

Nickle Tutorial

Robert Burgess, Keith Packard

Version 2.117

Table of Contents

1. Nickle Tour . . . 1
2. Nickle Basics . . . 4
    2.1. Invocation . . . 4
    2.2. Commands . . . 5
3. Language introduction . . . 11
    3.1. Nickle Datatypes . . . 11
    3.2. Nickle Expressions . . . 23
    3.3. Statements in Nickle . . . 28
    3.4. Nickle Functions . . . 33
4. Builtins . . . 35
    4.1. Input and Output . . . 35
    4.2. Math . . . 38
    4.3. Strings . . . 40
5. Advanced topics . . . 43
    5.1. Copy Semantics and Garbage Collection . . . 43
    5.2. Nickle Namespaces . . . 45
    5.3. Nickle Exceptions . . . 47
    5.4. Threads and Mutual Exclusion in Nickle . . . 50
    5.5. Nickle Continuations . . . 54
Colophon . . . 56

Chapter 1. Nickle Tour

The following is an example Nickle session, interspersed with comments.

```
$ nickle
```

Arithmetic works as expected, with a rich set of operators.

```
> 1 + 1
2
> 2 ** (2 + 2)
16
> 5!
120
```

Rationals are represented exactly, but printed as decimals rounded to 15 digits. Math is done with arbitrary precision. Notice that integer division (//) is different from rational division (/). Nickle provides some conveniences, such as `.` denoting the last value printed.

```
> 1 / 3
0.333333333333333
> . * 3
1
> 1 // 3
0
```

Variables can be declared implicitly at the top level, as well as explicitly with type. The results of statements are not printed; terminating an expression with a semicolon makes it a simple statement. C-like control structures may also be used at the top level; the `+` prompt indicates an incomplete line to be continued.

```
> x = .
0
> int y = x;
> ++y;
> for (int i = 0; i < 25; i++)
+   x += 0;
> x
0
> for (int i = 1; i < 9; i += 2)
+   x += i;
> x
16
```

When performing square roots, Nickle will stay exact when possible. If the result is irrational,

however, it will be stored as an inexact real. Imprecision is contagious; if a rational operator combines an imprecise variable with a precise one, the result will be imprecise. Squaring the square root of 5 prints as 5, but subtracting 5 shows that it is off in the 77th decimal place.

```
> sqrt(x)
4
> sqrt(2)
1.414213562373095
> .**2
2
> sqrt(5)
2.23606797749979
> . ** 2
5
> . - 5
6.908934844075556e-77
```

Functions can also be typed at the top level. Since functions, as most things, are first-class in Nickle, they may be declared and assigned as below.

```
> real foo(real x, real y) {
+   return x * y;
+ }
> foo(2, 3)
6
> foo(4, 2)
8
> real(real, real) bar = foo;
> bar(4, 2)
8
>
```

Nickle is guaranteed never to dump core; it has a simple yet powerful exception system it uses to handle all errors. When Nickle reads commands from a terminal, an unhandled exception raised inside a function leads to the debugger, which uses a - prompt. The debugger can be used to trace the stack, move up and down on it, and check values of variables.

```
> (-1) ** (1/2)
Unhandled exception invalid_argument ("sqrt of negative number", 0, -1)
/usr/share/nickle/math.5c:18:     raise invalid_argument("sqrt of negative number", 0,
v);
    sqrt (-1)
/usr/share/nickle/math.5c:1548:     result = sqrt(a);
    pow (-1, 0.5)
<stdin>:1:     -1 ** (1 / 2);
- done
> quit
```

```
$
```

Large chunks of code can be placed in a separate text file and loaded when needed: `load` reads a file by its name, and `library` searches Nickle's library directories, which hold the examples that come with Nickle. The `print` command can be used to inspect variables, functions, namespaces, and other names. `import` brings the names in a namespace into scope. The `::` operator can be used to view those names without importing them. (These can also be used with several namespaces built in to Nickle, such as Math and File.)

```
$ nickle
> library "examples/cribbage.5c"
> print Cribbage
namespace Cribbage {
    public int score(int[*] v);
}
> print Cribbage::score
public int
score(int[*] v)
{
    int n = dim(v);
    return 2 * countsum(15, v, n) + 2 * countpairs(v, n);
}
> import Cribbage;
> score((int[*]) { 5, 5, 10, 7, 8 })
8
> quit
$
```

Chapter 2. Nickle Basics

Nickle is a powerful desktop calculator language with many features of advanced languages and support for arbitrary precision numbers. It can run interactively to fulfill its role as a calculator, evaluate single expressions, and execute Nickle scripts. It also has an array of useful top-level commands for interacting with the interpreter.

2.1. Invocation

> `nickle`
> [`--help`|`--usage`]
> [`-f` *file*]
> [`-l` *library*]
> [`-e` *expr*]
> [`-V`]
> [*script*] [`--`] [*arg* ...]
> `nickle -F`|`-P`|`-C` [*file* ...]

`-f` *file*, `--file` *file*

Evaluate *file*.

`-l` *library*, `--library` *library*

Evaluate *library* like `-f`, but find it in Nickle's library path, `Command::nickle_path`. That is `$NICKLEPATH` if it is set, and Nickle's installed library directory otherwise.

`-e` *expr*, `--expr` *expr*

Evaluate a Nickle expression, e.g.

```
$ nickle -e 3**4
81
$
```

`-P` *file* ..., `--pretty` *file* ...

Print each *file* formatted, in order, instead of running it, laying out each command and declaration as `print` does, comments and all. Commands that load code, such as `autoimport` and `library`, still run, as the code after them needs their names. A *file* of `-`, or none, reads standard input.

```
$ cat square.5c
/* Print a square */
int square(int x) { return x*x; }
printf("%d\n", square(3));
$ nickle --pretty square.5c
/* Print a square */
```

```
int square(int x) = x * x;

printf("%d\n", square(3));
$
```

`-F` ***file*** **…**, `--format` ***file*** **…**

Format each *file* as `--pretty` prints it, writing it back when that changes it. A file that can't be formatted, as one with a syntax error, stays as it was, and Nickle exits with status 1.

`-C` ***file*** **…**, `--check-format` ***file*** **…**

Report the first line of each *file* that formatting would change, and exit with status 1 if formatting would change any. A *file* of `-`, or none, reads standard input.

```
$ nickle --check-format square.5c
square.5c:2: formatting changes this line
$
```

`-V`, `--version`

Print Nickle's version and exit.

`--help`, `--usage`

Print a summary of these options and exit.

`--`

End the options: the next argument is the script, even if it starts with `-`.

`script`

If Nickle encounters an unflagged argument, it assumes it to be the name of a script, which it runs. A script named `-` is read from standard input. No more arguments are processed; the rest of the line is given to the script as its arguments. Nickle puts the script's directory at the front of its library path, so the script can load libraries kept beside it.

Options may be repeated, and run in the order given. A syntax or compile error in any of them, or in a script, makes Nickle exit with status 1.

Without `-e` or a script as an argument, Nickle runs interactively, accepting standard input and writing to standard output, after evaluating any files given with `-f` or `-l`. When it reads commands from a terminal, it loads the file named by `$NICKLERC`, or `~/.nicklerc` if that isn't set, before anything else, as Python does with `PYTHONSTARTUP`.

2.2. Commands

The following are commands that the Nickle interpreter understands, not actual language constructs. They may be issued only at the top level.

2.2.1. Expressions

If an expression is issued at the top level, such as `3**4` or `100!`, its value is printed to standard output, formatted by the `format` variable, `"%g"` to start with. Setting `format` to `"%.-g"` shows values exactly, rationals with any repeating digits in curly braces. If the expression ends with a # sign and another expression, its value is printed in whatever base the second expression evaluates to.

```
$ nickle
> 10!
3628800
> 3**4
81
> 3**4 # 3
10000
> 1/3
0.333333333333333
> format = "%.-g";
> 1/3
0.{3}
>
```

Statements, from expressions terminated by semicolons to complicated control structures, are executed but have no value to print. Statements are not commands but actual syntax, so they may be used in scripts. A newline ends a command, except inside parentheses, brackets, or braces, or partway through a control statement such as `for` or `if`. There, Nickle reads more lines, prompting with `+`, until the command is complete, e.g.

```
$ nickle
> int x = (1 +
+ 2);
> x
3
>
```

2.2.2. Quit

The `quit` command exits Nickle. An optional argument specifies the return value.

```
$ nickle
> quit
$
```

2.2.3. Print

The `print` command provides information such as visibility, type, and value, about a name. It need not be the name of a variable; functions, namespaces, etc. may also be printed. A namespace prints

as a summary of its public names, functions without their bodies; `print Name::` prints all of it, as written, and `print ::` every name. A file name in quotes, as in `print "prog.5c"`, prints the file formatted, without running it.

```
$ nickle
> int x = 2;
> print x
global int x = 2;
> print String::length
public int length(string)
    /*
     * int length (string s)
     *
     * Return the number of characters in 's'.
     */ <builtin>
> void function hello() { printf("hello, world\n"); }
> print hello
void
hello()
{
    printf("hello, world\n");
}
>
```

Comments stay with the code they describe, and `print` shows them re-indented, with their lines as written. A comment that starts on the same line as code trails that code, and any other comment leads the code after it; a function with comments on its parameters prints one parameter per line. `pretty-printer.md`, installed with Nickle's documentation, gives the full rules.

```
$ nickle
> /* Count the vowels in s */
> int vowels(string s /* lower case */) {
+     int n = 0;        /* found so far */
+     for (int i = 0; i < String::length(s); i++)
+         if (String::index("aeiou", String::new(s[i])) >= 0)
+             n++;
+     return n;
+ }
> print vowels
/* Count the vowels in s */
int
vowels(
    string s /* lower case */)
{
    int n = 0; /* found so far */

    for (int i = 0; i < String::length(s); i++)
        if (String::index("aeiou", String::new(s[i])) >= 0)
            n++;
```

```
    return n;
}
>
```

2.2.4. Undefine

A defined name can be undefined. The `undefine` command takes several names, separated by commas, and stops at the first one it can't find, e.g.

```
$ nickle
> print x
"x": name not found
> int x = 0;
> print x
global int x = 0;
> undefine x
> print x
"x": name not found
>
```

2.2.5. Loading files

The `load` command evaluates the file it names, like the `-f` flag, and `library` finds the file in Nickle's library path, like the `-l` flag. `library` also loads C extensions, shared libraries named like `libcairo-5c.so.0`, only the first time. Unlike those flags, an error in such a file, or a file that can't be found, is reported and Nickle continues.

```
$ nickle
> load "nosuch.5c"
Cannot load "nosuch.5c": No such file or directory
> library "nosuch.5c"
Cannot load library "nosuch.5c": not found in "/usr/share/nickle"
>
```

The `autoload` command loads the library for each namespace it names, unless the namespace is already defined. The library's file name is the namespace's name in lower case, with a dash before each capital letter that follows a lower-case one, and `.5c` appended, so `PRNG` comes from `prng.5c`. The `autoimport` command does the same and then imports each namespace, and `reimport` loads the libraries even when the namespaces are already defined.

```
$ nickle
> autoimport PRNG
> randint(6) < 6
true
>
```

2.2.6. Edit

The `edit` command invokes `$EDITOR` on the name given as an argument, as `print` shows it, comments included, so they survive the edit. This is particularly useful to change a function while in interactive mode.

```
$ nickle
> void function hello() { printf("hello, world\n"); }
> edit hello
47
4
    printf("hello, world\n");
c
printf ("goodbye, cruel world\n");
wq
52
> print hello
void
hello()
{
    printf("goodbye, cruel world\n");
}
>
```

2.2.7. History

The `history` command retrieves the values of the last ten expressions. With an argument, it instead retrieves that many of the most recent values. With two arguments, it retrieves the specified range in history.

```
$ nickle
...
> history
$177    5
$178    0
$179    12
$180    12
$181    -2
$182    2
$183    2
$184    0
$185    10
$186    32
> history 3
$184    0
$185    10
$186    32
> history 180,185
$180    12
```

```
$181    -2
$182    2
$183    2
$184    0
$185    10
```

These history items may be named and used directly:

```
> $180 ** 2
144
>
```

2.2.8. Char

The `char` command prints each code point it is given in decimal and hexadecimal, with the character it encodes.

```
$ nickle
> char 0x263a
9786 0x263a: "☺"
>
```

Chapter 3. Language introduction

In this chapter, the features of Nickle such as datatypes, expressions, control statements, and functions will be discussed. By the end, most of the basic language features will have been covered.

3.1. Nickle Datatypes

3.1.1. Primitive datatypes

Nickle has a large set of primitive datatypes. Instead of overloading existing datatypes to represent fundamentally distinct objects, Nickle provides additional primitive datatypes to allow for typechecking. For instance, instead of using small integers to identify file handles, Nickle provides a `file` datatype.

Numeric datatypes

Nickle has four numeric datatypes:

- `int`
- `rational`
- `real`
- `complex`

Int and rational values are represented exactly to arbitrary precision so that computations need not be concerned about values out of range or a loss of precision during computation. For example,

```
> 1/3 + 2/3 == 1
true
> 1000! + 1 > 1000!
true
```

As rationals are a superset of the integers, rational values with denominator of 1 are represented as ints. The builtin `is_int` demonstrates this by recognizing such rationals as integers:

```
> rational r = 1/3;
> is_int(r)
false
> is_int(r*3)
true
```

Real values are either represented as a precise int or rational value or as an imprecise value using a floating point representation with arbitrary precision mantissa and exponent values. Imprecision is a contagious property; computation among precise and imprecise values yields an imprecise result.

```
> real i=3/4, j=sqrt(2);
```

```
> is_rational(i)
true
> is_rational(j)
false
> is_rational(i*j)
false
```

Upward type conversion is handled automatically; divide an int by an int and the result is rational. Downward conversion only occurs through builtin functions which convert rational or real values to integers.

```
> int i=4, j=3;
> is_int(i/j)
false
```

Complex values have real and imaginary parts, each of which may be any of the other numeric types. The `Complex` namespace holds the functions that work on them.

```
> complex c = Complex::sqrt(-4);
> c * c
-4
```

Boolean datatype

The `bool` datatype holds `true` or `false`, the results of comparisons and logical operators. The conditions of `if`, `while`, `for`, and `?:` must be bools; unlike C, Nickle doesn't treat numbers as truth values.

```
> bool b = 3 > 2;
> b
true
```

String datatype

A `string` holds a read-only sequence of Unicode characters. Several builtin functions accept and return this datatype. Elements of a string are accessible as integer code points by using the array index operators. See the section on Strings.

```
string foo = "hello";
string bar = "world";
string msg = foo + ", " + bar;

printf("%s\n", msg); /* hello, world */
```

File datatype

The `file` datatype provides access to the native file system. Several builtin functions accept and return this datatype. See the section on input and output.

```
file f = File::open("file", "w");

File::fprintf(f, "hello, world!\n");
File::close(f);
```

Concurrency and control flow datatypes

Nickle has builtin support for threading, mutual exclusion, and continuations, along with the associated types:

- `thread`
- `semaphore`
- `continuation`

Threads are created with the `fork` expression, the result of which is a thread value. Semaphores are a synchronization datatype, and the `Mutex` library builds mutexes on them. See the section on Concurrency and Continuations.

```
thread t = fork 5!;

/* do stuff... */
printf("5! = %d\n", Thread::join(t)); /* 5! = 120 */
```

Continuations capture the dynamic state of execution in much the same way as a C `jmp_buf` except that the `longjmp` operation is not limited to being invoked from a nested function invocation. Rather, it can be invoked at any time causing execution to resume from the point of `setjmp`. See the section on Continuations.

Poly datatype

Non-polymorphic typechecking is occasionally insufficient to describe the semantics of an application. Nickle provides an 'escape hatch' through the `poly` datatype. Every value is compatible with poly; a variable with this type can be used in any circumstance. Nickle performs full run-time typechecking on poly datatypes to ensure that the program doesn't violate the type rules.

```
> poly i = 3, s = "hello\n";
> i + 3
6
> s + 3          /* can't add string and int */
Unhandled exception invalid_binop_values ("incompatible operands for +", "hello\n", 3)
<stdin>:3:     s + 3;
> printf(i)     /* printf expects a string */
```

```
Unhandled exception invalid_argument ("Incompatible argument", 0, 3)
<stdin>:4:     printf(i);
> printf(s)
hello
>
```

Void datatype

To handle functions with no return value and other times when the value of an object isn't relevant, Nickle includes the `void` datatype. This is designed to operate in much the same way as the `unit` type does in ML. Void is a type that is compatible with only one value, `<>`. This value can be assigned and passed just like any other value, but it is not type compatible with any type other than void and poly.

3.1.2. Composite datatypes

Nickle allows the basic datatypes to be combined in several ways.

- `struct`
- `union`
- arrays
- hashes
- pointers
- references
- functions

Structs

Structs work much like C structs; they composite several datatypes into an aggregate with names for each element of the structure. One unusual feature is that a struct value is compatible with a struct type if the struct value contains all of the entries in the type. For example:

```
typedef struct {
    int  i;
    real r;
} i_and_r;

typedef struct { int i; } just_i;

i_and_r i_and_r_value = { i = 12, r = 37 };

just_i  i_value;

i_value = i_and_r_value;
```

The assignment is legal because `i_and_r` contains all of the elements of `just_i`. `i_value` will end up

with both `i` and `r` values.

Unions

Unions provide a way to hold several different datatypes in the same object. Unions are declared and used much like structs. When a union element is referenced, Nickle checks to make sure the referring element tag is the one currently stored in the union. This provides typechecking at runtime for this kind of polymorphism. Values can be converted to a union type by specifying a compatible union tag cast. A control structure `union switch` exists to split out the different tags and perform different functions based on the current tag:

```
typedef union {
    int  i;
    real r;
} i_and_r_union;

i_and_r_union u_value;

u_value.i = 37;

union switch (u_value) {
case i:
    printf("i value %d\n", u_value.i);
    break;
case r:
    printf("r value %g\n", u_value.r);
    break;
}

u_value = (i_and_r_union.r) 1.2;
printf("u_value %g\n", u_value); /* u_value r = 1.2 */
```

Enums

An enum names a fixed set of values. A value is written with the enum's type and the name, and `enum switch` chooses among them:

```
typedef enum { red, green, blue } color;

color c = color.green;

enum switch (c) {
case red:
    printf("red\n");
    break;
case green:
    printf("green\n"); /* green */
    break;
case blue:
```

```
    printf("blue\n");
    break;
}
```

Arrays

Array types in Nickle determine only the number of dimensions and not the size of each dimension. Therefore they can be declared in one of three ways:

```
int[*]   a;
int[...] b;
int[3]   c;
```

By these declarations, `a`, `b` and `c` are of the same type (one-dimensional array). The specification of the size of `c` actually has no effect on its declaration but rather on its initialization. Declaring `b` also gives it a value, an empty resizable array, and declaring `c` gives it an array of three uninitialized elements, but `a` starts out uninitialized. See Initializers below. Declaring multidimensional arrays in Nickle is different than in C; C provides only arrays of arrays while Nickle allows either:

```
int[3, 3] array_2d = {};
int[3][3] array_of_arrays = { (int[3]) {} ... };

array_2d[0, 0] = 7;
array_of_arrays[0][0] = 7;
array_of_arrays[1] = (int[2]) { 1, 2 };
```

These two types can be used in similar circumstances, but the first ensures that the resulting objects are rectangular while the second allows for each row of the array to have a different number of elements. The second also allows for each row to be manipulated separately. The final example shows an entire row being replaced with new contents.

Array values created with '...' in place of the dimension information are resizable; requests to store beyond the bounds of such arrays will cause the array dimensions to be increased to include the specified location. Resizable arrays may also be passed to the `setdim` and `setdims` built-in functions.

Hashes

Hashes provide an associative mapping from arbitrary keys to values. Any type may be used as the key. This allows indexing by strings, and even composite values. They are called `hashes` instead of associative arrays to make the performance characteristics of the underlying implementation clear.

Hashes are declared a bit like arrays, but instead of a value in the brackets, a type is placed:

```
int[string] string_to_int = { "hello" => 2, => 0 };
real[real]  real_to_real = { 2.5 => 27 };

string_to_int["bar"] = 17;
```

```
string_to_int["foo"] += 12;
real_to_real[pi] = pi / 2;
```

The initializer syntax uses the double-arrow `=>` to separate key from value. Eliding the key specifies the "default" value — used to instantiate newly created elements in the hash.

Pointers

Pointers hold a reference to a separate object; multiple pointers may point at the same object and changes to the referenced object are reflected both in the underlying object as well as in any other references to the same object. While pointers can be used to point at existing storage locations, anonymous locations can be created with the reference built-in function; this allows for the creation of pointers to existing values without requiring that the value be stored in a named object.

```
*int pointer;
int  object;

pointer = &object;
*pointer = 12;

printf("%g\n", object); /* 12 */

pointer = reference(37);
(*pointer)++;

printf("%g\n", *pointer); /* 38 */
```

References

References, like pointers, refer to objects. They are unlike pointers, however; they are designed to provide for calls by reference in a completely by-value language. They may eventually replace pointers altogether. They are declared and assigned similarly, but not identically, to pointers:

```
&int ref;
int  i;

i = 3;
&ref = &i;
```

`ref` is declared as a reference to an integer, `&int`. An integer, `i`, is declared and given the value 3. Finally, the assignment carries some interesting semantics: the address of the reference is set to the address of `i`. References may also be assigned otherwise anonymous values with `reference`, e.g.

```
&int foo;
&foo = reference(3);
```

References, unlike pointers, need not be dereferenced; they are used exactly as any other value. Changing either the value it refers to or the reference itself changes both.

```
printf("%g\n", i);   /* 3 */
printf("%g\n", ref); /* 3 */

++ref;

printf("%g\n", i);   /* 4 */
printf("%g\n", ref); /* 4 */
```

Functions

Nickle has first-class functions. These look a lot like function pointers in C, but important semantic differences separate the two. Of course, if you want a function pointer in Nickle, those are also available. Function types always have a return type and zero or more argument types. Functions may use the void return type. The final argument type may be followed by an ellipsis (...), in which case the function can take any number of arguments at that point, each of the same type as the final argument type:

```
int(int, int) a;
void(int ...) b;

a(1, 2);
b(1);
b(1, 2);
b(1, "hello"); /* illegal, "hello" is not compatible with int */
```

See the section on Functions.

3.1.3. Declarations

A declaration in Nickle consists of four elements: publication, storage class, type, and name. Publication, class, and type are all optional but at least one must be present and they must be in that order.

- Publication is one of `public` or `protected`, which defines the name's visibility within its namespace. When publication is missing, the name is private: it will not be visible outside of the current namespace. See the section on Namespaces.
- Class is one of `global`, `static`, `auto`, or `const`. When class is missing, Nickle uses `auto` for variables declared within a function and `global` for variables declared at the top level. See Storage classes below.
- Type is some type as described here, for instance

  ```
  type                    /* primitive type */
  *type                   /* pointer to type */
  ```

```
&type                   /* reference to type */
type[*]                 /* array of type */
type[...]               /* resizable array of type */
type[*, *]              /* two-dimensional array of type */
type[type]              /* hash from type to type */
type(type, ...)         /* function with type arguments and return value */
struct { ... }          /* struct of types */
union { ... }           /* union of types */
type[*](type, ...)      /* array of functions */
/* etc. */
```

When type is missing, Nickle uses poly, which allows the variable to hold data of any type. Nickle checks types when it compiles the code, and again at runtime where a value's type isn't known until then, as with poly.

3.1.4. Initializers

Initializers in Nickle are expressions evaluated when the storage for a variable comes into scope. To initialize array and structure types, expressions which evaluate to a struct or array object are used:

```
int    k = 12;
int    z = floor(pi * 27);
int[3] a = (int[3]) { 1, 2, 3 };

typedef struct {
    int  i;
    real r;
} i_and_r;

i_and_r s = (i_and_r) { i = 12, r = pi / 2 };
```

As a special case, initializers for struct and array variables may elide the type leaving only the bracketed initializer:

```
int[3]  a = { 1, 2, 3 };
i_and_r s = { i = 12, r = pi / 2 };
```

An array initializer may also compute each element from its indices, giving names for them in brackets:

```
int[5]    squares = { [i] = i * i }; /* 0, 1, 4, 9, 16 */
int[3, 3] identity = { [i, j] = i == j ? 1 : 0 };
```

Instead of initializing structures by their position in the declared type, Nickle uses the structure tags. This avoids common mistakes when the structure type is redeclared.

An array initializer followed by an ellipsis (`...`) is replicated to fill the remainder of the elements in that dimension:

```
int[4, 4] a = { { 1, 2 ... }, { 3, 4 ... } ... };
```

This leaves `a` initialized to an array whose first row is `{ 1, 2, 2, 2 }` and subsequent rows are `{ 3, 4, 4, 4 }`. It is an error to use this ellipsis notation when the associated type specification contains stars instead of expressions for the dimensions of the array. Variables need not be completely initialized; arrays can be partially filled and structures may have only a subset of their elements initialized. Using an uninitialized variable causes a run time exception to be raised.

3.1.5. Identifier scope

Identifiers are scoped lexically; any identifier in lexical scope can be used in any expression (with one exception described below). Each compound statement creates a new lexical scope. Function declarations and statement blocks also create new lexical scopes. This limits the scope of variables in situations like:

```
if ((int i = foo(x)) > 0)
    printf("i in scope here %d\n", i);
else
    printf("i still in scope here %d\n", i);
printf("i not in scope here\n");
```

Identifiers are lexically scoped even when functions are nested:

```
int
foo(int x)
{
    int y = 1;

    int bar(int z) = z + y;

    return bar(x);
}
```

3.1.6. Storage classes

There are four storage classes in Nickle:

- `auto`
- `static`
- `global`
- `const`

The storage class of a variable defines the lifetime of the storage referenced by the variable. When the storage for a variable is allocated, any associated initializer expression is evaluated and the value assigned to the variable.

Auto variables

`auto` variables have lifetime equal to the dynamic scope where they are defined. When a function is invoked, storage is allocated which the variables reference. Successive invocations allocate new storage. Storage captured and passed out of the function will remain accessible.

```
*int
foo(int x) = &x;

*int a1 = foo(1);
*int a2 = foo(2);
```

`a1` and `a2` now refer to separately allocated storage.

Static variables

`static` variables have lifetime equal to the scope in which the function they are declared in is evaluated. A function value includes both the executable code and the storage for any enclosed static variables, function values are created from function declarations.

```
int()
incrementer()
{
    return func() {
                static int x = 0;
                return ++x;
            };
}

int() a = incrementer();
int() b = incrementer();
```

`a` and `b` refer to functions with separate static state and so the values they return form independent sequences. Because static variables are initialized as the function is evaluated and not during function execution, any auto variables declared within the enclosing function are not accessible. This is the exception to the lexical scoping rules mentioned above. It is an error to reference auto variables in this context. Additionally, any auto variables declared within an initializer for a static variable exist in the frame for the static initializer, not for the function.

```
poly
foo()
{
    static poly
```

```
    bar(*int z)
    {
        *z = (*z)!;
    }

    static x = ((int y = 7), bar(&y), y);

    return x;
}
```

The static initializer is an anonymous function sharing the same static scope as the function containing the static declarations, but having its own unique dynamic scope.

Global variables

`global` variables have lifetime equal to the global scope. When declared at global scope, storage is allocated and the initializer executed as soon as the declaration is parsed. When declared within a function, storage is allocated when the function is parsed and the initializer is executed in the static initializer of the outermost enclosing function.

```
poly
foo()
{
    poly
    bar()
    {
        global g = 1;
        static s = 1;

        g++;
        s++;
        return (int[2]) { g, s };
    }

    return bar();
}
```

Because `g` has global scope, only a single instance exists and so the returned values from `foo` increment each time `foo` is called. However, because `s` has static scope, it is reinitialized each time `bar` is reevaluated as the static initializer is invoked and returns the same value each time `foo` is called.

Const variables

`const` variables are initialized when declared and can't be assigned afterward; an assignment to one is a compile error.

```
> const x = 3;
```

```
> x = 4;
->     x = 4;
<stdin>:2: Attempt to assign to const "x"
```

3.2. Nickle Expressions

See Precedence at the end of this section for how the operators group.

3.2.1. Variable declarations

Variable declarations are expressions in Nickle; the value of a declaration is the value of the last variable it declares. For example,

```
> int val;
> val = (int i=2, j=3);
> print val
global int val = 3;
```

If that variable has no initialization expression, it is an error to use the value of the expression.

```
> val = (int i, j);
Unhandled exception uninitialized_value ("uninitialized value")
<stdin>:3:     val = (int i, j);
```

Because they are expressions, declarations can be used in constructs such as:

```
for (int i = 0; i < 10; i++) {
}
```

3.2.2. Anonymous function declarations

Functions may be declared anonymously and used immediately, e.g.

```
> (int func(int a, int b) { return a + b; })(2, 3)
5
```

Any context available to the function at definition time will be available whenever it is executed. See Storage classes in the section on Nickle Datatypes.

3.2.3. Binary operators

Addition

`a + b`

Subtraction

```
a - b
```

Multiplication

```
a * b
```

Division, Integer division

```
a / b
a // b
```

Remainder

```
a % b
```

Exponentiation

```
a ** b
```

Left shift, Right shift

```
a << b
a >> b
```

Binary xor, Binary and, Binary or

```
a ^ b
a & b
a | b
```

Boolean and, Boolean or

```
a && b
a || b
```

Conditional

`a ? b : c` evaluates `b` if the bool `a` is true and `c` otherwise.

Equal, Not equal

```
a == b
a != b
```

Less than, Less than or equal to

```
a < b
a <= b
```

Greater than, Greater than or equal to

```
a > b
a >= b
```

Assignment

```
a = b
```

Assignment with operator

```
a += b    a -= b    a *= b    a /= b    a //= b   a %= b
a **= b   a <<= b   a >>= b   a ^= b    a &= b    a |= b
a &&= b   a ||= b
```

3.2.4. Unary operators

Dereference

```
* a
```

Reference

```
& a
```

Negation

```
- a
```

Bitwise inverse

```
~ a
```

Logical not

```
! a
```

Factorial

```
a !
```

Increment

```
++a
a++
```

Decrement

```
--a
a--
```

3.2.5. Constants

Integer constants

Integer constants with a leading zero or 0o are interpreted in octal, with a leading 0x in hexadecimal and with a leading 0b in binary. Underscores may separate digits, and a character in single quotes is its code point. Here are some examples:

```
12     /* 12, decimal */
014    /* 12, octal */
0o14   /* 12, octal */
0xc    /* 12, hex */
0b1100 /* 12, binary */
1_000  /* 1000 */
```

```
'a'    /* 97 */
```

Rational constants

Rational constants are the combination of an integer part, a mantissa with an initial and repeating part, and an exponent. All of these pieces are optional, but at least one of the parts (other than the exponent) must be present. Curly braces mark the repeating part, so `.{56}` is exactly 56/99. If the value is a whole number, its type is int rather than rational. The values are exact, though Nickle displays them rounded to 15 decimal places. Some examples:

```
> 12
12
> 12.5
12.5
> .34
0.34
> .{56}
0.565656565656566
> .{56} == 56/99
true
> .34e3
340
> .{56}e12
565656565656.565656565656566
```

Hexadecimal, octal, and binary constants may have fractions and exponents too; the exponent of a hexadecimal constant follows `p` and is a power of two, as in C, so `0x1p4` is 16.

String constants

String constants are surrounded in double quotes, e.g. `"hello, world"`. Characters preceded by a backslash stand for themselves, including double-quote (`"`) and backslash (`\`). The following backslashed characters are special:

- `\n` is newline
- `\r` is carriage return
- `\b` is backspace
- `\t` is tab
- `\f` is formfeed
- `\v` is vertical tab
- `\` followed by octal digits is the character with that code, so `"\101"` is `"A"` and `"\0"` is a NUL character
- `\x` followed by hexadecimal digits is the character with that code, so `"\x41"` is also `"A"`

As in C, a numeric escape takes every digit that follows it.

3.2.6. Variables

The name of a variable is replaced by the value of that variable in expressions. If the name is of a pointer, preceding it with the * dereference operator yields the value of its target.

3.2.7. Struct and union references

The construct

str . *name*

yields the element *name* of the struct or union *str*. To retrieve elements from a struct or union pointer, as in C, use

str -> *name*

3.2.8. Array references

array [*expr*]
array [*expr* , *expr* , ... , *expr*]

Elements of an array are indexed with square braces. Elements of multidimensional arrays have indices in each dimension as in the second form. Pointers to arrays can be indexed by placing the * dereference operator between the name of the array and the first square brace:

array *[*expr*]

This is, however, deprecated in favor of using references to arrays, which have no such problems.

3.2.9. The fork operator

The fork operator evaluates its argument in a child thread. See the section on Concurrency.

3.2.10. Comma operator

expr , *expr*

Evaluates each expression in turn, the resulting value is that of the right hand expression. For example,

```
> 1+2, 5!, 3**4, 27/3
9
```

3.2.11. Precedence

From the tightest-binding to the loosest, Nickle's operators group as follows. Operators on the same line have the same precedence.

```
a[i]  a.b  a->b  f(x)
```

```
++  --  -  !  ~  *  &  (prefix)  ++  --  !  (postfix)
(U.tag) a                         (union cast)
**                                (groups from the right)
*  /  //  %
+  -
<<  >>
<  <=  >  >=
==  !=
&
^
|
&&
||
?:
fork
=  +=  -=  and the rest           (group from the right)
,
```

Unlike C, the unary operators bind tighter than `**`, so `-2**2` is 4.

3.2.12. Unicode spellings

Many operators also have an alternate Unicode spelling, which means the same as the ASCII one: `≠`, `≤`, and `≥` for the comparisons, `×` and `÷` for multiplication and division, `⌿` for integer division, and superscript digits for constant exponents, as in `x²` or `2¹⁰`. Nickle prints code the way it was written, and the manual page lists every alternate spelling.

3.3. Statements in Nickle

3.3.1. Simple statements

expr `;`
`; /* null statement */`
`{` *statement ...* `}`

The simplest statement is merely an expression terminated by a semicolon; the expression is evaluated. A semicolon by itself is allowed but does nothing. One or more statements may be grouped inside curly braces to form one compound statement; each is executed in order. Any statements may compose the statement list, including control statements and other curly-bracketed lists.

3.3.2. Conditionals

`if (` *expr* `)` *statement*
`else` *statement*

`if` is used to execute a section of code only under some condition: If `expr` is true, `statement` is executed; otherwise control skips over it. For example:

```
if (x == 0)
    printf("x is zero.\n");
```

In this case, the message will be printed only if `x` is zero.

`else` allows for a choice if the condition fails. It executes its `statement` if the most recent `if` did not. For example,

```
if (x == 0)
    printf("x is zero.\n");
else
    printf("x is not zero.\n");
```

More than one option may be presented by nesting further 'if's in 'else' statements like this:

```
if (x == 0)
    printf("x is zero.\n");
else if (x < 0)
    printf("x is negative.\n");
else
    printf("x is positive.\n");
```

3.3.3. Twixt

`twixt (` *expr* `;` *expr* `)` *statement*

Ensures that the first `expr` will always have been evaluated whenever control flow passes into any part of `statement` and ensures that the second `expr` will be evaluated anytime control flow passes out of `statement`. *That order is guaranteed.* If a `longjmp` target is inside `statement`, the first `expr` will be executed before control passes to the target. If `statement` raises an exception, `longjmp`s out of the `twixt`, leaves it with `break`, `continue`, or `return`, or its thread is killed, the second `expr` will be evaluated. Thus, `twixt` is useful in locked operations where the statement should only be executed under a lock and that lock must be released afterwards.

```
twixt (get_lock(); release_lock())
    locked_operation();
```

3.3.4. Switch

`switch (` *expr* `) { case` *expr* `:` *statement-list* ... `default:` *statement-list* `}`

Control jumps to the first `case` whose `expr` evaluates to the same value as the `expr` at the top. Unlike in C, these values do not have to be integers, or even constant. The optional case `default` matches any value. If nothing is matched and there is no `default`, control skips the `switch` entirely. This example prints out a number to the screen, replacing it by a letter as though it were a poker card:

```
switch (x) {
case 1:
    printf("A\n"); /* ace */
    break;
case 11:
    printf("J\n"); /* jack */
    break;
case 12:
    printf("Q\n"); /* queen */
    break;
case 13:
    printf("K\n"); /* king */
    break;
default:
    printf("%d\n", x); /* numeric */
    break;
}
```

Notice the `break`s in the example. Once control jumps to the matching case, it continues normally: Upon exhausting that `statement-list`, *it does not jump out of the* `switch`; it continues through the subsequent statement lists. Here is an example of this 'falling through':

```
int x = 3;

switch (sign(x)) {
case -1:
    printf("x is negative.\n");
case 1:
    printf("x is positive.\n");
default:
    printf("x is zero.\n");
}
```

This prints:

```
x is positive.
x is zero.
```

Falling through may be desirable if several cases are treated similarly; however, it should be used sparingly and probably commented so it is clear you are doing it on purpose. This is a difficult error to catch.

3.3.5. Union switch

`union switch (` *union* `) { case` *name* `:` *statement-list* ... `default:` *statement-list* `}`

`union switch` is similar to `switch`. It matches the `case` based on what name currently applies to the

union's value. As always, `default` matches everything. The following example chooses the best way to print the union:

```
union {
    int    a;
    string b;
} u;

u.b = "hello";

union switch (u) {
case a:
    printf("%d\n", u.a);
    break;
case b:
    printf("%s\n", u.b);
    break;
}
```

In this case, it prints 'hello'.

An additional name may follow that of a case; the union's value will be available inside the case by that name. The switch above could have been written:

```
union switch (u) {
case a num:
    printf("%d\n", num);
    break;
case b str:
    printf("%s\n", str);
    break;
}
```

3.3.6. Loops

`while (` *expr* `)` *statement*
`do` *statement* `while (` *expr* `);`
`for (` *expr* `;` *expr* `;` *expr* `)` *statement*

`while` executes `statement` repeatedly as long as `expr` is true. Control continues outside the loop when `expr` becomes false. For example:

```
int x = 0;
while (x < 10) {
    printf("%d\n", x);
    ++x;
}
```

This prints the numbers from zero to nine.

`do-while` is like `while`, but tests the condition after each iteration rather than before. Thus, it is guaranteed to execute at least once. The final `;` may be left out. It is often used in input while testing for end-of-file:

```
file f = File::open("test", "r");

do {
    printf("%s\n", File::fgets(f));
} while (!File::end(f));

File::close(f);
```

`for` begins by evaluating the first `expr`, which often initializes a counter variable; since declarations are expressions in Nickle, they may be used here and the counter will be local to the loop. Then it executes `statement` as long as the second `expr` is true, like `while`. After each iteration, the third `expr` is evaluated, which usually increments or decrements the counter variable. The `while` example above can also be written as the following `for` loop:

```
for (int x = 0; x < 10; ++x)
    printf("%d\n", x);
```

3.3.7. Flow control

`continue`
`break`
`return` *expr*
`return`

`continue` restarts the nearest surrounding `do-while`, `while`, or `for` loop by jumping directly to the conditional test. The iterative statement of a `for` loop will be evaluated first.

`break` leaves the nearest surrounding `do-while`, `while`, `for`, or `switch` statement by jumping to its end. The iterative statement of a `for` loop will not be evaluated.

`return` returns from the nearest surrounding function with value `expr`, or with no value from a function of type `void`.

Each of these evaluates the second `expr` of any `twixt` it leaves.

3.3.8. Exceptions

`try`, `catch`, and `raise` handle errors and other exceptional conditions; see the section on Exceptions.

3.4. Nickle Functions

An example function might be declared like this:

```
int
gcf(int a, int b)
{
    int f = 1;

    for (int i = 2; i <= abs(a) && i <= abs(b); ++i)
        while (a // f % i == 0 && b // f % i == 0)
            f *= i;
    return f;
}
```

First comes the return type of the function, then the function name, then the list of arguments (with types), and finally the statement list in curly braces. If any types are left off, Nickle assumes `poly`. Nickle checks types when it compiles the function, and again at runtime where a type isn't known until then, as with `poly`.

A function that just returns the value of an expression can be written with `=` and that expression in place of its body. A declaration that ends with a semicolon instead of a body declares a function to be defined later:

```
int dbl(int x) = x * 2;

int fwd(int x);
```

A function can take any number of arguments. The final argument may be followed by an ellipsis (`...`) to indicate an arbitrary, variable number of succeeding arguments, each of the type of the final argument; the last argument becomes an array to store them.

```
> print sum
int
sum(int a, int b ...)
{
    for (int i = 0; i < dim(b); ++i)
        a += b[i];
    return a;
}
> sum(1, 2)
3
> sum(4)
4
> sum(1, 2, 4, 6)
13
```

Functions are called as in C, with their names followed by argument values in parentheses:

```
foo("hello", 7.2);
```

Since they are first class, functions can be assigned:

```
int(int, int) a = gcf;
```

See Type checking and subtyping in the section on Copy Semantics for details on what functions may be assigned to each other. Functions may also be declared and used anonymously:

```
(int func(int a, int b) = a + b)(2, 3); /* 5 */
(func(poly x) = x * 3)(5);              /* 15 */
```

Replacing the function name with the keyword `func` indicates its anonymity.

Chapter 4. Builtins

This chapter will explain various important builtin functions of Nickle, such as those for input and output and math. It will also discuss the various operators and builtin functions that manipulate strings.

4.1. Input and Output

Input and output in Nickle are mostly accomplished through the File builtin namespace; some top-level builtins refer to those functions. Nickle's input and output are modeled, as much of the language is, on C, but many changes have been made.

4.1.1. Opening and closing files

The functions in the File namespace use the `file` primitive type to describe filehandles. Three are predefined, with their usual meanings: `stdin`, `stdout`, and `stderr`. For many functions in File, there is a top-level builtin which assumes one of these standard streams. Other files may be read and written by opening them:

```
file File::open(string path, string mode)
```

The first string gives the path to the file to be opened; the second is one of:

- `"r"` to open read-only, starting at the beginning of the file.
- `"r+"` to open read-write, starting at the beginning of the file.
- `"w"` to create or truncate the file and open write-only.
- `"w+"` to create or truncate the file and open read-write.
- `"a"` to open write-only, appending to the end of the file.
- `"a+"` to open read-write, appending to the end of the file.

If successful, a filehandle will be returned that can then be used. Otherwise, `open` raises `File::open_error` with a message saying why.

Nickle can also run other programs connected via pipes, reading their output or writing their input; these are also treated as `file`s, and the difference is transparent to the functions that manipulate them. The `Process` library's `popen` runs a program and returns a file connected to it:

```
file Process::popen(Process::popen_direction d, bool nullf, string cmd, string args ...)
```

`d` is `popen_direction.read` to read the program's standard output or `popen_direction.write` to write its standard input. If `nullf` is true, the program's other standard stream is connected to `/dev/null` instead of Nickle's own. `cmd` is the program to run, and `args` are its arguments; by convention, the first should be the name of the program. For example,

```
> autoload Process
> file ls = Process::popen(Process::popen_direction.read, true, "ls", "ls", "-a");
> do printf("%s\n", File::fgets(ls));
```

```
+ while (!File::end(ls));
.
..
bin
man
```

For more control, `File::mkpipe` makes a pipe, returning its read and write ends as an array of two files, and `File::filter` runs a program with three given files as its standard input, output, and error.

When a file is no longer needed, it should be closed.

```
void File::close(file f)
```

```
> File::close(ls);
```

4.1.2. Flush

Output written to a file is not immediately written, but buffered until an appropriate time. Ordinarily, this is not noticed; if, however, it is important to know that all buffers have been written to a file, they can be flushed:

```
void File::flush(file f)
```

4.1.3. End

Returns true if the file is at end-of-file, otherwise returns false.

```
bool File::end(file f)
```

4.1.4. Characters and strings

Individual characters can be read and written using `getc`, `getchar`, `putc`, and `putchar`.

```
int File::getc(file f)
int getchar()
int File::putc(int c, file f)
void putchar(int c)
```

A character can be pushed back onto the stream with `ungetc` or `ungetchar`.

```
int File::ungetc(int c, file f)
void ungetchar(int c)
```

Strings can be read, a line at a time, using `fgets` and `gets`, and written with `fputs`.

```
string fgets(file f)
string gets()
void File::fputs(string s, file f)
```

These are like their C counterparts, with a few differences. At end of file, `getc` and `getchar` raise `File::io_eof` rather than returning a special value, so test with `File::end` before reading. `fgets` and `gets` return the line without its newline, and take no buffer or length. Characters are read and written as described in the following section.

4.1.5. Unicode and characters vs. bytes

Unicode is a standard for representing characters, like ASCII. However, Unicode is designed to be able to support a much larger range of characters; in fact, every character in every alphabet worldwide. It is optimized so standard ASCII characters retain their ASCII codes, and characters are not larger than they have to be. Because of its advantages, and the possibility that it may become more standard than ASCII, and because there is no reason not to, Nickle reads and writes Unicode. This is entirely transparent to the user/programmer.

However, there is one situation in which the programmer will notice: characters outside ASCII are *not* one byte long; they can be as many as four for the really obscure characters. Therefore, unlike in C, *characters cannot be counted on to be the same as bytes*. For this reason, Nickle provides the following functions:

```
int File::putb(int b, file f)
int File::getb(file f)
int File::ungetb(int b, file f)
void putbyte(int b)
int getbyte()
```

These operate the same as `putc`, `getc`, `ungetc`, `putchar`, and `getchar`, but will always read or write one byte at a time, regardless of character representation.

4.1.6. Formatted I/O

Nickle provides functions such as `printf`, `sprintf`, and `scanf` to perform formatted input and output. These functions perform like their C counterparts, with the following exceptions:

- The precision of a field in the format string may be specified to be `-`, which means arbitrary precision. The repeating digits of a fraction are then shown in curly braces, and an imprecise value is shown with just enough digits to identify it exactly:

  ```
  > printf("%.-f %.3f %.-g\n", 1/7, 1/7, imprecise(0.1, 53));
  0.{142857} 0.143 0.1
  ```

- The `%g` format specifier prints a number in the best way possible, and any other value as `%v` does. For example:

  ```
  > printf("%g %g %g\n", 1, 1/3, sqrt(2));
  1 0.333333333333333 1.414213562373095
  ```

- The `%v` format specifier will attempt to find the best way to print whatever value it is given. This

is a great way to print polys whose types will not be known ahead of time.

```
> printf("%v %v %v\n", 1/3, "hello", fork 4!);
(1/3) "hello" %38
```

Notice that it can even figure out difficult things like the thread returned by `fork`.

- `scanf` stores what it reads through pointers, as C's does:

```
> int n;
> scanf("%d", &n);
```

4.1.7. At the top level

Many functions in the File namespace have counterparts builtin at the top level; these do not need to be imported from File because they are automatically present.

- `void printf(string fmt, poly args...)` and `string sprintf(string fmt, poly args...)`, from `File::PrintfGlobals`, are the same as `File::fprintf` to `stdout` and `File::sprintf`.
- `int scanf(string fmt, *poly args...)`, from `File::ScanfGlobals`, is the same as `File::fscanf` from `stdin`.
- `getchar`, `ungetchar`, `putchar`, `getbyte`, `putbyte`, `fgets`, and `gets`, from `File::FileGlobals`, read and write `stdin` and `stdout`, except `fgets`, which reads the file it is given.

The other functions in File must be called with `File::` or imported as normal.

4.2. Math

4.2.1. Numbers

The four numeric types in Nickle—int, rational, real, and complex—have a hierarchical relationship. Specifically, int is a subset of rational, which is a subset of real, which is a subset of complex. Ints and rationals are stored internally in arbitrary precision, and printed as decimals rounded to 15 digits by default. Rational constants may give a repeating portion in curly braces, and printing with arbitrary precision shows it the same way; see the section on Expressions for a discussion of rational constants. Reals are stored in finite, floating-point representations. The mantissa of a new imprecise value is `float_precision` bits long, 256 to start with; assigning that global changes it, and `imprecise(x, p)` makes a value of `p` bits, which `precision(x)` reports.

```
> imprecise(1 / 3, 20)
0.33333349
> precision(sqrt(2))
256
```

Whenever performing calculations, Nickle will keep numbers in their most specific format. For

example, the result of `4/2` is an int, because although the result (2) is a rational, it is also an int, and int is more specific. Similarly, reals are not always in imprecise floating representation; if they are known exactly, they will be represented as rationals or ints. Nickle will only produce imprecise reals when it has to, as in square roots and logarithms.

4.2.2. Operators

In order to do the Right Thing for a desk calculator, Nickle provides several operators that are not present in C; these are extremely useful. To force division to produce an integer, even if the result would be a rational, use the `//` integer divide operator, which always produces an int: the floor of the quotient when the divisor is positive, and its ceiling when the divisor is negative, so the remainder from `%` is never negative. For example, `-7 // 2` is -4 and `-7 % 2` is 1. Nickle also has an exponentiation operator `**`, which behaves correctly for all exponents, including negative and fractional. Therefore, `sqrt(x)` is the same as `x**.5`, and `1/x` is the same as `x**-1`. Finally, it provides a factorial operator `!`.

4.2.3. The Math namespace

Nickle provides the builtin namespace Math for useful functions such as trigonometric functions, logarithms, as well as useful constants such as `pi` and `e`. Besides those below, it has `exp`, `pow`, `sqrt`, `cbrt`, `atan2`, the hyperbolic functions `sinh`, `cosh`, `tanh`, and their inverses, `gamma`, `erf`, `erfc`, the Bessel functions `j0`, `j1`, `jn`, `y0`, `y1`, and `yn`, and `choose`. The Complex namespace holds the corresponding functions for complex numbers.

Logarithms

```
real log(real a)
real log10(real a)
real log2(real a)
```

The logarithm of `a` in base e, ten, and two, respectively.

```
> log(Math::e)
1
> log10(16) / log10(4) /* change of base formula, log_4 16 */
2
> log2(16)
4
>
```

Trigonometric functions

```
real sin(real a)
real cos(real a)
real tan(real a)
real asin(real a)
real acos(real a)
real atan(real a)
```

The sine, cosine, and tangent of `a`, and the inverse functions.

```
> sin(pi) ** 2 + cos(pi) ** 2
1
> atan(1) * 4
3.141592653589793
>
```

Constants

```
protected real e
real pi
```

`pi` and `e` define the usual constants (3.14..., 2.72...). `e` is protected and must be called `Math::e` to allow ordinary use of the name `e`.

4.3. Strings

Unlike in C, strings in Nickle are not arrays of or pointers to individual characters. Consistent with its pattern of providing primitive datatypes for things that make sense (e.g. `file` instead of integer file handles), Nickle provides the `string` type. This has several interesting differences from C-style strings:

- In Nickle, strings are immutable—individual characters may not be changed.
- Strings hold Unicode characters, not bytes, and may contain any of them, including NUL.

4.3.1. Operators

Several operators have been overloaded to allow sane manipulation of strings: `+`, array subscript (`[]`), and the comparison operators.

Subscripting

Although they are not arrays of characters, it is often useful to access a string a character at a time; the array subscript operator has been overloaded to allow this. For example:

```
> string s = "hello, world";
> s[0]
104
> s[1]
101
> s[2]
108
> s[3]
108
> s[4]
111
```

```
>
```

Those are the Unicode code points of each character, which for ASCII characters are their ASCII codes. Each element is a whole character, however many bytes it takes; see Unicode and characters vs. bytes in the I/O section. So `"héllo"[1]` is 233, and the string's length is 5. The String namespace provides `new` to create a string from these code points:

```
string new(int c)
string new(int[*] cv)
```

For instance,

```
> String::new(s[0])
"h"
```

Concatenation

On strings, `+` is the concatenation operator. For example,

```
> string s = "hello", t = "world";
> s = s + ", ";
> t += "!";
> s+t
"hello, world!"
```

Comparison

`==`, `!=`, `<`, `<=`, `>`, and `>=` compare strings character by character, as C's `strcmp` does:

```
> "abc" < "abd"
true
> "b" > "abc"
true
```

4.3.2. String namespace

In addition, the String namespace provides several useful functions that facilitate using strings, including the following.

Length

```
int length(string s)
```

Returns the number of characters in `s`. For example,

```
> String::length("hello, world")
```

```
12
>
```

Index

`int index(string t, string p)`
`int rindex(string t, string p)`

Returns the index of the first occurrence of the substring `p` in `t`, or -1 if `p` is not in `t`; `rindex` returns the last occurrence instead. For example,

```
> String::index("hello, world", "or")
8
> String::index("hello, world", "goodbye")
-1
> String::rindex("hello, world", "o")
8
```

Substr

`string substr(string s, int i, int l)`

Returns the substring of `s` which begins at index `i` and is `l` characters long. If `l` is negative, returns the substring of that length which precedes `i` instead. For example,

```
> String::substr("hello, world", 8, 2)
"or"
> String::substr("hello, world", 8, -4)
"o, w"
>
```

Other functions

The String namespace also provides `split` and `wordsplit`, which break a string into an array of strings at separator characters; `chomp`, which trims white space from both ends, and `chump`, which trims trailing newlines; `reverse`; `dirname`, which returns the directory part of a path; and `parse_csv`, which splits a line of comma-separated values.

```
> String::split("a,b,c", ",")
(string [...]) {"a", "b", "c"}
> String::chomp("  hi  ")
"hi"
```

Chapter 5. Advanced topics

This chapter will discuss more advanced topics; these features make Nickle as powerful as it is. The semantics of copying and garbage collection, namespaces, exceptions, threading and mutual exclusion, and continuations will all be covered.

5.1. Copy Semantics and Garbage Collection

5.1.1. Copy by value

In Nickle, assignment, argument passing, and definitions—in short everything involving the values of variables—are all by-value. Nickle avoids the weird C-isms, like being by-value except for arrays and strings. Everything is copied. Consider the following example:

```
> int[*] foo = { 1, 2, 3 };
> int[*] bar = foo;
> foo[2] = 4;
> foo
(int [3]) {1, 2, 4}
```

What will `bar[2]` be?

```
> bar
(int [3]) {1, 2, 3}
```

Since assignment is by-value, bar has its own values—it is unchanged. Also consider function arguments:

```
> (void func(int[*] a) { a[0] = 9; printf("%v\n", a); })(foo);
(int [3]) {9, 2, 4}
```

Does `foo` still have its original value, or the one with 9? Since the function was modifying a copy—which was passed by-value--`foo` will be unchanged.

```
> foo
(int [3]) {1, 2, 4}
```

What if you want to pass something by reference? Nickle has a reference type to accomplish just that. (You could also use pointers, but references are The Right Way.) For example, to reimplement the example above using references:

```
> (void func(&int[*] a) { a[0] = 9; printf("%v\n", a); })(&foo);
(int [3]) {9, 2, 4}
```

```
> foo
(int [3]) {9, 2, 4}
```

Notice that `foo` was changed; it was passed as a reference (`&int[*]`). See References in the section on Nickle Datatypes for a discussion of references.

Strings can't be changed at all, only replaced, so they behave the same whether they are copied or shared.

5.1.2. Garbage collection

But if all those arrays and structs are copied entirely every time a function is called or an assignment made, won't there be a lot of unused, unreferenceable memory lying around? No. Nickle is fully garbage-collected; when a value no longer has any names, it is freed. This is invisible to the user/programmer, who need not worry about allocation, deallocation, or any other aspects of their assignments and argument passing.

In short, everything is by-value, and Nickle takes care of allocation and deallocation.

5.1.3. Type checking and subtyping

Type checking in Nickle is a combination of compile-time and runtime checking. At compile-time, Nickle will ensure that all assignments, argument passing, and other copying situations are sane, for instance that no strings are being assigned to integers. It will let some errors through if it cannot be sure they are errors. For instance, variables of type `poly` can hold any type; at compile-time, nothing is ruled out, but this does not mean you can't break typing at run-time.

At runtime, Nickle makes sure all assignments are actually valid. It does so by determining if one type is a subtype of the other, i.e. if the set of all values that can fit in it also fit into the other. As a concrete example:

```
> int i = 1;
> rational r = i;
> i = r/3;
Unhandled exception invalid_argument ("Incompatible types in assignment", 0,
0.333333333333333)
<stdin>:3:     i = r / 3;
```

The int can hold the integer value 1 without difficulty, because they are the same type. The rational can accept the same value because integers are a subset of rationals. However, attempting to assign a rational (1/3) to the integer raises an exception. This demonstrates that int is a subtype of rational; conversely, rational is the supertype of int. A variable can take on a value from any of its subtypes, but not from its supertypes—and if the two values do not share a sub/supertype relationship, they will not get past the compile-time checker.

A similar check occurs with structs. If one struct's elements are a subset of another's, it may take that value. For example,

```
> typedef struct { int i; string s; } i_and_s;
> typedef struct { int i; } just_i;
> i_and_s is = { i=2, s="hello" };
> just_i i = is;
> just_i ji = { i=2 };
> is = ji;
Unhandled exception invalid_argument ("Incompatible types in assignment", 0, {i = 2})
<stdin>:6:     is = ji;
```

Since `i_and_s` is a subtype of `just_i` (it has `i` and also `s`), the assignment to `i` from `is` worked. However, attempting to assign to `is` from a `just_i` failed, because it did not have an `s` to copy over.

Finally, a function can be assigned to a variable of function type when each of its argument types and its return type is a subtype or supertype of the variable's; functions with unrelated types are rejected at compile time. When such a function is called, its arguments are checked against its own argument types, and its result is checked wherever it is stored or returned.

5.2. Nickle Namespaces

Namespaces collect related variable and function names and allow control over visibility. A number of Nickle builtins are gathered into builtin namespaces that may be used. The following builtin namespaces have sections in this tutorial:

`Math`
: Useful mathematical functions.

`File`
: File input/output with the `file` type.

`Thread`
: Concurrent processing.

`Semaphore`
: Synchronization of threads.

`String`
: Useful functions for strings.

Other namespaces come from libraries that are loaded when needed, such as `Mutex` for mutual exclusion and `Process` for running other programs; see Loading libraries below.

An example namespace might be declared like this:

```
namespace Example {
    int        blah = 1;
    public int a = 0;

    int
```

```
    bar(int a)
    {
        ...
    }

    protected int
    foo(int a)
    {
        ...
    }
}
```

The keyword `namespace` is followed by the name of the namespace and a list of statements that declare names in the namespace. The publication of those declarations, e.g. `public` or `protected` defines how visible they will be outside the namespace. The namespace itself may be preceded by publication information, *but this has no bearing on the names within the namespace*; it defines the visibility of the name of the namespace. If the example above had been declared

```
protected namespace Example {
    ...
}
```

Then the names within `Example` would have the same visibility as always, but `Example` itself would be protected in whatever namespace it belongs to. In this case, it belongs to the top-level namespace, but namespaces can be nested within each other, which makes the visibility of their own names important.

5.2.1. Extend

`extend namespace` *name* `{` *statement-list* `}`

Names may be added to a namespace after it is initially defined with the `extend` command. The namespace `name` is reopened and the new `statement-list` is added to the previous ones. For example,

```
extend namespace Example {
    string[*] greeting = { "hello", "world" };
}
```

Adds `greeting` to the names already defined in `Example`.

5.2.2. Peering inside

namespace `::` *name*
`import` *namespace*

The `::` operator refers to a `name`, which is in `namespace`, analogously to a structure dereference. If

`name` also refers to a namespace, its names too are visible this way. Either `protected` or `public` names are visible in this way.

An `import` statement brings all the public names in `namespace` into scope, overshadowing conflicting names. Thereafter, those names may be used normally, and a name it overshadowed can still be reached with `::`. A namespace inside another is imported by its full name, as in `import Example::Inner`.

A variable is declared with one of three visibilities that defines how it is visible outside its namespace:

- `public` may be seen outside with `::` or imported
- `protected` may be seen outside with `::` but not imported
- if neither is specified, it may not be seen outside at all

Thus, in our example namespace `Example`:

- `blah`, `bar`, and `greeting` have no visibility specified and may only be used inside `Example`.
- both `a` (which is public) and `foo` (which is protected) may be seen with `::`.
- an `import` will only bring `a` into scope, as it is the only name that is public.

5.2.3. Loading libraries

`autoload` *namespace* `,` ...
`autoimport` *namespace* `,` ...
`reimport` *namespace* `,` ...

Many namespaces live in library files, loaded only when a program asks for them. `autoload` loads the library defining each namespace named, unless the namespace is already defined; `autoimport` also imports it, and `reimport` loads it again even if it is already defined. The library's file name is the namespace's name in lower case, with a dash before each capital letter that follows a lower-case one, and `.5c` appended, so `Mutex` comes from `mutex.5c`. The `library` command loads a file from the library path by its name. See Commands.

```
autoimport Mutex

mutex m = new();
```

5.3. Nickle Exceptions

Nickle has first-class exceptions for error handling and quick escapes from recursive algorithms. A number of exceptions are builtin to Nickle that it throws for various errors, including:

`exception uninitialized_value(string msg)`

Attempt to use an uninitialized value.

`exception invalid_argument(string msg, int arg, poly val)`

The argument with index `arg` to a builtin function had invalid value `val`; indices count from zero, so the first argument is 0. The interpreter's own checks, such as those of assignments, also report 0.

`exception readonly_box(string msg, poly val)`

Attempt to change the value of a read-only quantity to `val`.

`exception invalid_array_bounds(string msg, poly a, poly i)`

Attempt to access array `a` at index `i` is out of bounds.

`exception divide_by_zero(real num, real den)`

Attempt to divide `num` by `den` when `den` is zero.

`exception invalid_struct_member(poly str, string name)`

Attempt to refer to member `name` of the object `str`, which does not exist.

`exception invalid_binop_values(string msg, poly arg1, poly arg2)`

Attempt to evaluate a binary operator with arguments `arg1` and `arg2`, where at least one of these values is invalid.

`exception invalid_unop_value(string msg, poly arg)`

Attempt to evaluate a unary operator with invalid argument `arg`.

`exception out_of_memory(string msg)`

A result is too large to represent.

The following syntax may be used to declare a new exception:

`exception` *name* `(` *type name* `,` ... `)`

For example,

```
exception my_exception(string msg, int a, int b, int c);
```

5.3.1. Raise

`raise` *name* `(` *value* `,` ... `)`

Raises the named exception with the given arguments, e.g.

```
raise my_exception("message", 0, 1, 2);
```

Execution is broken and `my_exception` travels up the stack until it is caught by a try-catch block or it reaches the top level, where it prints an error message, with the line that raised it, such as:

```
Unhandled exception my_exception ("message", 0, 1, 2)
```

```
example.5c:2:     raise my_exception("message", 0, 1, 2);
```

When Nickle reads commands from a terminal, an unhandled exception raised inside a function then starts the debugger. Otherwise, Nickle exits with status 1.

5.3.2. Try - catch

`try` *statement*
`catch` *name* `(` *type name* `,` ... `)` `{` *statement-list* `}`

`try` executes `statement`; if it raises an exception whose name matches that of a succeeding `catch` block, the arguments are placed in the names specified and the associated `statement-list` is executed. The types may be left out; when given, they must match the exception's declaration. The name may be qualified with a namespace, as in `catch Mutex::deadlock (m, t)`, and if several catch blocks match, the first one is used. Control continues after the catch without continuing up the stack; if further propagation is desired, `statement-list` should re-raise the exception. Any number of catch blocks may be associated with a try statement. For example:

```
exception my_exception(string msg, int a, int b, int c);

try
    raise my_exception("blah", 1, 2, 3);
catch my_exception(string msg, int a, int b, int c) {
    printf("%s: exception successfully caught (%d,%d,%d).\n", msg, a, b, c);
}
```

This example raises an exception; since that exception matches the catch block, "blah", 1, 2, and 3 (the arguments) are put into `msg`, `a`, `b`, and `c` and the statement list is executed, which in this case merely prints out the arguments received and continues:

```
blah: exception successfully caught (1,2,3).
```

5.3.3. Twixt

`twixt` `(` *expr_enter* `;` *expr_leave* `)` *statement*

Nickle does not provide a `finally` clause to a `try-catch`. In order to ensure the order of some expressions, it provides `twixt` (See the section on Statements); its leave expression is evaluated when an exception, `break`, `continue`, `return`, or `longjmp` leaves its statement. For example,

```
exception my_exception(string msg, int a, int b, int c);

void
foo(string msg, int a, int b, int c)
{
    twixt (printf("entering twixt..."); printf("leaving twixt.\n"))
```

```
        raise my_exception(msg, a, b, c);
}

try
    foo("blah", 1, 2, 3);
catch my_exception(string msg, int a, int b, int c) {
    printf("%s: exception successfully caught (%d,%d,%d).\n", msg, a, b, c);
}
```

Will produce the output:

```
entering twixt...leaving twixt.
blah: exception successfully caught (1,2,3).
```

Notice the order of the printed messages: `twixt` finished up before the exception was handled by the `catch`. This is an elegant way to accomplish something that should be done finally, in this case printing the message `leaving twixt` for demonstration.

5.4. Threads and Mutual Exclusion in Nickle

5.4.1. Basic threading

Threads provide concurrent processing of calculations. They are created with the `fork` operator, which spawns a child thread to evaluate its argument and returns it as a variable of the first-class type `thread`:

`fork` *expr*

The thread it returns is typically stored like this:

```
thread t = fork x!;
```

In the above example, `fork` immediately returns a thread, which is stored in `t`. That thread will calculate the factorial of `x` in the background while the program continues; when the calculation is finished, the thread ends, keeping its value until another thread retrieves it with `join`.

Threads share names; if a thread changes the value of a variable, that change will occur in the other threads as well. See Mutual exclusion below.

5.4.2. Thread functions

The builtin namespace `Thread` has functions for manipulating threads once they have been forked.

`int kill(thread t ⋯)`

Kills the threads it takes as arguments, or the current thread if it is given none. A killed thread evaluates the leave expressions of any `twixt` statements it is in, as an exception would, and then

finishes. `kill` returns the number of those threads that weren't already finished or being killed.

`poly join(thread t)`

Waits for thread `t` to finish and returns the value of its expression, or `<>` if it was killed. This is how to get the value back out of a thread, and it may be done any number of times. For example,

```
thread t = fork 1000!;

/* something else... */
printf("1000! = %d\n", Thread::join(t));
```

will execute `something else` while `t` runs, then wait for it to complete and print out its value.

`thread current()`

Returns the currently running thread. `kill(current())` ends the current thread, like `kill()`, and `join(current())` raises `invalid_argument`, since a thread can't wait for itself.

`int set_priority(thread t, int i)`
`int get_priority(thread t)`

Priorities determine how runtime is divided among threads; a thread with higher priority will always run before one with a lower priority. `set_priority` sets the priority of `t` to `i` and returns the new priority. `get_priority` returns the priority of thread `t`.

5.4.3. Mutual exclusion

Consider the following situation:

```
import Thread;

void
para()
{
    printf("My next statement will be false.\n");
}

void
dox()
{
    printf("My previous statement was true.\n");
}

thread t = fork para();
thread s = fork dox();
join(t);
join(s);
```

When run, this prints out the less than clear message

```
MMyy  nperxetv isotuast esmteantte mweinltl  wbaes  ftarlusee..
```

Why? Because the two threads are running simultaneously and take turns printing out their messages; the result is that they are interleaved unreadably. The solution is in the `Mutex` namespace, which comes from a library loaded with `autoimport Mutex`. A mutex is an object which threads can use to coordinate conflicting sections of code. `Mutex` defines the following functions:

`mutex new()`

Creates a new mutex and returns it.

`bool acquire(mutex m)`
`bool try_acquire(mutex m)`

`acquire` blocks until `m` is free, then locks it and returns true. At the top of the conflicting code, each thread should acquire the mutex; since only one at a time can have it, they will take turns executing. `try_acquire` instead returns false immediately if `m` is in use. A thread acquiring a mutex it already owns raises `Mutex::deadlock`.

`void release(mutex m)`

When the thread which owns a mutex leaves the conflicting section of code, it should call release to free it for the next thread to acquire it. Releasing a mutex the thread doesn't own raises `Mutex::notowned`. Acquiring and releasing in a `twixt` statement makes sure the mutex is released however the code leaves.

`mutex_owner owner(mutex m)`

Returns the owner of `m`, a `mutex_owner` union: `owner` holding the thread which currently owns it, or `nobody` if it is free.

Each top-level command runs in a thread of its own, so a mutex acquired by one command isn't owned by the next.

An example

This is how the example above might work with mutual exclusion:

```
autoimport Mutex
import Thread;

mutex  m = new();

void
para()
{
    acquire(m);
    printf("My next statement will be false.\n");
```

```
    release(m);
}

void
dox()
{
    acquire(m);
    printf("My previous statement was true.\n");
    release(m);
}

thread t = fork para();
thread s = fork dox();
join(t);
join(s);
```

This prints out, as expected,

```
My next statement will be false.
My previous statement was true.
```

5.4.4. Semaphores

Nickle also has counting semaphores, implemented in the Semaphore namespace. Semaphores are similar to mutexes, but have some number of threads that may run that isn't necessarily one, as it is with mutexes. A semaphore with a count of one behaves just like a mutex.

```
semaphore new(int c)
```

Semaphores are created with `new`, which is unlike `Mutex::new` in that it takes an argument: the number of threads it will run simultaneously.

```
void wait(semaphore s)
void signal(semaphore s)
```

`wait` decrements the count of `s`, which starts with the initial value specified by `new`. If the count, after the decrement, is positive or zero, the thread continues to run; if it is negative, it blocks until a `signal` releases it. `signal` increments the count of `s` and wakes up the thread that has waited longest, if any are waiting. `Semaphore::test` waits only if that wouldn't block, returning whether it did.

Negative initial counts

If `new` is called with a zero or negative initial count `c`, every `wait` blocks until a `signal` releases it, and a `wait` proceeds without blocking only after 1 - `c` more signals than waits have been made.

Why semaphores?

Semaphores are useful in situations where several threads can run simultaneously, but not more than a certain number. They would be great, for instance, to work in a licensing system, where each

thread needs some command, but only a certain number may run at a given time.

Be careful

Semaphores, unlike mutexes, are very error-prone. They are not `owned`, in the sense that mutexes are, and therefore do not check what threads are signalling or waiting on them. Thus, situations like this are possible:

```
> import Semaphore;
> semaphore s = new(3);
> s
semaphore 1 (3)
> wait(s);
> s
semaphore 1 (2)
> wait(s);
> s
semaphore 1 (1)
> for (int i = 0; i < 100; ++i)
+   signal(s);
> s
semaphore 1 (101)
> wait(s)
> s
semaphore 1 (100)
>
```

Therefore, code must be written carefully so that threads do not signal the semaphore more than once, and only once they have waited on it.

5.5. Nickle Continuations

```
poly setjmp(&continuation c, poly retval)
noreturn longjmp(continuation c, poly retval)
```

Arbitrary flow control is accomplished in Nickle with first-class continuations and the functions `setjmp` and `longjmp`. These are similar to those in C, but without restrictions on the target.

Setjmp saves the state of the program, including program counter and names in scope, in `c` and returns `retval`.

Longjmp returns `retval` from the setjmp that set *c*. There can be two distinctions from this jump and the initial call to setjmp: the return value may differ, and variables that have changed retain their new values. Longjmp evaluates the leave expression of each `twixt` it jumps out of and the enter expression of each it jumps into.

Continuations are often used to implement control structures that do not exist in the language, interpreters, and escaping from recursive algorithms. For example, the following is a simple binary tree search that uses continuations to jump directly to the top instead of returning up each branch

once the result is found.

```
typedef struct {
    int   key;
    poly  data;
    &poly left, right;
} tree;

void
_search(tree t, int key, &continuation c)
{
    if (key < t.key && !is_void(t.left))
        _search(t.left, key, &c);
    else if (key > t.key && !is_void(t.right))
        _search(t.right, key, &c);
    else if (key == t.key)
        longjmp(c, t.data);
}

poly
search(tree t, int key)
{
    continuation c;
    poly         p = setjmp(&c, <>);

    if (is_void(p))
        _search(t, key, &c);
    return p;
}

tree t = {
    key = 2, data = "blah", left = reference(<>), right = reference(<>) };

search(t, 2) /* "blah" */
```

This is a pretty normal binary tree search, but notice how it is run: a continuation is set; if setjmp returns `<>` (which it will the first time), a value is searched for. If an actual value is returned, it must be from the longjmp in `_search`, and the value is returned. This optimizes the return from what can be a very deeply nested search.

This sort of escape from a nested search can also be done with exceptions, raising one when the value is found and catching it at the top, passing the value as an argument to the exception.

Colophon

The cover photograph, US Five Cent Coin 1979 Obverse, is by Slashme, from Wikimedia Commons, https://commons.wikimedia.org/wiki/File:US_Five_Cent_Coin_1979_Obverse.jpg, used under the Creative Commons Attribution-ShareAlike 3.0 Unported license, https://creativecommons.org/licenses/by-sa/3.0/.