4 type envs used: Size of each cell is 4, each subcell holds vals of
#   respective TE as follows: {0} {3} {2,2} {1,1,1,1}
# - 2 type envs used: Size of cell is 2, first subcell holds the val of wider
#   TE (TE 0 is before type eng. 3), each subcell holds as many vals as
#   possible, e.g. if type env 3 and 1 are used, cell will be of format:
#   {3} {1,1,1,1}

~"cmn_general.cmn"
~"cmn_bytecode_extra.cmn"

~CMN_iState             # interp state
~_CMN_iInputEnded

~32
  ~_CMN_iStack:0        # start of the stack within CMN_iMem
  ~_CMN_iCallStack:0    # start of the call stack inside CMN_iMem
  ~_CMN_iCallStackTop:0
  ~_CMN_iMemEnd:0       # last cell of CMN_iMem
  ~_CMN_iCellSizeShift  # says bit shift associated with stack cell size
  ~_CMN_iOffsets:3      # holds helper offsets for type envs 1, 2 and 3 (offsets
                        # for TE 0 are always 0), each cell here holds subcell
                        # offset within stack cell in lower 16 bits and PT
                        # offset in higher 16 bits
  ~_CMN_iPushValue      # val to be pushed
  ~_CMN_iJumpAddr       # addr to jump to, -1 means none
~0

~_CMN_iPopPush          # lowest bit says if a val should be pushed, remaining
                        # bits encode the num of vals to be potentially popped
# interp states:
CMN_iSTATE_RUNNING:    0 . # still running
CMN_iSTATE_DONE:       1 . # finished running program
CMN_iSTATE_ERROR:      2 . # further unspecified interp error
CMN_iSTATE_ERROR_MEM:  3 . # accessing memory outside bounds
CMN_iSTATE_ERROR_CALL: 4 . # call stack over/underflow
CMN_iSTATE_ERROR_OP:   4 . # operation error (zero div. etc.)
CMN_iSTATE_ERROR_BC:   5 . # malformed BC error

# arg: te 3PtrI
# ret: 3addr
# Converts ptr index as encoded in an instruction to a mem addr.
_CMN_iPtrIToAddr:
  0 # here table index will go
  ~32
    0

    $1 15 > ?
      >< 15 -
    .

    >0
    CMN_iPtrTableSeek

    $_CMN_g3P1 >< -
  ~0
.

# arg: te recordIndex
# Sets _CMN_g3P1 to the cell of requested PT record, doesn't check arg validity.
CMN_iPtrTableSeek:
  ~32 0 ~0 >32
  ~32 0 ~0 >32
  ~32
    ?' # for TE 0 offset will also be 0, so we'll just do nothing
      $_CMN_iOffsets>_CMN_g3P1
      --
      $+_CMN_g3P1
      $_CMN_g3P1 16 |>
    .

    $CMN_iMem>_CMN_g3P1
    +
    $+_CMN_g3P1
  ~0
.

# arg: 3te
# ret: 3shift
# Computes a val (3shift) that's a bit shift representing how many subcells of
# TE 3typeEnv fit into one cell in TE 32., (i.e. e.g. 1 => one cell in TE 32 can
# hold 2^1 subcells).
_CMN_iTypeEnvToCellShift: ~32 $0 2 = >< 1 = 1 |< | ~0 .

# arg: 3te
# ret: 3binMask 3maskSize
# Similar to _CMN_iTypeEnvToCellShift but returns two vals: binary mask for the
# cell and num of 1 bits in it.
_CMN_iTypeEnvToCellMask:
  ~32
    $0 0 = $1 3 = | 5 |< $1 1 = 3 |< | >< 2 = 4 |< | # mask size
    +xffffffff 32 $2 - |> ><
  ~0
.

# arg: 3addr 3te
# ret: 3subAddr
# Seeks _CMN_g3P1 to mem cell that holds requested emulated stack cell. the
# returned 3subAddr is the remaining modulo subaddr within the seeked cell. The
# func doesn't check for stack boundaries.
_CMN_iStackSeek:
  ~32
    $0 $:_CMN_g3V1

    _CMN_iTypeEnvToCellShift

    $1 $1 $0 2 = | & >< $<0 >< $>0 # subcell modulo addr, store it down

    |>
    $_CMN_iCellSizeShift |<

    $_CMN_g3V1 ?' # get cell offset within block
      --
      $_CMN_iOffsets>_CMN_g3P1
      $+_CMN_g3P1
      $_CMN_g3P1
      +xffff &
    .

    +
    $_CMN_iStack>_CMN_g3P1
    $+_CMN_g3P1
  ~0
.

# ret: 3bool
# Checks if _CMN_g3P1 is within bounds of emulated stack.
_CMN_iCheckStackBounds:
  ~32 $_CMN_iStack=_CMN_g3P1 1 != $_CMN_iMemEnd=_CMN_g3P1 2 != & ~0
.

# arg: 3val 3te 3addr
# Sets cell in given TE of the interp's emulated stack to given val. If addr is
# out of bounds if accessed, args are consumed and interp state is set to error.
CMN_iStackSet:
  ~32
    $1 # keep the val at $1 for later use
    _CMN_iStackSeek

    _CMN_iCheckStackBounds ?          # stack:
      ><                              # val, subAddr, typeEnv
      _CMN_iTypeEnvToCellMask         # val, subAddr, mask, maskSize
      $<0 >< $<0 & $>0 $>0 $>0 *      # val&mask, mask, subAddr*maskSize
      |< $>0 >< $<0 |< $_CMN_g3P1 $>0 # val&mask<<, subCell, mask<<
      ! & |                           # (val&mask<<)|(subCell&!mask<<)

      $:_CMN_g3P1
    ;
      ^ ^ ^ # 3val 3te 3ret_from_stackSeek
      ~0 CMN_iSTATE_ERROR_MEM $:CMN_iState ~32
    .
  ~0
.

# arg: te ptrIndex 3shiftSigned
# Adds given signed val to given ptr in PT.
CMN_iPointerShift: CMN_iPtrTableSeek ~32 $_CMN_g3P1 + $:_CMN_g3P1 ~0 .

# arg: 3te 3addr
# ret: 3val
# Same as CMN_iStackSet but retrieves val instead of setting it, val 0 is
# returned in case of error (args will be consumed in either case).
CMN_iStackGet:
  ~32
    $1 _CMN_iStackSeek

    _CMN_iCheckStackBounds ?  # stack:
      ><                      # subAddr, typeEnv
      _CMN_iTypeEnvToCellMask # subAddr, mask, maskSize
      $<0 >< $>0 *            # mask, subAddr*maskSize
      $_CMN_g3P1 >< |> &      # subCell>>&mask
    ;
      ^ 0
    .
  ~0
.

# arg: te 3val
# Pushes given val onto emulated stack.
CMN_iPush:
  ~32 0 ~0
  $0 >32

  0 CMN_iPtrTableSeek

  ~32
    $_CMN_g3P1
    ++
    $0 $:_CMN_g3P1
    CMN_iStackSet
  ~0
.

# arg: te n
# ret: 3val
CMN_iGetStackTopMinusN:
  ~32 0 ~0 >32 ~32 0 ~0 $0 >32
  ~32 >< ~0

  0 CMN_iPtrTableSeek

  ~32
    $_CMN_g3P1 >< - CMN_iStackGet
  ~0
.

# arg: te
# ret: 3val
CMN_iGetStackTop: 0 CMN_iGetStackTopMinusN .

# arg: te
# ret: 3valY 3valX
CMN_iGetStackXY:
  ~32 0 ~0 $0 >32
```

```
  0 CMN_iPtrTableSeek
  ~32
    $_CMN_g3P1
    $1 $1 CMN_iStackGet
    >< $<0 >< $>0 -- CMN_iStackGet
    ><
  ~0
.

# arg: callStackSize
# Initializes interp for running BC loaded in CMN_bBytecode, i.e. the BC must be
# present in the mem when this is called, preliminary analysis of it will be
# done by this func., callStackSize is size of call stack in cells that will be
# allocated inside CMN_iMem and should be kept reasonably low to leave space for
# PTs and emulated stack.
CMN_iInit:
  CMN_iSTATE_RUNNING $:CMN_iState
  0 $:_CMN_iInputEnded

  CMN_eAnalyze

  ~32 0 ~0 ^ >32
  ~32 0 ~0 ^ >32
  ~32 0 ~0 ^ >32
  ~32 0 ~0 ^ >32

  # now callStackSize is still on the stack for later...
  ~32
    # compute a val that says which type envs are used
    $0 0 != 3 |<
    $2 0 != 2 |< |
    $3 0 != 1 |< |
    $4 0 != |

    $:_CMN_g3V1 # store the val here for now

    # set stack start:
    $CMN_iMem>_CMN_iCallStack
    $3 $3 $3 $3 + + +

    $+_CMN_iCallStack
    $_CMN_iCallStack>_CMN_iCallStackTop

    0 # here we'll move 0callStackSize
    ~0 >32 ~32
    $_CMN_iCallStack>_CMN_iStack
    $+_CMN_iStack

    # set PT offsets:
    $0 $:_CMN_iOffsets
    $>_CMN_iOffsets
    +
    $0 $:_CMN_iOffsets
    $>_CMN_iOffsets
    +
    $:_CMN_iOffsets
    ^
    # now go back and compute the cell offsets

    $_CMN_g3V1 +b1001 = ? 1 ; $_CMN_g3V1 +b0111 >= 3 * .
    $_CMN_iOffsets 16 |< |
    $:_CMN_iOffsets
    $<_CMN_iOffsets

    $_CMN_g3V1 $0 +b0011 = >< +b1010 = | ? 1 ; $_CMN_g3V1 +b0111 >= 2 * .
    $_CMN_iOffsets 16 |< |
    $:_CMN_iOffsets
    $<_CMN_iOffsets

    $_CMN_g3V1 +b0101 >=
    $_CMN_iOffsets 16 |< |
    $:_CMN_iOffsets

    # cell size shift
    $_CMN_g3V1 +b0101 & $_CMN_g3V1 1 |> +b0101 & +
    $0 +b0011 & >< 2 |> +
    -- $0 3 = -

    $:_CMN_iCellSizeShift

    # zero the whole interp mem:
    $CMN_iMem>_CMN_g3P1

    CMN_iMEM_SIZE @'
      0 $:_CMN_g3P1 $>_CMN_g3P1
      --
    .
    ^

    # set mem end ptr:
    $CMN_iMem>_CMN_iMemEnd
    CMN_iMEM_SIZE $+_CMN_iMemEnd

    0 CMN_bCurrentInstrSetAddr
  ~0
.

# Performs another interp step (executes the next instruction). This func may
# call the user defined I/O funcs (even several times). If interp state is not
# OK (e.g. error or end), nothing happens. Nothing is returned, status is kept
# in CMN_iState.
CMN_iStep:
  $CMN_iState CMN_iSTATE_RUNNING = ?
    ~32 -1 $:_CMN_iJumpAddr ~0

    CMN_bCurrentInstrGet CMN_bInstrOpcode

    0 $:_CMN_iPopPush

    $0 4 |> 2 < ?
      # special instruction

      $0 +b00010000 & ?
        $0 +b00001000 & ?
          # 00011xxx

          $0 CMN_bOPCODE_CON = ?
            # COC does pop/push in different order, so we handle the push here
            # and only let the later code only handle the pop

            CMN_bCurrentInstrTypeEnv
            ~32 CMN_bCurrentInstrGetConst32 ~0
            CMN_iPush

            +b10 $:_CMN_iPopPush
          ;
          $0 CMN_bOPCODE_POP = ?
            CMN_bCurrentInstrGetConst ++ 1 |< $:_CMN_iPopPush
          ;
          $0 CMN_bOPCODE_SWP = ?
            CMN_bCurrentInstrTypeEnv
            $0 $0 # 3 vals: for following func, then for two pushes later
            CMN_iGetStackXY

            CMN_bCurrentInstrGet CMN_bInstrWontPop !! ?
              CMN_bCurrentInstrTypeEnv
              0 ~32 -2 ~0 CMN_iPointerShift
            .

            CMN_iPush
            CMN_iPush
          ;
          $0 CMN_bOPCODE_CND = ?
            CMN_bCurrentInstrTypeEnv
            $0 2 CMN_iGetStackTopMinusN
            CMN_iGetStackXY
            ~32 ??  $0 >0       $:_CMN_iPushValue ~0
            +b111 $:_CMN_iPopPush
          ;
          $0 CMN_bOPCODE_TRA = ?
            CMN_bCurrentInstrTypeEnv
            CMN_iGetStackTop
            CMN_bCurrentInstrGetConst
            ~32 0 ~0 $0 >32
            0 CMN_iPtrTableSeek
            ~32 $_CMN_g3P1 ~0
            CMN_iStackSet
            +b10 $:_CMN_iPopPush
          ;
          $0 CMN_bOPCODE_OUT = ?
            CMN_bCurrentInstrTypeEnv
            CMN_iGetStackTop
            0 ~32 >0 ~0
            CMN_iOut
            +b10 $:_CMN_iPopPush
          . . . . . .
        ;
        # 00010xxx

        $0 CMN_bOPCODE_MGE = ?
          CMN_bCurrentInstrTypeEnv
          ~32 0 ~0 $0 >32
          CMN_bCurrentInstrGetConst32
          _CMN_iPtrIToAddr
          CMN_iStackGet
          ~32 $:_CMN_iPushValue ~0
          +b01 $:_CMN_iPopPush
        ;
        $0 CMN_bOPCODE_MEX = ?
          CMN_bCurrentInstrTypeEnv
          $0 CMN_iGetStackTop
          ~32 0 ~0 $0 >32
          CMN_bCurrentInstrGetConst32
          _CMN_iPtrIToAddr
          CMN_iStackSet
          +b10 $:_CMN_iPopPush
        ;
        $0 CMN_bOPCODE_PAC = ?
          CMN_bCurrentInstrTypeEnv
          0 # here ptr index will go

          ~32
            CMN_bCurrentInstrGetConsts32
            CMN_ePtrIToPtrTableI >0

            $0 +b1000 & ? # 4bit two's compl., sign extend if needed
              +xfffffff0 |
            .

            CMN_iPointerShift
          ~0
        ;
        $0 CMN_bOPCODE_PCO = ?
          CMN_bCurrentInstrTypeEnv
          CMN_bCurrentInstrGetConsts32
          0 ~32 CMN_ePtrIToPtrTableI >0 ~0
          $1 0 ~32 CMN_ePtrIToPtrTableI >0 ~0
          CMN_iPtrTableSeek
          ~32 $_CMN_g3P1 ~0
          CMN_iPtrTableSeek
          ~32 $:_CMN_g3P1 ~0
        ;
        $0 CMN_bOPCODE_PUX = ?
          CMN_bCurrentInstrTypeEnv $0 CMN_iGetStackTop
          0 ~32 >0 ~0 CMN_iGetStackTopMinusN
          ~32 $:_CMN_iPushValue ~0
          +b11 $:_CMN_iPopPush
        ;
        $0 CMN_bOPCODE_PAX = ?
          CMN_bCurrentInstrTypeEnv
          $0 CMN_iGetStackTop

          $0 1 = 0 ~32 $0 +x80 & >0 ~0 && ?
            ~32 +xffffff00 | ~0
          ;
          $0 2 = 0 ~32 $0 +x00008000 & >0 ~0 && ?
            ~32 +xffff0000 | ~0
          . .

          0 # for PT index
          ~32
            CMN_bCurrentInstrGetConst32
            CMN_ePtrIToPtrTableI
            >0
          ~0

          $0 ?
            +b10 $:_CMN_iPopPush # don't pop if we're shifting ST
          .

          CMN_iPointerShift
        ;
        $0 CMN_bOPCODE_PCM = ?
          CMN_bCurrentInstrTypeEnv $0
          CMN_bCurrentInstrGetConsts32

          ~32
            >< _CMN_iPtrIToAddr >< _CMN_iPtrIToAddr
            - $0 0 << 1 |< >< 0 >> | $:_CMN_iPushValue
          ~0

          +b01 $:_CMN_iPopPush
        ;
        # CMN_bOPCODE_PSC
          CMN_bCurrentInstrTypeEnv 0

          ~32
            CMN_bCurrentInstrGetConsts32
            CMN_ePtrIToPtrTableI
            >0
            CMN_iPtrTableSeek
```

```
          $:_CMN_g3P1
        ~0
      . . . . . . .
    .
  ;
  $0 +b00001000 & ?
    # 00001xxx

    $0 CMN_bOPCODE_RET = ?
      ~32
        $<_CMN_iCallStackTop # TODO: check bounds (or just trust BC?)
        $_CMN_iCallStackTop
        ++ $:_CMN_iJumpAddr
      ~0
    ;
    $0 CMN_bOPCODE_CAE = ?
      CMN_bCurrentInstrGetConst
      CMN_iCallExt
    ;
    $0 CMN_bOPCODE_INI = ?
      $0>_CMN_gP1
      CMN_iGetArgs
      0 $:_CMN_gV1

      @@
        $_CMN_gV1 ++ $:_CMN_gV1
        $>_CMN_gP1

        $_CMN_gP1=0 1 = ?
          !@
        .

        ~32 0 ~0 $_CMN_gP1 >32 0
        CMN_iPush
      .

      $_CMN_gV1 -1 * $+0
    ;
      # CMN_bOPCODE_JIA, CMN_bOPCODE_JNA, CMN_bOPCODE_JMA
      $0 CMN_bOPCODE_JMA = # val that says whether to jump or not

      $0 !! ?
        CMN_bCurrentInstrTypeEnv CMN_iGetStackTop
        $1 CMN_bOPCODE_JNA = ? ~32 !! ~0 .
        ~32 >0 ~0
        +b10 $:_CMN_iPopPush
      .

      ? # do jump?
        ~32 CMN_bCurrentInstrGetConst32 $:_CMN_iJumpAddr ~0
      .
    . . .
  ;
    # 00000xxx

    $0 CMN_bOPCODE_CAL = ?
      ~32
        CMN_bCurrentInstrGetConst32
        $CMN_bCurrentInstrAddr $:_CMN_iCallStackTop
        $>_CMN_iCallStackTop
        $:_CMN_iJumpAddr

        # check call stack overflow:
        $_CMN_iCallStackTop=_CMN_iStack 2 != ?
          ~0 CMN_iSTATE_ERROR_CALL $:CMN_iState ~32
        .
      ~0

    ; $0 CMN_bOPCODE_END = ?
      CMN_iSTATE_DONE $:CMN_iState
    . .
  .
.
;
  # typical stack instrs

  $0 CMN_bOPCODE_ADR < ?
    # instrs that belong to subgroups

    $0 +b00000011 & # get the mode, push instr args accordingly to TE 32

    $0 CMN_bMODE21 = ?
      CMN_bCurrentInstrTypeEnv CMN_iGetStackXY
      +b0101 $:_CMN_iPopPush
    ;
    $0 CMN_bMODE1C1 = ?
      CMN_bCurrentInstrTypeEnv CMN_iGetStackTop
      CMN_bCurrentInstrGetConst32
      +b0011 $:_CMN_iPopPush
    ;
    $0 CMN_bMODE11 = ?
      ~32 0 ~0
      CMN_bCurrentInstrTypeEnv CMN_iGetStackTop
      +b0011 $:_CMN_iPopPush
    ;
    # CMN_bMODE01
      ~32 0 0 ~0
      +b0010 $:_CMN_iPopPush
    . . .

    ^ # opcode mode

    $0 +b11111100 & # extract the group
    $0 +b01100000 & # take two bits by which we accelerate the search
    ><

    $1 +b00100000 = ?
      ~32 $1 $1 ~0

      $0 CMN_bOPCODE_ADX = ? ~32 + ~0 ;
      $0 CMN_bOPCODE_SUX = ? ~32 - ~0 ;
      $0 CMN_bOPCODE_MUX = ? ~32 * ~0
      ;
        # CMN_bOPCODE_DIX, CMN_bOPCODE_DSX, CMN_bOPCODE_MOX, CMN_bOPCODE_MSX

        ~32 $0 0 = ? ~0
          # division by zero
          CMN_iSTATE_ERROR_OP $:CMN_iState
          ~32 ^ ^ 0 ~0
        ;
          $0 CMN_bOPCODE_DIX = ? ~32 / ~0 ;
          $0 CMN_bOPCODE_MOX = ? ~32 % ~0 ;
          $0 CMN_bOPCODE_DSX = ?
            CMN_bCurrentInstrTypeEnv
            $0 1 = ?  ~8 0 ~32 >8 ~8 0 ~32 >8 0 ~8 >< // >32 ~0       ;
            $0 2 = ?  ~16 0 ~32 >16 ~16 0 ~32 >16 0 ~16 >< // >32 ~0  ;
                      ~32 // ~0 # 0 and 3
            . .
            ^
          ;
          # CMN_bOPCODE_MSX
            CMN_bCurrentInstrTypeEnv
            $0 1 = ?  ~8 0 ~32 >8 ~8 0 ~32 >8 0 ~8 >< %% >32 ~0       ;
            $0 2 = ?  ~16 0 ~32 >16 ~16 0 ~32 >16 0 ~16 >< %% >32 ~0  ;
                      ~32 %% ~0 # 0 and 3
            . .
            ^
          . . .
        .
      . . .

        ~32 $:_CMN_iPushValue ~0
      ;
      $1 +b01000000 = ?
        ~32 $1 $1 ~0

        $0 CMN_bOPCODE_GRX = ? ~32 >  ~0 ;
        $0 CMN_bOPCODE_GEX = ? ~32 >= ~0 ;
        $0 CMN_bOPCODE_SMX = ? ~32 <  ~0 ;
        $0 CMN_bOPCODE_SEX = ? ~32 <= ~0
        ;
          # signed comparisons

          $0 +b100 & 0 ~32 = >0 ~0 && ? # equality check and equal?
            ~32 1 ~0
          ;
            $0 CMN_bOPCODE_SSX = $0 CMN_bOPCODE_LSX = | ?
              ~32 >< ~0
            .

            ~32 $1 $1 ~0

            CMN_bCurrentInstrTypeEnv

            $0 1 = ? ~8  0 ~32 >8  ~8  0 ~32 >8  0 ~8  >< >> >32 ~0 ;
            $2 2 = ? ~16 0 ~32 >16 ~16 0 ~32 >16 0 ~16 >< >> >32 ~0 ;
                     ~32 >> ~0
            . .
            ^
          .
        . . . .

          ~32 $:_CMN_iPushValue ~0
        ;
        $1 +b01100000 = ?
          ~32 $1 $1 ~0
          $0 CMN_bOPCODE_EQX = ? ~32 =   ~0 ;
          $0 CMN_bOPCODE_NEX = ? ~32 !=  ~0 ;
          $0 CMN_bOPCODE_BAX = ? ~32 &   ~0 ;
          $0 CMN_bOPCODE_BOX = ? ~32 |   ~0 ;
          $0 CMN_bOPCODE_BXX = ? ~32 |!  ~0 ;
          $0 CMN_bOPCODE_LAX = ? ~32 &&  ~0 ;
          $0 CMN_bOPCODE_LOX = ? ~32 ||  ~0 ;
          # CMN_bOPCODE_LXX #    ~32 |!! ~0
          . . . . . . . .
          ~32 $:_CMN_iPushValue ~0
        ;
        # +b00000000
          $0 CMN_bMODE11 | CMN_bOPCODE_BNO = ?
            ~32 $0 ! $:_CMN_iPushValue ~0
          .
          ;
          $0 CMN_bOPCODE_SRX = ?  ~32 $1 $1 |> $:_CMN_iPushValue ~0  ;
          $0 CMN_bOPCODE_SLX = ?  ~32 $1 $1 |< $:_CMN_iPushValue ~0
          . . .
        . . .

        ^ ^ # group and search bits

        ~32 ^ ^ ~0 # instr args
      ;
        $0 CMN_bOPCODE_INP = ?
          CMN_iIn
          $0 -1 = $_CMN_iInputEnded || ?
            ^ ~32 0 ~0
            1 $:_CMN_iInputEnded
          ;
            ~32 0 ~0 >32
          .

          ~32 $:_CMN_iPushValue ~0
          +b01 $:_CMN_iPopPush
        ;
        $0 CMN_bOPCODE_INU = ?
          $_CMN_iInputEnded !! ~32 0 ~0 >32 ~32 $:_CMN_iPushValue ~0
        ;
        ;
        # CMN_bOPCODE_ADR
          +b01 $:_CMN_iPopPush
          CMN_bCurrentInstrTypeEnv 0 CMN_iPtrTableSeek
          ~32 $_CMN_g3P1 $:_CMN_iPushValue ~0
        . .
      .
  . # end of opcode analysis

  ^ # opcode

  $_CMN_iPopPush ?' # now handle pops and pushes
    # first recompute new ST:
    ~32
      0
      ~0 $_CMN_iPopPush >32 ~32

      $0 1 & >< 1 |>

      ~0 CMN_bCurrentInstrGet CMN_bInstrWontPop ? ~32
        ^
      ;
        -
      .

      CMN_bCurrentInstrTypeEnv
      ~0 0 ~32
      CMN_iPointerShift
    ~0

    # now write to ST:
    $_CMN_iPopPush 1 & ?
      ~32 $_CMN_iPushValue 0 ~0

      CMN_bCurrentInstrTypeEnv >32

      ~32
        $_CMN_g3P1
        CMN_iStackSet
      ~0
    .
  .
  ^

  ~32
    $_CMN_iJumpAddr -1 = ?
      CMN_bCurrentInstrShift
    ;
```

```
        $_CMN_iJumpAddr CMN_bCurrentInstrSetAddr
      .
    ~0
  .
.
```

cmn_tokenizer.cmn

```
# comun self hosted implementation: tokenizer library
# module prefix: t
#
# This module contains resources that will help parsing comun source code as
# a stream of language tokens.

~"cmn_general.cmn"

~_CMN_tState
~_CMN_tSTBackup:0   # ST backup

CMN_tTOKEN_NONE:     0   . # no token detected
CMN_tTOKEN_COMMAND:  1   . # command such as "<=", ".", "+", "$>abc", ...
CMN_tTOKEN_NAME:     2   . # name of a func (call)
CMN_tTOKEN_FUNC_DEF: 3   . # name of a func with ":" appended
CMN_tTOKEN_NUMBER:   4   . # numeric literal
CMN_tTOKEN_STRING:   5   . # string literal
CMN_tTOKEN_ERROR:    255 . # badly formed token

_CMN_tSTATE_BLANK:   0   . # reading blank chars between tokens
_CMN_tSTATE_COMMENT: 1   . # inside comment
_CMN_tSTATE_STRING:  2   . # inside string literal
_CMN_tSTATE_NAME:    3   . # reading name (identifier)
_CMN_tSTATE_FUNDEF:  4   . # just read func definition
_CMN_tSTATE_SIGN:    5   . # starting + or -
_CMN_tSTATE_NUMBER:  6   . # num, next must be digit or end
_CMN_tSTATE_COMMAND: 7   . # reading command

CMN_tCharIsBlank: " " <= .  # TODO: handle also square brackets

CMN_tCharIsNameChar:
  $0 "0" >= $1 "9" <= &&
  $1 "a" >= $2 "z" <= && ||
  $1 "A" >= $2 "Z" <= && ||
  >< "_" = ||
.

# arg: t0 t1 ... tn
# ret: t0 t1 ... tn 0 tn ... t1 t0
# Converts a token from tokenizer format to a normal reversed zero ter. str.
_CMN_tTokenToStr:
  $0>_CMN_gP1
  0
  1 @
    $_CMN_gP1
    $<_CMN_gP1
    $_CMN_gP1=_CMN_tSTBackup
  .
.

CMN_tCharIsHexDig:
  $0 "0" >= $1 "9" <= &&
  $1 "a" >= $2 "f" <= && ||
  >< ^
.

CMN_tCharIsCommChar:
  $0 CMN_tCharIsBlank !!
  $1 "#" != &&
  >< 34 != && # '"'
.

# Initializes tokenizer, has to be called before tokenizer is used.
CMN_tInit:
  $0>_CMN_tSTBackup
  _CMN_tSTATE_BLANK $:_CMN_tState
.

CMN_tPopToken:
  $_CMN_tSTBackup>0
.

# arg: token pattern
# ret: bool
# Checks if currently read token matches given pattern. The token arg is
# currently read token (NOT a zero terminated string), pattern is either 0 or
# 1 terminated string, pushed backwards (just like normal comun strings);
# terminating 0 means the pattern has to match exactly, terminating 1 means
# the pattern may match only up to this val. The pattern arg will be popped,
# token won't.
CMN_tTokenMatch:
  1 $:_CMN_gV1 # matches?

  $_CMN_tSTBackup>_CMN_gP1
  $>_CMN_gP1

  @@
    $0 1 = ?
      ^
      !@
    .

    $0 $_CMN_gP1 != ?
      0 $:_CMN_gV1
    .

    !! ?
      !@
    .

    $>_CMN_gP1
  .

  $_CMN_gV1
.

# arg: token
# ret: token state 3val
# Extracts val from a token that contain numeric literal, expects token on input
# (in tokenizer's format, NOT a normal zero terminated string); this token will
# remain unpopped. Note the token may be a numeric literal but also another
# token that may contain such literal (e.g. "~myvar:100"). The val is returned
# as 32 bit integer in TE 32. Returns state, which may be: 0 (error), 1
# (positive) or 2 (negative, i.e. numeric literal was prepended with '-').
CMN_tLiteralVal:
   $0>_CMN_gP1

   0 # stopper

   @@ # go from right to the left, find start
     $_CMN_gP1 "+" != $_CMN_gP1 "-" != &
       $_CMN_gP1 CMN_tCharIsHexDig !! &
       $_CMN_gP1 "x" != &
       $0=_CMN_gP1 !! | ?
         !@
       .

       $<_CMN_gP1
     .

     $>_CMN_gP1

     0 # negative ?

     $_CMN_gP1 "+" = ?
       $>_CMN_gP1
     ; $_CMN_gP1 "-" = ?
       ^ 1 $>_CMN_gP1
     . .

     10 # base

     $_CMN_gP1 "d" = ?
       $>_CMN_gP1
     ; $_CMN_gP1 "x" = ?
       ^ 16 $>_CMN_gP1
     ; $_CMN_gP1 "b" = ?
       ^ 2 $>_CMN_gP1
     . . .

     ~32 0 ~0

     @@
       $_CMN_gP1 !! ?
         !@
       .

       $_CMN_gP1 # digit val

       $0 $0 "0" >= >< "9" <= &
         "0"
         "a" 10 - ??
       -

       $0 $2 >= ? # digit >= base => ERROR
         ^ ^ 0
         !.
       .

       ~32 0 ~0 $1 >32 ~32 * ~0
       ~32 0 ~0 >32    ~32 + ~0

       $>_CMN_gP1
     .

     ^ # base

     ? # nagative ?
       2 ~32 -1 * ~0
     ;
       1
     .

     >< ^ # pop stopper 0, leave only result
.

# arg: char
# ret: tokenType
# Feeds the next char to tokenizer, which may or may NOT be popped by this func,
# which the caller must leave so -- this behavior is such so that once the
# complete token has been read and detected, it will remain on the stack so that
# it can be analyzed (after which the whole token can be popped with
# CMN_tPopToken). Pushes the type of token detected; this val on the other hand
# MUST be popped by the caller before feeding another char. Feeding 0 means end
# of input.
CMN_tFeedChar:
  # TODO: order branches by probability?
  # BUT WATCH OUT, some order has to be kept (e.g. check digit before name!)

  $_CMN_tState _CMN_tSTATE_COMMENT = ?
    $0 "#" = >< 10 = || ?     # comment end?
      _CMN_tSTATE_BLANK $:_CMN_tState
    .
    # no pop is here, it was done by the above branch

    CMN_tTOKEN_NONE
  ;
  $_CMN_tState _CMN_tSTATE_STRING = ?
    $0 34 = ?    # '"' ?
      _CMN_tSTATE_BLANK $:_CMN_tState
      CMN_tTOKEN_STRING
    ;
      CMN_tTOKEN_NONE
    .
  ;
  $_CMN_tState _CMN_tSTATE_COMMAND = ?
    $0 CMN_tCharIsBlank ?
      ^
      _CMN_tSTATE_BLANK $:_CMN_tState
      CMN_tTOKEN_COMMAND
    .
    $0 "#" = ?
      ^
      _CMN_tSTATE_COMMENT $:_CMN_tState
      CMN_tTOKEN_COMMAND
    ;
      _CMN_tSTATE_COMMAND $:_CMN_tState
      CMN_tTOKEN_NONE
    . .
  ;
  $_CMN_tState _CMN_tSTATE_NAME = ?
    $0 ":" = ?
      _CMN_tSTATE_FUNDEF $:_CMN_tState
      CMN_tTOKEN_NONE
    ;
    $0 CMN_tCharIsBlank ?
      ^
      _CMN_tSTATE_BLANK $:_CMN_tState
      CMN_tTOKEN_NAME
    ;
    $0 "#" = ?
      ^
      _CMN_tSTATE_COMMENT $:_CMN_tState
      CMN_tTOKEN_NAME
    ;
    $0 CMN_tCharIsNameChar ?
      CMN_tTOKEN_NONE
    ;
      _CMN_tSTATE_COMMAND $:_CMN_tState
      CMN_tTOKEN_NONE
    . . . .
  ;
```

```
  $_CMN_tState _CMN_tSTATE_FUNDEF = ?
    # expecting end after ':'
    $0 CMN_tCharIsBlank ?
      ^
      _CMN_tSTATE_BLANK $:_CMN_tState
      CMN_tTOKEN_FUNC_DEF
    ;
    $0 "#" = ?
      ^
      _CMN_tSTATE_COMMENT $:_CMN_tState
      CMN_tTOKEN_FUNC_DEF
    ;
      CMN_tTOKEN_ERROR
    . .
  ;
  $_CMN_tState _CMN_tSTATE_SIGN = ?
    $0 "d" = $1 "x" = || $1 "b" = || $1 CMN_tCharIsHexDig || ?
      _CMN_tSTATE_NUMBER $:_CMN_tState
      CMN_tTOKEN_NONE
    ;
    $0 "#" = ?
      ^
      _CMN_tSTATE_COMMENT $:_CMN_tState
      CMN_tTOKEN_COMMAND
    ;
    $0 CMN_tCharIsBlank ?
      ^
      _CMN_tSTATE_BLANK $:_CMN_tState
      CMN_tTOKEN_COMMAND
    ;
      _CMN_tSTATE_COMMAND $:_CMN_tState
      CMN_tTOKEN_NONE
    . . .
  ;
  $_CMN_tState _CMN_tSTATE_NUMBER = ?
    $0 CMN_tCharIsHexDig ?
      CMN_tTOKEN_NONE
    ;
    $0 CMN_tCharIsBlank ?
      ^
      _CMN_tSTATE_BLANK $:_CMN_tState
      CMN_tTOKEN_NUMBER
    ;
    $0 "#" = ?
      ^
      _CMN_tSTATE_COMMENT $:_CMN_tState
      CMN_tTOKEN_NUMBER
    ;
      CMN_tTOKEN_ERROR
    . . .
  ;
  # _CMN_tSTATE_BLANK
    $0 "#" = ?                  # comment start?
      ^
      _CMN_tSTATE_COMMENT $:_CMN_tState
    ;
    $0 $0 "+" = >< "-" = || ?   # sign?
      _CMN_tSTATE_SIGN $:_CMN_tState
    ;
    $0 $0 "0" >= >< "9" <= && ? # decimal digit?
      _CMN_tSTATE_NUMBER $:_CMN_tState
    ;
    $0 34 = ?                   # '"', string start?
      _CMN_tSTATE_STRING $:_CMN_tState
    ;
    $0 CMN_tCharIsNameChar ?
      _CMN_tSTATE_NAME $:_CMN_tState
    ;
    $0 CMN_tCharIsBlank !! ?    # non-blank?
      _CMN_tSTATE_COMMAND $:_CMN_tState
    ;
      ^
    . . . . . .

    CMN_tTOKEN_NONE
  . . . . . .
.
```

comun.cmn

```
# comun self hosted implementation: comun all-in-one utility
#
# This is a general command line utility for the comun language, including
# compiler, optimizer, interpreter, debugger etc.
#
#  {}{}{} {}{}{} {}{}{} {}  {} {}{}{}
#  {}     {}  {} {}{}{} {}  {} {}  {}
#  {}{}{} {}{}{} {}  {} {}{}{} {}  {}

~v1                 # helper var
~p1:0               # helper ptr
~flags              # CLI flags as bits
~sourceLine         # counts source code lines
~sourceLatestChars:6 # buffer to keep lastest chars read
~state

flagToBit:
  $:v1
  0 "dcbvtoh"
  $v1

  1 $:v1

  @@
    $1 !! $1 $3 = | ?
      !@
    .

    $v1 1 |< $:v1
    $:1
  .

  @'
    ^
  .

  ^
  $v1
.

printPrefixStr: "=== " .

# arg: 3num 3base 3digits
_numPrint:
  0

  ~32
  @'
    ~0 0 ~32
    $2 $2 % $0 10 < "0" "A" 10 - ?? + >0
    $2 $2 / $:3
    --
  .
  ^ ^ ^
  ~0

  -->
.

numPrintD6: ~32 0 ~0 >32 ~32 10 6 ~0 _numPrint .
numPrintB8: ~32 0 ~0 >32 ~32 2 8 ~0  _numPrint .

isFlagSet: flagToBit $flags & .

verbosePrint:
  "v" isFlagSet ?
    -->
  ;
    @ .
  .
.

CMN_iIn: <- .
CMN_iOut: -> .
CMN_iGetArgs: 0 "abc" 0 "defgh" 2 .
CMN_iCallExt: 0 "EXT: " --> "0" + -> 10 -> .

CMN_cInclude:
  0 34 "include: " printPrefixStr verbosePrint
  verbosePrint # filename string
  0 10 34 verbosePrint
.

~32 CMN_iMEM_SIZE: 8192 . ~0
~32 ~CMN_iMem:8192 ~0

~32 CMN_cMEM_SIZE: 2048 . ~0
~32 ~CMN_cMem:2048 ~0

~32 CMN_bBYTECODE_SIZE: 32768 . ~0
~16 ~CMN_bBytecode:32768 ~0

~"cmn_all.cmn"

printHelp:
  0
  10 "flags (pass as single string):"
  10 CMN_gLANG_V_STR "lang. ver: "
  10 "comun: all-in-one utility"
  -->

  0
  10 "  d debug"
  10 "  c compile, output bytecode (else interpret)"
  10 "  b input bytecode (else input source code)"
  -->

  0
  10 "  v verbose"
  10 "  t compile, output text (else interpret)"
  10 "  o optimize"
  10 "  h help"
  -->

  # TODO: flag to analyze program (estimate, measure, ...)
.

printStackTops:
  0 "stack tops: " printPrefixStr -->
  $$ numPrintD6 " " ->
  0 ~32 $$ >0 ~0 numPrintD6 " " ->
  0 ~16 $$ >0 ~0 numPrintD6 " " ->
  0 ~8  $$ >0 ~0 numPrintD6 " " ->
  10 ->
.

printCompiler:
  0 10 "compiler" printPrefixStr -->
  0 "state = " -->        $CMN_cState numPrintD6
  0 "; addr. size = " --> _CMN_cDefaultAddrSize numPrintD6

  10 ->

  0 10 "  symbol table: " -->

  0 _CMN_cSymbolTableSize ~32 >0 ~0
  @'
    --
    0 "    " --> 0 $1 CMN_cSymbolGet --> " " -> 0 ~32 >0 ~0 numPrintD6 10 ->
  .
  ^

  # TODO

  10 ->
.

# Prints BC in CMN_bBytecode to output.
printBytecode:
  ~32 0 CMN_bCurrentInstrSetAddr ~0

  0 $:v1

  1 @
    CMN_bCurrentInstrGet
    CMN_bCurrentInstrShift

    $v1 numPrintB8 ":" -> " " ->

    $0 CMN_eInstrToStr -->

    $0 CMN_bInstrOpcode CMN_bOPCODE_DES = ?
      $0 8 |> +x0f &

      $0 CMN_bDES_IF = ?      0 " # if"      ;
      $0 CMN_bDES_LOOP = ?    0 " # loop"    ;
      $0 CMN_bDES_ELSE = ?    0 " # else"    ;
      $0 CMN_bDES_ENDIF = ?   0 " # endif"   ;
      $0 CMN_bDES_ENDLOOP = ? 0 " # endloop" ;
      $0 CMN_bDES_BREAK = ?   0 " # break"   ;
      $0 CMN_bDES_EXIT = ?    0 " # exit"    ;
      $0 CMN_bDES_FUNDEF = ?  0 " # func"    ;
      $0 CMN_bDES_LABEL =  ?  0 " # label"   ;
      $0 CMN_bDES_GOTO =  ?   0 " # goto"    ;
                              0
      . . . . . .

      -->

      ^
```

```
    .

    10 ->

    $v1 ++ $:v1
  .
.

readSourceChar:
  <-

  $0 10 = ?
    $sourceLine ++ $:sourceLine
  .

  $sourceLatestChars>p1

  5 @'
    $>p1
    $p1 $p1 10 != >< " " ??
    $<p1 $:p1 $>p1
    --
  .
  ^

  $0 $:p1
.

compile:
  0 10 "compiling" printPrefixStr verbosePrint

  0 CMN_cInit

  1 @
    readSourceChar
    CMN_cFeedChar

    $CMN_cState CMN_cSTATE_OK != <? !! | !!
  .

  $CMN_cState CMN_cSTATE_DONE = ?
    0 10 "compiled successfuly" printPrefixStr verbosePrint
    1 # state
  ;
    0 $:state

    0 "line "

    $CMN_cState CMN_cSTATE_ERROR_TOKEN = ?       "(bad token) "       ;
    $CMN_cState CMN_cSTATE_ERROR_UNEXPECTED = ?  "(unexpected) "      ;
    $CMN_cState CMN_cSTATE_ERROR_NAME_UNK = ?    "(unknown name) "    ;
    $CMN_cState CMN_cSTATE_ERROR_NAME_EXISTS = ? "(name redefined) "  ;
    $CMN_cState CMN_cSTATE_ERROR_SYMBOL_TAB = ?  "(symb. tab. mem.) " ;
    $CMN_cState CMN_cSTATE_ERROR_BC_TOO_BIG = ?  "(bytecode too big) "
    . . . . . .

    " " $CMN_cState 2 - "0" + "ERROR "
    -->

    $sourceLine numPrintD6

    34 ": " -> -> ->

    $sourceLatestChars>p1

    6 @'
      $p1 -> $>p1
      --
    .
    ^

    34 -> 10 ->

    0 10 "couldn't compile" printPrefixStr verbosePrint
    0 # state
  .
.

interpret:
  0 10 "interpreting" printPrefixStr verbosePrint

  32 CMN_iInit

  1 @
    CMN_iStep
    $CMN_iState CMN_iSTATE_RUNNING =
  .

  $CMN_iState CMN_iSTATE_DONE = ?
    0 10 "interpreting ended successfully" printPrefixStr verbosePrint
  ;
    0

    $CMN_iState CMN_iSTATE_ERROR_MEM = ?   " (memory access)" ;
    $CMN_iState CMN_iSTATE_ERROR_CALL = ?  " (call stack)" ;
    $CMN_iState CMN_iSTATE_ERROR_OP = ?    " (bad operation)" ;
    $CMN_iState CMN_iSTATE_ERROR_BC = ?    " (bad bytecode)"
    . . . .

    $CMN_iState 2 - "0" + "ERROR "
    -->
    10 ->

    0 10 "interpreting ended with error" printPrefixStr verbosePrint
  .
.

$:v1 # argument count

@@ # read arguments
  $v1 0 = ?
    !@
  .

  $v1 -- $:v1

  "-" = ?
    @'
      $v1 ><
      flagToBit $flags | $:flags
      $:v1
    .
  ;
    @'
      ^
    .
  .

  ^
.

"h" isFlagSet ?
  printHelp
  !.
.

0 10 "starting" printPrefixStr verbosePrint

#TMP_PRADDRESSES

"d" isFlagSet ?
  printStackTops
.

"b" isFlagSet ?
  0 10 "loading bytecode" printPrefixStr verbosePrint
  CMN_eLoadBC
;
  compile

  "d" isFlagSet ?
    printCompiler
  .
.

?
  "d" isFlagSet ?
    printStackTops
  .

  "t" isFlagSet ?
    printBytecode
  ;
  "c" isFlagSet ?
    CMN_bOutput
  ;
    interpret
  . .
;
  0 10 "error, stopping" printPrefixStr verbosePrint
.

"d" isFlagSet ?
  printStackTops
.

0 10 "ending" printPrefixStr verbosePrint
```

comun_includeproc.cmn

```
# comun: file include processor
#
# This is a tool to help preprocess comun source files as related to the file
# incude directive. As pure comun doesn't support file I/O and so can only take
# a single input stream of characters, in cases when a source code consists of
# multiple source files we have to merge them into one input stream. This tool
# helps with that. It works in following way:
#
# Feed all source files to the input of this utility, concatenated in the
# following format:
#
# <filename1><newline><filecontent1><zero><filename2><newline>...
#
# The first file will be considered the main file. The utility will output a
# structure (NOT the file itself!) of the final file as lines, each in format:
#
# <filename> <space> <startpos> <space> <endpos>
#
# This will says which parts of which files and in what order have to be
# concatenated to produce the final file.
#
# NOTE: making the utility output the whole final file could be done too of
# course, but it would require a lot of RAM to hold the whole file contents. We
# output only the file structure in hopes other tools will be able to make the
# final file more efficiently.
#
# Basic principles: we process the input and construct one big str in TE 8; this
# str contains the file names along with symbols (start/end of file, file
# include) and positions, we then print the string out while recursively
# printing out the "include" parts.

STATE_FILENAME:        0 . # reading filename
STATE_CONTENT:         1 . # reading file content
STATE_CONTENT_COMMENT: 2 . # reading file content (inside comment)
STATE_CONTENT_STRING:  3 . # reading file content (inside str lit.)
STATE_CONTENT_TILDE:   4 . # reading file content, read tilde
STATE_CONTENT_INCLUDE: 5 . # reading file content, read include filename

SYMBOL_EOF:            1 . # marks end of file
SYMBOL_SOF:            2 . # marks start of file
SYMBOL_SOF_DONE:       3 . # marks start of already included file
SYMBOL_INCLUDE:        4 . # marks start of file include
SYMBOL_SPACE:          5 . # holds place for spaces
SYMBOL_BLOCK_SEP:      6 . # separates blocks

~state        # state machine state

~8
  ~str1:2048  # the big str
  ~str2:0     # smaller tmp str, will be pointed to main stack
  ~ptr:0
  ~ptr2:0
  ~initAddr:0 # initial stack top

  $0>str2
  $str1>0
  $str2>ptr
  $0>initAddr
~0

~32
  ~pos1 # helper vars
  ~pos2
~0

# arg: char
# ret:
# Pushes a single char to str1.
pushChar:
  ~8 0 ~0 >8
.

# arg: 3n
# ret:
# Converts a number to string which will be pushed to str1.
pushNum:
  ~32
    0 ><
    @@
      $0 10 % "0" + ><
```

```
      10 /

      $0 0 = ?
        !@
      .
    .
    ^

    @'
      ~8 0 ~32 >8
    .
    ^
  ~0
.

# Pushes one whole file range block represented as string to str1.
pushBlock:
  pushStr2
  SYMBOL_SPACE pushChar
  ~32 $pos1 pushNum ~0
  SYMBOL_SPACE pushChar
  ~32 $pos2 $pos1 - pushNum ~0
  SYMBOL_BLOCK_SEP pushChar
.

# Pushes str2 to str1.
pushStr2:
  ~8
    $str2>ptr

    1 @
      $ptr $>ptr
      $ptr
    .
  ~0
.

# Helper that shifts ptr and increments pos.
shiftPtr:
  ~8 $>ptr ~32 ++ ~0
.

# ret: 3n
# Seeks position of a file within the main str, returns its position.
seekFile:
  ~32
    1 seekPos
    1
  ~0

  ~8
    0 # comparing?

    @@
      $ptr !! ? ~32 ^ -1 ~8 !@ .

      ?' # comparing?
        $ptr2 !! $ptr 0 ~0 SYMBOL_SPACE >8 ~8 = & ?
          1 @ # go back to filename start
            ~32 -- ~8
            $<ptr
            $ptr 0 ~0 SYMBOL_SOF >8 ~8 !=
          .

          !@
        .

        $ptr $ptr2 != ?
          ^ 0
        .

        $>ptr2
      .

      $ptr 0 ~0 SYMBOL_SOF >8 ~8 = ?  # TODO: make fun
        ^ 1
        $str2>ptr2
      ;
      $ptr 0 ~0 SYMBOL_SPACE >8 ~8 = ?
        ^ 0
      . .

      shiftPtr
    .

    ^
  ~0
.

# art: 3pos
# ret:
# Seeks ptr to given position within str1.
seekPos:
  ~8 $initAddr>ptr ~0

  ~32
    @' # go to the passed position
      ~8 $>ptr ~32
      --
    .
    ^
  ~0
.

# arg: 3strPos
# Prints all blocks of a file, potentially recursively printing included files.
printFile:
  ~32 $0 seekPos ~0 # we'll keep the position below, for recursion

  0 ~8 $ptr >0 ~0 SYMBOL_SOF_DONE = ? # file already included?
    ~32 ^ ~0
    !.
  .

  ~8 0 ~0 SYMBOL_SOF_DONE >8 ~8 $:ptr ~0 # mark done

  shiftPtr

  ~8
    @@ # go char by char
      $ptr 0 ~0 SYMBOL_EOF >8 ~8 = ? !@ .

      $ptr 0 ~0 SYMBOL_INCLUDE >8 ~8 = ?
        # load the included file name into str2
        $str2>ptr2

        @@ # copy the filename to str2
          shiftPtr
          $ptr 0 ~0 SYMBOL_BLOCK_SEP >8 ~8 = ?
            !@
          .

          $ptr $:ptr2
          $>ptr2
        .

        0 $:ptr2

        seekFile
        10 ->

        ~32
          $0 -1 != ?
            printFile
          ;
            ^
          .
        ~8

        ~32 $0 seekPos ~8
      .

      $ptr 0 ~0 SYMBOL_BLOCK_SEP >8 ~8 != ?
        $ptr 0 ~0 SYMBOL_SPACE >8 ~8 != $ptr " " ?? ->
      .

      shiftPtr
    .
  ~0

  ~32 ^ ~0

  10 ->
.

1 @ # read all input
  <-
  $state STATE_FILENAME = ?
    $0 10 = ? # newline?
      ^

      SYMBOL_SOF pushChar
      STATE_CONTENT $:state

      ~32
        0 $:pos1
        0 $:pos2
      ~0
    ;
      pushChar ~8 $:ptr ~0
    .

    ~8 $>ptr 0 $:ptr ~0
  ; # STATE_CONTENT*
    ~32 $pos2 ++ $:pos2 ~0

    $0 0 = ?
      pushBlock
      SYMBOL_EOF pushChar
      ~8 $str2>ptr 0 $:ptr ~0
      STATE_FILENAME $:state
    ;
      $0 "#" = $state STATE_CONTENT = & ?
        STATE_CONTENT_COMMENT $:state
      ;
      $0 "#" = $1 10 = | $state STATE_CONTENT_COMMENT = & ?
        STATE_CONTENT $:state
      ;
      $0 34 = $state STATE_CONTENT = & ?
        STATE_CONTENT_STRING $:state
      ;
      $0 34 = $state STATE_CONTENT_STRING = & ?
        STATE_CONTENT $:state
      ;
      $0 "~" = $state STATE_CONTENT = & ?
        STATE_CONTENT_TILDE $:state
      ;
      $state STATE_CONTENT_TILDE = ?
        $0 34 = ?
          ~32 $pos2 2 - $:pos2 ~0 # don't include the two previous chars
          pushBlock
          ~32 $pos2 2 + $:pos2 ~0
          STATE_CONTENT_INCLUDE $:state
          SYMBOL_INCLUDE pushChar
        ;
          STATE_CONTENT $:state
        .
      ;
      $state STATE_CONTENT_INCLUDE = ?
        $0 34 != ?
          $0 pushChar
        ;
          ~32 $pos2 ++ $:pos1 ~0
          STATE_CONTENT $:state
          SYMBOL_BLOCK_SEP pushChar
        .
      . . . . . .
    .


    ^
  .

  <?
.

0 pushChar # terminate str1

~32 1 printFile ~0
```