CX Programming Language Specification

Alex Shinn, John Cowan, and Arthur A. Gleckler (*Editors*)

Steven Ganz
Aaron W. Hsu
Bradley Lucier
Emmanuel Medernach
Alexey Radul
Jeffrey T. Read
David Rush
Benjamin L. Russel
Olin Shivers
Alaric Snell-Pym
Gerald J. Sussman

Richard Kelsey, William Clinger, and Jonathan Rees
(Editors, Revised^5 Report on the Algorithmic Language Scheme)

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten
(Editors, Revised^6 Report on the Algorithmic Language Scheme)

August 26, 2017

SUMMARY

CONTENTS

INTRODUCTION

CX is both a compiled and interpreted, strongly typed programming language. The distinctive features of CX are affordances, which tell us what actions can be done on an element of a program; program serialization, which can be performed even to a running program; and program stepping, which allows the programmer to control a program's execution to go backward or forward a desired number of steps.

In order to provide the core features previously mentioned, a CX implementation is required to reach a very flexible program structure. CX programs can have access to their own internal structure during compilation and runtime; they can control their own execution, so, in theory, they can step back, modify their structure, and command themselves to continue. In a distributed system, a CX program could fully or partially serialize itself, send its serialized self to another server, and resume its execution there.

Affordances in CX represent a finite set of actions that can be performed over a CX program element, e.g., adding arguments to a function call or adding an expression to a function definition. There are primitive rules which determine what actions are allowed or restricted, such as type restrictions (an operator can only receive arguments of certain types) and what are the currently defined functions in the program. CX provides a more complex mechanism for determining an element's affordances, which is based on Prolog clauses and queries. Using facts and rules, a programmer can describe the environment in which a module is compiled or run, and the module will change its behaviour accordingly.

The CX specification describes a set of constructs (called "base language"), which can then be used by other programs (particularly, parsers) to generate CX programs. As a consequence, any programming language can be regarded as a CX dialect as long as they internally follow these base constructs. The present specification file also includes the description of an implementation of a CX base language and lexical and syntactic analyzers for a programming language that generates CX base code. This CX implementation is programmed in Go.

DESCRIPTION OF THE LANGUAGE

1. Overview of CX

1.1. Semantics

CX is a strongly typed language. There is no implicit casting in CX, which means that if the programmer wants to send a 64 bit integer to a function requiring a 32 bit integer, this argument needs to be explicitly casted to a 64 bit integer, even if this argument is a literal (e.g., 5 or 10). Because of this design decision, there are no overloaded functions in CX: if one wants to calculate the sum of two 32 bit integers, a native function called "addI32" needs to be used. If an argument of a different type than i32 is sent to "addI32", an error is raised.

CX is lexically scoped. Every call to a function is bound to an array of definitions, called a *state*. Each of these definitions is defined by a name, a type, and a value. The name is used to uniquely identify the definition among the other definitions in the state. The value represents an array of bytes that can be read in a way dictated by the definition's type.

The primitive types in CX are: boolean, string, byte, 32 bit integer, 64 bit integer, single and double precision floating-point real numbers, and arrays of these types. As mentioned before, arithmetic operations among the different primitive types are not permitted, and an explicit cast must be performed beforehand.

CX must provide the programmer a collection of basic functions (called "native functions") that enable the construction of more complex, user defined functions. These functions are present in a pre-loaded in a "core" module, which is pre-loaded in every CX program.

Any program in CX must have at least two modules: the previously mentioned core module, and a main module. The main module must have defined at least one function: the main function, which acts as the entry point for a CX program.

Every element in a CX program is defined by a group of variables. This group of variables can be seen as a *struct* in languages like C or Go. For example, a CX program can be defined as a collection of modules and a call stack. Next, a module can be defined by a name to uniquely identify itself among the other modules, a collection of imported modules, a collection of functions, etc. Each of these modules are linked to the main program struct, and each of the functions, structs, etc., defined in a module, is linked to their corresponding module. This design allows a representation of a CX program as an abstract syntax tree.

Arguments are always passed to functions by value in CX. For example, if a 32 bit integer array is sent to the native function "writeI32A" to change one of the values of the array, a new array is created with the altered value.

In CX, a variable or constant that is created outside of any function is called a definition. Definitions have global scope, which means that any function in the current module or in modules which import the current module have access to it. Definitions can be declared of any of the primitive types or user-defined types.

User-defined types are created using CX structs. A struct, as in other languages such as C or Go, are a group of variables that are collectively identified by a name.

1.2. CX REPL

CX is both a compiled and interpreted language. In its interpreted form, CX provides the programmer a REPL (read, eval, print loop), where programs can be constructed and executed interactively. The programmer can append new CX elements by entering functions, expressions, definitions, etc., as in writing a program in a source code file, and can also use meta-programming commands that can manipulate a program's structure and execution.

1.3. Identifiers

1.3.1. Base and

1.3.2. Error situations and unspecified behavior

1.3.3. Entry format

1.3.4. Evaluation examples

1.3.5. Naming conventions

2. Adders

In CX, adders are the basic building tools to construct programs. A CX program begins as an empty object, and new elements can be attached to it by the use of an adder. There exists one adder for each type of element that can be attached to a program. An adder will take the element to be attached, and will perform the binding to a parent element. In this binding process, the element can be pre-processed before

The adder can pre-process the element before this binding process, e.g., the expression adder can also attach the module to whom it will belong to (which should be its parent function's module). For this reason, it is convenient to have a series of helper functions solely for the task of creating an element, and restricting the adders to

only perform the binding process. In this specification file, *makers* (see Section 7) are suggested for element creation. However, it is important to note that adders could be in charge of both creating the element and performing the binding process.

An adder must be used whenever a CX program needs to change its element structure. For example, one could be tempted to directly append an element to another; considering a CX implementation which follows a Go-like syntax:

```
1   program.Modules = []Module{mod1, mod2}
```

The above code could raise multiple problems. One of these problems is that it could potentially lose all the previously appended modules. Another problem is that the modules being added don't have a reference to the main program structure.

The output value of an adder is not specified.

The following sections describe the adders for each of the elements that can be present in a CX program.

2.1. AddModule

AddModule takes a CX element with at least a name which uniquely identifies it among the other modules. This means that a bare representation of a module in CX is a name. This adder should attach to the module a reference to the program structure

```
1   mod.Program = &program
```

as in the example above. *AddModule* must also inform the program struct that the current module (see Section 11) will be the module just added. An example of this process is:

```
1   prgrm.CurrentModule = &mod
```

As mentioned above, a module must have a name that uniquely identifies it. This implies that there can not be two modules with the same name, and *AddModule* should be in charge of avoid duplicated modules. It is not specified if *AddModule* should or should not redefine a module if a duplicated module is sent as an argument.

2.2. AddDefinition

AddDefinition takes a CX element with at least a name, an array of bytes representing the definition's value, and a type. The name is used to uniquely identify the definition among other definitions. The type is used for instructing a compiler or an interpreter how should the array of bytes be read. This adder should attach to the definition a reference to the program structure, as well as a reference to the module which is going to contain it. An example of this process is:

```
1   def.Program = &prgrm
2   def.Module = &mod
```

A definition must have a name that uniquely identifies it, which implies that there can not be two definitions with the same name. *AddDefinition* should be in charge of avoiding duplicated definitions. If a definition with an already defined name is sent as an argument to *AddDefinition*, the default behaviour must be to redefine the definition. If this was not the case, every definition would be treated as constants in a CX program. It is a task for the parser and code generator to differentiate between a constant and a variable definition.

2.3. AddFunction

AddFunction takes a CX element with at least a name, which is used to uniquely identify the function among other functions defined in a module. This adder should attach to the function a reference to the program structure, as well as a reference to the module which is going to contain it

```
1   fn.Program = &prgrm
2   fn.Module = &mod
```

as in the example above. *AddFunction* should also set the module's current function (see Section 11) to the one which is being added. An example of this process is:

```
1   prgrm.CurrentFunction = &mod
```

A function must have a name that uniquely identifies it, which implies that there can not be two functions with the same name. *AddFunction* should be in charge of avoiding duplicated functions. It is not specified whether a function with a duplicated name should or should not redefine the old instance.

2.4. AddStruct

AddStruct takes a CX element with at least a name, which is used to uniquely identify the struct among other structs defined in a module. This adder should attach to the struct a reference to the program structure, as well as a reference to the module which is going to contain it

```
1   strct.Program = &prgrm
2   strct.Module = &mod
```

as in the example above. *AddStruct* should also set the module's current structure (see Section 11) to the one which is being added. An example of this process is:

```
1    strct.CurrentStrct = &strct
```

2.5. AddImport

An import element in CX is a synonym for a CX module element. As a consequence, *AddImport* must accept the same type and number of arguments as *AddModule*. The difference between these two adders is in the process: *AddModule* attaches a new module to a program, which can hold definitions, functions, structs, etc., while *AddImport* attaches a module reference to another module. Once a module is attached as an import to another module, elements in the module have access to the elements contained in the imported module (see Section 1).

Addimport should be in charge of avoiding duplicated imported modules. As redefining an import in a module should not add any behaviour to a CX program, a duplicated import being sent to *AddImport* should be discarded.

2.6. AddObject

Prolog objects (see Section 1 and Section 11) are contained in CX module elements. As objects are stored as just a collection of names, the only task of *AddObject* should be to append a new name to this collection. Despite the process being so simple, it should be encapsulated for maintainability purposes.

2.7. AddClauses

Prolog clauses (see Section 1 and Section 11) are contained in CX module elements. As clauses are stored as just a string of characters, the only task of *AddClauses* should be to redefine this property of a CX module element. Despite the process being so simple, it should be encapsulated for maintainability purposes.

2.8. AddQuery

Prolog queries (see Section 1 and Section 11) are contained in CX module elements. As a query is stored as just a string of characters, the only task of *AddClauses* should be to redefine this property of a CX module element. Despite the process being so simple, it should be encapsulated for maintainability purposes.

2.9. AddField

As is explained in Section 11, a CX field element consists of only a name and a type, the process of adding a field to a CX struct element should be simple. *AddField* should be in charge of avoiding duplicate fields in a CX struct element and perform the actual appending of the field to the struct. If the field name is not present in the fields collection, it should be appended.

It is not specified whether a field should be redefined or not if the field name to be appended is already present in the fields collection.

2.10. AddExpression

AddExpression takes a CX element with at least a CX function element. This adder should attach to the expression a reference to the program structure, a reference to the module which is going to contain it, as well as a reference to the function which is going to call it during program execution. An example of attachments to these references is shown below

```
1    expr.Program = &prgrm
2    expr.Module = &mod
3    expr.Function = &fn
```

AddExpression should also set the function's current expression (see Section 11) to the one which is being added. An example of this process is:

```
1    fn.CurrentExpression = &expr
```

Lastly, *AddExpression* should also be in charge of assigning a line number (see Section 11) to the expression being added. This line number should be equal to number of expressions contained by the function before the current expression is added.

2.11. AddInput

As is explained in Section 11, a function's input is defined by a CX parameter element, and this element consists of only a name and a type. As a consequence, the process of adding an input to a CX function element should be simple: append the new parameter to the list of inputs attached to the function, and avoid duplicated input parameter elements by name.

If a parameter to be appended is already present in the inputs collection, this parameter must be discarded.

2.12. AddOutput

As is explained in Section 11, a function's output is defined by a CX parameter element, and this element consists of only a name and a type. As a consequence, the process of adding an output to a CX function element should be simple: append the new parameter to the list of outputs attached to the function, and avoid duplicated output parameter elements by name.

If a parameter to be appended is already present in the outputs collection, this parameter must be discarded.

As explained in Section 1, a feature of CX is its capability of defining functions with multiple outputs. This behaviour implies that one can call *AddOutput* multiple times to append new outputs, and the output would not be redefined.

2.13. AddArgument

AddArgument takes a CX element with at least a byte array which defines its value, and a type which instructs the compiler or interpreter how should the array of bytes be read. The CX argument element is appended to an ordered collection of arguments attached to an expression.

Any argument can be attached to any expression, and no constraints or limitations exist, as it's the job of the compiler or interpreter to raise an error in case there exists an input number or type mismatch.

2.14. AddOutputName

AddOutputName takes a character string which is used to construct a CX definition. The constructed definition is appended to a collection of output names attached to a CX expression element. *AddOutputName* must take information from the expression's operator in order to determine the definition's type. The definition's value must be initialized to a zero-equivalent value (e.g., 0 for an i32 type, an empty string for a str type).

3. Affordances

Affordances are part of a CX mechanism that informs the programmer or the CX program itself about the possible actions that can be performed over a CX element. For example, if a function call (a CX expression) requires two arguments as input parameters, and only one has been provided, it is said that the expression can afford another argument to be added to its collection of arguments. In this case, a programmer can make a call to a GetAffordances function to obtain a list of possible CX elements that can act as an argument to the given expression.

However, the GetAffordances function, in this case, should restrict the list of affordances to the operator's required type for that particular input. This means that if the operator's second input type is i32, only arguments, identifiers, globals or function calls which return i32 outputs (note that this is not an exhaustive list of the possibilities) could be indicated as candidates to be appended to the expression's collection of arguments.

At least, the following CX elements can be asked for affordances: the CX program itself, modules, structs, functions, and expressions. In its most basic functionality:

- **Program affordances** should enlist the action for appending a new module.
- **Module affordances** should enlist the actions for adding a new definition, for each of the available types to the module (primitive types, and user defined types via structs); adding a new import, excluding those that have already been imported; adding a new function; and adding a new struct.
- **Struct affordances** should enlist the action for appending a new field, for each of the available types to the module (primitive types, and user defined types via structs).
- **Function affordances** should enlist the actions for appending new input or output parameters, for each of the available types to the module (primitive types, and user defined types via structs); and adding a new expression, where its operator can be any of the functions defined in its parent module or in any of the imported modules by its parent module.
- **Expression affordances**, perhaps the most complex type of affordances, should enlist actions for adding new arguments and new output names. Compared to the other type of affordances, expression affordances are complex to determine because there are many types of CX elements that can act as an argument to an expression. Likewise, output names could be virtually any identifier in a CX program which matches the required type by an input parameter in a given expression. The elements that could act as arguments include, and are not limited to, input parameter names, output parameter names, global and local definitions, literals, and expressions. The elements that could act as output names include, and are not limited to, input parameter names, output parameter names, and global and local definitions.

How the affordance mechanism chooses a name or a value for those elements which require either or both of those is not specified. A CX program could ask the programmer for a desired name or value, or the program could choose

a randomly generated name. In any case, the mechanism must ensure that names be unique for new elements, and that values comply with the element's required type.

More detailed descriptions of each of the types of affordances can be found in the next Subsections.

3.1. Program Affordances

The most basic action a program can afford is the addition of new modules. Nevertheless, program affordances have the potential to enlist any action that could be performed to any element contained in a CX program.

As a CX program element also contains a reference to the call stack, affordances could enlist possible actions related to them. For example, a program can afford to go back N steps in the call stack, or be executed N steps. Other more obscure actions could be afforded by the call stack element, such as changing local definitions in previous, open calls, change the return address of a call, etc.

Another kind of program affordances would be to change the program's output. Instead of sending the output argument to the operating system, it could be sent to another device.

3.2. Module Affordances

A CX module element

3.3. Struct Affordances

3.4. Function Affordances

3.5. Expression Affordances

3.6. Filtering Affordances

3.7. Applying Affordances

4. Execution

The execution of a CX program must start with the creation of a CX call element. This call must be associated with the main function, from the main module (as explained in Section 1, every CX program must contain a main function and main module, which act as an entry point). After creating this call, it must be appended to the ordered collection of calls, and start the execution of the program with this initial call.

Execution can be stepped in CX. This implies that the process must be aware of the amount of calls that have been added to the call stack since the last time the program was paused. The execution process must also have access to the call stack, and have permission to manipulate the call stack: if a program is required to go back N steps, N calls must be popped from the call stack. If a program goes back N steps, the information from these N calls must be discarded, i.e., any change performed by the N calls must be rolled back, and any local definitions contained in the state of the calls must be discarded.

Every time a CX program is instructed to evaluate a function, a call is created and pushed onto the call stack. A call must have a reference to the function which is going to be evaluated, the line number at which the program execution is at (i.e., the index of the expression in a CX function definition). A reference to the return address (the call which created the current call) must be accessible; once a call finishes its execution, it can return an output value by appending a CX definition to the state of the caller by using the this reference, and it must be popped out from the stack. A call finishes its execution once all the expressions contained in the function being called have been evaluated. Whenever a call is created, an internal counter is incremented so program stepping can be performed.

If a function did not explicitly assign values to its output parameters, a call should return the output values returned by its last expression.

The execution process is also in charge of ensuring that all the arguments sent to every function call are of the correct type, and that the correct number of arguments have been sent. If there is a mismatch either in the arguments types or in the arguments number, the execution process should halt the program without killing the program's process (see Section 4.1 below).

Every expression contained in the function definition of the callee must be evaluated by the execution process. Before performing the evaluation of an expression, its arguments also need to be evaluated. If an argument is another function evaluation, another call will be created. If the argument is an identifier which is referencing a value determined in a previous expression, a global definition, or a function input or output parameter, the identifier needs to be resolved in order to find the actual value and type that the identifier is referring to. If the argument is a native function (see Section 4.2) (determined by checking against a list of all the native functions defined in the core module), the execution process sends the CX argument elements to the required function. Native functions are defined in the host language, and they provide basic functionality, like arithmetic operations. The arguments sent to a native function need to be converted to be readable by the host language. This conversion process usually

involves using the argument's type to know how to interpret the bytes stored in the argument's value. Once the native function has finished processing its arguments, the output needs to be converted back to a CX argument element. Lastly, if an argument to a function call is a literal (e.g., 10, "a string", 3.14), these are directly sent to the callee.

If the program is seen as an abstract syntax tree, every leaf in a CX AST will always be a native function, i.e., every CX program could be decomposed into a structured series of native calls.

4.1. Call Stack Debugging

A CX program should be resilient to crashing. If possible, a CX program should always halt its execution when an error is encountered, inform the programmer about the error, and give the programmer the opportunity to change the program's state and structure.

Whenever an error is encountered, CX should enter REPL mode (see Section 1.2), if not already in it, so the programmer has access to the meta-programming commands (see Section 12) to modify the program's structure and execution. The last call in the call stack will be discarded or popped out (which is equivalent to stepping back one step), and the programmer can start altering the program's structure. When an error is encountered, a CX implementation should give the program as much relevant information as possible. At a minimum, when an error is encountered, the function which caused the error, the arguments that were sent to the function, the line number of such expression, and a call stack trace should be printed for the programmer. The amount of calls in the call stack to be printed is not specified.

4.2. Native Function Calling

When an expression is called, the execution process must look for the name of the callee's operator in the collection of functions attached to the current module, or the collection of functions of imported modules if the identifier name starts with the name of a module followed by a period (see Section 1.3 for more information about identifiers). If the name is found in one of these modules, the actual CX function element is retrieved and executed with the arguments provided in the caller's function definition. If the name is not found in the current module or any of the imported modules, the name is looked for in the collection of native functions in the core module. If the name is not found in the collection of native functions, an error should be raised, and if it is found, the function is called with the provided arguments.

Native functions are implemented in the programming language in which the CX implementation is being programmed in. A native function should not call any CX functions, and this implies that a native function can be considered as a terminal node in the CX program structure. All native functions must be able to receive CX argument elements, and must be able to convert them to native arguments to the host programming language. For example, the native function "addI32" should receive two CX argument elements which hold 32 bit integers as values, and are of type "i32." The native function can have the responsibility of checking if the arguments are of the expected types, and then should proceed with the conversion from CX 32 bit integers, to the host programming language's equivalents. The integers should be added together and the resulting value must be converted to a CX 32 bit integer, which must be attached to a CX argument element that will serve as the native function's output parameter's value.

4.3. Compilation

A compilation process can be called at any time (see Section 4.3) by a CX program. Compilation is in charge of optimizing the program structure to require fewer system resources, such as processing power and memory. A CX program element must have access to a byte array which represents a memory heap. The compilation process can place certain values from different elements in the heap to access them in a constant time.

10. Serialization

10.1. Derived expression types

11. Structs

Every struct name should start with the letters "CX", to avoid conflicts with existant definitions in the host programming language, and for clarification purposes.

11.1. CXProgram

11.2. CXCallStack

11.3. CXCall

11.4. CXModule

11.5. CXDefinition

11.6. CXStruct

11.7. CXField

11.8. CXType

11.9. CXFunction

11.10. CXParameter

11.11. CXExpression

11.12. CXArgument

11.13. CXAffordance

12. Meta-Programming

AN IMPLEMENTATION OF THE LANGUAGE

13. Implementation of a CX Language

This Section describes the most important aspects of the CX implementation that can be found in `http://github.com/skycoin/cx`, in order to clarify the concepts covered in the Sections regarding the CX specification. The code presented in the code listings in this Section is not guaranteed to be up-to-date as the code present in the official repository.

13.1. Adders

13.2. Affordances

13.3. Execution

13.4. Native Functions

13.5. Getters

13.6. Makers

13.7. Removers

13.8. Selectors

13.9. Serialization

13.10. Structs

14. Lexical Analyzer

14.1. Derived expression types

15. Syntactic Analyzer

15.1. Derived expression types

15.2. Derived expression types

This section gives syntax definitions for the derived expression types in terms of the primitive expression types (literal, variable, call, `lambda`, `if`, and `set!`), except for `quasiquote`.

Appendix A. Standard Libraries

This section lists the exports provided by the standard libraries. The libraries are factored so as to separate features which might not be supported by all implementations, or which might be expensive to load.

A.0.1. addI32, addI64, addF32, addF64

Syntax:

addI32 i32 i32 => i32

addI64 i64 i64 => i64

addF32 f32 f32 => f32

addF64 f64 f64 => f64

Description:

Returns the sum of the two numbers provided. If the two numbers are of different type, an error must be signaled, as type conversions are explicit in CX.

Examples:

```
1 sum := addI32(3, -5)
2 addI64(i32ToI64(3), i32ToI64(5))
3 printF32(addF32(3.5, 3.7))
```

A.0.2. subI32, subI64, subF32, subF64

Syntax:

subI32 i32 i32 => i32

subI64 i64 i64 => i64

subF32 f32 f32 => f32

subF64 f64 f64 => f64

Description:

Returns the difference of the two numbers provided, and the order of the arguments is equivalent to the order in a traditional infix substraction. If the two numbers are of different type, an error must be signaled, as type conversions are explicit in CX.

Examples:

```
1 diff := subI32(10, 5)
2 subI64(i32ToI64(3), i32ToI64(5))
3 printF32(subF32(10.0, 5.0))
```

A.0.3. mulI32, mulI64, mulF32, mulF64

Syntax:

mulI32 i32 i32 => i32

mulI64 i64 i64 => i64

mulF32 f32 f32 => f32

mulF64 f64 f64 => f64

Description:

Returns the product of the two numbers provided. If the two numbers are of different type, an error must be signaled, as type conversions are explicit in CX.

Examples:

Listing 13.1: Example of an AddModule adder

```
func (cxt *CXProgram) AddModule (mod *CXModule) *CXProgram {
        mod.Context = cxt
        cxt.CurrentModule = mod
        found := false
        for i, md := range cxt.Modules {
                if md.Name == mod.Name {
                        cxt.Modules[i] = mod
                        found = true
                        break
                }
        }
        if !found {
                cxt.Modules = append(cxt.Modules, mod)
        }
        return cxt
}
```

Listing 13.2: Example of an AddObject adder

```
func (mod *CXModule) AddObject (obj string) *CXModule {
        mod.Objects = append(mod.Objects, obj)

        return mod
}
```

Listing 13.3: Example of an AddClauses adder

```
func (mod *CXModule) AddClauses (clauses string) *CXModule {
        mod.Clauses = clauses
        return mod
}
```

```
func (strct *CXStruct) GetAffordances() []*CXAffordance {
        affs := make([]*CXAffordance, 0)
        mod := strct.Module
        types := make([]string, len(BASIC_TYPES))
        copy(types, BASIC_TYPES)
        for _, s := range mod.Structs {
                types = append(types, s.Name)
        }
        for _, imp := range mod.Imports {
                for _, strct := range imp.Structs {
                        types = append(types, concat(imp.Name, ".", strct.Name))
                }
        }
        for _, typ := range types {
                fldGensym := MakeGenSym("fld")
                fldType := MakeType(typ)
                affs = append(affs, &CXAffordance{
                        Description: concat("AddField ", fldGensym, " ", typ),
                        Action: func() {
                                strct.AddField(MakeField(fldGensym, fldType))
                        }})
        }
        return affs
}
```

```
func (cxt *CXProgram) Run (withDebug bool, nCalls int) {
        var callCounter int = 0
        if cxt.CallStack != nil && len(cxt.CallStack.Calls) > 0 {
                lastCall := cxt.CallStack.Calls[len(cxt.CallStack.Calls) - 1]
                lastCall.call(withDebug, nCalls, callCounter)
        } else {
                if mod, err := cxt.SelectModule("main"); err == nil {
                        if fn, err := mod.SelectFunction("main"); err == nil {
                                state := make([]*CXDefinition, 0)
                                mainCall := MakeCall(fn, state, nil, mod, mod.Context)
                                cxt.CallStack.Calls = append(cxt.CallStack.Calls, mainCall)
                                mainCall.call(withDebug, nCalls, callCounter)
                        }
                } else {
                        fmt.Println(err)
                }
        }
}
```

```
if nCalls > 0 && callCounter >= nCalls {
                return
}
callCounter++
```

```
if call.Line >= len(call.Operator.Expressions)
```

Listing 13.4: Popping out an Element from the Stack

```
call.Context.CallStack.Calls =
call.Context.CallStack.Calls[:len(call.Context.CallStack.Calls) - 1]
```

```
panic(fmt.Sprintf("output '\%s' is of type '\%s'; '\%s' requires output type '\%s'",
out.Name, out.Typ.Name, call.Operator.Name, call.Operator.Outputs[i].Typ.Name))
```

Listing 13.5: Example of a Native Function

```
func divI32 (arg1 *CXArgument, arg2 *CXArgument) (*CXArgument, error) {
        if arg1.Typ.Name != "i32" || arg2.Typ.Name != "i32" {
                return nil, errors.New("divI32: wrong argument type")
        }

        var num1 int32
        var num2 int32
        encoder.DeserializeAtomic(*arg1.Value, &num1)
        encoder.DeserializeAtomic(*arg2.Value, &num2)

        if num2 == int32(0) {
                return nil, errors.New("divI32: Division by 0")
        }

        output := encoder.SerializeAtomic(num1 / num2)

        return &CXArgument{Value: &output, Typ: MakeType("i32")}, nil
}
```

Listing 13.6: Example of a Getter

```
func (cxt *CXProgram) GetDefinition (name string) (*CXDefinition, error) {
        if mod, err := cxt.GetCurrentModule(); err == nil {
                var found *CXDefinition
                for _, def := range mod.Definitions {
                        if def.Name == name {
                                found = def
                                break
                        }
                }

                if found == nil {
                        return nil, errors.New(fmt.Sprintf("Definition '\%s' not found", name))
                } else {
                        return found, nil
                }
        } else {
                return nil, err
        }
}
```

Listing 13.7: Example of a Maker

```
func MakeModule (name string) *CXModule {
        return &CXModule{
                Name: name,
                Definitions: make([]*CXDefinition, 0),
                Imports: make([]*CXModule, 0),
                Functions: make([]*CXFunction, 0),
                Structs: make([]*CXStruct, 0),
        }
}
```

Listing 13.8: Example of a Remover

```
func (mod *CXModule) RemoveFunction (fnName string) {
        lenFns := len(mod.Functions)
        for i, fn := range mod.Functions {
                if fn.Name == fnName {
                        if i == lenFns - 1 {
                                mod.Functions = mod.Functions[:len(mod.Functions) - 1]
                        } else {
                                mod.Functions = append(mod.Functions[:i], mod.Functions[i+1:]...)
                        }
                        break
                }
        }
}
```

Listing 13.9: Example of a Remover

```
func (expr *CXExpression) RemoveArgument () {
        if len(expr.Arguments) > 0 {
                expr.Arguments = expr.Arguments[:len(expr.Arguments) - 1]
        }
}
```

Listing 13.10: Example of a Getter

```
func (mod *CXModule) SelectFunction (name string) (*CXFunction, error) {
        var found *CXFunction
        for _, fn := range mod.Functions {
                if fn.Name == name {
                        mod.CurrentFunction = fn
                        found = fn
                }
        }

        if found == nil {
                return nil, errors.New("Desired function does not exist")
        }

        return found, nil
}
```

Listing 13.11: Example of a Struct

```
type CXProgram struct {
        Modules []*CXModule
        CurrentModule *CXModule
        CallStack *CXCallStack
        Outputs []*CXDefinition
        Steps []*CXCallStack
        ProgramSteps []*CXProgramStep
        Heap *[]byte
}
```

```
product := mulI32(-1, -20)
mulI64(i32ToI64(0), i32ToI64(3))
printF32(mulF32(1.1, 1.1))
```

A.0.4. divI32, divI64, divF32, divF64

Syntax:

divI32 i32 i32 => i32

divI64 i64 i64 => i64

divF32 f32 f32 => f32

divF64 f64 f64 => f64

Description:

Returns the division of the two numbers provided. If the two numbers are of different type, an error must be signaled, as type conversions are explicit in CX. If the denominator (second argument) is 0, a *division by 0* error must be raised, and the program's execution must be halted.

Examples:

```
div := divI32(10, 2)
divI64(i32ToI64(30), i32ToI64(0))
printF32(divF32(1.1, 1.1))
```

A.0.5. modI32, modI64

Syntax:

modI32 i32 i32 => i32

modI64 i64 i64 => i64

Description:

Returns the modulus of the two numbers provided. If the two numbers are of different type, an error must be signaled, as type conversions are explicit in CX. If the denominator (second argument) is 0, a *division by 0* error must be raised, and the program's execution must be halted.

Examples:

```
rem := modI32(10, 2)
modI64(i32ToI64(30), i32ToI64(0))
```

A.0.6. printStr, printBool, printByte, printI32, printI64, printF32, printF64, printBoolA, printByteA, printI32A, printI64A, printF32A, printF64A

Syntax:

printStr str => str

printBool bool => bool

printByte byte => byte

printI32 i32 => i32

printI64 i64 => i64

printF32 f32 => f32

printF64 f64 => f64

printBoolA []bool => []bool

printByteA []byte => []byte

printI32A []i32 => []i32

printI64A []i64 => []i64

printF32A []f32 => []f32

printF64A []f64 => []f64

Description:

A print function exists for each of the primitive types. These functions display the argument's value to the terminal. In the case of arrays, the values are enclosed in brackets and separated by spaces. Each of the print functions return their argument.

Examples:

```
printStr(``Hello world!'')
addI32(printI32(30), printI32(50))
```

A.0.7. idStr, idBool, idByte, idI32, idI64, idF32, idF64, idBoolA, idByteA, idI32A, idI64A, idF32A, idF64A

Syntax:

idStr str => str

idBool bool => bool

idByte byte => byte

idI32 i32 => i32

idI64 i64 => i64

idF32 f32 => f32

idF64 f64 => f64

idBoolA []bool => []bool

idByteA []byte => []byte

idI32A []i32 => []i32

idI64A []i64 => []i64

idF32A []f32 => []f32

idF64A []f64 => []f64

Description:

An identity function exists for each of the primitive types. These functions return thir argument as output, and correspond to the mathematical function $f(x) = x$. These functions are mostly used in functional programming constructs, and are used by CX to parse certain statements like if/else and loops.

Examples:

```
1 idI32(5)
```

A.0.8. readBoolA, readByteA, readI32A, readI64A, readF32A, readF64A

Syntax:

readBoolA []bool i32 => bool

readByteA []byte i32 => byte

readI32A []i32 i32 => i32

readI64A []i64 i32 => i64

readF32A []f32 i32 => f32

readF64A []f64 i32 => f64

Description:

This set of functions are used to retrieve values from arrays of their corresponding type. The first argument is an array of the required type, and the second argument is the index of the value to be retrieved. The output is the value in the array at the indicated index.

If either a negative index or an index which exceeds the length of the array minus one (CX arrays are zero-indexed) is provided, an out of bounds error must be raised.

Examples:

```
1 readBoolA([]bool{true, false, true}, 0)
2 addI32(readI32([]i32{0, 1}, 1), 5)
```

```
addI32                  subI32
mulI32                  divI32
addI64                  subI64
mulI64                  divI64
addF32                  subF32
mulF32                  divF32
addF64                  subF64
mulF64                 divF64

printStr                 printByte
printI32                printI64
printF32               printF64
printByteA             printI32A
printI64A              printF32A
printF64A              printBool

idStr                    idByte
idI32                   idI64
idF32                  idF64
idByteA                idI32A
idI64A                  idF32A
idF64A

readByteA              writeByteA
readI32A              writeI32A
readI64A              writeI64A
readF32A              writeF32A
readF64A              writeF64A

i64ToI32              f32ToI32
f64ToI32              i32ToI64
f32ToI64              f64ToI64

i32ToF32              i64ToF32
f64ToF32              i32ToF64
i64ToF64              f32ToF64
byteAToStr

ltI32                   gtI32
eqI32                  ltI64
gtI64                  eqI64

sleep
setClauses            addObject
setQuery               remObject
remExpr               addExpr
exprAff                initDef
evolve                  goTo
```

Appendix B. Standard Feature Identifiers

An implementation may provide any or all of the feature identifiers listed below for use by `cond-expand` and `features`, but must not provide a feature identifier if it does not provide the corresponding feature.

`r7rs`

All R^{7}RS Scheme implementations have this feature.

`exact-closed`

All algebraic operations except `/` produce exact values given exact inputs.

`exact-complex`

Exact complex numbers are provided.

LANGUAGE CHANGES

Incompatibilities with R^{5}RS

This section enumerates the incompatibilities between this report and the "Revised^5 report" [**?**].

This list is not authoritative, but is believed to be correct and complete.

ADDITIONAL MATERIAL

The Scheme community website at `http://schemers.org` contains additional resources for learning and programming, job and event postings, and Scheme user group information.

EXAMPLE

The procedure `integrate-system` integrates the system

$$y_k' = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a promise to deliver the rest of the stream.

```
(define head car)
(define (tail stream)
  (force (cdr stream)))
```

The following illustrates the use of `integrate-system` in integrating the system

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

```
(define (damped-oscillator R L C)
  (lambda (state)
    (let ((Vc (vector-ref state 0))
          (Il (vector-ref state 1)))
      (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
              (/ Vc L)))))

(define the-states
  (integrate-system
    (damped-oscillator 10000 1000 .001)
    '#(1 0)
    .01))
```

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[3] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. `http://www.ietf.org/rfc/rfc2119.txt`, 1997.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

The principal entry for each term, procedure, or keyword is listed first, separated from the other entries by a semicolon.