Nickle: Language Principles and Pragmatics*

Bart Massey
Computer Science Department
Portland State University
Portland, Oregon USA 97207–0751
bart@cs.pdx.edu, http://www.cs.pdx.edu/~bart

Keith Packard
SuSE Inc.
keithp@suse.com, http://keithp.com

Abstract

Nickle is a vaguely C-like programming language for numerical applications, useful both as a desk calculator and as a prototyping and implementation language for numerical and semi-numerical algorithms. Nickle abstracts a number of useful features from a wide variety of other programming languages, particularly functional languages. Nickle's design principles and implementation pragmatics mesh nicely to form a language filling a useful niche in the UNIX software environment. The history of Nickle is also an instructive example of the migration of an application from an idea to a freely-available piece of software.

1 Introduction

The past 20 years have seen an explosion of "utility-belt programming languages". Often implemented as true or byte-code interpreters and designed to operate smoothly in the UNIX environment (in the spirit of `sed` [McM78], AWK [AWK88], `bc` [MC78a], `dc` [MC78b], and the like), these languages are intended both to address specific classes of tasks and to be usable for general-purpose programming. Other examples include Perl [WCS96], Python [Lut01], Java [GJSB00], various Scheme [CE92] implementations, and ML [MTH90].

During this time frame, the authors have been intermittently involved in the development of a utility-belt programming language initially tailored to scratch-pad-style numerical calculation, and reflecting design principles including:

- Simplicity of design and implementation.
- Separability of concerns, such that language features can be implemented and used independently.
- Use of best-practice programming language technologies.
- Practical problem-solving utility.

The result is a language with a low learning curve for experienced UNIX programmers that allows the integration of offline programs with online calculations in a flexible yet safe notation.

2 Programming With Nickle

A small example (Figure 1) may help give a feel for the Nickle language. Note that the variables `i` and `t` are declared at first use. The language feels much like C, but with some of the bothersome details of declaration and typing optional.

This much code would typically be placed in a file, and read into a running Nickle session ("`> `" is the default Nickle prompt here and throughout)

```
> load "countsum.5c"
```

During reading (which occurs with no perceptible delay) the input file is incrementally compiled into the running session. We can then interactively create a sample array to work with.

*To appear. Proceedings of the 2001 Usenix Technical Conference, FreeNIX Track.

```
function countsum(c, v, n) {
  if (c < 0)
    return 0;
  int t = 0;
  for (int i = 0; i < n; i++)
    t += v[i];
  if (t == c)
    return 1;
  if (t < c)
    return 0;
  return countsum(c, v, n - 1) +
   countsum(c - v[n - 1], v, n - 1);
}
```

Figure 1: Function returning the number of ways that the first `n` elements of the array `v` can be added to produce `c`.

```
> v = [5]{1, 2, 3, 4, 5}
1 2 3 4 5
```

The square brackets are the array constructor, and in this case create a dynamically-typed array of 5 elements. The curly braces, as in C, are being used to surround an initializer list which values the array elements `a[0] = 1` ... `a[4] = 5`. Now we can invoke `countsum()` (which is extracted from a Cribbage scoring program).

```
> countsum(15, v, dim(v))
1
> countsum(12, v, dim(v))
2
```

If a new definition is given interactively for `countsum()`, it will scope out the current definition. As expected, the storage for `v` is allocated automatically, and released when `v` becomes unreachable.

3 Language Features

Nickle strives to be a simple yet expressive programming language: always a difficult goal. A reasonable number of simple language features may be combined in powerful ways to solve problems.

3.1 Names, Lifetimes, Types and Values

Nickle has a C-like syntax and procedural imperative semantics. In addition to borrowing from other imperative languages like Java and Modula-3 [Nel91], Nickle incorporates a number of useful notions from the functional programming community.

3.1.1 Names

Nickle supports first-class function values with full static scope. The visibility of a name is controlled by its scope, and by its relationship to its namespace. Nickle namespaces are similar to Java modules, except that they are purely syntactic rather than having anything to do (in principle or in current practice) with the filesystem. The closest notion is probably namespaces in C++. In brief, a namespace is opened by a `namespace` declaration. Each name declared in a namespace may have one of three possible visibilities:

- `public` The name should be visible outside the current namespace, and should be automatically imported.
- `protected` The name should be accessible from outside the namespace via an explicit path, but should not be made directly available by import declarations.
- *(no keyword)* The name should be completely inaccessible outside the namespace.

Names in a namespace may be referenced in one of two ways. First, `public` and `protected` names may be accessed via an explicit namespace qualifier `::`. Thus, the name `Foo::bar` refers to the name `bar` in the namespace `Foo`. Second, the `public` names in a namespace may be brought into unqualified scope using an `import` declaration. For some example uses of namespaces see Figure 3 below and the Appendix.

3.1.2 Lifetimes

There are three options for the lifetime of a Nickle name, as opposed to C's `auto` and `static`. This extra control is a natural consequence of the distinction between global functions and nested functions, which is lacking in C. It manifests noticeably in the times at which initialization occurs. The three lifetimes for a Nickle declaration are:

- `global` The lifetime of the named value is the lifetime of the interpreter evaluating the program. Objects declared `global` are initialized once, when the definition of the function containing them is first encountered.

```
int() function f () {
  int function g () {
     global x = 0;
     return ++x;
  }
  return g;
}

> g = f();
> g()
1
> g()
1
> g = f();
> g()
1
```

Figure 2: A variable `x` with `auto` scope.

`static` The lifetime of the named value is the lifetime of the function in which the definition occurs. Objects declared `static` are initialized whenever the function containing their definition is evaluated. In the absence of nested scopes, this lifetime is the same as `global`.

`auto` The lifetime of the named value is the lifetime of the current function. Objects declared `auto` are initialized whenever their definition is evaluated.

The default lifetimes are as in C: `global` for top-level objects, and `auto` for those local to a function. Note that lifetime is different than scope: different function definitions, for example, may have different global objects named `x`.

Figure 2 is useful illustrating the difference between `auto`, `global` and `static` scope. Imagine that the `auto` declaration of Figure 2 was instead a `static` declaration. In this case, `x` would be initialized whenever the definition of `g` was evaluated, and thus the second invocation of `g()` at top-level would return 2. If `x` was instead declared `global`, it would be initialized only once, when the definition of `f` was compiled, and thus the successive invocations of `g()` would return 1, 2, and 3.

3.1.3 Types

Nickle has two sets of type consistency rules. First, any object in the program may be statically typed. The static type system strongly resembles that of C or Java. Syntactically, Nickle types are required to be like Java's optional "left-hand" syntax: type declarations appear to the left of program objects, and are modified type-by-example. Semantically, there are subtle but significant differences between the type systems of these languages. Nickle allows objects to be explicitly typed `poly`, indicating that the type need not be statically checked. Unlike C, arrays with different sizes are not of different type. Nickle has more types than C, including a disjoint union type and multi-dimensional array types. Structure types obey a subtype relationship over their members. The `void` type is handled slightly differently than in C: in Nickle, there is a single value of type `void`, written as `<>`, allowing the `void` type to interact more smoothly with the rest of the language without significant loss in security. All of this adds up to a type system which provides full security while retaining reasonable expressiveness.

Secondly, all operations are currently checked for type consistency (as well as performing array bounds checking and the like) at runtime. While in principle many of these runtime checks could be removed by static type checking, and others could be hoisted in order to improve performance, runtime type checking is not currently believed to be a performance bottleneck, and the implementation is greatly simplified by this choice. The combination of strong static checking and complete runtime checking does normally mean that program defects will be caught "as early as possible", providing confidence in execution correctness and aiding debugging.

3.1.4 Values

Nickle supports first-class structured values. Any value of any type may be created and used in any legal expression context. This is important for a language designed for a desk calculator. It also allows the programmer to determine which values in the program need to be named, instead of being forced to produce names by the language semantics.

For example, an anonymous array can be created whenever needed, and treated just like any other array object. In this example, a two-element string array is created and then dereferenced:

```
> ((string[*]){"a","b"})[1]
"b"
```

3.2 Implementation Features

The current implementation of Nickle is an interactive byte compiler, in the style of many Scheme implementations. All expressions and statements typed at the command line are rapidly compiled into an intermediate byte-code representation that is then evaluated in the compiled top-level context. The language is designed to support implementations using offline compilation to native code, for greater efficiency. While Nickle's predecessors were at one time pure interpreters, the current structure of the language would make such an implementation difficult, and offers no obvious advantages.

The statement syntax of Nickle, as with C, is essentially that of infix expressions, lending itself to scratch-pad-style calculation. Full interactive compilation and first-class support for all values means that it is quite easy to interact with and modify Nickle code written online, to develop new code online, and to calculate interactively in this environment. The easy loading and import of appropriate namespaces allows a custom calculation environment to be quickly set up.

Many of the features of Nickle are predicated upon the existence of automatic storage management. The Nickle implementation includes a tracing mark-sweep garbage collector, which is used in two principal ways. First, storage is implicitly allocated for Nickle objects during definition. Second the C code that implements Nickle makes use of the garbage collector as a general-purpose storage allocator. This is facilitated by the non-copying nature of the collector, and by CPP macros which allow easy rooting of the garbage collector in a C frame.

3.3 Numeric Features

To some degree, the reason for the existence of Nickle is the support for infinite-precision integer and rational arithmetic. By default, non-integer quantities are represented as rationals, which ensures that precision will never be lost by the computation. For convenient comparison and for familiarity's sake, rationals behave syntactically like floating-point numbers: they are input and output using a floating decimal point.

To avoid loss of precision due to decimal conversion, both output and input representations of rational numbers support "repeating decimals". For example, the constant `0.1{6}` is the Nickle decimal representation of 1/6. Using sophisticated techniques based on a factored representation, Nickle is capable of calculating the minimum-length repeat for the decimal representation of an arbitrary rational number. This can be very expensive in some cases, however, so by default repeating representations beyond a certain number of digits will be truncated. While this may lead to loss of precision on input or output, it can be disabled, and in any case all calculations will still be performed internally to full precision.

Some real numbers (irrational numbers such as $\sqrt{2}$ and transcendental numbers such as $acos(0)$) are not precisely representable as rational numbers. In order to perform calculations involving these quantities, Nickle provides its own implementation of floating-point arithmetic with user-settable mantissa precision and infinite precision exponents. A sensible numeric type hierarchy and well-defined rules for combining various precisions means that Nickle is generally very good at retaining precision in all numeric calculations.

3.4 Flow Of Control Features

In order to cope with situations where a function should not return normally, either as a result of a semantic error such as division by zero or at the user's request, Nickle provides support for the declaration, generation, and handling of *exceptions*. Exceptions are not first-class objects. They are declared in the style of void functions (including arbitrary typed formal parameters), and are scoped identically. At any point during program execution, any in-scope exception may be thrown via a `raise` statement. A Java-style `try`–`catch` block allows the handling of raised exceptions. Currently, there is no mechanism for "restarting" a computation which has raised an exception: once an exception has been raised, execution will resume at the nearest applicable dynamically enclosing `catch`, or at the top level if none is found.

In addition to exceptions, Nickle provides first-class "continuations", which capture an execution locus and environment. These are not true continuations, as they do not restore the values of variables modified by assignment between the capture of a continuation and its use: however, the actual semantics supported is both much cheaper to execute and arguably more usable for an imperative language. The closest C analog is `setjmp()`/`longjmp()` (and indeed, the Nickle equivalents share these names). However, unlike C, a `longjmp()` can occur anywhere, not just in a dynamically enclosing function. In addition, since the Nickle primitives are builtins rather than library functions, variables modified between a `setjmp()` and a `longjmp()` have well-defined values after the `longjmp()`.[1]

3.4.1 Threads

Nickle contains a continuation-based thread system, implemented from scratch entirely in UNIX-portable C code at the byte-interpreter level. Thread scheduling utilizes real-time thread priorities with round-robin scheduling among equal-priority threads. Support is provided for thread synchronization via built-in semaphores or mutex variables.

Threads in Nickle are important for at least three reasons. First, some calculations are most naturally performed concurrently. Consider a simple generate-and-test example like the odd-number example of Figure 3. While in this simple example saving the generator state would be easy, in practice more complicated cases (such as for example a chess move generator) are much easier to write if the generation and testing are performed in separate threads.

Secondly, prototyping of parallel algorithms is most easily performed using a language with concurrent features. The concurrency primitives provided by Nickle are very similar to those of low-level parallel systems, allowing prototyping of code like that of Figure 3 above for later translation to a truly parallel environment.

Finally, the construction and use of Nickle itself is made easier by the availability of threads. In particular, the combination of continuations and threads eases the implementation of debugging in the presence of exceptions. Normal command-line Nickle execution is handled by a thread separate from the command parser. When an unhandled exception occurs, the debugger can simply expose the thread state, including the exception continuation, to the user for inspection.

3.4.2 Flow Control Primitives

Some minor modifications to the Nickle language to help support threads, continuations, and exceptions have proven to be particularly convenient.

[1]In order to support optimization of separately compiled C programs while providing `setjmp()` and `longjmp()` as library functions, the C standards allow variables after a `longjmp()` to have any value they might have attained since the corresponding `setjmp()`, except for `volatile` variables.

```
import Semaphore;

semaphore prod, cons;
int i;

void function ints() {
  while(i < 1000) {
    wait(prod);
    i++;
    signal(cons);
  }
}

void function odd_ints() {
  i = 0;
  prod = new(1);
  cons = new(0);
  thread t = fork ints();
  while(i < 1000) {
    wait(cons);
    if (i & 1)
      printf("%d\n", i);
    signal(prod);
  }
  Thread::join(t);
}
```

Figure 3: An odd-number printer using threads.

Threads are created by a low-precedence `fork` operator: the semantic is that the expression to which the `fork` operator is applied will be evaluated in the new thread returned by the operator. Any thread can retrieve the result of this computation via the `Thread::join()` built-in function. The principal alternative to making the `fork` operator a syntactic part of the language was to make it a built-in function accepting a closure or continuation. This is much less convenient for the user.

Another important syntactic feature concerns the semantics of operations under locks or other temporary invariants. Java provides a `finally` clause of its `try` block, whose primary purpose is to ensure that the lock is restored. This choice has some unfortunate consequences. Consider typical Java code to execute a function under a lock of some sort, shown in Figure 4. First, this code is syntactically complicated. There are many blocks, and the flow of control is not obvious. Second, even though no `catch` clauses are present the `try` is necessary merely to obtain the `finally` clause. Third, the establishment of the lock is syntactically indistinguishable from the rest of the surrounding code,

```
if (get_lock(&l)) {
  try {
     locked_operation();
  } finally {
     release_lock(&l);
  }
} else {
  throw new LockException("failed");
}
```

Figure 4: Locking in Java using `finally`.

```
twixt(get_lock(l); release_lock(l))
  locked_operation();
else
  raise lock_exception("failed");
```

Figure 5: Locking in Nickle using `twixt`.

which means that the language can have no idea whether locks and unlocks are matched, and can provide no assistance with locking errors.

In order to remedy these deficiencies, Nickle separates the invariant-management functionality of `try` from the exception-handling functionality. The Nickle `twixt` statement is syntactically similar to a C `for` loop, but with two arguments instead of three, and an optional `else` clause. A natural reimplementation of Figure 4 using `twixt` is shown in Figure 5. With this syntax, the `get_lock(l)` and `release_lock(l)` operations are syntactically distinguished, and their correspondence is visually obvious. Equally obvious is the fact that the exception is raised as the result of the failure of `get_lock()`. Finally, the gratuitous `try` clause has been eliminated, and the nesting has been regularized.

As a final feature, consider the situation where a continuation is captured using Nickle's `setjmp()` inside `locked_operation()`. Because the `get_lock(l)` operation is known to be protecting the context in which the continuation is captured, Nickle will re-execute it (!) upon `longjmp()` back to the continuation. This would be difficult or impossible in most languages.

3.5 Planned Enhancements

In its current state, Nickle is a quite useful tool. A number of enhancements to Nickle are planned to increase that usefulness, by making Nickle easier to use and harder to make mistakes in.

First and foremost, while Nickle's current full static type system is a marked improvement over dynamic typing alone, it is not the end of the road. Explicitly stating the types of names is tedious and error-prone, which may partially explain the lack of strong static typing in most recent scripting language designs. There are two possible improvements over explicit static typing which might be adapted to Nickle with good effect.

Parametric polymorphism, as found in ML and Generic Java (GJ) [BOSW98] (and its close conceptual cousin type templates, as found in C++) tries to capture the idea that an expression that is independent of the details of its input and output types should not be tied to those details. Parametric polymorphism avoids much "copy-and-paste" coding while still retaining the benefits of full static typing. For example, linked list code which is independent of the type of list elements need not be repeated for each possible structure type, a savings in code size and in the potential for error. Since polymorphic typing is fully safe if properly designed and implemented, the downside is minimal: a slight decrease in the expressiveness of the language due to extra constraints on the code.

ML goes a step further, and adds automatic static type inference to the language. Instead of having to explicitly declare all types, undeclared types are inferred from context, and the resulting type assignments are checked for consistency. Experience has shown that polymorphic type inference has much of the flexibility of dynamic typing while still retaining most of the safety of static typing. For a language like Nickle which is to be used as a calculator and for algorithmic prototyping, this seems like an especially ideal combination.

The Nickle implementation is currently several times slower than equivalent C code. This is ameliorated somewhat by the fact that its primitive operations on numbers are well-tuned and exceptionally fast, and by the ability to implement much more sophisticated algorithms in Nickle in equivalent time and code. Nonetheless, there are applications where better performance would be desirable. The obvious approaches of optimization of byte code or compilation to native code and optimization thereof should be explored. There is no reason in principle why Nickle programs should be dramatically slower than C programs, although there is some overhead inherent in the language.

An interesting tack in this direction would be to produce a Nickle compiler to byte code for the

Java Virtual Machine (JVM). Running Nickle code under the JVM would bring a couple of important advantages. First, portability to non-UNIX environments would be greatly enhanced. Second, significant performance improvements might be available "for free"[2] as a result of the JVM runtime native-code compilation and dynamic code optimization commonly available in many implementations. On the downside, the mismatch between Java's object-oriented model and Nickle's imperative-functional model might make things difficult, and the filesystem-based module system of the typical JVM implementation is ill-suited to Nickle's requirement for separating files from namespaces.

It would be nice if Nickle supported a wider range of built-in types, constants, and operators. The limited range of numeric types is particularly problematic: obvious candidates include the natural numbers, finite fields (particularly $\mathrm{GF}(2^{32})$), and various extensions of the reals, particularly complex numbers. Better semantics and more powerful operators for dealing with vectors, matrices and tensors would also be a plus.

The natural numbers have been in and out of the language at various times in its history. They are currently out, to simplify the language, but they are being reconsidered since their inclusion would close the one known hole in the static type system of the language.[3] C-style 32-bit integers were supported at one time, but were deemed to be too much of a special case to expose to the user. Consideration is currently being given to adding general finite field constructors and operators, but there are notable complications.

The lack of complex numbers is slightly embarrassing, but ideas about regularizing them and implementing them in a sane and compatible way have so far been hard to come by. Issues such as the representation of complex coefficients are vexing: for instance, are complex integers desirable? In addition, it is not totally obvious that the complex numbers should be preferred to other extensions of the reals, yet including multiple generalizations makes it difficult to find a suitable static type lattice and correspondingly to choose runtime promotions.

In addition to numeric types, Nickle could arguably use a larger range of modern structured datatypes, such as lists, sets, and curried functions. There are several reasons why we have not provided these to date. Notably, until a polymorphic type system is implemented, static typing issues are difficult to resolve reasonably. In addition, it is problematic to provide a built-in implementation of datatypes which might be sensibly implemented in multiple ways having different properties. Sets are a particularly good example of this: Pascal's bitsets have very different runtime properties from sets of integers represented as search tries, and both representations are difficult to generalize to sets of values, such as structures, which have no natural inherent ordering. It is questionable whether the language should make choices for the user about representation issues, and so far it has been shied away from.

While Nickle's supporting libraries are already well along, further work needs to be done here. The floating-point math support needs to be validated and extended: it would be especially nice to make it compliant with the rounding and precision rules of the IEEE floating point arithmetic standards [IEE85, IEE87]. The string support is extremely rudimentary. Built-in support for array slicing, array comprehensions, and similar features would occasionally be useful. Some implementations of standard Abstract Data Types (ADTs) such as priority queues might occasionally ease algorithm development.

Support for built-in operators on vectors and arrays of numbers would be a real plus, and has no obvious downside except a slight increase in language complexity. The main reason for their lack of current inclusion is the lack of a need for them in the authors' work.

It would be nice to add some sort of support for workspaces or the like to Nickle, to aid scratchpad calculation. A `save` command would be a good start, but in principle Nickle could allow capturing a true continuation and saving it to a file, which would lead to a nice implementation of both workspaces and checkpointing.

3.6 Omitted Features

Perhaps as important as what to include is what not to include. For example, Nickle contains no Object-Oriented Programming (OOP) features. Class-based OOP in the style of C++ and Java is well understood and accepted. However, the combination of this style of OOP with some of the current

[2] As in beer.

[3] Nickle's exponentiation operator `**` with an integer base has an integer result when the exponent is a natural number, but a rational result with negative integer argument. This proves to be quite problematic in balancing usability and static typing, and static typing currently loses.

and proposed features of Nickle, especially polymorphic type inference and first-class functions and continuations, verges on open research questions. In addition, the most useful domains of application of OOP, such as large-scale programming, easy reuse of opaque constructs, and graphics and window system programming, are outside the intended scope of Nickle's applicability.

Other features deliberately omitted from Nickle include:

- Support for graphics and GUI implementation. The impact of this support on the portability and simplicity of the Nickle implementation is potentially large, and the perceived benefit for expected Nickle applications is presently small. Nonetheless, this decision may be revisited in the future.
- Language-level support for interfacing with native code. In actuality, as discussed below, it is quite easy to integrate native code with Nickle programs, but only by integrating the native code into the Nickle implementation.
- Ad-hoc polymorphism, operator overloading, user-defined operators, and related features. This is driven by concern about the complexity and difficulty of implementation of these features, and their perceived negative impact on readability and portability of code. The general philosophy of Nickle is that anything important enough to require these features is important enough to embed in the language definition.

4 History

Nickle began life around 1985 at Reed College as `ec`, a compiler written by Packard for translating arbitrary-precision arithmetic into high-performance interpreted byte code. In addition, around that time, both authors were experimenting with the design and implementation of Kalypso, an interpreter (and later a compiler) for a purely functional dialect of LISP. Inevitably, the desire for LISP-like numerical expressions with C-like syntax led to the construction of `ic`, an "interpreted C" which also incorporated concepts borrowed from earlier work by Packard and others at Tektronix, Inc. on incremental C and Pascal compilation.

The original `ic` was a pure tree-walking interpreter with arbitrary-precision integer and rational datatypes allocated and destroyed statically. This allowed the memory management issues to be finessed. Incremental compilation was one of the first enhancements, and was accompanied by new datatypes which necessitated a reference-counted memory management scheme with a custom storage allocator.

By about 1993, accumulated incremental changes prompted a complete reworking of the `ic` implementation. The reference-counted storage management was replaced by a tracing mark-sweep collector (borrowed from another of the authors' projects, a functional LISP subset implementation known as Kalypso), first-class functions were added, and the syntax and semantics of the language were revised somewhat. The resulting language was known as Nick. Later additions included first-class continuations, threads, and, by about 1996, namespaces.

In the last 6 months, the static type system has finally been implemented, the platform-native floating point representation has been supplanted by a platform-independent arbitrary-precision implementation,[4] many builtins have been added, the disjoint union type has been added, the user interface has been improved (including support for GNU `readline`), the documentation has been largely completed, the examples have been collected and regularized, a multitude of bugs and misfeatures have been repaired, and other improvements too numerous to list have been made. The result is Nickle as it exists today.

A few months before the public source release, only minor changes were planned, except for the implementation of polymorphic type inference. As the release was finalized, it became clear that the polymorphic type inference system would have to wait: the features described in the previous paragraph became clear priorities and absorbed all of the available time of both authors. The lesson here is clear and, in retrospect, obvious: the first 90% of Nickle development took 15 years, and the remaining 90% took the last three months. The result appears, so far, to be worth the work: Nickle has never been faster, more stable, or more pleasant to use.

The first public source release of Nickle was in mid-April of 2001. As of the first week since it was announced on `freshmeat.net`, about 100 copies have been downloaded. So far, there have been no bug reports, and contributions have already been made to the project by users who helped with creating

[4]Several potential floating point representations were considered. In particular, interval arithmetic [Kea96], while in some ways preferable, was not chosen due to performance concerns.

alternative binary packages for the distribution.

Nickle was designed to be highly portable within the UNIX environment: so far, that goal appears to have been met. Prerelease, it was compiled on a variety of UNIXes with no problems. GNU `autoconf` was invaluable here: while difficult to use, it does its intended job admirably.

5 The Nickle Implementation

Nickle's current implementation consists of about 25,000 lines of C code in about 45 files, together with about 1000 lines of builtins written in Nickle itself. Great attention has been paid to modularity in the implementation: the current structure is the result of literally years of refactoring and reorganization effort. The rough breakdown of the implementation is as follows:

- Builtin datatypes: approximately 20 files, with one `.c` file and one `.h` file per type.
- Memory management: 6 files
- Internal ADT implementations: about 8 files
- Execution infrastructure: about 8 files

The remainder consists of miscellaneous support routines.

As noted above, the implementation uses a highly stereotyped interface to the garbage collector, which allows the C code to easily allocate and reference storage in the Nickle heap. Since this strategy was perfected, memory reference errors in the C infrastructure of the implementation have become extremely rare. Of course, the increasing maturity of the implementation is also a factor here.

Because of the ease of memory management and the extreme modularity, adding new C code to the Nickle implementation is quite easy, even for someone not overly familiar with the internals. As mentioned above, this is one reason why the lack of a native-code interface as part of the Nickle programming language is regarded as unobjectionable. Massey has added C builtins to the implementation on a couple of occasions, and has found the overhead due to learning curve and extra code requirements to be on the order of a few hours. It is not clear that this could be improved with a JNI-style builtin native interface. Portability is an issue, however: the integrated C code should be able to run on arbitrary UNIX (at least) platforms.

6 Experience With Nickle

The various incarnations of Nickle have been used for a range of tasks. First and foremost, Nickle is the calculator program of choice: it is an altogether superior[5] replacement for UNIX `bc`, `dc`, `expr`, and the like.

Nickle is also a very nice general purpose programming language, especially for numerical work. Nickle programming projects distributed with the reference implementation include

- The Cribbage scoring implementation mentioned above.
- A DSP filter design package.
- Sample data generation for DSP verification.
- A full RSA implementation, including Miller-Rabin probabilistic prime number key generation.
- An implementation (now converted from Nickle to C for use inside Nickle) of Weber's accelerated GCD algorithm.
- A port of the C reference code for the Rijndael encryption algorithm.

In addition, numerous other projects have been assisted by Nickle, including

- Graphics chip clock calculation, and XFree86 "mode line" calculation.
- Probability calculations for Collectible Card Games.
- Course grading.

As noted above, the performance of Nickle is not spectacular, but is adequate for the tasks for which it is intended. For example, the Miller-Rabin implementation typically spends 5–15 seconds generating a 512-bit probabilistic prime on a 700MHz Athlon with adequate memory. This is about a factor of 5 slower than the C-based probabilistic prime generator of OpenSSH. As another example, the Nickle implementation of the Weber GCD code mentioned above is typically 10 times slower on a given input than the C implementation. On the other hand, the Nickle implementation was *much* easier to develop.

[5] Nickle's overhead is considerably larger than those of the listed programs. In practice, this is not a noticeable problem on modern UNIX platforms.

The end result of this experience has been that Nickle has become part of the authors' standard toolkit. It has reached its design goals: the current version is simple to use, extend, and modify.

7 Related Work

The design ideas behind Nickle have been drawn from a number of language implementations, as mentioned above. Relevant languages include C, C++, Icon, Java, ML, Modula-3, Perl, Python, Scheme, UNIX `sed`, AWK, `bc`, `dc` and `expr`, and a host of others. A detailed comparison with each of these language is precluded by space considerations, but some important considerations and principles emerge.

7.1 What Nickle Is Not

First, Nickle is not a text-processing language. While it does include some rudimentary support for strings, and support for file I/O and formatting comparable to (and modeled after) UNIX `stdio`, it has no native support for such niceties as regular-expression-based pattern matching, implicit stream processing, textual variable substitutions, text editing, etc. While the authors have considered the problem of designing a modern text processing language, it would not look a great deal like Nickle; in addition, it is not clear that there is a niche for yet another text processing language given the popularity of many existing candidates.

Second, Nickle is not a language for building large applications. While it does have some support for syntax-level modularity, the implementation is currently rather dependent on whole-program compilation. In addition, the exclusion of OOP and GUI features, as well as the relative inefficiency of the current implementation, augers ill for Nickle's acceptance as a replacement for Ada.

Third, Nickle is not a symbolic algebra package. Its domain is strictly numeric. While a great deal of the Nickle feature set might be useful in a symbolic algebra package, constructing such a thing is probably beyond the purview of a two-person team inexperienced in such matters, and certainly would vastly exceed the current 25K lines of code.

7.2 Comparison With Other Languages

Given the design goal for Nickle—a language for desk calculation and prototyping of numerical and semi-numerical algorithms—it is constructive to compare its feature set and implementation properties with those of a few of the languages listed above.

First and foremost, unlike all of the languages listed above except certain Scheme implementations, Nickle supports a wide variety of exact or highly precise numeric types, organized in a sensible fashion and properly checked statically and dynamically. The true power of Nickle is not apparent until one tries to add up 10,000 probabilities, and finds that in Nickle they sum to 1; not 0.998 or 1.02, but just plain 1. The ability to select the mantissa precision for floating point computation is similarly useful: it is notable that Nickle is quite usably fast with the default mantissa precision of 256 bits.

A more fair comparison is with R5RS or later Scheme implementations supporting the full numeric model. Such an implementation provides a quite usable calculator and programming language, comparable in some ways to Nickle. The principal differences here include the C-like syntax (indeed, any syntax at all), the static type system, namespaces, structures, etc., all of which ease the sort of programming at which Nickle is aimed. (The lack of built-in list support is a missed feature, as noted above.)

ML has the potential to do much of what Nickle does. Its support for functional programming is obviously far superior to Nickle's, and its syntax, type system, and the like are comparable. The learning curve for ML tends to be fairly steep by most accounts. The experience of the authors is that C programmers pick up Nickle immediately, not just because of the C-like syntax, but also because of the first-class support for imperative programming: Nickle does not try to change one's programming paradigm. Of course, the support for numeric types in Nickle is also superior to that of any ML implementation of which the authors are aware.

Some interest has been expressed in the relationship between Nickle and Perl. First and foremost, as noted above, Perl, `sed` and AWK are aimed primarily at text processing, and are well-suited to this sort of task. The support for numeric programming in Perl is limited: until recently, the only numeric representation supported was IEEE floating point numbers. In addition, the complex syntax and semantics of the language tends to make for a steep learning curve even for experienced programmers [Sch93]. Finally, Perl's complicated system of types and values is somewhat error-prone [McC01] and contains

Table 1: Benchmark execution time in seconds.

	`bc`	Nickle	GMP
`ifact`	67.6	5.7	3.4
`rfact`	67.8	6.0	3.3
`choose`	130.	6.3	1.8
`comp`	31.7	9.6	2.6

little support for static typing.

7.3 Performance

Nickle's performance appears to be around 5 times slower than equivalent C code using the GNU GMP multiple-precision library, and quite a bit faster than GNU `bc`. Some simple benchmarks were run to compare the performance of Nickle 1.99.3, GNU `bc` 1.05, and C using GNU GMP 2.0. Four benchmarks were utilized: `rfact` computes 20000! using the obvious recursive implementation, `ifact` computes 20000! iteratively, `choose` computes $\binom{20000}{5000}$ (using `ifact` in the C and `bc` versions), and `comp` applies the Miller-Rabin test to the prime number 31957 for every possible base from 1 to 31956. (The source of all of these benchmarks is available with the Nickle distribution.)

Table 1 shows Nickle execution times on an Athlon 700 with 256MB of RAM running Linux kernel 2.4.1 in single-user mode. All times are the minimum of 5 insignificantly different consecutive runs. (Nickle's built-in ! operator, while more convenient, produced similar timings to the hand-coded versions.) Nickle and GMP spent about 50% of total time on the factorial benchmarks generating and printing the decimal result (since there appears to be no easy way to inhibit this behavior in `bc`). The runtimes for these benchmarks are thus somewhat inflated. In general, the performance results are positive: the small performance hit over C code is more than made up for in ease of use.

8 Lessons Learned

Certainly, a number of pragmatic lessons about language design and implementation have emerged over the years of Nickle development. It turns out, for example, to be difficult to give an LALR(1) grammar for such a strong superset of C. Garbage collection turns out to be a huge win over the alternatives: in practice the authors have never observed a problem related to collector performance, and the ease of implementation and the quality of the user experience have been tremendously improved. The principle of least surprise has proven a good guiding principle for the design: the authors as well as novice users seem to be able to use Nickle without deep thought or constant reference to the documentation.

It was also interesting to observe how two people with similar backgrounds and tendencies can have quite different opinions about even broad details of language design. While the authors always largely agreed on where they were going, there was much involved discussion about the best way of getting there.

In particular, the influence of other languages on the Nickle design was complex and varied: both authors learned a lot about a variety of language options and about how to keep a clear head when evaluating and implementing them. Corner cases in existing language features proved to be problematic: Nickle tended to adopt in a piecemeal fashion features that other languages were designed around, and understanding the best methods of fitting these features in usually required a significant effort.

In preparing the initial draft of this paper, the authors wrote:

> The degree of meticulousness [involved in finalizing Nickle] is admittedly unusual in a public utility-belt programming language release. However, it should be understood that the authors have refrained from a public release over a 15-year period precisely to reach this level of quality while there was still room to experiment. Following a successful public release, it will become much harder to make major specification or implementation changes. This drives a desire to get it largely right the first time.

As public release drew near, it became apparent that a number of significant last-minute changes to the language and the implementation were not only desirable but necessary. To a large degree, however, these changes were intended to articulate the goal quoted above: to get the language as "right" as possible before the first public release.

9 Conclusion

Nickle has been an interesting and quite successful experiment in utility-belt programming language design and implementation. It has increased the authors' understanding of various programming language options, proved out some of their opinions, and been instrumental in getting some of their other work done. We hope it will be useful and interesting for the computing public as well.

Acknowledgments

Thanks to the authors' various employers and institutions over the past 15 years for their tolerance and even encouragement of an effort devoted to a marginally work-related endeavor. Thanks to the Usenix referees for their insightful comments on the paper, and to Clem Cole in particular for his careful (and patient) reading and many helpful suggestions that vastly improved the presentation. Thanks to the pioneers of programming language development, who made the implementation of Nickle both possible and desirable. And a big thanks for the invaluable contributions of users of Nickle and its predecessors over the years, including advice, kibitzing, bug reports and moral support.

Availability

Nickle and a variety of supporting materials are freely available in both source and binary forms from the Nickle web site: `http://www.nickle.org`.

References

[AWK88] A. V. Aho, P. J. Weinberger, and B. W. Kerninghan. *The AWK programming language*. Addison-Wesley, 1988.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Phillip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Conference on Object-Oriented Programing systems, Languages and Applications (OOPSLA '98)*. SIGPLAN, ACM, October 1998.

[CE92] William Clinger and Jonathan Rees (Editors). Revised ^{4} report on the algorithmic language Scheme. Technical Report CIS-TR-91-25, University of Oregon, February 1992.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.

[IEE85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, NY, 1985. Revised 1990.

[IEE87] IEEE. *IEEE 854-1987, Standard for Radix-Independent Floating-Point Arithmetic*. IEEE, New York, NY, 1987. Revised 1994.

[Kea96] R. Baker Kearfott. Interval Computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.

[Lut01] Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., second edition, 2001.

[MC78a] Robert Morris and Lorinda Cherry. *BC - An Arbitrary Precision Desk-Calculator Language*. AT&T Bell Laboratories, 1978. Unix Programmer's Manual Volume 2, 7th Edition.

[MC78b] Robert Morris and Lorinda Cherry. *DC - An Interactive Desk Calculator*. AT&T Bell Laboratories, 1978. Unix Programmer's Manual Volume 2, 7th Edition.

[McC01] Jamie McCarthy. Sophomore uses list context; cops interrogate. *Slashdot*, March 2001. URL `http://slashdot.org/yro/01/03/13/208259.shtml`, accessed April 18, 2001 03:52 UTC.

[McM78] Lee E. McMahon. *SED - A Non-interactive Text Editor*. AT&T Bell Laboratories, 1978. Unix Programmer's Manual Volume 2, 7th Edition.

[MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nel91] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Sch93] Randal L. Schwartz. *Learning Perl*. O'Reilly & Associates, Inc., 1993.

[WCS96] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., second edition, 1996.

Appendix: The Nickle Tour

Material in `typewriter font` is taken from a Nickle interactive session. Material in *italics* is commentary.

```
$ nickle
```

The obvious calculations work, including fancy operators and arbitrary precision.

```
> 1 + 1
2
> (2 ** 4)!
20922789888000
```

Rationals are representend exactly, but printed in decimal. Integer division with // is different from rational division.

```
> 1 / 3
0.{3}
> . * 3
1
> 1 // 3
0
```

Expected conveniences, like the value . denoting the last value printed, work. Implicit declarations work at top level, as do explicit typed declarations. Using a statement form at top level results in no value being printed.

```
> x = .
0
> int y = x;
```

C-style statements can be typed at the command line. The + prompt denotes an incomplete statement.

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

Exact integer square roots will be represented by integers. For irrational roots, a 256 bit floating point representation is used; the printed representation is indistinguishable.

```
> xsqrt = sqrt(x)
5
> sqrt(2)
1.4142135623730
> . * .
2
> sqrt(5)
2.2360679774997
> . * .
4.9999999999999
```

Functions may be typed at the command line. Argument and result types are optional. This function returns nonsense for non-integers.

```
> function sqr(x) {
+   auto s = 0;
+   for (int i = 1;
+        i < 2 * x + 1;
+        i += 2)
+     s += i;
+   return s;
+ }
> sqr(5)
25
> sqr(5.1)
36
```

Functions are first-class: untyped functions can be assigned to statically typed function variables.

```
> int(int) isqr = sqr;
> isqr(5)
25
> isqr(5.1)
->     isqr ((51/10))
Incompatible types 'int', 'rational'
argument 0
```

Operators try to behave properly in as many cases as possible.

```
> 5.1 ** 2
26.01
> -5.1 ** 2
26.01
> -5.1 ** 3
-132.651
> -5.1 ** 3.1
Unhandled exception "invalid_argument"
at /usr/local/share/nickle/math.5c:196
"log: must be positive"
0
(-51/10)
> quit
```

For reasonable sized chunks of code, it is normal to use a separate text file.

```
$ cat > stack.5c
namespace Stack {
  typedef frame;
  typedef struct{
    poly val;
    *frame next;
  } frame;
  public typedef * *frame stack;
  public exception stack_underflow();

  public stack function
  new() {
    return reference(0);
  }

  public void function
  push(stack s, poly xval) {
    *s = reference((frame){
      next = *s,
      val = xval
    });
  }

  public poly function
  pop(stack s) {
    if (*s == 0)
      raise stack_underflow();
    poly xval = (*s)->val;
    *s = (*s)->next;
    return xval;
  }
}
^D
```

Here is the Stack ADT in action.

```
$ nickle
> load "stack.5c"
> print Stack
namespace Stack {
    public typedef **frame stack;
    public stack function new ();
    public void function
             push (stack s, xval);
    public function pop (stack s);
}
> import Stack;
> stack s = new()
&0
> push(s, "x")
> push(s, 3)
> pop(s)
3
> pop(s)
"x"
```

Uncaught exceptions lead to the debugger.

```
> pop(s)
Unhandled exception "stack_underflow"
at stack.5c:23
- trace
    raise stack_underflow ();
pop (&0)
    pop (s)
- s
&0
- done
> quit
$
```