The Design of a Self-Compiling C Transpiler Targeting POSIX Shell

Laurent Huberdeau
laurent.huberdeau@umontreal.ca
Université de Montréal
Montréal, QC, Canada

Cassandre Hamel
cassandre.hamel.1@umontreal.ca
Université de Montréal
Montréal, QC, Canada

Stefan Monnier
monnier@iro.umontreal.ca
Université de Montréal
Montréal, QC, Canada

Marc Feeley
feeley@iro.umontreal.ca
Université de Montréal
Montréal, QC, Canada

Abstract

Software supply chain attacks are increasingly frequent and can be hard to guard against. Reproducible builds ensure that generated artifacts (executable programs) can be reliably created from their source code. However, the tools used by the build process are also vulnerable to supply chain attacks so a complete solution must also include reproducible builds for the various compilers used.

With this problem as our main motivation we explore the use of the widely available POSIX shell as the only trusted pre-built binary for the reproducible build process. We have developed `pnut`, a C to POSIX shell transpiler written in C that generates human-readable shell code. Because the compiler is self-applicable, it is possible to distribute a human-readable shell script implementing a C compiler that depends only on the existence of a POSIX compliant shell such as `bash`, `ksh`, `zsh`, etc. Together, `pnut` and the shell serve as the seed for a chain of builds that create increasingly capable compilers up to the most recent version of the GNU Compiler Collection (GCC) that is a convenient basis to build any other required tool in the toolchain. The end result is a complete build toolchain built only from a shell and human-readable source files. We discuss the level of C language support needed to achieve our goal, the generation of portable POSIX shell code from C, and the performance of the compiler and generated code.

SLE '24, October 20–21, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695639

CCS Concepts: • **Software and its engineering → Software development process management**; **Source code generation**; • **Security and privacy → Software security engineering**.

Keywords: Compiler, Bootstrapping, Reproducible Builds

ACM Reference Format:
Laurent Huberdeau, Cassandre Hamel, Stefan Monnier, and Marc Feeley. 2024. The Design of a Self-Compiling C Transpiler Targeting POSIX Shell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3687997.3695639

1 Introduction

Traditional C compilers generate executable programs that are not *portable*, meaning that the produced program's execution is limited to a specific processor type and operating system. In our work we have explored the peculiar choice of using the POSIX shell [17] as a compilation target. POSIX shell was standardized over 30 years ago and it is an exceptionally portable execution environment for these reasons:

- It has multiple conforming implementations, including Korn Shell (`ksh` [4]), Bourne-Again Shell (`bash` [7]), Z Shell (`zsh` [11]), Almquist Shell (`ash` [2]), Debian Almquist Shell (`dash` [10]), and Yet Another Shell (`yash` [13]).
- It is cross-platform, with implementations on all the major operating systems (Linux, macOS, and Windows), including old versions of those operating systems and currently less mainstream operating systems such as MS-DOS, AIX, and VxWorks.
- It is readily accessible, as many operating systems have a POSIX shell preinstalled. On Windows, the Windows Subsystem for Linux (WSL) or one of the many `bash` packages for Windows can be used.
- It is a widely used standard for scripts controlling critical aspects of the operating system, so it is unlikely to evolve in a backward incompatible way.

The execution speed and portability of the generated shell scripts are clearly important issues. One of our contributions is to find ways to implement certain critical C language features, such as pointers and file I/O, that are reasonably efficient across the various shell implementations for our use cases. Moreover, the readability of the generated scripts is an issue our work addresses. We have developed `pnut`, a C *transpiler* that preserves the structure of the C source code. `Pnut` has a compiler backend that generates readable and portable POSIX shell scripts from C source code.

`Pnut` has been written in C and its main source code file, `pnut.c`, is compilable by `pnut`. This self-compilation produces the `pnut.sh` shell script which is a portable version of the `pnut` compiler. The two step procedure to create `pnut.sh` using an existing C compiler is shown in Figure 1. Once created, `pnut.sh` can compile and execute C programs on any system with a POSIX shell, and with no requirement for a preinstalled C compiler or other tool. This allows C programmers with little experience in shell programming to develop system utilities (installers, configurators, daemons, etc) that work as is on a wide variety of computers.

`Pnut`'s shell backend was designed to produce shell script code that is easy for humans to understand and review. Consequently, a motivated software development team can do a code audit of scripts generated by `pnut`, including `pnut.sh` itself, to verify that it does not contain malicious code. The scripts generated by `pnut` avoid shell constructs that would hinder this auditing process. As explained in Section 2, this makes `pnut.sh` particularly interesting as a trusted seed for the purpose of *reproducible builds*.

In this paper, we motivate the design choices we took to make `pnut` a practical tool for compiling C. The lessons learned about portable and efficient shell scripting in the context of compiling C are likely to be applicable to compilers for other languages that generate portable shell scripts. For reference, when we measure performance to support our claims and choices it is done in the hardware/software environment which corresponds to a typical development computer with recent versions of various shells:

OS:	Ubuntu 22.04.2 with Linux 6.5.0-41
Processor:	AMD Ryzen 9 5900X @ 4.95GHz
RAM:	128GiB
shells:	`bash` 5.1.16(1)-release, `dash` 0.5.11, `ksh` 93u+m/1.0.0-beta.2, `yash` 2.51, `zsh` 5.8.1

Section 2 explains the compiler's use for reproducible builds, which is the main motivation for creating `pnut`. Section 3 explains the main challenges to compiling C to shell code and our solutions. Section 4 goes over the architecture and optimizations used in `pnut`. `Pnut`'s C library is described in Section 5. Section 6 gives a performance evaluation of `pnut` on various shells. Finally, related work is in Section 7.

To bring the reader up to speed on the POSIX shell language, we provide a brief overview of the relevant aspects of the language in Appendix B.

2 Reproducible Builds and Bootstrapping

A *reproducible build* is a process of building software in a way that ensures that the resulting executable programs can be reproduced bit-to-bit. To achieve this, the source files must be the same for every build and the compiler used must be deterministic. Moreover, the compiler version must be fixed, as different versions can generate different outputs.

Ideally, the compiler used would itself be built reproducibly, which brings the question of how to compile the compiler. Historically, this was seen just as a technical hurdle which was to be overcome via something called a *bootstrap* but whose actual details were largely considered as irrelevant, as long as it ended up delivering a functional compiler.

The dark art of the bootstrap consists sometimes in various ad-hoc workarounds, as documented for example in [14, Appendix C], and in the general case it relies on the use of an older but already compiled version of the same compiler, which can be discarded once the new version has been successfully compiled. The latter approach is generally less ad-hoc but can still be delicate, as discussed by Appel [3].

Yet, the technical difficulties of bootstrapping a compiler have nowadays been overtaken by other concerns such as having to keep around a precompiled version of the compiler, that it complicates porting the compiler to a new platform, or the fact that it fundamentally undermines the ideals of Free Software since it requires distributing precompiled code (basically a "binary blob") alongside the source code. The more serious problems come down to reproducibility and security and often presents themselves as software supply chain attacks. These attacks are problematic because compromised software that is distributed in an executable form is opaque, making it hard for the user to detect any tampering (addition of viruses, backdoors, etc) done prior to its distribution.

A given compiler's source code may result in not bit-for-bit identical executables depending on which older compiler is used to bootstrap it, even after recompiling itself, and those different executables can behave differently, which can be abused using the famous *trusting trust attack* [18] where compromised compiler can also compromise the executable form of the software even if the source code has not been compromised. Because a toolchain is also software built by another toolchain, this argument applies recursively all the way back to some primordial tools that must be trusted and from which all the other toolchains can be built.

We propose that a practical choice for a universally available "root language" is a POSIX shell, as a shell will be used for some of the preparation steps of the build process, such as obtaining the sources, decompressing them, checking file hashes, etc. In this case, the shell has to be a trusted tool, otherwise it could compromise the sources at the very start of the build process. Moreover, the many independent implementations of the POSIX shell make it possible to use diverse double compilation [21] to increase confidence in the result.

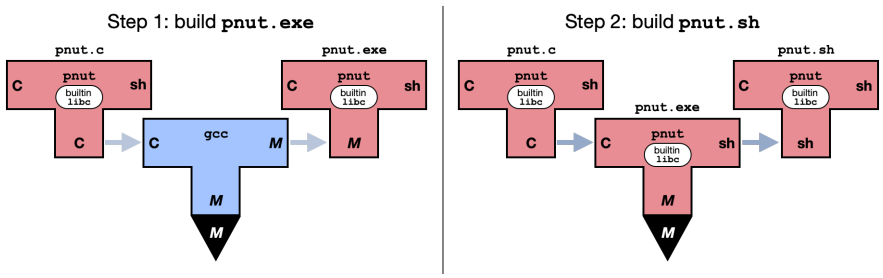

Figure 1. The file `pnut.sh` is created by a two step bootstrap procedure. The file `pnut.c` is compiled by an existing C compiler, such as `gcc`. The executable compiler `pnut.exe` that is produced can then be used to compile `pnut.c` again to create `pnut.sh`. The traditional "T" notation [1] for compilers is used, indicating the source/target/host language triplet. A given color indicates a specific translator, ignoring the host language. Red is used for the `pnut` compiler, which compiles C to POSIX shell. The inverted black triangle represents a machine of type *M*, whose architecture and OS don't matter here.

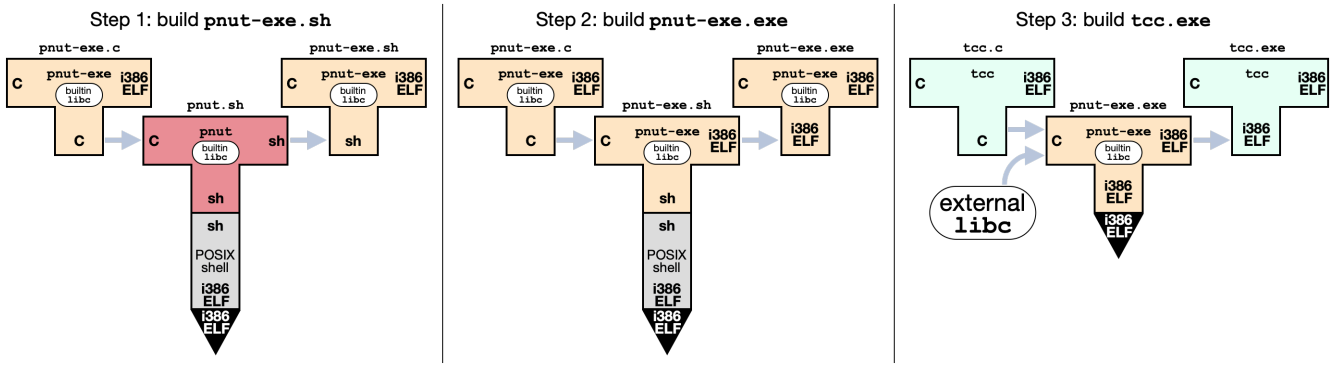

Figure 2. The first steps of a reproducible build process on an `i386` Linux computer. The `pnut.sh` C to shell compiler (red) is executed with a POSIX shell to compile the `pnut-exe.c` program (beige) to the `pnut-exe.sh` shell script which is then used to compile `pnut-exe.c` once more to produce the `pnut-exe.exe` executable compiler. This compiler can then be used to compile more feature-full C compilers, here TCC (green).

This is an important motivation for developing the `pnut` *transpiler* that compiles a subset of the C language to the POSIX shell language. One can view it as a tool to help write shell scripts, or as a way to bootstrap a compiler written in C with an executable version that is human-readable, making it possible to prevent attacks through a compromised compiler.

Because reproducible builds are an important use case for `pnut`, we have also developed the `pnut-exe` C compiler, a variant of `pnut` where the shell backend has been replaced by one that generates Linux ELF format executable programs for the Intel x86 architecture. Compiling the `pnut-exe.c` C source file using `pnut.sh` produces the `pnut-exe.sh` shell script, that can then be used to compile `pnut-exe.c` again to obtain the `pnut-exe.exe` executable that is much faster than the `pnut-exe.sh` shell script. `pnut-exe.exe` can then be used to build increasingly powerful compilers in a chain, such as the Tiny C Compiler (TCC [9]) that can then be used to build other C compilers up to the GNU Compiler Collection (GCC). The first few steps of the reproducible build process are based on `pnut` are shown in Figure 2. A key point is that the party interested in performing a reproducible build from a POSIX shell only needs to obtain the files `pnut.sh` and `pnut-exe.c` to start the process, and these are source code files that can be audited for security issues. We will refer to Figures 1 and 2 in subsequent sections to explain some of our design requirements.

3 Compiling C to POSIX Shell

Given `pnut`'s goal to have easily auditable generated code, these design principles were adopted:

- Preserve the general structure of the C source code.
- Generate regular shell code and avoid special cases.
- Avoid the use of hardcoded constants and magic numbers in the shell code.
- Avoid the use of the shell `eval` command and other kinds of dynamic code evaluation, both for readability and efficient safety auditing.
- Prioritize readability over performance, but don't ignore performance.

As a consequence, optimizations that would improve the execution speed such as inlining, dead code elimination, loop unrolling and constant folding are not performed by `pnut`.

These constraints and the low-level nature of the C language make taking advantage of certain features of the shell language difficult. For example, the shell string processing utilities are too high-level to be useful for the low-level operations on C strings. The opposite is also true where some C constructs are difficult to map to shell. As a result, certain C features are not supported, this includes:

- `goto` and `switch` case fall-through: The shell doesn't have a direct way to implement this kind of control flow, and supporting it would require transformations that would diminish its readability.
- The *address of* (`&`) operator on local variables. Supporting this would prevent the direct mapping of C local variables to shell variables.
- Floating point numbers: Floating point numbers are not supported natively by the shell and would require a large amount of code to implement fully (recall that making use of external programs such as the Unix `bc` is forbidden by our design constraints).

These features are not needed on a large class of programs and, in particular, were not used for writing `pnut.c` and `pnut-exe.c`. We note that these limitations stem from the requirement for readability and auditability of the shell code. The `pnut-exe` compiler, which targets machine code, is not subject to these limitations and is thus suitable for building TCC whose source code uses these features. That being said, `pnut.sh` supports a comfortable subset of the C language including all signed arithmetic operators, structures, enums, pointers, pointer arithmetic and C preprocessor directives.

Also, as some of the implementation choices taken are at the cost of some performance, we evaluate their impact compared to the other options. The following table gives the time taken by `pnut.sh` to compile `pnut-exe.c` on different shells with default options which will be used as the baseline when comparing different code generation approaches:

ksh	dash	bash	yash	zsh	
24.6 s	43.8 s	60.7 s	61.9 s	664.2 s	Time

3.1 Shell Variables and Functions

Because all shell variables live in the same namespace, it is important to use a naming scheme that prevents conflicts between the internal variables used by `pnut` and the variables of the compiled program. To this end, the underscore character serves as a prefix to create separate namespaces for the variables of the compiled program. Because local variables are most common, they are assigned the empty namespace and have the same name as in the C code. Global variables are prefixed with one underscore to help distinguish them from local variables, and internal variables are prefixed with two underscores. This naming scheme is short and easily recognizable and is used throughout the shell code generated by `pnut`. This prevents the use of C variables starting with an underscore, but it is an acceptable compromise as it improves readability to preserve the same name.

One kind of internal variable is temporary variable created to store intermediate results. These use the `__t` prefix and are numbered starting from 1. By using the internal variable namespace, it makes it clear that these variables are not present in the C code and were generated by the compiler.

In addition, certain variables such as `IFS` and `PATH` have a special meaning in the shell. These variables are not allowed by `pnut`. While `pnut` could assign different variable names to these special variables, that would make the generated shell code less readable and these variables can easily be avoided by the programmer. An exception to this rule is the `argv` variable which has a special meaning in `zsh` and is commonly used in C for the command line arguments received by `main`. This variable is mapped to `argv_` in the generated shell code, and the C variable `argv_` is forbidden to ensure that there are no naming conflicts.

All function names are prefixed with underscore with no restriction regarding the use of underscores in the function name. This leaves the namespace with no prefix available for the runtime library functions and prevents conflicts with the shell's built-in functions and utilities.

3.2 Calling Convention

A major difference between POSIX shell and C is the treatment of functions. POSIX shell procedures cannot declare local variables, as shell variables are globally scoped, and they cannot return a result value. We will nevertheless refer to them as shell *functions* as this is the commonly used term.

3.2.1 Returning from Functions. One way of returning a value from a function is to assign the value to a global variable, which can then be read by the caller of the function. Using the fact that shell allows assigning to variables with computed names using string and arithmetic expansion, we choose to have functions take as their first argument the name of the variable that receives the result, the *return variable*. We think this style reads easily as all function calls have the same structure. Also, since the results of C functions are often assigned to variables, this choice makes it possible for a function to assign the result directly where the caller wants the result to be. Concretely, the C function call
`x = f(y, 1)` is mapped to the shell code `_f x $y 1`.

In the cases where the result of a function is ignored, the internal variable `__` is used as the return variable as a convention. This shell variable cannot be accessed by the C program, and so any value written to it is effectively discarded. The presence of `__` in a function call is a reminder that the result of the function is ignored.

3.2.2 Local Variables. We use the term *local variable* to mean a programmer-declared local variable or parameter, or a compiler introduced temporary. Since POSIX shell offers only globally scoped variables, local scoping of variables is

obtained with a *shallow binding* approach. For this, we use a callee-save calling convention. Every function starts by saving the local variables to a stack so that the function is free to use them. The variables are then assigned their local value by the function. The variables get restored from the stack when the function returns. Unlike the traditional C stack implementation, this stack is used only for saving local variables and is not used for control flow, meaning that there is no stack space used when there are no local variables.

To make the bookkeeping of local variables easy to read, it is abstracted using the `let` and `endlet` functions:

```
__SP=0
let() { # $1: variable name, $2: value (optional)
  : $((__SP += 1)) $((__$__SP=$1)) # Push
  : $(($1=$2+0))                   # Init
}

endlet() { # $1: return variable
           # $2...: function local variables
  __ret=$1 # Don't overwrite return value
  : $((__tmp = $__ret))
  while [ $# -ge 2 ]; do
    : $(($2 = __$__SP)) $((__SP -= 1)); # Pop
    shift;
  done
  : $(($__ret=__tmp)) # Restore return value
}
```

The `let` function takes the name of a local variable as its first argument and optionally takes the initial value as the second argument. It saves the value of the variable to the stack and assigns its initial value or 0 if the initial value is not specified. The `endlet` function takes the name of the return variable as its first argument, followed by the names of the variables to restore. It makes sure not to overwrite the return variable in case a local variable of the callee has the same name as the return variable specified by the caller.

Calls to `let` appear at the head of function bodies, and calls to `endlet` appear last. These statements always come together, but `endlet` can undo the effect of multiple `lets` and may appear in the middle of a function returning early.

This bookkeeping has a cost, since all functions, no matter how small, require it as they all have the return variable parameter when compiled to shell. Fortunately, the POSIX specification includes local scoping for positional parameters – the initial value of a function parameter can be retrieved using its corresponding positional parameter. This feature can be used for parameters that stay constant over the execution of a function and removes the need to save them to the stack. This is done for the return variable which is always `$1`. This allows functions without parameters and local variables to avoid the overhead of local variable management.

In a function, the positional parameters can also be set using the `set` command. This command replaces the positional parameters with the arguments given to it and can be used as a way to store data that's visible only in the function call. This feature makes it possible to save the value of local variables in the positional parameters instead of saving them to the stack with `let`, greatly speeding up the calling convention. However, `set` is less readable than using `let`, so we use `let` by default and `set` is kept as a compilation option if performance is needed since the difference in performance can be significant, as shown here by comparing `pnut.sh` compiling `pnut-exe` with this option to the baseline:

ksh	dash	bash	yash	zsh	
15.0 s	34.5 s	38.6 s	35.4 s	252.2 s	Time
0.61 ×	0.79 ×	0.64 ×	0.57 ×	0.38 ×	Ratio

3.2.3 Function Declaration. Combining the callee-save calling convention with using a result variable, the prologue of the generated shell functions is minimal, consisting of calls to `let` followed by the initialization of the local variables if necessary. The `let` calls for parameters are placed on the same line as the function declaration, matching the position where parameters are declared in C and hiding the fact that function arguments are passed as positional parameters.

The epilogue is also short, with only an assignment to the result variable and a call to `endlet` before returning. The `fib` function in Figure 3 shows the prologue and epilogue and compares it to the `set` version of the same function.

3.3 Memory

Unlike `bash` and other more advanced shells, the POSIX standard doesn't provide arrays or any other data structure with indexing. The only way to store data is in variables, with each variable containing either a string or an integer (represented as a string). Fortunately, it is possible to define variables with a dynamic name using arithmetic expansion, which is how arrays and ultimately the heap can be implemented.

We assign each memory location the variable underscore followed by the address – address 0 corresponds to `_0`, address 1 corresponds to `_1`, etc. This prevents the use of C variables named with an underscore followed by a number, which are valid in C. We think this is an acceptable compromise, as the underscore is short and easily recognizable.

To reference dynamic variables, expansion is used to prefix the address with `_`. Arithmetic expansion can be nested multiple times and is expanded inside-out, permitting read and writes to dynamic variables, and so to memory locations. With this, pointers to objects are simply the address of the beginning of the object just like in C. For example, writing and reading the array `arr` at index `i` is achieved like this:

```
: $((_$((arr + i)) = 42)) # arr[i] = 42
: $((x = _$((arr + i)))  # x = arr[i]
```

3.3.1 Shell's Internal Representation of the Heap. The use of variables to represent the heap can cause performance issues. All shells (`dash`, `bash`, `ksh`, `zsh`, `mksh`, `yash`) store variables using one hash table for all variables, and so the time to access variables varies depending on the number

```c
int fib(int n) {
  if (n < 2) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

C source code

```
_fib() { let n $2
  let __t1; let __t2
  if [ $n -lt 2 ] ; then
    : $(($1 = n))
  else
    _fib __t1 $((n - 1))
    _fib __t2 $((n - 2))
    : $(($1 = __t1 + __t2))
  fi
  endlet $1 __t2 __t1 n
}
```

Using `let` and `endlet`

```
_fib() { set $@ $n $__t1 $__t2
  n=$2
  if [ $n -lt 2 ] ; then
    : $(($1 = n))
  else
    _fib __t1 $((n - 1))
    _fib __t2 $((n - 2))
    : $(($1 = __t1 + __t2))
  fi
  : $((__tmp=$1)) $((n=$3)) $((__t1=$4)) $((__t2=$5)) $(($1=__tmp))
}
```

Using `set`

Figure 3. Comparing `let`/`endlet` with `set` for Fibonacci

of variables in the environment, which directly correlates with the amount of memory allocated by the C program. A consequence of this internal representation of the heap is that as more memory is allocated, the slower many of their operations get. Also, because memory location variables are pieced together dynamically, the shell must hash the variable name, which can impact access time as longer variable names take longer to hash, meaning that smaller addresses are faster to access than larger addresses.

3.3.2 Memory Management. Because shell variables don't need to be declared before they can be used, new memory locations can be created by simply accessing new variables. In turn, those memory locations are added to the shell environment when they are first written to. This separates the allocation of memory and the use of hash table slots.

This simplifies `malloc`'s implementation as it doesn't need to initialize memory and a bump allocator is sufficient to reserve addresses. This allows large memory objects to be allocated without impacting performance as their memory locations occupy the shell environment only if they are touched.

POSIX shell includes the `unset` command, which removes variables from the shell environment. This command is used to implement `free`, to reduce the size of the environment. We note that with the bump allocator design, `free` does not reduce the memory usage towards the memory limit, but because the POSIX standard requires signed long integer arithmetic, meaning at least 2 GiB (2^{31}) can be accessed with this scheme. Here is the implementation of `malloc` and `free`:

```
__ALLOC=1   # 0 is reserved for NULL
_malloc() { # $1 = return variable, $2 = size
  : $((_$__ALLOC = $2))     # Track object size
  : $(( $1 = __ALLOC + 1))  # Assign return value
  : $((__ALLOC += $2 + 1))  # Update bump pointer
}

_free() { # $2 = object to free
  __ptr=$(($2 - 1))          # Start of object
  __end=$((__ptr + _$__ptr)) # End of object
  while [ $__ptr -lt $__end ]; do
    unset "_$__ptr"
    : $((__ptr += 1))
  done
}
```

3.4 Strings and I/O

Another difference between C and POSIX shell is how strings are represented. In C, strings are represented using null-terminated arrays of bytes, which can be used like any other array, while in shell, strings are a special type that can be modified only using the provided API. This API is limited, being mainly useful for concatenating, cutting, and comparing strings, and it does not allow random access to characters – one of the most common access patterns used in C programs.

3.4.1 String Representation. To reconcile this incompatibility, we settled on representing strings as arrays of integer character codes, This requires packing and unpacking the shell strings into arrays of bytes whenever a string crosses from C code to shell code and vice-versa. This operation is required to initialize string literals or to do any kind of I/O, which is very common for programs like compilers.

Converting from a C string to a shell string is relatively simple, as it can be done by outputting each character using the `printf \\`*octal_num* command and capturing the output in a command substitution in this manner:

```
_puts() { # $2 = address of string to print
  __addr=$2
  while [ $((_$__addr)) != 0 ]; do
    printf \\$((_$__addr/64))$((_$__addr/8%8))$((
    _$__addr%8))
    : $((__addr += 1))
  done
}

# $1 = address of string to pack
pack_string() { __str=$(_puts __ $1); }
```

This C string to shell string conversion may be slow on certain shells as it uses a subshell, but since almost all conversions are to then print the string, the subshell can be skipped and the characters be printed directly.

Going the other way is more costly, as extracting the characters is difficult in POSIX shell. Using variable expansion, it is possible to remove the first character of a string using `tail=${string#?}` where "?" matches any character, and then remove this string from the original string using

`${string%"$tail"}` to obtain the first character. This operation is repeated until the string is empty. The function below shows how this can be done:

```
unpack_string() { # $1 = string to unpack
  __str=$1
  _malloc __addr $((${#__str} + 1))
  __ptr=$__addr
  while [ -n "$__str" ] ; do
    __tail=${__str#?}          # Remove head char
    __head=${__str%"$__tail"}  # Get head char
    __byte=$(LC_CTYPE=C printf %d "'$__head")
    : $((_$__ptr = __byte))    # Store byte
    : $((__ptr += 1))          # Advance
    __str=$__tail
  done
  : $((_$__ptr = 0))
}
```

Shells are likely not optimized for this specific use case, so we can expect that extracting 1 character requires traversing the whole string at least once, and potentially allocating a substring. This work is repeated for each character in the string, meaning that the time complexity of unpacking a string is $O(n^2)$, where n is the length of the string. For particularly long lines, the quadratic time can cause considerable slowdown in the program's performance.

To mitigate this problem, we found that extracting longer chunks from long lines and iterating on those smaller substrings is much faster. Specifically, we break long lines into 256-character chunks, then again into 16-character chunks before extracting individual characters.

3.4.2 String Literals.

Given the cost of unpacking strings, the initialization of string literals requires attention as initializing all strings at the beginning of the program could noticeably increase startup time. Also, each string initialization allocates memory which grows the environment and contributes to slowing down the execution. As a result, string literals are initialized the first time they are used, using the `defstr` function below.

```
defstr() { # $1 = variable name, $2 = string
  if [ $(($1)) = 0 ]; then # check if undefined
    # Unpack string with escape sequences
    unpack_escaped_string $1 "$2"
  fi
}

defstr __str_0 "Hello"  # convert "Hello"
_puts __ __str_0        # print string
```

Each string literal is associated with a unique string variable that acts as a cache and is passed to `defstr`. When the variable is empty, the string is unpacked and the result is saved in the variable. The calls to `defstr` are placed just before the string literal is used, improving readability as the string variable is located close to where it is used.

3.5 Magic numbers

We define magic numbers as numbers that appear in the shell code but not literally in the C code. These numbers can obfuscate the code and potentially hide bugs as they can be difficult to verify. These come from the use of character literals, enums and structures, and can be given meaning by assigning each to a `readonly` global variable with a descriptive name.

For character literals, the character codes are assigned to global variables named *__character__* for alphanumerical characters, and *__name__* otherwise. For example, `'A'` is given the `__A__` variable, and `' '` the `__SPACE__` variable. C character literals are then mapped to these shell variables to make the code more readable. This indirection adds at most a few percent to the execution time of the shell, as shown in the table below comparing `pnut.sh` with the character code placed directly in the code to the baseline.

ksh	dash	bash	yash	zsh	
24.0 s	42.7 s	61.2 s	61.3 s	655.7 s	Time
0.98 ×	0.97 ×	1.01 ×	0.99 ×	0.99 ×	Ratio

3.5.1 Structures.

Similarly, because structures are just like arrays, with the fields placed in consecutive memory locations which are accessed by adding the corresponding offset to the structure's base address, the offset of each field is computed by pnut and assigned to a global variable. For example, here is a C program with `structs` and the generated shell code:

```
struct Point { int x; int y; };

struct Point *p;
void init_point() {
  p = malloc(sizeof(struct Point));
  p->x = p->y = 0;
}
```

```
# Point struct member declarations
readonly __x=0
readonly __y=1
readonly __sizeof__Point=2

_p=0
init_point() {
  _malloc _p $__sizeof__Point
  : $((_$((_p + __x)) = _$((_p + __y)) = 0))
}
```

3.6 Printf

C programs often output text with the `printf` function, which is usually implemented as a library function. Most calls to `printf` are with a constant format string, making it possible to decompose the function call into an equivalent sequence of calls to `_putstr`, a function that outputs a C string, and to the shell's `printf` function.

This makes it possible to not include the `printf` implementation in the scripts produced by `pnut`, and save many

N	ksh	dash	bash	yash	zsh
10000	0.03	0.03	0.04	0.05	0.05
100000	0.25	1.27	0.44	0.50	0.44
500000	1.21	34.96	2.14	2.62	2.26
1000000	2.47	312.49	4.24	5.09	4.60

Figure 4. Time in seconds to initialize an array of size N

packing and unpacking of format strings and arguments. This optimization is particularly useful for programs that output a lot of text using `printf`, which includes `pnut` itself.

3.7 Readability

The auditing of `pnut.sh` being an obligatory step to ensure that the existing C compiler hasn't tampered in any way with the generated shell code, `pnut` can include the C source code as comments in the shell code. With this option, each top-level declaration is prefixed with a comment containing the C code from which it was generated. This includes comments from the C code that often contain important information about the code, such as the purpose of a function or the meaning of a variable. This makes it easy to see the correspondence between the C code and the shell code and to verify that the shell code is correct.

Because it can almost double the size of the scripts generated, this option is disabled by default and is only used when generating `pnut.sh` using an existing compiler to ensure that the generated shell code is correct.

3.8 Performance Portability

In general, we observe that the fastest shells are `ksh` and `dash`, while the slowest shells are `bash` and `zsh`. This rule is not absolute, as the performance of a shell can vary greatly depending on the nature of the programs it runs.

3.8.1 Memory Use. As detailed earlier, performance can degrade when the shell environment grows too large, which is proportional to the amount of memory used by the program. This degradation highly depends on the shell, with `ksh` showing almost no sign of performance degradation, while `dash` quickly shows linear access times to variables. Figure 4 shows the time to initialize an array of variable size.

To maximize portability, programs should limit their memory usage or delay it until absolutely necessary. A few ways to do this are to reuse memory locations, free memory as soon as possible, and leave memory uninitialized for as long as possible. In `pnut`, the statically allocated arrays can be left uninitialized using a compile-time option to avoid increasing the environment size immediately on startup. This option is disabled by default, as it deviates from the C standard, but is used for `pnut.sh` to significantly speed it up.

	ksh	dash	bash	yash	zsh
subshell (1000)	0.06	3.12	6.23	4.99	5.21
subshell (10000)	0.06	3.67	8.80	5.70	5.68
subshell (100000)	0.06	6.77	22.77	11.59	9.53
fast (1000)	0.01	0.01	0.04	0.15	0.03
fast (10000)	0.02	0.03	0.04	0.15	0.04
fast (100000)	0.02	0.03	0.04	0.15	0.05

Figure 5. Time in seconds to convert characters using a subshell compared to the fast method with different environment size.

3.8.2 Faster String Conversion. The shell string to C string conversion plays a central part in all the input primitives of `pnut` and can take a significant time for programs that read from the standard input or files, even when the lines are short. This is because opening a subshell to convert each character is slow, and on certain shells becomes even slower as the environment grows as Figure 5 shows.

Instead, a lookup table can be used to convert characters to their character code. This works for alphanumerical characters that can form valid variable identifiers, as the lookup can be done using variable expansion. For other characters, the case statement can be extended to match them. The subshell is then required only for control characters and extended ASCII characters, which are much rarer.

```
__c2i_0=48 ... __c2i_z=122

__char="A" # Convert character to character code
case $__char in
  [a-zA-Z0-9]) __code=$((__c2i_$__char)) ;;
  " ") __code=32 ;;
  *) __code=$(LC_CTYPE=C printf %d "'$__char") ;;
esac
```

This method (fast) is much faster on all shells, which can be seen in the Figure 5 comparing the original method (subshell) to the one using the lookup table when converting the characters of a string containing a mix of alphanumerical and special characters. Because both methods are influenced by the size of the environment, the cases with 1000, 10000 and 100000 variables are shown.

4 Implementation of Pnut

Because the shell can be a tricky platform to build on, pnut was designed to be relatively simple and efficient. This means restricting the use of memory and I/O, which can be expensive on some shells, and trying to keep the code compact since the more code, the more time it takes to bootstrap. As a result, pnut is a single-pass compiler with no external dependency such as Yacc or Lex, using instead a hand-written lexer and recursive descent parser.

The single-pass nature of pnut is not constraining since most optimizations done in a multi-pass compiler are specifically avoided in pnut to favor readability. It also makes

managing memory usage easier as few objects are kept in memory at any given time and static buffers can be reused between each top-level declaration. On the other hand, it also means relatively few checks are done on the program which pnut assumes to be well-formed. This is a reasonable assumption as the compiler's main purpose is to compile itself and specific programs that are known to be good.

4.1 Lexer and Parser

Having no external dependency, pnut's lexer and recursive descent parser are hand-written using a fairly regular structure. Identifiers and keywords are interned, which makes comparisons between identifiers very fast and reduces memory usage as only one copy of each is kept in memory.

One particularity of the C language is its preprocessor, of which pnut supports most features: object and function-like macros, token pasting, stringification, conditional compilation and includes. The preprocessing is done as tokens are read, which somewhat complicates the lexer but allows preprocessing to be single-pass and minimize memory usage.

4.2 Code Generation

Shell code generation is heavily dependent on string concatenation, which can result in quadratic time and memory complexity if implemented naively. To solve this problem, pnut uses a tree-like data structure to represent concatenation. The data structure also differentiates between immutable (mainly coming from the string intern pool and from string literals) and mutable strings, and escaped strings to avoid copying strings when it is not needed. To output the compiled code, the structure is traversed and the strings are outputted to stdout directly, again avoiding allocating more memory than needed.

The tree nodes are stored in a preallocated buffer that is reset and reused between each top-level declaration. Reusing this buffer alone reduces total memory usage from almost 300K shell variables to 120K when compiling pnut-exe which has a significant impact on performance on some shells.

Since the runtime library is embedded in the C code, the functions used by the program are tracked and only the parts of the library that are needed are added to the compiled code. This reduces the amount of code generated which reduces compilation time and bloat of the generated scripts.

4.3 Other Optimizations

The generation of the runtime library is done at the end of the compilation and is more-or-less a series of `printf` calls. Because `printf` calls are handled as a special case where the shell's `printf` directly receives the string literal, this avoids much unpacking and packing of strings.

Local variable management can be costly and is best avoided for small functions that are called very frequently. Non-recursive functions can often have their local variables made global with a simple renaming to prevent conflicts with other global variables. This was done on many of the reader and lexer functions as they are small and called very frequently as an alternative to inlining them since the function call cost is small compared to the variable management cost. That transformation is not done automatically by pnut as it changes the structure of the generated code and cannot be implemented simply given the single-pass nature of the compiler.

```
1 void exit(int status) {
2   asm (
3     "mov   $1, %%eax\n"  /* 1 = SYS_EXIT */
4     ".byte 0xcd,0x80\n"  /* int 0x80 */
5     : : "b" (status)
6   );
7 }
```

Figure 6. An implementation of the `exit` function suitable for TCC and GCC when targeting a Linux computer with `i386` processor. The exit status is passed in the `ebx` processor register and the `int 0x80` instruction calls the Linux kernel to execute the system call specified in register `eax`.

5 C Library

The C language defines a number of standard functions that are part of the C library, a.k.a the `libc`, that is typically linked with all C programs. The prototypes of these functions are defined in header files such as `stdio.h` for the standard I/O functions (e.g. `printf` and `fopen`) and `stdlib.h` for miscellaneous utility functions (e.g. `malloc` and `exit`).

While many of the `libc` functions can be defined using plain C code, some require calling functions inside the operating system kernel. Traditionally, the `libc` functions are defined using a mix of C code and assembler code (possibly through the use of the `asm` construct, which is an extension supported by many C compilers). Figure 6 shows a typical implementation of the `exit` function with the `asm` construct for a Linux computer with `i386` processor.

Interfacing with the operating system through assembler code is inappropriate for `pnut`, so it does not support the `asm` construct. Instead, a specific set of `libc` functions are implemented directly by the code generator. This *builtin* `libc` is thus part of the compiler's logic, as shown in Figures 1 and 2. For example, the code generator knows that a call to the `libc` `exit` function can be translated to shell code that uses the shell's `exit` command. This approach is particularly interesting for implementing calls to the `libc` `printf` function by a direct translation to the `printf` command, which is a required command in the POSIX shell standard. There is a minor drawback that some advanced `printf` format specifiers of the C `printf` are not implemented by the shell `printf`. We believe this is an acceptable tradeoff because the generated code is easier to understand.

The builtin `libc` of `pnut` covers the following functions:

stdlib: exit, malloc, free
stdio: fopen, fclose, fgetc, printf, puts, putchar, getchar
unistd: read, write, open, close

Functions in this set that have no direct translation to standard shell commands, such as getchar, malloc, and open, are implemented by a small runtime system that is added to the generated script. Here too the code of this runtime system is embedded in the compiler, so that the pnut.sh compiler is fully self-contained.

The functions in the builtin libc are the only ones that are called in pnut.c and pnut-exe.c, allowing the second step of Figure 1 and the first two steps of Figure 2 to be done with no extra files needed (i.e. header files and libraries).

The pnut-exe compiler uses a similar approach except the builtin libc does not contain an implementation of printf. Instead, we make use of a custom *external* libc which implements printf and other useful functions in plain C code. In particular, the external libc's printf, which supports advanced formatting specifiers, calls write implemented by the builtin libc. The external libc can also be used by pnut to compile programs relying on a larger set of libc functions. It currently contains the set of libc functions needed to compile TCC (step 3 of Figure 2). Figure 7 lists this set.

6 Evaluation

Pnut is designed to be a practical tool for compiling C to POSIX shell. It meets the goals of readability, portability and reasonable performance, and has enough support of the C language for pnut to compile itself and pnut-exe in a reasonable time, which demonstrates that POSIX shell is a viable "root language" for bootstrapping larger toolchains.

6.1 Readability

While readability is subjective, pnut generates shell code that has the following properties:

- It follows the structure of the original C code, with indentation and using the same names for variables and functions.
- The code is regular, simple and predictable.
- It can include the C code as comment for auditing.

As a result, the pnut.sh script can be read and audited by a programmer without too much difficulty (Appendix A gives an example). As a rough measure of readability, we compare the size of the generated shell script (with no embedded C source code comments) to the size of the original C code (stripped of its comment) in the following table. The pnut.sh and pnut-exe.sh shell scripts have respectively 35% and 30% more LOC than the C code. This increase in size includes the runtime functions that are not present in the C code, making the actual difference smaller in practice.

Pnut version	C source	Shell code	Shell runtime
pnut.c	5270	7194 (1.35 ×)	634
pnut-exe.c	4746	6149 (1.30 ×)	415

stdlib: malloc, free, realloc, exit
stdio: fopen, fdopen, fclose, fputc, fwrite, fputs, puts, vfprintf, fprintf, printf, vsnprintf, snprintf, sprintf
string: memset, memcpy, memmove, memcmp, strlen, strcpy, strcat, strchr, strrchr, strcmp

Figure 7. The set of functions provided by the external libc.

6.2 Portability

Pnut is able to bootstrap itself on all tested shells, despite some shells deviating slightly from the POSIX standard. It has been tested on the shells ksh, dash, bash, yash, and zsh, and on Linux (x86_64), MacOS (ARM), and Windows (x86_64 on WSL), all of which can run pnut.sh and compile pnut-exe without any issues. This includes bash version 2.05b from 2002, which demonstrates the stability of the POSIX shell standard and the non-reliance on more modern shell features and bashisms. This is not surprising as no external utilities are used, nor any platform-specific code, making pnut.sh exceptionally portable.

6.3 Performance

Pnut's performance can be evaluated in two ways: the time taken by pnut.sh to compile a C program to shell and the execution time of the generated shell script.

In addition to the time taken by pnut.sh to compile pnut-exe.c, we also measure the time to compile smaller programs using pnut.sh. These programs include a simple hello world (hello.c), a program that prints the first 20 Fibonacci numbers (fib.c), a program counting the number of characters, words and lines of a file (wc.c), a file reading program (cat.c), program computing the SHA256 of a file (sha256sum.c), the C4 compiler[16] [1] (c4.c) and the Ribbit Virtual Machine [15] (repl.c) running a R4RS Scheme REPL. The compilation options used are the same ones that are in the baseline times of Section 3 using the let/endlet functions to manage local variables, character literals variables and the fast runtime library. The compilation times are shown in Figure 8 and are compared to pnut executables produced by GCC (with the -O3 optimization level) and by pnut.exe (i386 version).

Except for small programs, the execution of pnut.sh takes a few seconds even on the fastest shells. These long compilation times mean that using pnut.sh directly is not practical for compiling large programs. The slow compilation is due to the shell's slow execution speed as pnut compiled with GCC and pnut-exe are much faster and can compile large programs in a reasonable amount of time. pnut.sh being slow is not an issue since it is only meant to compile the

[1] The c4.c compiler is a minimal C compiler that can compile itself. It can be compiled by pnut with the addition of a single memset call and adjusting some buffer sizes.

	ksh	dash	bash	yash	zsh	pnut.exe (GCC)	pnut.exe (pnut-exe)
hello.c (5 LOC)	0.028s	0.019s	0.059s	0.059s	0.085s	0.001s	0.004s
fib.c (19 LOC)	0.089s	0.059s	0.190s	0.197s	0.314s	0.001s	0.005s
cat.c (41 LOC)	0.183s	0.119s	0.407s	0.427s	0.666s	0.001s	0.018s
wc.c (64 LOC)	0.281s	0.188s	0.655s	0.673s	1.246s	0.001s	0.021s
sha256sum.c (233 LOC)	1.512s	1.105s	4.058s	4.078s	9.007s	0.001s	0.034s
c4.c (529 LOC)	5.494s	5.274s	13.956s	14.259s	67.350s	0.002s	0.085s
repl.c (814 LOC)	13.361s	13.650s	23.543s	19.614s	105.406s	0.002s	0.079s
pnut.c (6698 LOC)	31.008s	72.997s	76.548s	76.455s	1094.586s	0.006s	0.469s
pnut-exe.c (6296 LOC)	24.991s	44.610s	61.606s	62.309s	660.516s	0.005s	0.368s

Figure 8. Compilation times of programs using pnut.sh per shell. The times taken for the pnut.exe executable made from GCC and pnut-exe to compile the same programs are included for comparison.

	ksh	dash	bash	yash	zsh	GCC	pnut-exe
hello.c (5 LOC)	0.002s	0.001s	0.001s	0.002s	0.001s	0.001s	0.001s
fib.c (19 LOC)	0.784s	0.399s	1.617s	1.948s	2.557s	0.001s	0.001s
cat.c (41 LOC)	2.828s	1.189s	3.611s	3.992s	4.508s	0.001s	0.001s
wc.c (64 LOC)	3.174s	1.627s	5.009s	5.611s	7.615s	0.001s	0.001s
sha256sum.c (233 LOC)	5.964s	2.446s	7.478s	8.286s	6.831s	0.002s	0.008s
c4.c (529 LOC)	1.691s	18.330s	3.903s	3.969s	23.144s	0.001s	0.058s
repl.c (814 LOC)	3.251s	21.117s	6.241s	6.469s	52.086s	0.001s	0.002s

Figure 9. Execution times of small programs produced by pnut.sh per shell. The time taken for the same executables compiled by GCC and pnut-exe are included for comparison.

faster pnut-exe executable, and this step is only required when used for reproducible builds.

The execution times of the generated shell files are shown in Figure 9. Again, we include for reference executables produced by GCC (with the -O3 optimization level) and from pnut-exe. C4 is given its own source code as input, the R4RS REPL receives a simple hello world program and other programs that take a file as input use a 64KiB file with 512 characters per line and 128 lines. The execution time of the hello.c is almost instant, showing that the initialization time of the generated script is minimal, fib.c takes a few seconds which is expected as it does mostly arithmetic and function calls, which the shell is not particularly good at. The other scripts that do I/O are between 1000 and 10000 times slower than the executables produced by GCC, but still only take a few seconds to read and output a file or to count the number of characters, words and lines of a file. The sha256sum and repl programs are the slowest as they combine I/O with many arithmetic operations, but are still usable on small inputs.

6.4 Use for Reproducible Builds

As of now, pnut is complete enough to bootstrap itself and compile pnut-exe. Work to build TCC with pnut-exe is ongoing. Once this is achieved, building GCC is easy as it can be done with a known recipe from TCC.

7 Related Work

As far as we know, Pnut is the only C to POSIX shell compiler that supports a large enough subset of C to be able to compile itself. There are some compilers for domain-specific languages that produce shell scripts, all with the explicit goal of simplifying shell scripting:

- Amber [12]: Aims to facilitate shell scripting, produces scripts that are Bash-compatible. Written in Rust.
- Batsh [8]: C-like language that can be compiled to both Bash and Windows Batch. Written in OCaml.
- Bish [6]: Aims to make Bash scripting easier with a more modern syntax. Written in C++.
- Powscript [20]: Transpiler from a CoffeeScript-like language to Bash, with an option to generate POSIX compatible code. Written entirely in Bash.

These projects show that there is interest for higher-level languages that compile to shell scripts. However, all of these projects require the use of a custom language, which can be a barrier to entry for some users. In contrast, Pnut works with most standard C code and doesn't require programmers to learn a new language. And because it is self-applicable with no external dependencies its installation is trivial which makes it accessible to an even wider audience.

On the reproducible build side, a similar project is Guix's Full-Source Bootstrap [5] whose goal is to root the Guix package ecosystem in a minimal set of trusted binaries. It does so by compiling the GNU toolchain (GCC, make, binutils)

starting from a 357-byte seed program that can be reasonably audited manually given its size. Since so much software is built on top of C, this allows most Guix packages to be built from source in a fully reproducible way.

This approach is fundamentally different from the one `pnut` takes as it tries to remove pre-built binaries as much as possible from the bootstrapping process while `Pnut` assumes the existence of a shell. The consequence of Guix's approach is a reliance on a large amount of x86 machine encoding of instructions and assembler code that require experts to audit. Both approaches are likely complementary by having `pnut.sh` or scripts generated with it be part of the seeds used in the full-source bootstrap project as they can be manually audited unlike pre-built binaries.

Time will tell if pnut can contribute to the Guix Full-Source Bootstrap as the project is still in progress. As noted by Tournier [19], the binaries for the `Guile` Scheme compiler, `Bash`, `tar`, `mkdir` and `xz`, totaling 25 MB in size, are required to initiate the bootstrap process. It is unclear how the project will rid itself of these dependencies without having to rewrite part of their source because so much of the full-source bootstrap now relies on them.

This dependency on `bash` is convenient for `pnut` since it is a POSIX shell. This means that it could potentially be used to reduce the size of the prebuilt binaries by compiling some of them, `tar` for example, using only `pnut.sh` and `bash`, and without having to rewrite them from scratch.

Additionally, by starting from a minimal seed, Guix's Full-Source Bootstrap includes numerous steps to reach TCC and GCC, and because so many of the tools are purpose-built, auditing the entire source code is a significant task. In contrast, we think `pnut` can bootstrap these binaries in fewer steps and less code to audit, with all code being written in C and readable shell, making the task easier.

8 Conclusion

We have presented `pnut`, a C to POSIX shell transpiler written in C that can compile itself and that generates human-readable shell code. Because the compiler is self-applicable, `pnut` can be distributed as a human-readable shell script, `pnut.sh`, that can run on any system with a POSIX compliant shell. This makes `pnut.sh` a practical tool for bootstrapping a complete build toolchain reproducibly as it can be easily audited, which defeats possible "trusting trust" attacks that are generally ignored as they are difficult to detect.

Additionally, because POSIX shell is not a typical compilation target, we discussed the design choices that make the generated code humanly readable while preserving the structure and semantics of the original C code. Special attention to performance was given to ensure that `pnut.sh` and other generated scripts are reasonably fast on all shells so that all POSIX compliant shells can make use of `pnut`, and performance pitfalls that happen when using the shell as a compilation target are documented with ways to avoid them.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada. We also thank Manuel Serrano for his ideas on ways to compile TCC with Pnut.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools* (2 ed.). Addison-Wesley. Visited on 2024-07-01.

[2] Kenneth Almquist. 1989. Almquist shell. https://www.in-ulm.de/~mascheck/various/ash/ Visited on 2024-07-01.

[3] Andrew W. Appel. 1994. Axiomatic bootstrapping: a guide for compiler hackers. *Transactions on Programming Languages and Systems* 16, 6 (1994), 1699–1718. https://doi.org/10.1145/197320.197336

[4] Morris Bolsky and David Korn. 1983. KornShell. https://github.com/ksh93/ksh Visited on 2024-07-01.

[5] Ludovic Courtès and Janneke Nieuwenhuizen. 2023. The Full-Source Bootstrap: Building from source all the way down. https://guix.gnu.org/en/blog/2023/the-full-source-bootstrap-building-from-source-all-the-way-down/ Visited on 2024-07-01.

[6] Tyler Denniston. 2015. Bish. https://github.com/tdenniston/bish Visited on 2024-07-01.

[7] Brian Fox et al. 1989. Bourne-Again SHell. https://git.savannah.gnu.org/cgit/bash.git Visited on 2024-07-01.

[8] Carbo Kuo et al. 2014. Batsh. https://github.com/batsh-dev-team/Batsh Visited on 2024-07-01.

[9] Fabrice Bellard et al. 2001. Tiny C Compiler. https://bellard.org/tcc/ Visited on 2024-04-15.

[10] Herbert Xu et al. 1997. Debian Almquist shell (DASH). https://git.kernel.org/pub/scm/utils/dash/dash.git Visited on 2024-07-01.

[11] Paul Falstad et al. 1990. Z shell. https://www.zsh.org/ Visited on 2024-07-01.

[12] Paweł Karaś et al. 2023. Amber. https://github.com/Ph0enixKM/Amber Visited on 2024-07-01.

[13] WATANABE Yuki et al. 2009. Yet another shell. https://github.com/magicant/yash Visited on 2024-07-01.

[14] Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol.* MIT Press.

[15] Léonard Oest O'Leary, Mathis Laroche, and Marc Feeley. 2023. A R4RS Compliant REPL in 7 KB. arXiv:2310.13589 [cs.PL] https://arxiv.org/abs/2310.13589

[16] Robert Swierczek. 2014. c4 - C in four functions. https://github.com/rswier/c4/tree/master Visited on 2024-07-01.

[17] The Open Group 2018. *The Open Group Base Specifications Issue 7.* The Open Group. https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html

[18] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (aug 1984), 761–763. https://doi.org/10.1145/358198.358210

[19] Simon Tournier. 2023. Is Guix full-source bootstrap a lie? https://simon.tournier.info/posts/2023-10-01-bootstrapping.html Visited on 2024-07-01.

[20] Leon van Kammen. 2016. Powscript. https://github.com/coderofsalvation/powscript Visited on 2024-07-01.

[21] David Wheeler. 2005. Countering Trusting Trust through Diverse Double-Compiling. In *Annual Computer Security Applications Conference.* 33–48. https://doi.org/10.1109/CSAC.2005.17

A Compilation Example

In this appendix we give a code generation example to illustrate some of the features of `pnut`.

Figure 10 is the compilation to POSIX shell code of the `accum_digit` function found in the `pnut.c` file, and thus this shell code is part of `pnut.sh`. The compilation was done while activating the option to include the C source code as comments in the shell code, so the C source code is readily accessible to any party interested in understanding and auditing the code (as well as you the reader of this paper).

`accum_digit` is part of the tokenizer for integers and string escapes. It looks at the current character, in the C global variable `ch`, and checks that it is a valid digit in the given `base`, and if so accumulates it in the C global variable `val`. It advances to the next character by calling `get_ch` before returning whether a digit was accumulated.

It is noteworthy that the shell code and C source code have the same structure, and are of a similar complexity to understand. The shell code uses variables to refer to specific characters, so the meaning is clear. The uses of `let`, `endlet` and the undescore prefix on the variables helps clarify the scope of the variables (`base` and `digit` are local, and `_ch` and `_val` are global).

```
###################### C code ######################
# int accum_digit(int base) {
#   int digit = 99;
#   if ('0' <= ch && ch <= '9') {
#     digit = ch - '0';
#   } else if ('A' <= ch && ch <= 'Z') {
#     digit = ch - 'A' + 10;
#   } else if ('a' <= ch && ch <= 'z') {
#     digit = ch - 'a' + 10;
#   }
#   if (digit >= base) {
#     return 0; /* char is not a digit in that base */
#   } else {
#     val = val * base - digit;
#     get_ch();
#     return 1;
#   }
# }
################### End of C code ###################
: $((digit = base = 0))
_accum_digit() { let base $2
  let digit
  digit=99
  if [ $__0__ -le $_ch ] && [ $_ch -le $__9__ ] ; then
    digit=$((_ch - __0__))
  elif [ $__A__ -le $_ch ] && [ $_ch -le $__Z__ ] ; then
    digit=$(((_ch - __A__) + 10))
  elif [ $__a__ -le $_ch ] && [ $_ch -le $__z__ ] ; then
    digit=$(((_ch - __a__) + 10))
  fi
  if [ $digit -ge $base ] ; then
    : $(($1 = 0))
  else
    _val=$(((_val * base) - digit))
    _get_ch __
    : $(($1 = 1))
  fi
  endlet $1 digit base
}
```

Figure 10. C to POSIX shell compilation of the `accum_digit` function found in the `pnut.c` file.

B POSIX Shell Overview

The POSIX shell language is relatively basic, offering few of the facilities of modern programming languages. Most shell implementations extend this language in various ways, but those extensions are not standardized and not portable. Being mainly used to manipulate various forms of strings, most of the shell's features center around this.

B.1 Strings and Expansion

Strings can be expressed with double quotes, single quotes, and no quotes when the string's characters have no special meaning for the shell. Shell variables are assigned string values with the syntax *name=string*.

Variables are accessed using the expansion mechanism. Commands are first expanded before being executed. Expansion is central to how the shell works and is how most operations are done. For our needs, the expansion steps are:

1. Variable expansion
2. Arithmetic expansion
3. Command substitution

The `$` character introduces all these expansion kinds. For variable expansion to happen, the syntax `$`*name* is used, or the alternative `${`*name*`}` which is useful in situations where the expansion needs to be immediately followed by a character that is valid for variable names. The value of the variable is then substituted in its place. Expansion of undefined variables is legal, and produces an empty string. Expansion also can be transformed with the following forms:

- `${#`*name*`}`: Length of the variable value
- `${`*name*`#`*pattern*`}`: Remove shortest prefix pattern
- `${`*name*`%`*pattern*`}`: Remove shortest suffix pattern
- `${`*name*`-`*value*`}`: `$`*name* if non-empty, otherwise *value*

In these forms the pattern is itself the subject of variable expansion, which is useful to match a computed pattern.

Arithmetic expansion runs second and is used to evaluate integer expressions with the syntax `$((`*expression*`))`. Section B.2 explains this in more details.

Command substitution comes next and handles the syntax `$(`*command*`)` and the alternative syntax `` `command` ``. This expansion is done by running the command in a subshell, and substituting it with the standard output of the command. A subshell works like a subprocess, inheriting all characteristics of its parent. This includes variables and function declarations. However, any changes in the subshell are discarded when it ends. It is the idiomatic way to capture the output of a command in a variable for later use, for example:

```
num=42 thing=" black cat"
x="$num${thing}s"  # x = "42 black cats"
echo "length of \"$x\" is ${#x}"
hex=$(printf "0x%x" $num)  # capture output
echo "$num = $hex"         # prints: 42 = 0x2a
```

B.2 Arithmetic Expansion

The arithmetic expansion `$((`*expression*`))` evaluates the expression using signed integer arithmetic with at least 32 bit precision[2]. Most of the C integer arithmetic operators are allowed in the expression, with the exception of the increment and decrement operators, the `sizeof` operator and prefix `&` and `*` operators. Moreover, function calls are not allowed in expressions. Operators have the same precedence as C. This close similarity is convenient for translating C to shell.

In expressions, variables are accessed without the `$` prefix, with empty or undefined variables replaced by 0. Assignment operators such as `=` and `+=` are supported, and update the variable outside the expression, declaring it if needed. For example, `$(( (x = y = 1 << 5) > 10 ))` expands to 1 (true) and assigns 32 to the variables `x` and `y`.

A very important and perhaps less known feature of arithmetic expansion is that it can be nested. In appearance, this is not any more useful than using 1 level of expansion, but because of the order of expansion, it can be used to define variables with dynamic names, as in this example:

```
a_6=42 a_7=5 i=7
: $(( a_$i += 1 ))        # Increment a_7
echo $(( a_$((a_$i)) ))  # Output a_6, i.e. 42
```

This level of indirection is essential to implement arrays in POSIX shell as it does not support arrays natively. This avoids having to use a more general evaluation method such as `eval` which is considered bad practice as it can execute any shell command. However, using arithmetic expansion to implement arrays limits the array elements to integer values.

B.3 Variables and Functions

Shell functions can be declared and called like so:

```
foo() {
  ...
}

foo 1 2 3  # Call foo with arguments 1, 2 and 3
```

The shell does have a `return` statement to immediately return from a function, but unlike C and other languages it does not allow specifying a value. Instead, it is customary to write the return value in a variable and have the caller access this variable to retrieve the value.

Shell functions do not declare their parameters. This is because functions can take any number of arguments which are received in the function using positional parameters – the first argument is in `$1`, the second in `$2` and so on. The syntax `$#` expands to the number of arguments passed to the function. It is also possible to refer to all parameters at once using the syntax `"$@"` which expands to the function parameters separated with whitespace.

[2]We infer this from the POSIX standard specification "Only signed long integer arithmetic is required."

The following commands interact with the parameters:

- `set` replaces the positional parameters with the arguments it receives.
- `shift` drops the first parameter of the function and shifts the positional parameters by 1 to the left. This can be used to iterate over a function's arguments:

```
concat() {
  string=
  while [ $# -ge 1 ]; do # While there are params
    string="$string$1"   # Concat string and $1
    shift                # Drop the first param
  done
}

concat "abc" "def" "ghi" "jkl" "mno"
printf "The alphabet begins with $string"
```

The positional parameters of a function and `$@` are locally scoped. As we will see, this is very convenient as all other shell variables are global.

B.4 Input and Output

The POSIX shell's standard ways to do I/O include:

- `echo`: Outputs its arguments to the standard output with a newline at the end.
- `printf` *format arg1 arg2...*: Formats a string and outputs it to the standard output. Supports `%s`, `%d`, `%o`, and `\`*octal_num* in the *format* string like C's `printf`.
- `read` *var1 var2...*: Reads a line from the standard input, splits it on the delimiters specified by the `IFS` (Internal Field Separator) shell variable and stores the words in the variables passed as arguments. The `-r` option disables "backslash escaping".

A noteworthy limitation of the `read` command is that it can't read null bytes, so it isn't possible for the shell to read binary files. However, `printf` can output any byte, so it can create binary files.

The output of commands can also be redirected to other file descriptors, such as standard error or a file. Files can be opened for reading and writing using the `exec` command and the `<` (read from), `>` (write to) and `>>` (append to) redirection operators, and file descriptors can be referred by their number prefixed with `&`. To close a file descriptor, the `exec` command with the `<&-` operator can be used. A text file `a.txt` can be copied to the file `b.txt` like this:

```
exec 3< a.txt  # Open a.txt in read mode as fd 3
exec 4> b.txt  # Open b.txt in write mode as fd 4

IFS=                        # Don't split line
while read -r line <&3; do  # Read line from a.txt
  printf "%s\n" "$line" >&4 # Write it to b.txt
done;

exec 3<&-  # Close files a.txt and b.txt
exec 4<&-
```

Received 2024-07-01; accepted 2024-08-30