Comefrom0x10 Documentation

Release 0x10.1

Joe Ulfers

Jan 19, 2018

Contents:

Comefrom0x10, pronounced "Come from sixteen" and commonly abbreviated to just "Cf0x10" is a programming language dedicated to bringing the awesome power of the `comefrom` statement into the modern age. Inspired by ComeFrom2 and Come Here, Comefrom0x10 advances the state of the art by embracing the structured programming movement.

Comefrom0x10 features:

- Named identifiers!
- Structured programming!
- Lexical closure!
- Clean, modern syntax!
- Memory management!

Download now!

CHAPTER 1

Hello world

It can't get much simpler:

```
'hello world'
```

CHAPTER 2

If you are coming from another language

Cf0x10 will be familiar and easily understood by anyone familiar with languages of equivalent expressiveness, like Perl, Java and MUMPS.

- Built-in types are string and number
- Values zero and the empty string are falsy
- Undefined variables implicitly coerced to a value of the correct type where necessary, so `n = n + 1` is `0 + 1` when n is undefined
- Assignment is illegal in `if` conditions.
- Indentation, blank lines and comments are syntactically significant. *Cf0x10 doesn't throw away your precious code.* But, unlike Python, where spaces versus tabs can create confusion, cf0x10 only allows spaces for indentation.
- No more confusion about case sensitivity: Cf0x10 only allows lowercase identifiers. Of course, this will still be half confusing because string comparison is case-sensitive.
- No need for traditional method overloading because the power of comefrom allows you to use the same block name as many times as you please for different purposes
- Strongishly typed

2.1 Best of breed influences

- Outputs expressions not otherwise captured, as in PowerShell
- Even less noisy syntax than Python
- Space is an operator, as in CSS
- No fatal errors, as in JavaScript/browser/on error resume next

CHAPTER 3

Why use Comefrom0x10?

It is well known that reading programs is often harder than writing them. The power of the comefrom statement, however, encourages programmers to avoid deeply-nested control structures and makes cf0x10 programs much easier to read than they are to write.[1]

Business-friendly licensing, under the WTFPL

Comefrom0x10 programs are, on average, shorter than programs written in other languages.[2]

Designed from the start with full Unicode support, including astral plane characters (handle emoji with ease). No more mucking about with surrogate pairs, no legacy non-Unicode cruft.

Ready for quantum computing: see *CUM 1: Schrödinger's conditionals*.

3.1 Install

Comefrom0x10 requires Python 3.4 or newer.

Install from source, because that's how real hackers install programs:

```
hg clone https://julfers@bitbucket.org/julfers/comefrom0x10
cd comefrom0x10
./setup.py install
```

Execute thus:

```
cf0x10 path/to/program.cf0x10
```

Run `cf0x10 --help` for more info.

[1] To the extent permitted by law, the author makes no warranties, express or implied, regarding whether the aforementioned programs are understandable by humans

[2] Not necessarily programs of equivalent functionality.

3.2 Tutorial

3.2.1 Hello World

Comefrom0x10 writes the result of any expression not captured by assignment to stdout. This makes it an ideal teaching language, as students don't need to worry about the weird way that computer programmers use the word "print." If you are a computer programmer, you might say that this prints "Hello, world":

```
'Hello, world' # prints "Hello, world"
```

Notice that # delimits comments.

3.2.2 Variables

Comefrom0x10 has variables just like most other languages. It uses duck-typing, because ducks are delicious, when prepared just so, with a nice crispy skin, and does away with pesky declaration nonsense. Just assign to a variable to create it:

```
greeting = 'Hello'
```

Strictly speaking, assignment doesn't create the variable. As with "hoisting" in JavaScript, names are in scope throughout the block. Before assignment, the name `greeting` exists, but with the value `undefined`.

In fact, it's even simpler than that. It (usually) doesn't matter whether a name is in scope. A Cf0x10 design principle is that your programs should never have fatal runtime errors, thus Cf0x10 programs, like shell scripts, are extremely robust against crashes. You are free to refer to a name that's never assigned at all, if you wish, so this is a complete and valid Cf0x10 program, that just prints nothing at all:

```
greeting
```

Whereas, this program prints "Hello":

```
greeting = 'Hello'
greeting
```

See *Types* in the documentation for specifics.

Later in this tutorial, we will see that assignment is closely related to control flow.

3.2.3 Strings

Cf0x10 allows double or single quotes to be used interchangeably. This avoids the considerable confusion found in languages where double and single quotes have subtly different meanings. These are exactly the same:

```
'Hello, world'
"Hello, world"
```

If you run the program above, you'll find there is a newline after the first "Hello, world," but none after the second. If you want an explicit newline, use backtick for the usual escapes that most programming languages have:

```
'`n is a newline'
'`` is a literal backtick'
'`' is a literal single quote'
```

Hint: If you do not want the newline between lines of output, use the continuation operator, three dots. This prints "Hello, world" all on one line:

```
'Hello, '...
'world'
```

String concatenation

Comefrom0x10 has very simple string concatenation, making it, like PHP, an ideal language for Web templating[1].

Unlike other languages that mostly throw away whitespace, Comefrom0x10 uses space as an operator, for string concatenation. This is like C-style string literal concatenation, but more powerful, as it's a true operator and, thus, not limited to string literals only:

```
who = "world"
"Hello, " who
```

The space operator is required. This does not compile:

```
who = "world"
"Hello, "who
```

3.2.4 Math

Cf0x10 has ordinary number literals and the usual math operators:

```
2 + 2 # prints 4
2 * 2 # prints 4
1 / 2 # prints 0.5
```

Notice that it does sensible division, with floating point as necessary. There is no floor (aka "integer") division operator, nor does Cf0x10 include confusing anti-features of other languages like "modulus" or "exponent."

Unlike most languages that don't do mathematically-correct division by zero, in Cf0x10, this gives the correct result:

```
1 / 0 # undefined
```

3.2.5 Types

Comefrom0x10 has two types of variables, strings and numbers. The space operator coerces its operands to strings, where `undefined` becomes the empty string:

```
2 2 # prints 22
'foo' undefined # prints "foo"
```

And mathematical operators coerce `undefined` to zero:

```
2 + undefined # prints 2
```

[1] Assuming you don't mind being pwned

Note: As in early JavaScript, `undefined` is not reserved, it's a perfectly fine variable name.

Being strongishly typed, however, Cf0x10 doesn't try to make sense of nonsensical operations like adding strings to numbers:

```
2 + 2 # prints 4
2 + '2' # prints nothing
```

In the second line, the result of adding a string to a number is `undefined`, so, as seen in *Variables*, it prints nothing at all.

Lists

As any Lisper knows, all data structures are just fancy forms of lists. In recent decades, however, a new movement, stringly-typed programming, has discovered that all lists are just fancy forms of strings. Cf0x10 embraces this movement, though there is a proposal to add list support, *CUM 2: Lists*.

Truthiness

As in C, Cf0x10 has no boolean type. Numbers are good enough:

```
0 # zero is falsy
1 # all other numbers are truthy
'' # empty strings are falsy
'false' # non-empty strings are truthy
1 / 0 # undefined is falsy
```

Next, we'll exercise Cf0x10's full complement of boolean operators to unlock this power.

3.2.6 Control flow

The unique power of Comefrom0x10 arises from its approach to control flow. It has only one control flow construct, the `comefrom` statement.

Control flow jumps to wherever you place a comefrom statement:

Program	Output

`1`
 	
`2`
`comefrom`
`3`
	`1`
	`3`

As previously seen, Cf0x10 is designed on the principle that *all* your code matters, including blank lines. Think of a blank line as a yield point, where the program is allowed to jump to an eligible `comefrom`.

If more than one is eligible, it jumps to the last:

Program	Output

`1`
 	
`2`
`comefrom`
`3`
`comefrom`
`4`
	`1`
	`4`

Conditionals

Comefrom statements may optionally include an `if` condition. Conditionals are more specific than bare `comefrom` statements, so they supercede the last-wins rule shown above (remember than non-zero numbers are truthy):

Program	Output

`1`
 	
`2`
`comefrom if 1`
`3`
`comefrom`
`4`
	`1`
	`3`
	`4`

Another distinction of `comefrom if` is that jumps are eligible when a variable changes; that is, consider assignments to be yield points.

Program	Output

`foo = 'truthy'`
`1`
`comefrom if foo`
`2`
	`2`

Hint: It may help to understand this as analogous to "pub-sub" frameworks. You can think of an assignment like `foo = 1` as publishing a "foo changed" event and `comefrom if foo` as subscribing to the "foo changed" event. Most languages require libraries to achieve this kind of action at a distance, but Comefrom0x10 builds it right in.

Assignments only cause jumps to conditional comefroms that refer to the variable being changed. This is handy to avoid infinite loops in many situations, for example:

```
comefrom if x
'x is ' x
x = 'foo'
```

The first time `x = foo` runs, `x` changes from `undefined` to "foo". The second time, however, `x` does not change, so the assignment does not trigger a jump back to the first line. This program outputs:

```
x is
x is foo
```

Recall that string concatenation coerces `undefined` to the empty string.

Of course, assignment never jumps to an unconditional comefrom.

Loops

Thanks to the incredible power of `comefrom`, Cf0x10 needs no loop keywords at all:

Program	Output

`comefrom if i < 4`
`i`
`i = i + 1`
	1
	2
	3

3.2.7 Blocks

So far, we've only seen examples of very straightforward Cf0x10 programs, which execute one line after another except for a `comefrom` jump. At this point, you might be thinking, "Meh, I could do that in Intercal," so it is time to see where Cf0x10 separates itself the rest: it allows structured programming with lexical scope!

That's right, it really is a language for the twentieth century.

Like all the good modern languages, Cf0x10 uses syntactically significant indentation. That is, indentation creates a block:

Program	Output

`leave_this_place`
`  "Goodbye, cruel world"`
 	
`"Hello, world"`
	`Hello, world`

In the above, execution begins in the top-level scope. Nothing ever jumps into the block named `leave_this_place`, so none of the code in that block executes.

For the moment, you can think of blocks as coroutines in other languages, but that's not really accurate, as we'll soon see.

Qualified comefrom

First, how to jump into and between blocks? In the previous part, we saw "conditional" comefrom, now we see "qualified" comefrom:

Program	Output

`foo`
`  1`
 	
`  comefrom bar`
`  3`
 	
`bar`
`  comefrom foo`
`  2`
 	
`  4`
	`1`
	`2`
	`3`
	`4`

When qualified, as in `comefrom foo`, the program only jumps to that location from yield points in the block named `foo`.

Let's break down how this works.

Execution begins at the line just after `foo`, which prints `1`. Cf0x10 has no magic names like "main;" if there is no code outside of a block, execution begins at the first block in the file. Next:

1. Blank line yields
2. Jumps to the `bar` block, at the line `comefrom foo`
3. Prints `2`
4. Blank line yields
5. Jumps back to the `foo` block, at the line `comefrom bar`
6. Prints `3`

So far this is all pretty obvious, but now we get to the amazing part. How does `4` get printed? We just jumped back to `foo`, so how do we get to `bar` again?

Recall that the program jumped from the middle of the `bar` to `foo`. After printing `3`, the `foo` block ends and execution returns from whence it came. The next line in `bar` prints `4`. Isn't structured programming great?

Notice that even though the previous jump was from `foo` to `bar`, execution never returned to the blank line in `foo`, otherwise the program would have printed another `3` at the end. This is one way Cf0x10 blocks are significantly different, (and simpler!) than coroutines.

Comefrom ordering

We previously saw that multiple eligible comefrom jumps are resolved in the order they appear in the source. This only applies, however, within the a single scope. When multiple comefroms are eligible targets in *separate* scopes, *all* of them will execute.

Comefrom0x10 picks exactly one comefrom in each scope to execute, then jumps to the comefroms in a well-defined order. The comefrom in the current scope executes last, that is, after comefroms in nested scopes.

This ordering is *not* recursive. That is, comefroms in nested scopes execute top-down, in source code order, depth-first.

While these rules may seem complex, they are actually the most intuitive known solution to the multiple-comefrom problem[1]. An example may make this more clear:

[1] In many languages with comefrom-based constructs, multiple comefroms are simply an error. See *Comefrom* for more motivation.

Program	Output

outside
  # blocks cannot start with blank line
 	
  "don't print this"
  comefrom
  5
  inside
  comefrom
  "don't print this"
  comefrom
  1
  deeper_one
  comefrom
  inception
  comefrom
  3
  2
  deeper_two
  comefrom
  4
	1
	2
	3
	4
	5

When this program arrives at the first blank line in the `outside` block, *all* `comefrom` statements are eligible targets. They execute in this order:

1. Second `comefrom` in the `inside` block, by the last-first and top-down rules
2. Only `comefrom` in `deeper_one`, by the first-first and top-down rules
3. Only `comefrom` in the `inception` block, by the depth-first rule
4. Only `comefrom` in `deeper_two` (figure out why)
5. Only `comefrom` in the `outside` block, by the current-scope-last rule

Notice that each block finishes executing before the program moves to the next comefrom.

Scope

Naturally, you'll need to modularize your Cf0x10 programs. Block-scoping lets you do this without worrying about things like "classes" in other languages. Block scope works essentially the same way as in every other modern language, so we can put this all together into an idiomatic Cf0x10 program:

```
'start program'
until = 6
loops = 0

'n is ' n ' after ' loops
count
  comefrom
  'start counting'
  n = 0
  #
  next
    comefrom if next
    next = 1/0
```

```
    loops = loops + 1
    n = n + 1

  comefrom next if n < until
  'n is ' n
  next = 1
  'done counting'

never_runs
  comefrom next if loops
  'foobar'
```

This is a rather complicated method of printing the numbers from 1 to 5. It prints:

```
start program
start counting
n is 0
n is 1
n is 2
n is 3
n is 4
n is 5
done counting
n is  after 6
```

This is a big jump in complexity, so let's break it down line-by-line.

The program starts at the first line, which just prints "start program":

```
'start program'
```

Since changing `until` does not affect any `comefrom` statements, there are no jumps at the assignment:

```
until = 6
```

Next, the blank line yields and the program jumps to the first line in the `count` block:

```
comefrom
```

Then it prints "start counting":

```
'start counting'
```

Initializing `n` to zero actually does not cause a jump, because the only `comefrom` that refers to `n` is qualified by `comefrom next`:

```
n = 0
```

The empty comment line is for aesthetic reasons, just to separate the block declaration for `next` from the lines above it:

```
#
```

Exercise

Why can't the empty comment be a blank line?

Execution passes the `next` block declaration. Blank lines that immediately follow blocks do *not* trigger jumps, so the next line that executes is a comefrom. To help write robust programs, Cf0x10 is designed to avoid side-effects as much as possible, thus, executing a `comefrom` lines never does anything:

```
comefrom next if n < until
```

Now, it prints the current value of `n`, that is "n is 0":

```
'n is ' n
```

Whoa! It looks like the next line redefines `next`. In fact, it does not. For easier programming, Cf0x10 doesn't have first-class blocks, as languages like Java has shown us that most programmers don't want or understand them. It's quite common in Cf0x10 to use a variable with the same name as a block. In this case, we're using it to cause a jump:

```
next = 1
```

Because `next`, the variable changes from `undefined` to `1`, the program continues in `next`, the block:

```
comefrom if next
```

Resetting `next`, the variable, to `undefined` does not cause a jump, since the `if next` evaluates to falsy after this assignment (assigning to zero would have worked just as well):

```
next = 1/0
```

The variable `loops` is defined in the outermost scope. This does not trigger a jump into the block `never_runs` because the block `next` is not visible from the scope of `never_runs`:

```
loops = loops + 1
```

Incrementing `n` triggers a jump back into the `count` block:

```
n = n + 1
```

So, `count` continues at:

```
comefrom next if n < until
```

And it prints the updated value of `n`, "n is 1":

```
'n is ' n
```

Buzzword!

This is "cooperative multitasking"

Resetting `next`, the variable, jumps back into `next`, the block, and repeats printing incremented values of `n` until `6`, when `if n < until` evaluates to false. Instead of jumping, `next`, the block, finishes execution. The last line executed before jumping into `next` was `next = 1`, so the next line to execute prints "done counting":

```
'done counting'
```

Finally, the program returns to the line after the very first jump and prints "n is after 6", illustrating that `n` is defined only within the lexical scope of the `count` block, but `loops` is visible:

```
'n is ' n ' after ' loops
```

3.3 Reference

3.3.1 Syntax

Comefrom0x10 is **line-oriented**. Every line is a single statement and in fact, you can't escape the newline at the end of a line or do any line continuation at all because that usually makes code harder to read.

Blocks in cf0x10 are defined by **indentation** and create new lexical scopes. They look similar to functions in other languages, but they are not actually functions since they do not take arguments or return values:

```
block_name
   statements
```

> **Buzzword!**
>
> You can't "call" a block at all (don't call me, I'll call you). This is also known as "inversion of control."

Blank lines are significant, as yield points for jumps, except when they immediately follow blocks.

Comments are significant, not for content, but because they can change what counts as a blank line. Comments begin with a mesh character and continue until the end of the line:

```
# this is a comment
```

If there are no top-level statements in a program, execution begins at the first block, in source-code order. If there are top-level statements execution begins at the first top-level statement.

Source code is always read as UTF-8.

Identifiers

Only lowercase ASCII letters, digits and underscore are allowed in identifiers, because uppercase is reserved for future use. Identifiers cannot start with a digit.

Numbers

Number literals are always American-style decimal format, that is, with a full stop to separate the fractional part:

```
1.5
```

There is no syntax for octal, hex, or other such nonsense. This is a high-level language after all.

Strings

Both single and double quotes are allowed in string literals. There is no difference, but idiomatic Cf0x10 tends to use single quotes, as they are easier to type:

```
"A string"
'A string'
```

Backtick is the escape character:

```
'`n is a newline'
"`" is a literal double-quote"
```

Only `` `n `` is a valid escape sequence. No others, such as `` `t `` or Unicode escapes are implemented in the current parser, though Unicode literals are allowed. Just write UTF-8 source files.

Operators

In order of precedence[1]

Operator	Meaning
`(`	Grouping
`*`	Multiplication
`/`	Division
`+`	Addition
`-`	Subtraction
`<space>`	Concatenation
`<`	Less than
`>`	Greater than
`is`	Equals (comparison)
`=`	Assignment
`...`	Continues (no newline between output)

Conspicous by their absence are the boolean operators `and`, `or` and `not`. These can easily be simulated by other constructions within the language[2] and their inclusion would tempt program authors to write complicated, unreadable boolean expressions.

Keywords

Keyword	Meaning
`die`	Crash and burn. Intended for debugging. Cf0x10 is nothing if not pragmatic.
`comefrom`	The control flow statement to rule them all
`if`	Introduce a condition to limit, valid in `comefrom if` and `die if`.

3.3.2 Comefrom

> We don't know where to GOTO if we don't know where we've COME FROM. This linguistic innovation lives up to all expectations.
>
> —R. Lawrence Clark

Comefrom is a ludicriously powerful flow-control innovation that obviates the need for those sirens of structured programming, `if then` and `while`.

[1] Except where there are ties
[2] For weak values of "easy"

Bare `comefrom` statements are eligible for jumps from any blank line, provided they are in a scope visible from the blank line.

Comefrom also comes in two varieties, the *qualified* `comefrom [blockname]` and *conditional* `comefrom if [condition]`. A statement may be both qualified and conditional, that is, in the form `comefrom [blockname] if [condition]`.

Multiple comefrom dispatch

One of the first things that people think when they encounter `comefrom` is, "What if there is more than one eligible?"[1]

This is really quite silly, as the same problem happens with the known evil, `goto`:

```
10 PRINT "Joe is great"
10 PRINT "Joe is dumb"
20 GOTO 10
```

Different languages handle this in different ways, but most languages with first-class comefrom support call it an error.[2]

> A runtime error is cowardly evasion!

With a few seconds of thought, it becomes obvious that the modern fascination with "pub-sub" frameworks is merely an attempt to shoehorn `comefrom` into languages without it. Such attempts are clearly inferior to the first-class support in Cf0x10, but they do illustrate that people have no fundamental problem with multiple comefrom dispatch; it should, therefore, not be an error.

Buzzword!

When nerds say "undecidable," they mean, "bad news."

The problem such frameworks all encounter is that programs written with them turn out to be very hard to understand. They've figured out an ordering when multiple handlers react to the same event, but, because they are tacked on after a language is designed, they must determine the order dynamically. In other words, which part responds to what event becomes undecidable.

Cf0x10, however, embeds publish-subscribe right in the core of its design, enabling you to determine in lexical scope what parts of the program may respond to an event. It follows a few simple rules:

1. Blank lines yield to any `comefrom` statement in inverted lexical scope
2. When more than one `comefrom` is an eligible jump target in a single scope, the last wins, in source-code order
3. Execution jumps to the current scope after jumping to eligible `comefrom` statements in nested scopes
4. Nested-scope `comefrom` statements execute depth-first, as seen in source code indentation, where blocks that are first in the source are executed first

That's it! With four simple rules, you can see at a glance which code will run upon what condition.

See also

- Datamation article introducing comefrom
- Come from in Intercal

[1] According a scientific poll of one, yours truly
[2] What I think it would do in Intercal, if I were actually smart enough to write an Intercal program.

- Origin of comefrom, with some interesting info about similar instructions in real languages
- The original Intercal paper, which did not use comefrom
- Comefrom0x10 on Esolangs

3.3.3 Input

Comefrom0x10 programs can receive input from all the normal places, standard input, files and the command line.

Command-line arguments

The global `argv` stores command-line arguments to Cf0x10 programs. This example echoes its command line arguments:

```
# In the file echo.cf0x10
argv
```

Invoked thus:

```
cf0x10 echo.cf0x10 hello world
```

Prints `hello world`.

Standard input

This example prompts for input from the user and exits when the user enters a blank line:

```
prompt
    comefrom echo
    'Type something: '...
    stdin = ''

echo
    comefrom stdin if stdin
    'You typed "' stdin '"'

    # jumps back to prompt
```

Input is blocking and line-buffered. Changing the `stdin` global causes the program to wait for input, so it actually pauses at the line `stdin = ''`, until stdin is set.

Each line automatically gets its trailing newline dropped (automatically chomped, in Perl-speak).

End-of-file (EOF) sets stdin to `undefined`, so a Cf0x10 program can trivially detect the end of piped input.

Files

Setting the `read_path` global causes a read from the given path, whose contents will be stored in the `file` global. This cat program reads a single file and prints its contents to standard out:

```
read_path = argv
file
```

Setting the `write_path` global causes the contents of the `file` global to be written to `write_path`. This program writes "Hello, world" to a path given on the command line:

```
file = 'Hello, world'
write_path = argv
```

Because Cf0x10 has no binary data type, files are always encoded and decoded as UTF-8.

3.3.4 Types

Comefrom0x10 has two data types, Unicode strings and numbers.

Strings are truly Unicode, that is, you don't need to worry about legacy surrogate pair nonsense giving you a wrong string length. Which would be relevant if Cf0x10 had a way to retrieve string length.

Numbers are integers or floating point in the most sensible way possible, so division gives approximately exact answers:

```
1/3 # prints 0.333333
```

Naturally, in this spirit of mathematical correctness, Cf0x10 knows how to handle division by zero:

```
0/0 # undefined
```

Undefined

The great third datatype, or, perhaps, meta-datatype is `undefined`. Known in other languages as `None`, `Nothing`, `nil`, `void` or, if you're in JavaScript *both* `null` and `undefined`, Cf0x10 adopts the great success of SQL in how it handles the unknown:

```
undefined is undefined # falsy
```

The example above is misleading however, because `undefined` is not a keyword. The program above would be just as easily written:

```
frobnicate is foobar
```

Inspiration

JavaScript started this way, and, cowardly, backed down.

In Cf0x10, there is nothing special about the name `undefined`. It is a concept, the absence of value. In the first example, `undefined` is just the name of a variable that is… not defined. Thus, this is a perfectly valid program, that prints… nothing:

```
undefined = 1
'fear nothing'
comefrom if undefined is 1
```

Coercion

`4 2`	String: “42”
`4 + 2/0`	4
`4 + '2'`	Undefined
`'4' + '2'`	Undefined (adding strings is nonsense)
`'4' '2'`	String: “42”
`'4' 2/0`	String: “4”

3.3.5 Standard library

Comefrom0x10 comes with a suprisingly complete standard library: it includes everything you need to solve literally any computable problem.

car
: Grabs the first character of a string:

```
car = 'abc'
car # prints 'a'
```

cdr
: Grabs the rest of the string:

```
cdr = 'abc'
cdr # prints 'bc'
```

itoa
: Converts a number that represents a Unicode code point to a string:

```
itoa = 65
itoa # prints 'A'
```

atoi
: The opposite of `itoa`. Together, they can be used to represent lists of numbers as strings.

3.3.6 Python bindings

Cf0x10 programs may be easily invoked from Python:

```
>>> from cf0x10.interpreter import Program
>>>
>>> p = Program('"Hello, world"')
>>> p.execute(sys.stdin, sys.stdout, [])
Hello, world
```

In your Python program, just decorate functions with `@comefrom` to make them eligible jump targets from Cf0x10 programs:

```
>>> from cf0x10.bindings import comefrom
>>>
>>> @comefrom('if who')
... def cf_greeting(io, scope):
...     io.stdout.write("Hello, %s" % scope['who'])
>>>
```

```
>>> p = Program('who = "world"', additional_bindings=[cf_greeting])
>>> p.execute(sys.stdin, sys.stdout, [])
Hello, world
```

class cf0x10.bindings.**Binding**(*fn*)

The `Binding` class encapsulates a Comefrom0x10 block bound to a Python function. Client code normally will not need to instantiate bindings directly, as the decorators provide all required functionality.

cf0x10.bindings.**comefrom**(*comefrom_what*)

Indicates that the decorated function should be invoked when the interpreter encounters a condition that satisfies `comefrom_what`. This uses a syntax identical to the tail of a `comefrom` statement, in an ordinary Cf0x10 program, so:

```
@comefrom('foo if bar')
def example(io, scope):
    pass
```

Is essentially equivalent to defining this block in a cf0x10 program's global scope:

```
example
  comefrom foo if bar
```

cf0x10.bindings.**setvar**(*name*)

Declares that the decorated function expects to change the value of the indicated variable. This declares a global variable in the cf0x10 program.

The bound function can set variables by assigning the value in the scope object. Values must be of a type that can be converted to a Comefrom0x10 type, that is, numbers or strings:

```
@comefrom('')
@setvar('foo')
def example(io, scope):
    scope['foo'] = 'bar'
```

Assignment may cause jumps to comefrom statements, the same way as in cf0x10 code, except:

- Jumps from bound functions are queued and executed *after* the function returns
- No self-jumps

Bound functions can only trigger jumps by assignment, that is, they cannot trigger unconditional jumps.

3.4 Examples

Because the best way to learn a language is copy-paste.

3.4.1 Fibonacci

Prints Fibonacci numbers less than ten thousand:

```
fib
  b = 1
  comefrom fib
  a
  next_b = a + b
```

```
  a = b
  b = next_b

  comefrom fib if a > 10000
```

3.4.2 Factorial

The classic recursion example, that looks like recursion in Cf0x10, but really isn't:

```
n = 5 # calculates n!
acc = 1

factorial
  comefrom

  comefrom accumulate if n < 1

accumulate
  comefrom factorial
  acc = acc * n
  comefrom factorial if n is 0
  n = n - 1

acc # prints the result
```

3.4.3 Fizzbuzz

```
fizzbuzz
  mod_three = 3
  mod_five = 5
  comefrom fizzbuzz
  n
  comefrom fizzbuzz if n is mod_three
  comefrom fizzbuzz if n is mod_five
  n = n + 1

  fizz
    comefrom fizzbuzz if n is mod_three
    'Fizz'...
    mod_three = mod_three + 3
    linebreak
      # would like to write "unless mod_three is mod_five"
      comefrom fizz if mod_three - mod_five - 3
      ''

  buzz
    comefrom fizzbuzz if n is mod_five
    'Buzz'
    mod_five = mod_five + 5

  comefrom fizzbuzz if n is 100
```

3.4.4 99 bottles of beer

Using lyrics from http://99-bottles-of-beer.net/

```
bottles = ' bottles '
remaining = 99
one_less_bottle = remaining
depluralize
  comefrom drinking if one_less_bottle is 1
  bottles = ' bottle '

drinking
  comefrom if remaining is one_less_bottle
  remaining bottles 'of beer on the wall, ' remaining bottles 'of beer.'
  'Take one down and pass it around, '...
  one_less_bottle = remaining - 1
  one_less_bottle bottles 'of beer on the wall.`n'
  remaining = remaining - 1

  comefrom if one_less_bottle is 0
  'no more bottles of beer on the wall.'
  ''
  'No more bottles of beer on the wall, no more bottles of beer.'
  'Go to the store and buy some more, 99 bottles of beer on the wall.'
```

3.4.5 Brainfuck interpreter

Just in case anyone doubted that Cf0x10 is Turing complete

```
# A Brainfuck interpreter in Comefrom0x10
#
# Uses a Unicode string as storage, so behavior is undefined if cell value goes lower␣
↪than zero
# or higher than 0x10ffff.
#
# Storage is unbounded on both sides of the pointer.
#
# Invoke thus:
#
#   cf0x10 brainfuck.cf0x10 bfprogram.b
#
pointer_alpha = 1/0
pointer_numeric = 1/0
tape_behind = ''
tape_ahead = 1/0
tape_pos = 0 # only for debugging
array_behind = 1/0
array_ahead = ''
set_tape_ahead = array_ahead
array_ahead = 1/0
#
shift
  comefrom if array_ahead is array_ahead
  cdr = 1/0
  cdr = array_ahead
  shift_tail = cdr
  new_cell
```

```
    comefrom shift if shift_tail is ''
    itoa = 0
    shift_tail = itoa
  car = 1/0
  car = array_ahead
  array_behind = car array_behind
  done = shift_tail
  array_ahead = shift_tail
  comefrom shift if array_ahead is done

set_pointer_alpha = 1/0
set_pointer_alpha
  comefrom if set_pointer_alpha
  atoi = set_pointer_alpha
  cdr = tape_ahead
  set_tape_ahead = set_pointer_alpha cdr
  set_pointer_alpha = 1/0

set_tape_ahead = 1/0
set_pointer_vals
  comefrom if set_tape_ahead
  tape_ahead = set_tape_ahead
  car = tape_ahead
  pointer_alpha = car
  atoi = pointer_alpha
  pointer_numeric = atoi
  set_tape_ahead = 1/0

pointer_change = 1/0
change_pointer_val
  comefrom if pointer_change
  car = tape_ahead
  cdr = tape_ahead
  itoa = pointer_numeric + pointer_change
  set_tape_ahead = itoa cdr
  pointer_change = 1/0

file = 0 # initialize to something other than undefined so jump from file works when␣
↪read fails
read_path = argv
error_reading_program
  comefrom file if file + 0 is 0
  'Error: cannot read Brainfuck program at "' read_path '"'
  ''

program_loaded
  comefrom file if file is file
  program_behind = ''
  program_ahead = file

  run
    comefrom program_loaded
    opcode = 1/0
    opcode_numeric = 1/0
    in_buffer = '' # cf0x10 stdin is line-buffered
    jumping = 0
    moving = 1
    comefrom run
```

```
comefrom execute if opcode_numeric is 0
''
execute
  comefrom run if moving
  # can be useful for debugging:
  #program_ahead moving ':' jumping '@' tape_pos ':' pointer_numeric
  car = program_ahead
  atoi = car
  opcode_numeric = atoi
  opcode = car
  opcode = 1/0

  #

program_forward
  comefrom execute if moving > 0
  array_behind = program_behind
  array_ahead = 1/0
  array_ahead = program_ahead
  program_behind = array_behind
  program_ahead = array_ahead

  forward_jump
    comefrom execute if opcode is '['

    jump
      comefrom forward_jump if pointer_numeric is 0
      jumping = jumping + 1
      moving = 1
    match_brace
      comefrom forward_jump if jumping < 0
      jumping = jumping + 1
      stop_jump
        comefrom match_brace if jumping is 0
        moving = 1

program_backward
  comefrom execute if moving < 0
  array_behind = program_ahead
  array_ahead = 1/0
  array_ahead = program_behind
  program_behind = array_ahead
  program_ahead = array_behind

  backward_jump
    comefrom execute if opcode is ']'

    jump
      comefrom backward_jump if pointer_numeric > 0
      jumping = jumping - 1
      moving = -1
    match_brace
      comefrom backward_jump if jumping > 0
      jumping = jumping - 1
      stop_jump
        comefrom match_brace if jumping is 0
        moving = 1
```

```
op
  comefrom execute if opcode

  moving = 1
  do_op = opcode
  comefrom op if jumping
  #
  forward
    comefrom op if do_op is '>'
    tape_pos = tape_pos + 1
    array_ahead = 1/0
    array_behind = tape_behind
    array_ahead = tape_ahead
    tape_behind = array_behind
    set_tape_ahead = array_ahead
  backward
    comefrom op if do_op is '<'
    tape_pos = tape_pos - 1
    array_ahead = 1/0
    array_behind = tape_ahead
    array_ahead = tape_behind
    tape_behind = array_ahead
    set_tape_ahead = array_behind

  increment
    comefrom op if do_op is '+'
    pointer_change = 1
  decrement
    comefrom op if do_op is '-'
    pointer_change = -1

  print
    comefrom op if do_op is '.'
    pointer_alpha...
  read
    comefrom op if do_op is ','
    #
    cdr = 1/0
    cdr = in_buffer
    car = in_buffer
    set_pointer_alpha = car
    cdr = in_buffer
    in_buffer = cdr
    comefrom stdin if stdin + 0 is 0
    #
    block_for_input
      comefrom read if cdr is ''
      stdin = ''
      in_buffer = stdin
      cdr = in_buffer
      comefrom stdin if stdin + 0 is 0
```

3.5 Design patterns

- *All your assignment are belong to us*

3.5.1 All your assignment are belong to us

Using a variable of the same name as a block to simulate return values. Also known as AYARBTU.

3.6 Enterprise support

Enterprise support plans start at one million US dollars per year. While this may seem high, it's trivial compared to the cost of time uninitiated software engineers can spend on Cf0x10 software.[1]

To contact a Comefrom0x10 Solutioneer, send an email to cf0x10sales@mailinator.com. Due to the volume of requests, do not be discouraged should you not receive a prompt reply.

Finally, don't forget to mark your calendars for this year's **ComeFest**, happening all major cities around the world.

3.7 Contribute

Seriously, you don't have anything better to do?

Ok, you can report bugs on Bitbucket or find something to fix.

Discuss enhancements via the Comefrom Update Process, or CUP, for short. Proposals are called Comefrom Update Manuscripts, or CUMs, for short.

Note: The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in all CUMs are to be interpreted as normally understood in English.

Interested in writing a Cf0x10 implementation? Take a look at `output.yaml` in the reference implementation, which can be considered a defacto specification.

Cf0x10 programmers are in high demand. If you want to show off your Comefrom0x10 knowledge, contribute examples on Rosetta Code.

3.7.1 CUM 1: Schrödinger's conditionals

A form of boolean expression wherein a variable can be considered to simultaneously have multiple values, a "superposition".

Status of this document

Final draft, not yet accepted by CUP.

[1] Or the initiated, for that matter

Motivation

Though this may seem confusing, it really is quite natural. Consider this looping idiom:

```
comefrom if n is n + 1
# ... do things ...
n = n + 1
```

The condition would be considered true when incrementing variable n, because it can be satisfied by using the value of n before assignment on one side of the conditional and the value of n after assignment on the other. It is not necessary that the value before assignment be used, that is, this conditional is also true, provided n is defined:

```
comefrom if n is n
```

In this case, the conditional is satisfied by using the value of n after assignment on both sides.

Superpositions come into play only when the same variable appears on both sides of a boolean operator. If no boolean operator is involved, or if the variable only appears on one side of an operator, it is viewed as having collapsed to its value after assignment.

3.7.2 CUM 2: Lists

Some people claim that strings aren't really convenient enough for all your data structure needs, so Comefrom0x10 should implement lists.

Status of this document

Draft, not yet accepted by CUP.

Syntax

Use square brackets, as this matches the state of the art and is unlikely to confuse programmers:

```
[item1, item2, etc]
```

Dereference an item in a list:

```
['a', 'b', 'c']@0 # extract the 'a' value
```

Even though languages much more commonly use square brackets to extract list items, that syntax is annoying and sure to be obsolete in future languages. The `@` syntax only requires typing one character.

This requires some careful consideration of the precedence of `@`. It probably should be lower than all mathematical operators, for ease of using computed indices without requiring parens.

The `@` operator probably should be right-associative. Consider:

```
['a','b']@[1,1,0]@2
```

Which would be nonsensical with left-associativity.

Naturally, retrieving values at nonexistent indices should be undefined:

```
[]@2 # undefined
```

Negative indexing should, of course, be allowed.

Slicing

Slices should use Python-style half open ranges and the syntax can be quite similar:

```
[]@start:end:stride
```

Immutability

All data types in Cf0x10 are currently immutable, for robustness[1]. Lists should be no different.

Relationship to strings

Indexing and slicing should also be implemented for strings.

Prior art

I think I've seen a language with a single-character list operator, but I can't remember where.

3.8 About

Comefrom0x10 is a project by Josiah Ulfers. Source code and the content of this site is published under the WTFPL:

```
            DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
                    Version 2, December 2004

Copyright (C) 2004 Sam Hocevar <sam@hocevar.net>

Everyone is permitted to copy and distribute verbatim or modified
copies of this license document, and changing it is allowed as long
as the name is changed.

            DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
  TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

 0. You just DO WHAT THE FUCK YOU WANT TO.
```

3.8.1 Version compatibility

Comefrom0x10 was originally written in 2016. Like all great languages of the twentieth century, such as COBOL-60, it uses the year of its design as its major version. The minor version is separated by a spot, so the first release is version `0x10.1`.

Buzzword!

This is "semantic versioning"

[1] Not that immutability matters, since variables can, um, vary

Major versions are backward-compatible, so version `0x10.2` and so forth will run programs written for the `0x10.1` interpreter, and so forth. Backward compatibility is not guaranteed for major version changes.

Python Module Index

C

Index