Documentation for the Nasal Scripting Language

Philosopher & Hooray

September 21st, 2013

Contents

Prerequisites

This manual assumes familiarity with the C programming language and requires programming experience, including advanced concepts such as pointers and bit manipulation, data structures, memory management and (possibly) the ability to build programs from source. If you need to brush up on your C knowledge, we recommend checking out some of the following resources first, some of which may only take a couple of minutes to review/work through to get up to speed:

1. `http://www.slideshare.net/petdance/just-enough-c-for-open-source-programmers`
2. `http://www.slideshare.net/amraldo/introduction-to-c-programming-7898353`
3. `http://www.slideshare.net/olvemaudal/deep-c`
4. `http://mindview.net/CDs/ThinkingInC/beta3`
5. `http://www.physics.drexel.edu/courses/Comp_Phys/General/C_basics/`
6. `http://www.learnconline.com/2010/03/introduction.html`
7. `http://publications.gbdirect.co.uk/c_book/`
8. `http://c-faq.com/index.html`

For web-based programming experiments, you may want to check out some online compilers, such as:

1. `http://www.learn-c.org/`
2. `http://codepad.org/`
3. `http://ideone.com`

At least basic knowledge of Nasal (or a similar language like JavaScript) will be helpful, i.e. understanding the basic syntax (for the parsing section) will go a long way, for example:

```
# declare a function with a named argument
var hello = func(name) {
  print("Hello ", name);
}
hello("world"); # call the function with one argument
```

If you are connected to the internet, you can learn more about the Nasal language itself by visiting these links:

1. `http://plausible.org/nasal/sample.nas` (example of syntax and usage)
2. `http://plausible.org/nasal/lib.html` (documentation of library functions)

3. `http://wiki.flightgear.org/Nasal` – see some more specific articles like these to get started:

    i. `http://wiki.flightgear.org/Nasal_Conditionals`

    ii. `http://wiki.flightgear.org/Nasal_Variables`

    iii. `http://wiki.flightgear.org/Nasal_Loops`

    iv. `http://wiki.flightgear.org/Nasal_Operators`

1 Introduction to Nasal

Nasal is an Open-Source, Mult-Platform, Small, and Easily Embeddable Scripting Language designed by Andy Ross and released under the GNU LGPL license. It originally started out as "NASL" (acronym for Not Another Scripting Language) but was renamed due to a naming conflict with another, unrelated scripting language. Nasal was specifically designed to be used as an extension language in other programs and to be embedded by developers into their own programs, without any platform bloat. It supports various programming paradigms, such as functional programming, procedural, and object oriented programming (OOP), and runs on both 32bit and 64 bit platforms. It is written in ANSI C99 and its source code is small, compact, and completely self-contained, with a handwritten lexer and parser and no external dependencies, other than the standard C library (though it has several library bindings that can be optionally compiled). It's main usage has been in FlightGear (where it got first added in 2003), an open-source flight simulator, and in AlgoScore, an algorithmic music composition software, where Nasal is used to write the algorithms. Nasal can also be run standalone, through a command-line "interactive" REPL (read-eval-print-loop) interpreter using the GNU readline library.

Nasal is available from several different sources. Ross has a repository on GitHub (`https://github.com/andyross/nasal`), but he has said that he considers the SimGear sources to be the standard version (`http://gitorious.org/fg/simgear/trees/next/simgear/nasal` – SimGear is a component in FlightGear, it currently does not include a standalone interpreter). AlgoScore also has Nasal in it's source tree (`http://svn.gna.org/viewcvs/algoscore/trunk/src/nasal/`), but it has not been updated in a while and SimGear's is more actively maintained. Most source code references made here are referring to Ross's GitHub repository for now.

Nasal combines the advantages of a dynamically-typed scripting language with a clean syntax close to C and Javascript – Nasal being inspired by ECMAScript – and it has many modern features like multiple assignment expressions and named function arguments, as well as unique ones like access to namespaces in the native "hash" datatype. It supports object-oriented programming through hashes that can contain a mutable "parents" vector used in implementing multiple-inheritance. It has automatic memory management through a garbage collector (GC) and is designed to be minimally threadsafe, without requiring a global interpreter lock (GIL). Nasal as a language is very flexible, and both its integration into FlightGear and the variety of libraries that have been created in Nasal attest to this (e.g. see *FG*/src/Scripting/FGNasalSys.cxx and the *Andy*/lib/ and *Andy*/contrib/Ampere/ directories for an illustration of this flexibility).

For more on Nasal syntax and features, see `http://plausible.org/nasal` and `http://wiki.flightgear.org/Nasal_scripting_language`. Reading Andy's design document is also recommended, as many of the things that are alluded to will be explained here: `http://plausible.org/nasal/doc.html`.

2 Maintaining this Manual

This manual is very far from being complete, or up-to-date. If you want to help out by contributing, please contact Philosopher or Hooray (the authors) on the forum. It is written in LaTeX but shouldn't be difficult to pick up on sight. Both grammar/wording edits and contributions/revisions of topics will be welcome!

This manual is hosted on both writeLaTeX.com and in FGData.

3 Document layout

This document is focused on the C/C++ sides of working with Nasal. There are many great tutorials out there for Nasal programming itself, please see the links under "Prerequisites".

First described are some of the building blocks of the source code – basic things that are used throughout and need to be understood. Next the types of variables and their usage are detailed – manuipulating variables is essential to creating a usable API. From those building blocks, methods for creating an API are described for those needing to integrate Nasal into a existing program. CPPBind – an extension found in SimGear – is the focus of the next chapter: it allows easy manipulation of Nasal types from C++, including automatic conversions using templates.

For those who are interested, there are many parts describing different core pieces of nasal. Steps of parsing: lexing, block & precedence parsing, and code generation. Code running in the virtual machine – this provides insight into many of the actual workings of the Nasal code that programmers write. There's also an article on Nasal's garbage collection and some of the problems FlightGear has faced regarding that. Efforts to improve the GC are welcome! Error and exception handling is the last topic, including how to recognize and handle errors while programming in C.

Cool examples of work done on improving Nasal or that exemplifies the concepts described above are found at the end of the document. This includes ongoing work on "introspection" (enhancing the ability of Nasal to debug itself and complementing the existing metaprogramming capabilities), potential

"Nasal maintenance" tasks, and descriptions of the existing bindings for PCRE, SQLite, Cairo/GTK, FlightGear's Canvas and more.

Part I
Creating an API: useful knowledge

4 Globals and Contexts

A Nasal context is basically the "housekeeper" struct that manages data for a single instance of an interpreter. *All* interpreter-specific stuff is managed via the context, which is why you have pointers and data such as:

1. the execution-frame stack, its current top
2. the operand stack
3. various GC related data structures (marking, free pool memory elements)
4. error handling (via setjmp/longjmp/jmp_buf)
5. support for sub contexts, i.e. Nasal scripts calling other scripts, calling other scripts via naCall() etc

Look at some of the callbacks - the only way to know what data to operate on is by having a handle to the context, which in turn contains handles to the relevant program (opcodes), operand stack, frames, GC stuff and so on. Each variable that gets created via naNew*() needs to be tied to a context for GC'ing.

All extension functions operate via naContexts – so the context is basically the "instance" of the interpreter. Also, if we have multiple contexts around, for example using threading, naContexts are a way to keep things separate, even though we may have multiple threads running their own Nasal interpreter instance.

4.1 Using Nasal Sub Contexts

Subcontexts are needed whenever an extension function is called which in turn invokes another Nasal function as a callback. The most common use in Nasal is using the `call()` library function in code.c to explicitly call a function, while specifying the complete environment. Using the `call()` function is also the only way to do exception handling in Nasal.

Additional examples on subcontext API usage can be found in the sort() implementation (sort_cmp in lib.c), where the extension function executes a callback that is passed via a naRef argument. Another example is to be found in the readline bindings (readline.c), where the completion function can be provided as a Nasal callback. Also, the SQLite bindings (see sqlitelib.c) provide another example on using Nasal sub contexts, where the `run_query()` extension function provides support for running a Nasal-space callback.

Sub contexts are usually used to issue a `naCall()` from inside a Nasal extension function. After use, sub contexts should be freed using `naFreeContext()`.

5 Variables

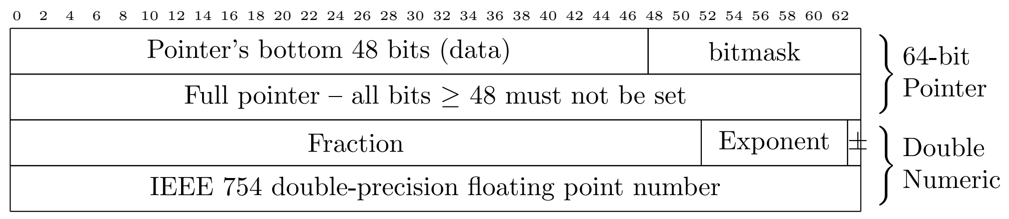
The main object type in Nasal is `naRef` and it either holds a reference to a Nasal object or it stores a number. A `naRef` is a union between a double-precision floating-point number and a pointer to another data type. Designating it as a pointer is done by setting a bitmask in the top bits of the double, which includes the whole exponent, sign, and a couple more bits in the mantissa. Setting the bits in this way makes the double a NaN (Not a Number, an invalid number) that will usually cause a floating-point error if used, since it is a singalling NaN (SNaN) versus a quiet NaN (QNaN). Depending on the system configuration (processor and OS), setting the bits has to be accomplished in different ways. For supported 64-bit systems/OS combinations, where all of the data in the pointer lies in the bottom 48 bits, the bits above that are set with a bitmask simple. For 32-bit systems, the layout in endianness-dependent (since it has to be in the "top" bits) and a magic number (see the `REFMAGIC` macro) is stored in the top bits, again making it a NaN:

```c
/* On supported 64 bit platforms (those where all memory returned from
 * naAlloc() is guaranteed to lie between 0 and 2^48-1) we union the
 * double with the pointer, and use fancy tricks (see data.h) to make
 * sure all pointers are stored as NaNs. 32 bit layouts (and 64 bit
 * platforms where we haven't tested the trick above) need
 * endianness-dependent ordering to make sure that the reftag lies in
 * the top bits of the double */

#if defined(NASAL_LE)
typedef struct { naPtr ptr; int reftag; } naRefPart;
#elif defined(NASAL_BE)
typedef struct { int reftag; naPtr ptr; } naRefPart;
#endif

#if defined(NASAL_NAN64)
typedef union { double num; void* ptr; } naRef;
#else
typedef union { double num; naRefPart ref; } naRef;
#endif
```

If that is a bit confusing to understand, consider the following picture (for 64-bit systems):



With this layout, we have a pointer that has 48 bits of data in its lower half overlaid on top of a double whose exponent, sign, and some of the mantissa are in the top 16 bits. Once these bits are set with a bitmask, the double becomes a NaN. For more on this, see data.h and naref.h.

The pointers to the other types are stored in a `naPtr` union that contains all the Nasal types, some of which, like `naStr` and `naHash`, can be used from Nasal; others, like `naObj` and `naCode`, cannot be directly accessed from Nasal, but are instead used by the underlying C code.

```c
typedef union {
    struct naObj* obj;
    struct naStr* str;
    struct naVec* vec;
    struct naHash* hash;
    struct naCode* code;
    struct naFunc* func;
    struct naCCode* ccode;
    struct naGhost* ghost;
} naPtr;
```

These `naRefs` are the fundamental 'variable' structure of Nasal, and the closer one gets to the actual running of Nasal code, the more `naRefs` are used (e.g. from codegen.c to code.c, which is going from barely any `naRefs` to essentially as rich with `naRefs` as it gets).

An interesting thing to note is that some of the attributes of private, internal objects, like a function, are implemented using `naRefs` instead of their "raw" counterparts, like `naFunc`*. This is probably because all the methods that work with the types, e.g. `naHash_get`, are implemented with `naRefs` (and those use `PTR(foo).hash`, etc., to fetch the raw data type), and so it is often much more convenient to use those methods since there are no "raw" equivalents to use. It also could be for consistency, i.e. a `naRef` is always going to be a fixed size independent of application (it is, after all, just a pointer/double combination). Sometimes this leads to assumptions about the type of a member/variable. This document uses the convention of `naRef:`⟨`type`⟩ to denote assumptions about type, like `naRef:naHash` for hashes, `naRef:naCode/naCCode` for executable code, and `naRef:double` for numbers. Violating these assumptions may cause code to crash!

5.1 Scalars

Nasal tries to keep a notion of "scalars" as numbers or strings, like Perl – in fact, this is what the builtin `typeof()` function will return instead of "number" or "string". While numbers and strings are still separate types behind the scenes, Nasal will try to automatically convert from one to the other when running code, e.g. making a number into a string when concatenating it, or vice-versa for mathematical operators. Most of the Nasal extension function libraries will convert between them as well, but be warned that some do not! Those that do not use simple checks like `IS_STR()` and `IS_NUM()` instead of converting first using `naStringValue()` and `naNumValue()`; this means that only if a value literally represents a string or number will it be used as such. Simple ways to convert on the Nasal side are using concatenation with an empty string and addition with 0 or using `num()`.

5.2 Types

Here's a reference to all of the raw types used in Nasal:

5.2.1 Strings/T_STR/`naStr`

The `naStr` struct stores a string's length and either its data (if it is less than 16 characters) or a pointer to its data. Unlike C, which uses null-termination to determine the length of a string, Nasal explicitly stores the length of a string so that it can contain any characters whatsoever – though it is null-terminated as well, since C code often needs to deal with it like it actually was a C `char`* array. Note that when using `naStr_fromdata`, only a copy of the data is stored, so that Nasal code can mutate the string without affecting the original bytes.

By default strings are mutable, including results of concatenation or `substr()`. When used as a key in a hash, a particular string has its "hashcode" member set to non-zero, making it immutable. Using this method, constant strings in the code (literals enclosed in double- or single-quotes) are made immutable.

API functions:

```
// String utilities:
int naStr_len(naRef s); // get the length of a string
char* naStr_data(naRef s); // pointer to null-terminated data
naRef naStr_fromdata(naRef dst, const char* data, int len); // set a string's data and length
naRef naStr_concat(naRef dest, naRef s1, naRef s2); // concatenate two strings
naRef naStr_substr(naRef dest, naRef str, int start, int len); // take a subscript of a string
```

5.2.2 Vectors/T_VEC/`naVec`

Vectors are resizeable arrays with automatic reallocation, using a standard multiply-by-two rule for expanding. In comparison with bigger languages like Python, Nasal's vector operations are rather basic, but they do provide enough functionality for most uses. Most extensions can be scripted using Nasal "classes" or optimized as C code.

API functions:

```
// Vector utilities
```

```
int naVec_size(naRef v); // get the size of a vector
naRef naVec_get(naRef v, int i); // get a specific element by index
void naVec_set(naRef vec, int i, naRef o); // set a specific element by index
int naVec_append(naRef vec, naRef o); // append to a vector
naRef naVec_removelast(naRef vec); // remove the last element
naRef naVec_removefirst(naRef vec); // remove the first element − only in SimGear!
void naVec_setsize(naRef vec, int sz); // set the size of a vector, padding with naNil()
```

5.2.3 Hashes/T_HASH/naHash

Hashes are the unordered, associative container type in Nasal (no fancy name like "dictionary"). Each string used as a hash key is made immutable via having its "hashcode" set – this is simply storage for a special number used in looking up and setting strings.

API functions:

```
// Hash utilities:
int naHash_size(naRef h); // get the size of a hash − number of elements
int naHash_get(naRef hash, naRef key, naRef* out); // get a key − returns 1 if it is found
naRef naHash_cget(naRef hash, char* key); // get a key using char* − DEPRECATED!
void naHash_set(naRef hash, naRef key, naRef val); // set a key to a value
void naHash_cset(naRef hash, char* key, naRef val); // set a key using char* − DEPRECATED↩
    !
void naHash_delete(naRef hash, naRef key); // delete a key from a hash
void naHash_keys(naRef dst, naRef hash); // get a vector of all keys in the hash
```

5.2.4 Functions/T_FUNC/naFunc

A Nasal function is essentially a wrapper for executable objects, which can be naCodes (written in Nasal) and naCCodes (written in C, i.e. extension functions). This allows one common type for executing both of these. It is also the only one of these three types which should appear in Nasal space: the other two are more 'raw' and need to be wrapped and/or bound to a context before they are exposed. Each naFunc stores three naRefs: *code*, *namespace*, and *next*. *code* is the naRef:naCode/naCCode that should get executed, *namespace* is the parent namespace of the function (i.e. the first level of recursion after the locals), and *next* is the naRef:naFunc:naCode that is the next link in the chain of namespaces. Note that the last two members are only applicable to naCodes since naCCodes do not have their own Nasal-space symbols.

There is no API for functions.

5.2.5 Nasal Fuctional Code/T_CODE/naCode

This carries the data behind either a Nasal func{} expression or a complete parsed file/script. In either case, these objects are generated during parsing, in particular the "code generation" stage, and returned by naParseCode. They consist of a *constants* block – basically a static heap of allocated storage. This contains naRef constants, both scalars and func{} constructs, the bytecode of the function, and info on line numbers versus opcode indices. These are accessed separately using macros in data.h, in particular since the constants are `naRefs` while the others are `unsigned shorts`.

There is no API for code objects.

5.2.6 Ghosts/T_GHOST/`naGhost`

Garbage Collected Handle to Outside Things, i.e. raw C pointers wrapped in a `naGhost` struct and managed via the GC, using explicitly registered allocation/release functions for each ghost type.

API functions:

```
// Ghost utilities:
typedef struct naGhostType {
    void(*destroy)(void*);
    const char* name;
} naGhostType;
naRef naNewGhost(naContext c, naGhostType* t, void* ghost);
naGhostType* naGhost_type(naRef ghost);
void* naGhost_ptr(naRef ghost);
int naIsGhost(naRef r);
```

5.2.7 Extension functions/`naCFunction`/`naCCode`

Nasal Extension functions are usually functions with static linkage written in C and callable from Nasal. They can, of course, work with C variables, like from a pointer stored in a Nasal ghost variable, from a C library, or anything else. They can also perform operations on Nasal variables through the Nasal API (see nasal.h and the next section) – essentially anything that can be done from Nasal, and in a potentially more efficient manner because it is compiled down to native machine code. Each of these extension functions must match this signature:

```
// The function signature for an extension function:
typedef naRef (*naCFunction)(naContext ctx, naRef me, int argc, naRef* args);
```

This uses a scheme like the `main()` function where there is *argc* – the number of arguments – and *args* – a pointer to the actual arguments, which are `naRefs` in this case. The *me* reference is like “self” in general Python usage or “this” in C++, and is set to the value of left-hand side of an expression if it is a method call (e.g. the value of foo in `foo.bar()`) or is nil otherwise (e.g. even in `foo['bar']()`, or just `fn()`). (Note that this would primarily be used for methods operating on ghost objects.) The context is of course the currently executing context, which is used to bind new `naRefs` to, etc.

```
// Extension functions *must* be static and convention is to use names that start with 'f_'
static naRef f_demo(naContext ctx, naRef me, int argc, naRef* args)
{
    // Write your C code here
    return naNil();
}
```

6 The Main Nasal API in C

All of the C API is included in the nasal.h file, which in turn gets included by every module or application wanting to use Nasal. It includes the type-specific manipulation functions shown above along with all of the types (`naRef`, `naPtr`, etc.) and other parts of the API.

7 How to Create an API

8 Using CPPBind

Also see: `http://wiki.flightgear.org/Nasal/CppBind`.

C++Bind is a really cool API for interfacing with Nasal through C++ classes, methods, and advanced STL functionality.

Part II
Internals: Parsing through Running

9 Overview of Parsing

"Parsing" a file involves a many-tiered hierarchy of transformations that results in a piece of executable code stored in a newly-allocated `naCode` object. This is essentially what a Nasal func{⟨*contents of file*⟩} expression would do, but of course it is through parsing files that function expressions can be generated, not the other way around.

Parsing of a script starts out in parse.c, the Nasal parser, which initializes a new parse object (`struct Parser p`) and then runs the string through the lexer, via `naLex()` (lex.c, line 338), generating list of "tokens" that represent each lexeme (builtin Nasal keyword/operator), symbol, and constant. This then is returned to parse.c, which runs a block parser over the tokens, adding tree structure "up" and "down". After this the precedence,,,,,, parser goes throuhin blocks and transforms them into a structured representation of operations. This list, though still made of tokens, represents a more abstract view of the expression that is being parsed, but in a form that is easier for the code generation to understand due to the fact that operators are tightly connectected with their often complex operands. Finally, the expression list is given to `naCodeGen()` (codegen.c, line 702), which takes the tokens and creates opcodes – low-level instructions that will later be executed by the Nasal Virtual Machine each time the function is called. The code object (as returned by `naCodeGen`) is then returned from `naParseCode()`, to be saved to a `naCode` object or, by extension, a `naRef` referencing such.

9.1 Lexing

"Lexing" is the process of converting a stream of characters (which is hard for a computer to directly read) to a series of objects (`structs` in C) that are easier for a computer understand. In Nasal, a string is made into a linked list of `struct Tokens`, each of which can represent either a "lexeme" (a symbol that has special meaning in Nasal, like '+=') or some other type of token, like a string or numeric literal, or a symbol. Each `Token` can store a string and its length or a double in addition to its type (TOK_LITERAL, etc.) and its line number.

9.2 Block and Precedence Parser

[Note: for this section, I can do a good job of explaining how the algorithms work in an abstract manner, but if you want to understand how the C functions fit together to do this work or learn what exactly is done, then I recommend reading through the relevant functions in parse.c (precChildren, precBlock, and parsePrecedence); it really isn't a difficult task and will set you up for being able to understand the algorithm and modify it —end note]

The first job that parse.c does is create the tree of tokens with regard to adding "blocks". Some tokens open a block and have a matching ending token; namely parentheses ('(' and ')'), brackets ('[' and ']'), and curly braces ('{' and '}'). When the block parser comes upon an opening token, it parses succesive tokens into children of the opening one until it finds an ending one. If another opening brace is encountered, another block parser parses that and gets to use up the tokens in the stream until its block has ended, which then returns control back to the previous block parser at the token after the ending one. A parse error occurs if a closing brace is encountered that does not match the token that started the block or if the parse finds the end of the file without encountering a suitable ending token. Aside from matching pairs, blockoids (if/elsif/else, for/while, func, etc.) are also parsed at this stage. Parts of the blockoid structure (parethesis as "arguments" to the loopoid, braces or braceless bodies, and extra else/elsif's after an if) get put as children of the blockoid token. Parethesis and braces obviously get parsed by the previously described system, and a braceless body gets parsed by a special one which looks for a semi-colon, miscellaneous end-of-block (like an extra closing parenthesis or even an else token), or end of the file. The middle condition allows the end-of-block to be parsed at a higher level. Consider this Nasal statement:

```
setlistener("/sim/foo", func print("activated"));
```

The function body parser reads "print", then a matching pair of parentheses, and finds a closing parenthesis. This parenthesis looks out of place, but it actually belongs to the token after setlistener, so to allow the parse to succeed the extra parenthesis must be declared as one-past-end-of-block and be the next token the parent parser sees – which will successfully end the block since they match.

Anyways, a blockoid construct ends up with potentially three children: a parenthesized argument list, a body with braces or without (braceless bodies end up as children of a TOK_BRACE, so the body always has braces going into the next phase of parsing), and if the blockoid is an "if" then it can have an else or elsif as its last child.

The next part of parsing is one of the most fun ones. The precedence parser creates a binary tree out of the blocks of tokens that were previously generated. There are three transformations that this parser can handle: PREC_BINARY/PREC_REVERSE, PREC_PREFIX/PREC_SUFFIX, and no precedence (single token). PREC_BINARY and PREC_REVERSE are opposites; same with PREC_PREFIX and PREC_SUFFIX. The main point of precedence parsing is to break a long stream of tokens into a binary tree with a large expression formed by an operator and optional expressions on either side. Each unary or binary operator is given a precedence label at the top of parse.c. That table in order of parse-first–parse-last is as follows:

Precedence:	**Token(s):**		
PREC_REVERSE	TOK_SEMI	TOK_COMMA	
PREC_SUFFIX	TOK_ELLIPSIS		
PREC_PREFIX	TOK_RETURN	TOK_BREAK	TOK_CONTINUE
PREC_REVERSE	TOK_ASSIGN TOK_MULEQ	TOK_PLUSEQ TOK_DIVEQ	TOK_MINUSEQ TOK_CATEQ
PREC_REVERSE	TOK_COLON	TOK_QUESTION	
PREC_PREFIX	TOK_VAR		
PREC_BINARY	TOK_OR		
PREC_BINARY	TOK_AND		
PREC_BINARY	TOK_EQ	TOK_NEQ	
PREC_BINARY	TOK_LT TOK_GTE	TOK_LTE	TOK_GT
PREC_BINARY	TOK_PLUS	TOK_MINUS	TOK_CAT
PREC_BINARY	TOK_MUL	TOK_DIV	
PREC_PREFIX	TOK_MINUS	TOK_NEG	TOK_NOT
PREC_SUFFIX	TOK_LPAR	TOK_LBRA	
PREC_BINARY	TOK_DOT		

When the precedence parser gets run, it starts at the top row, and if a condition is fullfilled, then the linked-list of Tokens is split into top (the matching token), left, and right (which are added as the binary children of top, or 0 if there is none to add). The left and right tokens are binary trees formed by extra calls to the precedence parser, with four different rules according the the precedence type. The prefix/suffix operators are simple: they are only recognized at the beggining (if prefix) or end (if suffix) of the list. The tokens from the one inside of the matching token to the end of the opposite side are parsed as either the left or right child (this is right for prefix, left for suffix). Any children are precedence parsed as well and added on the other side, or 0 if there are no children. (Note: children are only going to be present on parentheses or brackets in the role of a suffix operator.)

Binary and reverse operators are a bit different: they can occur anywhere *but* the endcaps (not the start token, not the end one, but anywhere in between). For binary operators, the rightmost one is grabbed, and vice-versa for reverse operators. (The way I remember this is that the statement separator (;) is a reverse, but the top of the list of them is going to be the leftmost ones; thus binary operators must have the top be the rightmost one, opposite of reading order.) All the tokens from the start to one left of the operator is parsed as the left child, and similarly for the right child. One side is parsed at the same precedence (because there might be more of the same precedence level that lie undiscovered) and the other is parsed at the next precedence level.

If the parser reaches a state where the end and start of the list are the same, then that token is added to the tree as either a leaf (if it has no children) or another branch (with both left and right pointers pointing to the top of the precedence-parsed tree formed from its children). The end result of this sorting is a tree of tokens, where each may have 0, 1, 2, or duple children (duple meaning that both children are the same, which is true in the last case we discussed). This tree for an example file looks something like this:

```
var x = 1.5 - 0.5;
```

```
// Notation:
// Prefix Name = value (Precedence)
// Prefix:
// l Left child
// r Right child
// b Both children (left == right)
// . (Spacer)
// Precedence: what rule was used to parse

TOP (Synthetic)
b SEMI (Reverse)
. l ASSIGN (Binary)
. . l VAR (Prefix)
. . . r SYMBOL = x (None)
. . r MINUS (Binary)
. . . l LITERAL = 1.5 (None)
. . . r LITERAL = 0.5 (None)
. r EMPTY (Synthetic)
```

10 Code Generation – producing bytecode

Code generation is the next step of parsing and is implemented in the file code-gen.c. It takes the expression tree generated by the lexer and parse.c recursively traverses it to create opcodes – little "numbers" (or an `enum`) that are interpreted by the Nasal Virtual Machine as instructions. With the exception of blockoids (which have a slightly different structure with up to three children), the token structure that code generation gets is a tree where the most important (highest precedence) tokens are on top, and tokens of lesser- and equal-importance tokens on the sides; laws of grouping are followed as well (like parenthesis and such) by having both left and right children point to the top of the tree for the expression inside of the block.

The example above shows what is really neat about this approach: when it is parsing, say, an assignment operator (=), it knows that the left side is going to be a symbol and that the right side can be any type of expression. Seeing it this way makes much more sense then first seeing the symbol, looking ahead for an assignment, and then hoping the rest of the tokens parse correctly – that would be a more text-based approach and it is not scalable for complicated syntaxes with oodles of operators with different semantics.

The method used in parse.c allows for the minimal-step, maximally-recursive code generator present in Nasal: parsing a mathematical operator is as simple as generating the left and right expressions, and then emitting the correct VM instruction afterwards – the function handling the mathematical operator does not need to know what lives below the operator, only that it exists. Each step of the generator (implemented in the `genExpr` function and surrounding static functions) takes the smallest step it can and lets the parse tree work itself out by calling helper functions to handle expressions and expression lists.

Local macros helping with abstraction include:

1. `LEFT(tok)` to get the left-hand branch of a token (i.e. `tok->children`).
2. `RIGHT(tok)` to get the right hand branch of a token (i.e. `tok->lastChild`).
3. `BINARY(tok)` to check if a token is a binary expression (if there is only two children: `tok->children` and `tok->lastChild`, and nothing in between, meaning that they reference each other in their *next/prev* members).

The low-level workhorses of the code generator are the `emit()` and `emit Immediate()` functions, which directly write bytecode into the parser structure's codegen pointer.

```c
static void emit(struct Parser* p, int val)
{
    // Reallocate if necessary
    if(p->cg->codesz >= p->cg->codeAlloced) {
        int i, sz = p->cg->codeAlloced * 2;
        unsigned short* buf = naParseAlloc(p, sz*sizeof(unsigned short));
        for(i=0; i<p->cg->codeAlloced; i++) buf[i] = p->cg->byteCode[i];
        p->cg->byteCode = buf;
        p->cg->codeAlloced = sz;
    }
    // And add our next opcode
    p->cg->byteCode[p->cg->codesz++] = (unsigned short)val;
}
```

```
static void emitImmediate(struct Parser* p, int val, int arg)
{
    emit(p, val);
    emit(p, arg);
}
```

This "immediate" value is associated with the opcode and retrieved later in the VM stage. In general it is just an `unsigned short`, but it can mean several things.

Some immediate-mode opcodes refer to the constants table – this is a chunk in the `naCode` which holds scalar constants (string/numeric literals) and code constants (func{} blocks in the code). These opcodes always need the constants index added when `emitImmediate()` is called, using either `findConstantIndex()` or a precomputed index, such as returned from `newConstant()`. Of this type, there are only these three:

1. OP_PUSHCONST
2. OP_MEMBER
3. OP_LOCAL

In addition, these two use their immediate value to represent their number of arguments; this is because the VM needs to be able to pop the right amount of arguments from the operand stack:

1. OP_FCALL
2. OP_MCALL

And lastly, branching instructions are also immediate-mode; their immediate value refers to the index of the command to jump to. The specific opcodes are:

1. OP_JIFTRUE
2. OP_JIFNOT
3. OP_JIFNOTPOP
4. OP_JIFEND
5. OP_JMP
6. OP_JMPLOOP

When jumping backwards, the address is saved away earlier and emmitted with the OP_JMP instruction, but when jumping forwards, the code it is jumping to does not exist yet and therefore its index is not known. To get around that, there are some functions to handle forward-jumps. `emitJump` makes a new jump with a dummy index:

```c
// Emit a jump operation, and return the location of the address in
// the bytecode for future fixup in fixJumpTarget
static int emitJump(struct Parser* p, int op)
{
    int ip;
    emit(p, op);
    ip = p->cg->codesz;
    emit(p, 0xffff); // dummy address
    return ip;
}
```

As suggested by the comment, `fixJumpTarget` is what comes along at the desired jump-to point and adds in the correct index:

```c
// Points a previous jump instruction at the current "end-of-bytecode"
static void fixJumpTarget(struct Parser* p, int spot)
{
    p->cg->byteCode[spot] = p->cg->codesz;
}
```

For typical usage, here's a stripped down `genIf` example:

```c
static void genIf(struct Parser* p, struct Token* tif, struct Token* telse)
{
    int jumpNext, jumpEnd;
    genExpr(p, tif->children); // the test
    jumpNext = emitJump(p, OP_JIFNOTPOP);
    genExprList(p, tif->children->next->children); // the body
    jumpEnd = emitJump(p, OP_JMP);
    fixJumpTarget(p, jumpNext);
    if(telse) {
        if(telse->type == TOK_ELSIF) genIf(p, telse, telse->next);
        else genExprList(p, telse->children->children);
    } else {
        emit(p, OP_PUSHNIL);
    }
    fixJumpTarget(p, jumpEnd);
}
```

The first jump points to right after the second jump, since `fixJumpTarget` is called after `emitJump` is called the second time. After the else{} clause is handled, the second jump is made to point after all of that. The power in this is allowing arbitrary chunks of code (e.g. as emitted from `genExprList`) to be jumped over.

10.1 Initialization

Code generation itself is started in `naCodeGen()` where the code generation related data structures are allocated and initialized, including:

1. an `naCode`* code object
2. argument list processing

Afterwards, `genExprList()` is invoked, which in turn recursively invokes itself and `genExpr()` while traversing the semi-colon-seperated statements in the parse tree, the latter is the primary workhorse of the code generator in that it will recognize and turn tokens into bytecode primitives.

10.2 Constants Table

This stores constants used in a function's body, both scalars and code. Implemented during codegen as a `naVec` for the purposes of cheap alloc handling, this gets turned into a static table as part of the `naCode` structure. `internConstant()` is used to return a reference to the existing constant if one already exists; it relies on `newConstant()` to add the constant if needed. Code for both (in reverse order of declaration):

```c
// Interns a scalar (!) constant and returns its index
static int internConstant(struct Parser* p, naRef c)
{
    int i, n = naVec_size(p->cg->consts);
    if(IS_CODE(c)) return newConstant(p, c);
    for(i=0; i<n; i++) {
        naRef b = naVec_get(p->cg->consts, i);
        if(IS_NUM(b) && IS_NUM(c) && b.num == c.num) return i;
        else if(IS_NIL(b) && IS_NIL(c)) return i;
        else if(naStrEqual(b, c)) return i;
    }
    return newConstant(p, c);
}

static int newConstant(struct Parser* p, naRef c)
{
    int i;
    naVec_append(p->cg->consts, c);
    i = naVec_size(p->cg->consts) - 1;
    if(i > 0xffff) naParseError(p, "too many constants in code block", 0);
   return i;
}
```

11 The Code Generation API

12 Code Generation Examples

13 Virtual Machine: running code

As mentioned before, the Nasal bytecode gets executed in code.c using an infinite interpreter loop with a huge switch/case block in the form of the `run()` function in code.c. This is the "Virtual Machine" (VM) of Nasal. In general, machines take some form of operations (e.g. Nasal bytecode) and execute them based upon other factors, like other inputs, and then produce an output of some kind. (A *virtual* machine is simply such a machine that is implemented in software not hardware – e.g. C/C++ code versus circuit boards). Some of the simplest machines are stack machines; this is what is used for the Nasal VM, stack machines are generally a popular concept, and e.g. also used by Java. The Nasal stack machine uses two main stacks and *pushes* and *pops* from those stacks. Each stack is an array of variables with a fixed size – exceeding this size is an error. The first stack is where all the opcodes are stored and it is for all purposes of type `unsigned short` or `int`, since each opcode is part of an enumeration. This also allows other integers to live on the stack as well (more on this later). The other stack is made up of `naRefs`, which are the "operands" that the opcodes operate on.

For most typical opcodes, the VM pops some arguments from the stack – that is, it takes the top item on the stack and saves it to a variable if needed, then it decrements the top of the stack so that the popped argument "disappears" from the stack. After performing the operation, the VM will push a result back onto the stack – this does the opposite of popping as it increments the top of the stack and sets the new top item. Thus pushing makes a "new" item on the top of the stack while popping takes one away. Eventually this change in state cascades down as a result of operations and a useful output is acheived. While the Nasal VM can do a fair amount of variable-handling operations on its own, `naCFunctions` can add more operations (e.g. `f_setsize()`) or can provide access to anything accessible from C space, like `f_print()` and the io library do.

These two stacks live in different places. The opcode stack lives in the `naCode` object under the constants block and is extracted using the `BYTECODE()` macro defined in data.h, which typecasts the `naRef` to an `unsigned short`* (see data.h, line 147). The "cursor" for the bytecode is stored in the current frame's *ip* member. A common operation is then `int op = BYTECODE(cd)[f->ip++];` (which is used in code.c to retrieve the opcode and/or an argument, implicitly typecasting to an integer). By this method, the opcode "stack" is not a "true" stack in the fact that it is immutable (should **not** be changed except using stack operations like push/pop!) and can only move one direction (adds to it during code generation, moves down it during the running of the code). The other stack – the one of `naRefs` – is actually a real stack. The items are stored in the context's *opStack* member, which has `MAX_STACK_DEPTH` number of terms, and its cursor is stored in the *opTop* member. Note that indexing *opStack* with the current *opTop* index is not a valid operation – the true top item on the stack is one below is at `STK(1)` or `ctx->opTop-1`. There are various helper macros to help out with operations on this stack, like `PUSH()` and `POP()` and in particular `STK(n)`, which directly retrieves an item on the stack (where $n = 1$ is the top member as previously mentioned, and $n = 2$ is the second, etc.). Note that the `STK()` macro requires manual movement of the *opTop* index, i.e. the programmer has to use `ctx->opTop--` or `ctx->opTop-=2`. For more info, see code.c, line 246 for `PUSH()`, and lines 520–523 for the other macros.

Also, there are two other relevant stacks. The first one is the stack of frames. Since a function can be called in an overlapping fashion (e.g. a function calling itself recursively, or being called in a multithreaded fashion from different threads), Nasal needs separate places to store information for each separate call. This stack is made up of `struct Frames` and lives in the *fStack* member, which has size of `MAX_RECURSION`, and its index is in the *fTop* member. New calls are pushed in at the *fTop* index and the current frame is `ctx->fStack[ctx->fTop-1]` while the first frame is `ctx->fStack[0]`.

13.1 An example

To take an example of the VM in action, let's say that we execute this Nasal statement:

```
var foo = 1.5−0.5;
```

This takes and sets the local variable "foo" with the result of 1.5 minus 0.5 (code generation does not optimize any). This would produce a stack of opcodes like this:

```
Stack of op codes:   Stack of arguments:
[start]              ...      ctx->opTop
OP_PUSHCONST                  [end]
00003
OP_PUSHCONST
00004
OP_MINUS
OP_PUSHCONST
00005
OP_SETLOCAL
...
OP_RETURN
```

The first two opcodes are OP_PUSHCONSTs, which take a constant and pushes it onto the (previosuly empty) stack at the top. This illustrates the fact that each function call starts with a clean stack of operands, and all constants have to be initialized. This particular one points to constant '3', which should hold an naRef with it's number set to 0.5. Then we push another number, which will be 1.5. Then we get this stack:

```
Stack of op codes:   Stack of arguments:
...                  ...      ctx->opTop
OP_MINUS             naRef  1.5   STK(1)
OP_PUSHCONST         naRef  0.5   STK(2)
00005                         [end]
OP_SETLOCAL
...
OP_RETURN
```

The first opcode is OP_MINUS, which takes the first argument (1.5) and subtracts the second (0.5) and places the result (1.0) where the second used to be, then decrements the top of the stack. The stack now looks like this:

```
Stack of op codes:   Stack of arguments:
...                  ...      ctx->opTop
OP_PUSHCONST         naRef  1.0   STK(1)
```

```
00004                         [end]
OP_SETLOCAL
...
OP_RETURN
```

Now we see OP_PUSHCONST and a number after it. This push operation takes an argument of type `int` from the stack of opcodes, not `naRefs`, and retrieves the constant `naRef` stored at the specified index in the constants block of the currently executing code using the `CONSTARG()` macro (code.c, line 520). During code generation, this constant is saved away inside of the code's constant block for each required lvalue, and this opcode pushes the correct one onto the stack – which should be a `naRef` representing "foo" and is pushed in before the result of the OP_MINUS, setting us up for the next step:

```
Stack of op codes:    Stack of arguments:
...                   ...       ctx->opTop
OP_SETLOCAL           naRef  foo    STK(1)
...                   naRef  1.0    STK(2)
OP_RETURN                       [end]
```

We finally get to OP_SETLOCAL, which sets a local variable (i.e. one using the 'var' keyword) to a value. It takes the first argument on the stack as the lvalue (symbolic name) of the variable and the second as the value to set it to. In this case, we see those are "foo" and 1, respectively, and so the variable foo correctly gets set to 1. Note that this operation does not explicitly "return" anything, i.e. it does not push a 'result', but it does have the implicit return of the value set, since that is the last argument left on the stack (since the OP_SETLOCAL operation only moves down 1 on the stack). This illustrates an interesting feature of Nasal: it always has its current working state and every operation returns *something*. We will see this in action in the next paragraph.

```
Stack of op codes:    Stack of arguments:
...                   ...       ctx->opTop
OP_RETURN             naRef  ??     STK(1)
```

We execute some more statements in between and eventually get to a return opcode – this is required otherwise the Nasal VM will get stuck in an infinite loop and possibly do some very weird things. Anyways, sometimes this return is implicit and is synthesized by the code generation, if so it returns the first argument on the stack – remember how we called it the "current working state"? An equivalent expression that makes this return explicit goes like so:

```
return (...);
```

Substitue the last expression you want to execute into the parentheses and you are set. A great mistake would be to say "Oh, I just need to type return-⟨semicolon⟩ and I'll be set." This is incorrect, since code generation makes an OP_PUSHNIL before any return statement like this:

```
return (...);
```

Thus it would really return nil, not the last expression that was evaluated. See line 631 of codegen.c for more information.

OP_RETURN
...
...
...
...

13.2 VM Operators/Opcodes

Here is a semi-comprehensive list of all the operators and their functions.

13.2.1 Unary operators

Operators that modify STK(1) in place:

OP_NOT Boolean inverse (i.e. `!a`, $\neg a$).

OP_NEG Additive inverse (i.e. `-a`, $-a$).

13.2.2 Binary operators

Operators that pop two arguments and push a result:

OP_PLUS Addition (i.e. `a+b`, $a + b$).

OP_MINUS Subtraction (i.e. `a-b`, $a - b$).

OP_MUL Multiplication (i.e. `a*b`, $a \times b$).

OP_DIV Division (i.e. `a/b`, $a \div b$).

OP_CAT Concatenate either two vectors or two strings (i.e. `a~b`).

OP_LT Strictly less-than (i.e. `a<b`, $a < b$).

OP_GT Strictly greater-than (i.e. `a>b`, $a > b$).

OP_LTE Less-than-or-equal-to (i.e. `a<=b`, $a \leq b$).

OP_GTE Greater-than-or-equal-to (i.e. `a>=b`, $a \geq b$).

OP_EQ Equality or equivalence (i.e. `a==b`, $a \equiv b$; tests for numerical esuality if possible and acts recursively on hashes and vectors besides comparing pointers).

OP_NEQ Non-equality/equivalence (i.e. `a!=b`, $\neg(a \equiv b)$).

13.2.3 Push and/or pop operations

Operations that push or pop a `naRef` directly from the stack:

OP_POP Pop one `naRef` off of the stack (actually, it just moves the top of the stack down one).

OP_DUP Push a duplicate of the top item.

OP_DUP2 Push duplicates of the top two items (e.g. a, b becomes a', b', a, b).

OP_PUSHONE Push a `naRef` '1'.

OP_PUSHZERO Push a `naRef` '0'.

OP_PUSHNIL Push a `naNil()`.

OP_PUSHEND Push an `endToken()`.

OP_PUSHCONST Take an integer argument from the **opcode** stack then retrieve and push the constant found at that index in the code's constants block. See the `CONSTARG()` macro, line 520 in code.c, for more details. Also note that if the constant is a `naCode` object (inside of its `naRef` shell), then it gets bound to the current context/frame via `bindFunction()` (which turns it into a `naFunc`).

OP_NEWVEC Push a new vector.

OP_NEWHASH Push a new hash.

13.2.4 Jump operations

Operations that jump within either stack:

OP_JMP Execute a non-conditional jump to the bytecode index specified by the next item on the opcode stack (i.e. an integer one).

OP_JMPLOOP Same, but execute `naCheckBottleneck()` first.

OP_JIFTRUE Jump if the top `naRef` on the stack is true. Does *not* pop from the stack!

OP_JIFNOT Same, but jumps if false.

OP_JIFNOTPOP Same as OP_JIFNOT but *does* pop from the stack.

OP_JIFEND Jumps if the top `naRef` is an `endToken()` (i.e. the pointer equals `(void*)1`). Pops only if it is an `endToken()`.

OP_MARK Pushes a mark (the current size of the `naRef` stack) onto the mark stack.

OP_UNMARK Pops off a mark and discards it.

OP_BREAK Takes and restores a mark position.

OP_BREAK2 Same, but pops it off instead of leaving it.

13.2.5 Function calls

These opcodes either call a `naCCode` or push another stack frame for a `naCode` call. The number of arguments is compiled during code generation and is thus fixed. By default, the top *nargs* on the stack are used as the arguments, another `naRef` is popped off as the function, and for method calls, another is popped off for the 'me' reference. For named arguments, there is only 1 argument that is taken from the stack, which becomes the *locals* hash for the new frame, otherwise the *locals* are made by the `setupArgs()` function.

OP_FCALL A basic function call.

OP_MCALL A method call – sets the 'me' symbol.

OP_FCALLH A function call with named arguments (the 'H' is probably for "hash-like").

OP_MCALLH A method call with named arguments.

13.2.6 Set methods

These methods set a variable and usually leave the value that was set on the stack, thus in most cases where you want to set something and don't care about the return, these should be followed by OP_POP.

OP_SETLOCAL Set the local variable named by STK(1) to the value given by STK(2).

OP_SETSYM Like OP_SETLOCAL but tries to find the variable in successive namespaces (i.e. each `func->next` in turn). If it is not found there, place it in the locals.

OP_SETMEMBER STK(1) is the name of the member, STK(2) is the hash, and STK(3) is the value.

OP_INSERT Like OP_SETMEMBER but more generic for the case of vectors as well (i.e. `hashvec[index]` versus `hash.member`).

OP_HAPPEND Another hash set but this time with the arguments in a funny order: STK(1) is the value to set, STK(2) the name, and STK(3) the hash.

13.2.7 Symbol "get" methods

OP_LOCAL Takes a constant argument and pushes

OP_MEMBER Retrieve a member from STK(1) with the name given by a `CONSTARG()`. Recurses into the parents vector if necessary. This is unique – no other function goes into the parents vector!

13.2.8 Miscellaneous

Other opcodes:

OP_XCHG Swap the top two items on the stack.

OP_XCHG2 Swap the top three items on the stack (i.e. first becomes third and the two others get shifted 'up' in the stack, thus a 1-2-3 order gets converted to a 2-3-1 order). An OP_XCHG2 followed by an OP_XCHG results in a total reversing (i.e. 3-2-1) and OP_XCHG followed by an OP_XCHG2 results in a swap of the lower two items (i.e. 1-3-2).

OP_EACH Works like a vector get: STK(1) is the index and STK(2) is the vector; it then increments the index and pushes the result.

OP_INDEX Same, but pushes a copy of the index instead of the variable at that position in the vector.

OP_UNPACK Take the contents of the vector on the top of the stack and push its contents one-by-one for the number of times specified by the argument on the opccode stack.

OP_RETURN If this is the top of the call stack (i.e. *fTop* is 1) then it returns from the `run()` function with the value of STK(1), else it resets to the next call up the stack setting the new STK(1) with the value of the old STK(1).

13.3 Stack Frames

Stack frames are essential to running a programming language. Functions call other functions, and their data must be stored on a stack. In Nasal, each `struct Context` object has a stack of frames in its `struct Frame` *fStack[]* member, whose top is stored in the `int` *fTop* member. The definition of each element is:

```
struct Frame {
    naRef func;
    naRef locals;
    int ip;
    int bp;
};
```

The *func* member stores the `naFunc` that created the Frame. The *locals* member is a new hash that is loaded with the arguments to the function and is used as the "running namespace"; it is discarded when the function returns, unless saved by some means (either making a func{} that is findable by the GC or using the `caller()` API). Thus, despite many calls to the same function, especially in recursion, each invocation will safely run in its own, separate "workspace" and will not interfere with other calls of the same function. The *ip* and *bp* members refer to the position in the opcode (instruction pointer and base pointer) and `naRef` stacks, respectively. The former begins at 0 while the latter is equivalent to the `ctx->opFrame` at the time and is used to pop `ctx->opFrame` back to its old value, so that the function can access its own data on that stack again.

As a function call is executed, its frame is pushed onto the stack at the current *fTop* index, which is then pushed up one. Thus the frame of the currently executing code is `ctx->fStack[ctx->fTop-1]`.

The relationship of of the operand stack with function calls is rather weird. All functions on a context's call stack end up using the same operand stack, and they all have access only to the top of it – specifically they should only access those that they pushed themselves. When a function call is done, any of its data on the stack gets popped by setting `ctx->opFrame = ctx->fStack [ctx->fTop-1].bp`, its return gets pushed, and now the old function can use its old operands, with the return now added on top.

14 Garbage Collection

14.1 Mark & Sweep

When using a mark-and-sweep collector, unreachable objects are not immediately reclaimed. Instead, garbage is allowed to accumulate until all available memory has been exhausted. When that happens, the execution of the program is suspended temporarily while the mark-and-sweep algorithm collects all the garbage. Once all unreferenced objects have been reclaimed, the normal execution of the program can resume.

Obviously, the main disadvantage of the mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs. In particular, this can be a problem in a program that must satisfy real-time execution constraints (like a flight simulator). For example, an interactive application that uses mark-and-sweep garbage collection becomes unresponsive periodically.

The mark-and-sweep algorithm is called a tracing garbage collector because is traces out the entire collection of objects that are directly or indirectly accessible by the program. The objects that a program can access directly are those objects which are referenced by local variables on the processor stack as well as by any static variables that refer to objects. In the context of garbage collection, these variables are called the roots. An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object. An accessible object is said to be live. Conversely, an object which is not live is garbage.

The mark-and-sweep algorithm consists of two phases: In the first phase, it finds and marks all accessible objects. The first phase is called the mark phase. In the second phase, the garbage collection algorithm scans through the heap and reclaims all the unmarked objects. The second phase is called the sweep phase.

Nasal's implementation of `sweep()` (called `reap()`) works such that it is always executed for a handful of different type-specific memory pools. In addition, it also makes sure to allocate new memory if required.

14.2 Nasal Memory Pools

A memory pool is basically a preallocated region of memory, which is dynamically resized as required. The Nasal GC works such that it manages a handful of global memory pools for all native Nasal types (strings, functions, vectors, hashes etc). At the moment, the hard coded defaults ensure that 25–50% of additional object ”slots” (memory blocks) are kept available during each execution of reap().

Whenever new memory is requested to create a new type (such as a vector or a string), the available memory in the corresponding pool will be checked, reachable objects will be marked, and dead objects will be removed from all pools using a mark/sweep collector, new memory blocks will be allocated if necessary. All of this happens atomically, i.e. single-threaded, in a “stop-the-world” fashion.

All Nasal memory pools are implemented using memory blocks, a memory block is implemented as a singly linked list. Each block contains a field to store its size, a pointer to the allocated memory, and another pointer to the next block.

```
struct Block {
    int size; // size of the block
    char* block; // pointer to the memory region
    struct Block* next; // pointer to the next block
};
```

For each of the 7 Nasal data types, there is a separate storage pool used, declared as part of the ”Globals” structure. As can be seen, each storage pool is addressed by its enum index (0..6):

```
enum { T_STR, T_VEC, T_HASH, T_CODE, T_FUNC, T_CCODE, T_GHOST, NUM_NASAL_TYPES };
```

The 7 storage pools are declared in code.h as part of the ”Globals” structure:

```
struct Globals {
    // Garbage collecting allocators:
    struct naPool pools[NUM_NASAL_TYPES];
    int allocCount;
    // Dead blocks waiting to be freed when it is safe
    void** deadBlocks;
    int deadsz;
    int ndead;
//...
};
```

New memory blocks are allocated using `newBlock()` and the memory is initialized with 0:

```c
static void newBlock(struct naPool* p, int need)
{
    int i;
    struct Block* newb;
    if(need < MIN_BLOCK_SIZE) need = MIN_BLOCK_SIZE;
    // allocate a new Block
    newb = naAlloc(sizeof(struct Block));
    // initialize the Block
    newb->block = naAlloc(need * p->elemsz); // number of elements * size of element
    newb->size = need; // set block size
    // memory blocks are circular linked lists:
    newb->next = p->blocks;
    p->blocks = newb;
    naBZero(newb->block, need * p->elemsz);

    if(need > p->freesz - p->freetop) need = p->freesz - p->freetop;
    p->nfree = 0;
    p->free = p->free0 + p->freetop;
    // mark all new blocks as unreachable and add them to the pool's free list
    for(i=0; i < need; i++) {
        // initialize each new new memory blocks by setting up an naRef (the container for all↩
            Nasal references)
        struct naObj* o = (struct naObj*)(newb->block + i*p->elemsz);
        o->mark = 0;
        p->free[p->nfree++] = o;
    }
    p->freetop += need;
}
```

15 Error and Exception Handling

15.1 Parser error handling

This information is relevant to lex.c, parse.c, and codegen.c.

If an error occurs, often relevant information is lacking as there is no "state" of the generator, so most errors end up saying "parse error" and give you a line number, instead of what could be helpful information such as "inside a foreach loop, this error occured".

15.2 VM error handling

15.3 Exception handling via `call()`

Part III
Examples and Existing Work

16 The Nasal Standard Library

17 Multithreading Support

18 Nasal bindings

18.1 SQLite

18.2 PCRE (regex)

18.3 Cairo

18.4 Gtk

18.5 OpenCL

19 Embedding Nasal

19.1 Basic Nasal Integration Example

1. create a new context using `naNewContext()`
2. parse the code using `naParseCode()`
3. create the standard namespace using `naInit_std()`
4. add any custom symbols using `naAddSym()` (optional)
5. add other required bindings/libraries (optional)
6. use `naCall()` to call the code in the namespace
7. do error checking using `naGetError()`

```
// A Nasal extension function (prints its argument list to stdout)
static naRef print(naContext c, naRef me, int argc, naRef* args)
{
    int i;
    for(i=0; i<argc; i++) {
        naRef s = naStringValue(c, args[i]);
        if(naIsNil(s)) continue;
        fwrite(naStr_data(s), 1, naStr_len(s), stdout);
    }
    return naNil();
}
```

19.2 Nasal Integration in FlightGear

In FlightGear, Nasal is added as a SGSubsystem in the form of the *FGNasalSys* class in *$FG_SRC*/Scripting/NasalSys.cxx, which is mainly responsible for initializing the interpreter at the moment, as most runtime code is invoked through callbacks that are run via timers and listeners. It notably introduces FlightGear-specific extension functions, like setlistener()/removelistener(), settimer(), and systime() functions, and a new area of development is exposing C++ classes and alternative Nasal APIs via ghosts, like the maketimer() function does.

19.2.1 Initialization

At the moment, *FGNasalSys* loads all the Nasal code currently. This will hopefully be changed in the future to allow a Nasal script to do all of the loading, making it a more flexible and transparent process, which is important to those who do not compile from source or understand that source but need to know how things work.

Initialization is a big part of running Nasal code in FlightGear and most Nasal libraries/subsystems get loaded this way. But to keep running intermittently, to provide an alternative method of loading, or to run a Nasal "action" script, there are three other ways to run Nasal in FlightGear: timers, listeners, and bindings.

19.2.2 Timers

Timers can be used to repetively run bits of code in a loop, somewhat like a proper subsystem in FlightGear, to delay a response to something, or to distrubute work across frames. settimer() is mainly used for this right now, and it simply registers a function to be run after a certain amount of time – this is checked every frame, and so at maximum it can only run at frame rate. To run in a loop, the callback must re-register itself in a new timer, which is inefficient. Recently, maketimer() was introduced, which returns a Nasal ghost that is a virtual C++ timer object. This timer can be stopped, (re-)started, and told to run at a certain interval (solving the need to reregister a timer), or to just run once (like settimer() does, though maketimer() objects can be reused). These both illustrate how to constantly run code in FlightGear, and this can be very useful though being limited to the frame rate.

19.2.3 Listeners

Another solution to looping is hooking a Nasal callback to a property that is called by some subsystem updating at a potentially higher rate – like the rate of the underlying flight dynamics model (FDM). These callbacks are called listeners, and for long-standing reasons they may not work with all properties created by C++ code, but there are some places where they are applicable. Other uses are waiting for a user input (like adjusting a preference, which may require 3D objects or even Nasal modules to be loaded or unloaded), communicating between systems, receiving direct input from C++ or other languages interfacing with FlightGear, and more. In the future, all Nasal modules other than necessary libraries might be loaded from listeners, as they provide a convient way to run Nasal code that transcends all languages in FlightGear by using the property tree.

19.2.4 Bindings/FGCommands

Bindings are the third and final way to run code. FlightGear has a core set of commands – called fgcommands – that interact with their caller via the property tree and can be executed from both C++ and Nasal (via the `fgcommand()`function in Nasal). These are often wrapped in "bindings", which are property nodes that store the name of their command and arguments to the command side-by-side. One of these commands is "nasal", which executes a Nasal snippet in a "script" node. Other commands provide access to C++ functionality, such as the flight plan. Nasal can actually register its own fgcommands, so Nasal code can even be registered to run via bindings that do not use the "nasal" command.

One of the exciting things about bindings is that they are prevalent in many areas of FlightGear. In particular, they are often used to handle user input (joystick and mouse events and "pick" animations on 3D objects) and can be incorporated as elements of other systems, like checklists that can be "run" by executing sequential bindings. These three parts of FlightGear, timers, listeners, and bindings, allow for a big diversity in how Nasal code can be executed within FlightGear. It also illustrates well what they have done with integrating Nasal into its core – perhaps someday achieving the dream of using Nasal and C/C++ side by side.

20 Nasal Maintenance

If Nasal is not maintained well and code bases around it evolve too much, then it might become a broken part of whatever it is integrated into, or even being broken by itself. When integrating Nasal into a new place, developing infrastructure to cement Nasal and the host application together will be needed, and one even might want to extend core Nasal functionality. This section aims to teach how some of these tasks can be done.

20.1 Adding a new Token

20.2 Adding a new OP code

20.3 Adding a new binary operator

20.4 Adding a new loopoid

20.5 Adding a new extension function

21 Known Bugs & Issues

22 Maintenance Examples

Some ideas (brainstorming)

1. iterating through hashes (Philosopher; done)
2. serialize/unserialize functions
3. json support
4. native tasking primitives
5. DbC: preconditions/postconditions
6. protected types(via hashes) with implicit locking
7. do{}while() construct or other loopoid (Philosopher)
8. custom typing
9. list comprehension

22.1 Perl's Spaceship Operator

See `http://wiki.flightgear.org/How_to_add_a_new_binary_operator_to_Nasal` for the full example.

22.2 Slicing Strings

Currently, Nasal only supports slicing vectors, not strings. Let's say that we want the ability to slice strings as well; how would we go about adding this? First we want to consider what stage of Nasal will be affected: parsing (parse.c, codegen.c) or running (code.c)? It's obvious that since the type of the sliced variable is a runtime property, we will only be editing code.c, since anything before that has no knowledge of the types of a variable – right? Well I thought so too, but then I looked more closely and saw that codegen.c actually pushes a new vector when it sees a slice. This is an optimization that obviously does not work well with strings, and so I introduced a new opcode to push a slice of the appropriate type – I called it OP_NEWSLICE.

22.3 Tom's String Methods

23 VM Extensions

23.1 Dumping internal data structures

23.2 Implementing support for parser hooks

23.3 Implementing debugging support via VM hooks and custom opcodes

23.4 Implementing instrumentation/profiling support via codegen hooks

23.5 Implementing a bytecode optimizer via VM hooks

23.6 Implementing RAII support via VM events

23.7 Exposing GC behavior to Nasal space

23.8 Implementing a VM debugger

1. breakpoints (via special opcodes or setjmp exceptions)
2. stepping
3. interactive, bash-style interface to stepping/executing code and viewing the state of the VM. Could also be useful to have a "Nasal explorer" as well. FGCanvas too, maybe?

24 Debugging Nasal

24.1 Dumping Tokens

24.2 Dumping the Parse Tree

24.3 Dumping Bytecode

24.4 Dumping Opcode/Operand Stacks

24.5 Dumping Stack Frames

24.6 Dumping GC State

25 Nasal Syntax