EIS-2 - A general purpose, high performance input deck and maths parser

Chris Brady, University of Warwick

Warwick RSE

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk)

EIS-2

- Library to control a code from a structured text file using rich mathematical notation
- Intended to replace the existing system in the EPOCH plasma physics code
  - EPOCH Input System - Version 2
- Funded under eCSE 13-19 as part of a package of improvements for EPOCH
- EIS-2 has been written as a general purpose library that can be used in a wide variety of codes

EPOCH

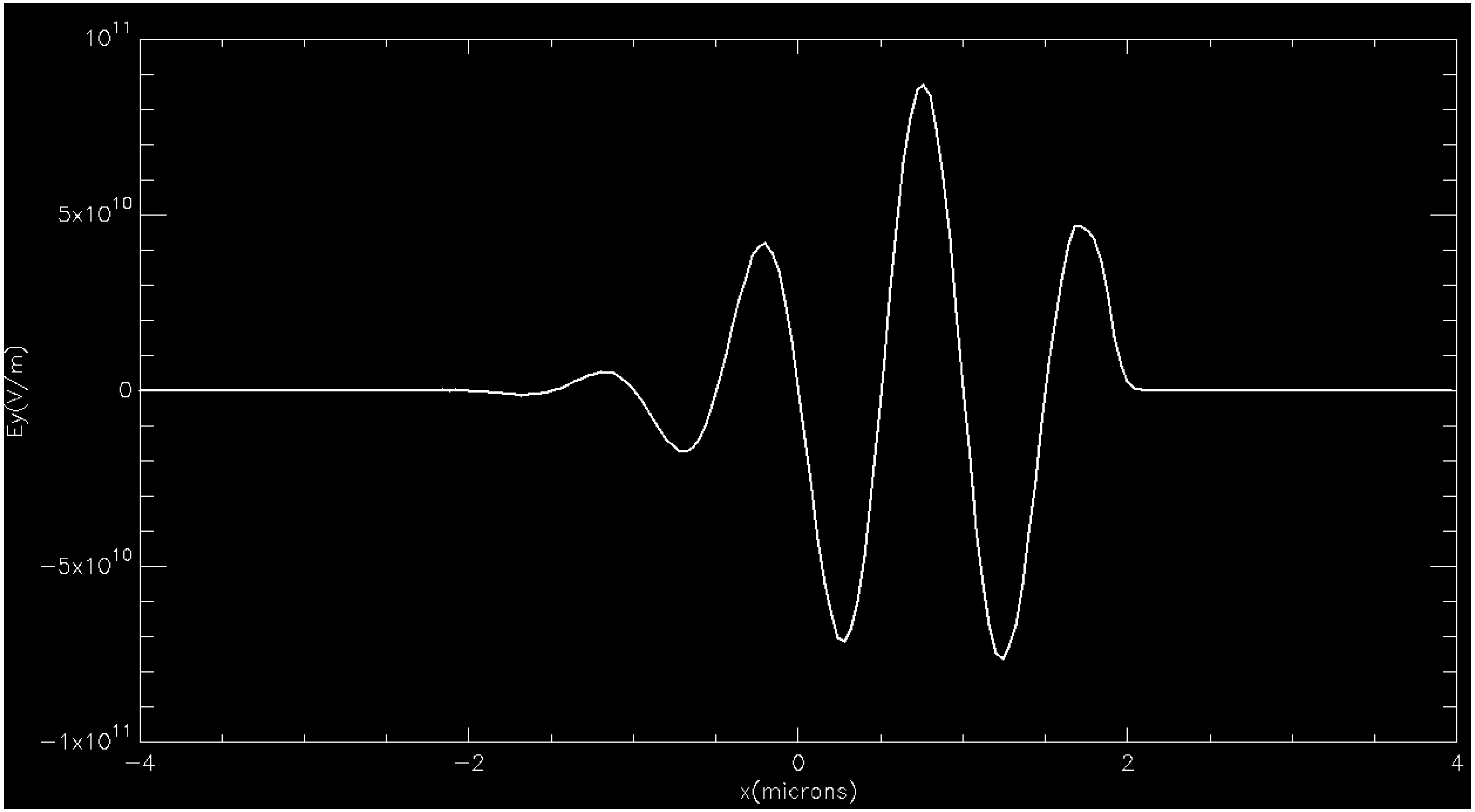
- Electromagnetic Particle in Cell code (PIC code) for plasma physics simulations, written in 2007 using Fortran 95 and very widely used in the community
  - Now moved to newer Fortran 2003 standard opening new options
- Has a computational mesh you solve Maxwell's equations on
- Has a set of particles representing ions and electrons that freely move over the grid
- MPI parallel typically run on 1000s to 10000s cores - median job size last month on ARCHER 1200 cores

- Input system has to provide a **lot** of different things
    - Properties for fields on the grid
    - Properties for boundaries
    - Properties for particles
    - Properties for mechanical parts of the code

- Any replacement needs to have
  - Good performance in parallel
    - Limited comms (EIS-2 does not itself use MPI)
    - No dynamic libraries (EIS-2 is a static library)
  - Identical behaviour to existing deck parser
    - Too many users to change syntax without there being a very substantial benefit to them

- Aims for improvement
  - Faster
  - More maintainable
  - More extensible
  - Can be moved to languages other than Fortran

```
begin:laser
  boundary = x_min
  intensity_w_cm2 = 1.0e15
  lambda = 1 * micron
  t_profile = gauss(time,4*femto,4*femto)
  t_end = 14 * femto
end:laser
```


10^{11}
5×10^{10}
0
-5×10^{10}
-1×10^{11}
Ey(V/m)
−4
−2
0
2
4
x(microns)

Two parts to EIS-2

- Deck parser to deal with blocks and key/value pairs
- Maths parser to deal with evaluating the mathematical expression in the values
    - Has to only trigger when appropriate since some keys in EPOCH's deck really are strings

Maths parser

EIS-2 Maths Parser

- Idea of a maths parser is to convert a mathematical expression into a data structure that the computer can use to evaluate the expression

- Technically made up of three parts

  - Tokenizer/Lexer

  - Parser

  - Evaluator

EIS-2 Maths Tokenizer

`sqrt(epsilon0 * kb * background_temp / background_density / qe^2)`

- Termed “tokenization” or “lexing”
- Converting the input string into tokens that describe the individual parts
- The next step is “parsing” and that converts the tokens into a form that can be executed
  - Dijkstra’s Shunting Yard Algorithm

EIS-2 Maths Parser

`sin(2 * pi)`

Output

Intermediate

EIS-2 Maths Parser

`sin(2 * pi)`

^

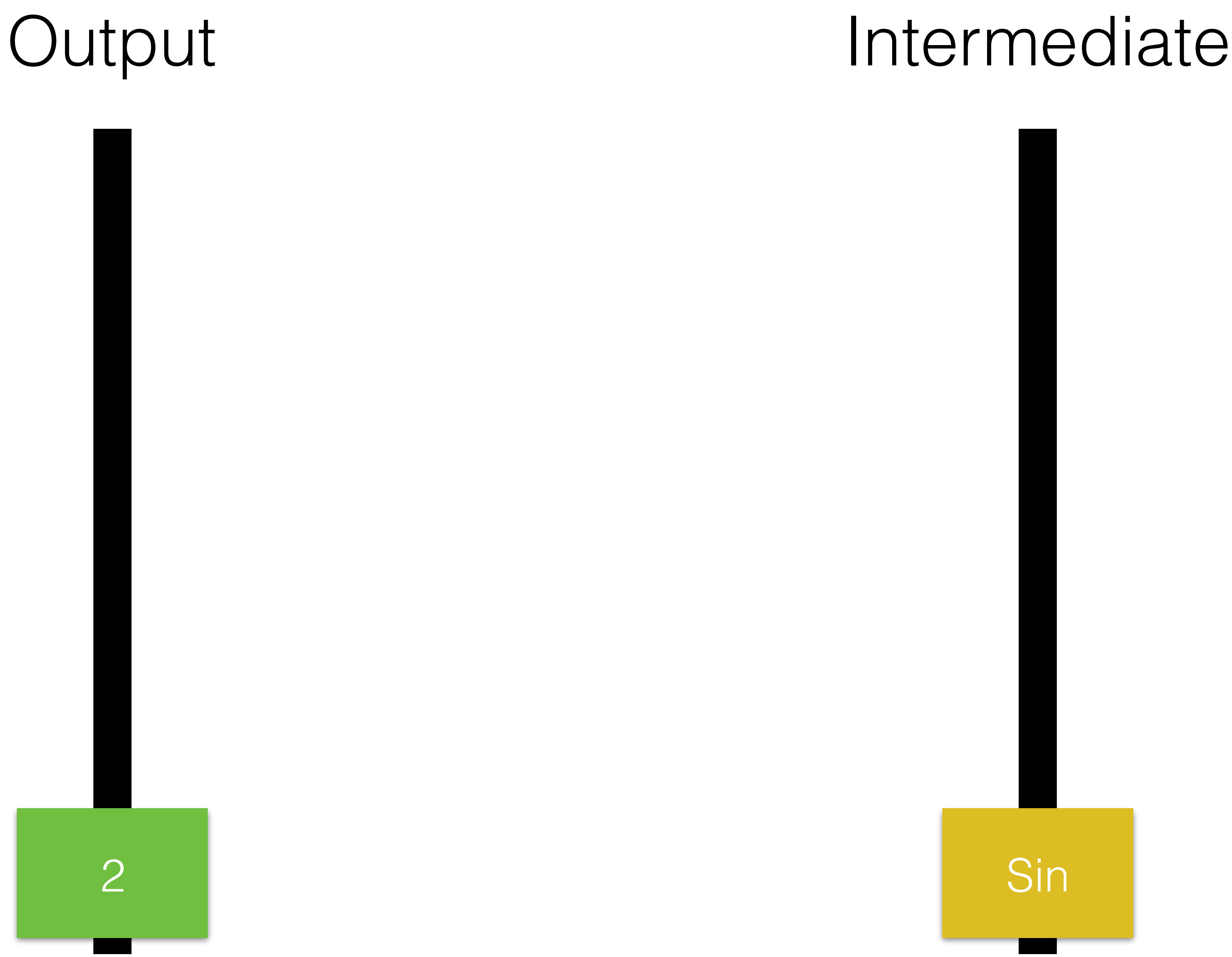
Output

Intermediate

EIS-2 Maths Parser

`sin(2 * pi)`

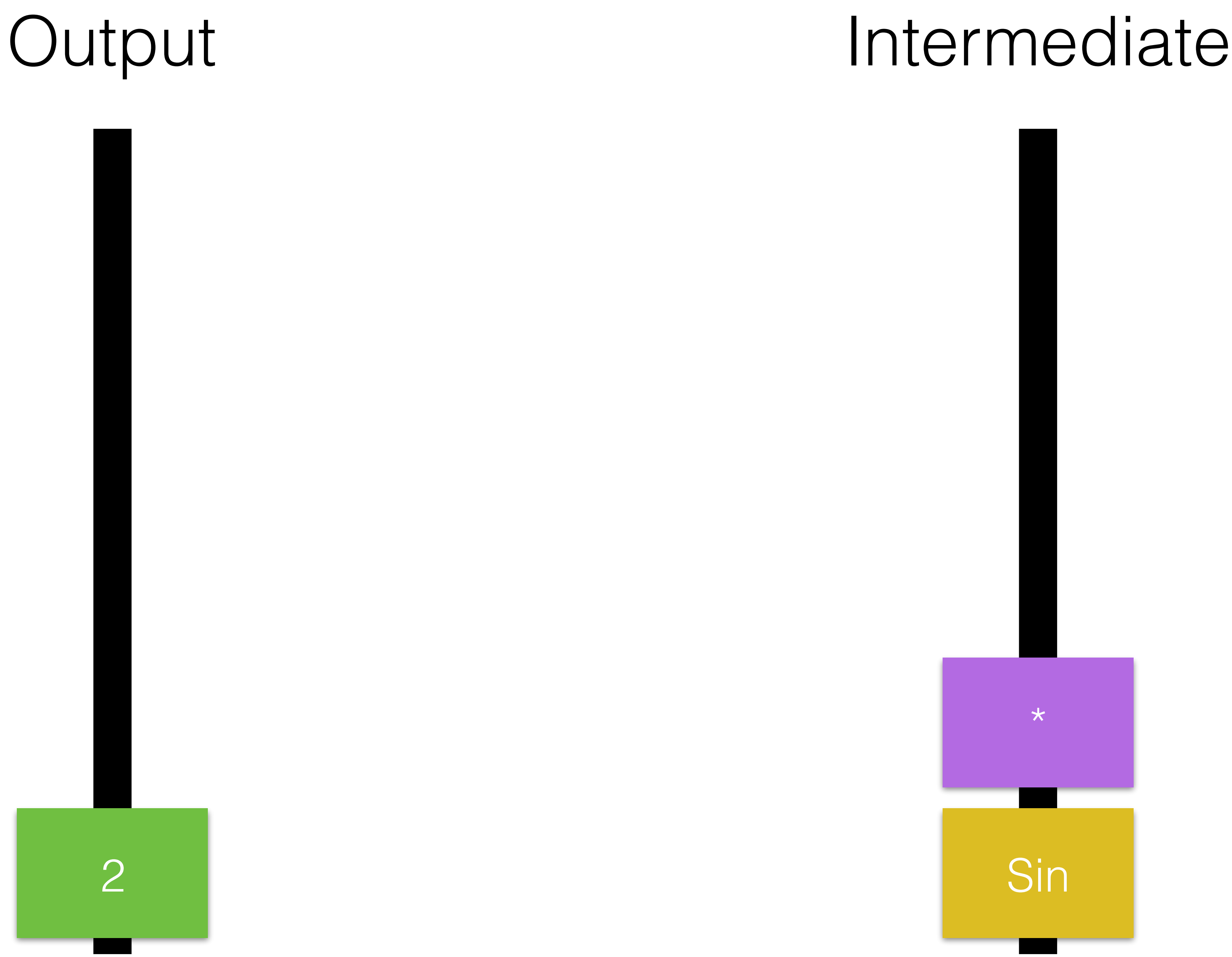
^



EIS-2 Maths Parser

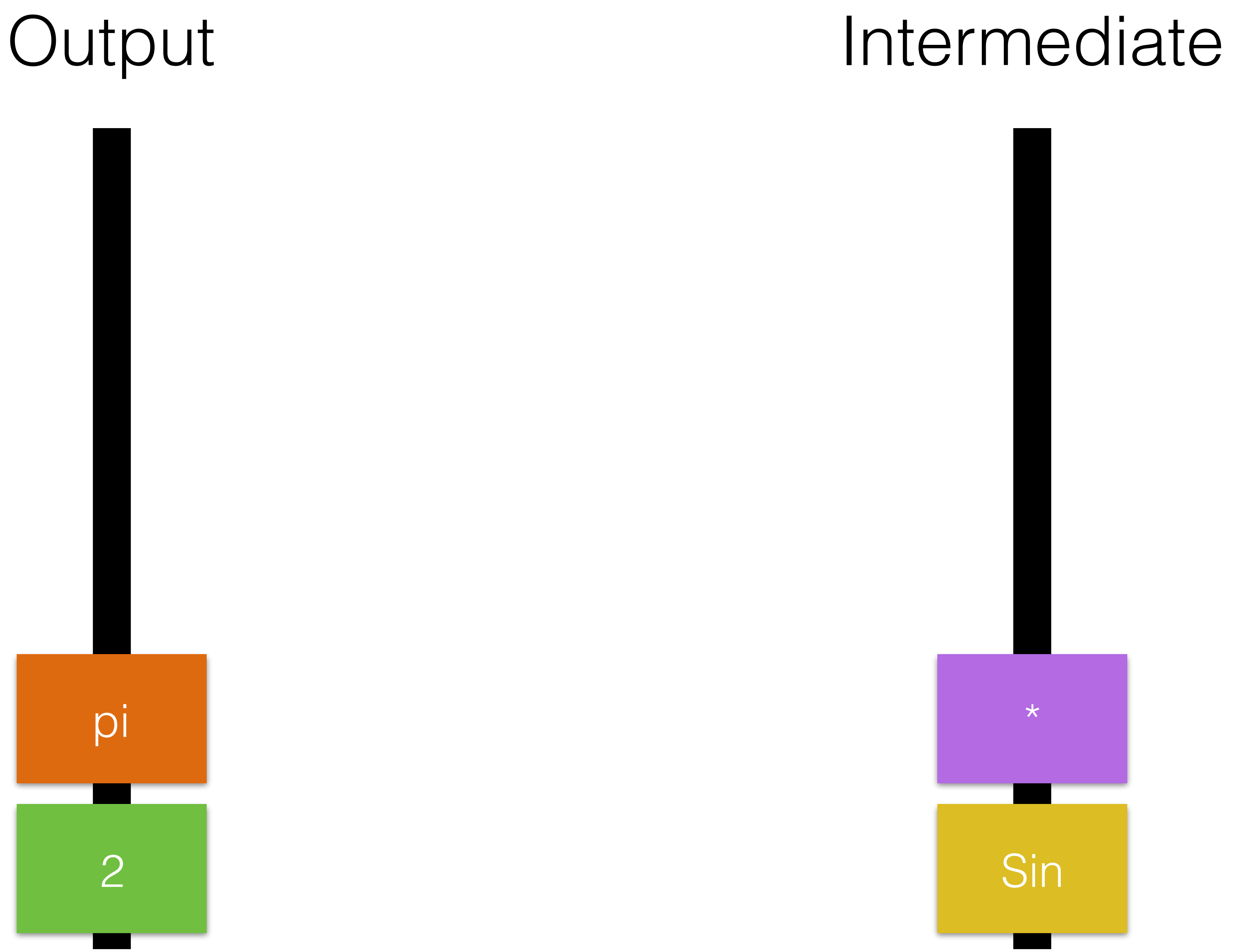
`sin(2 * pi)`

^



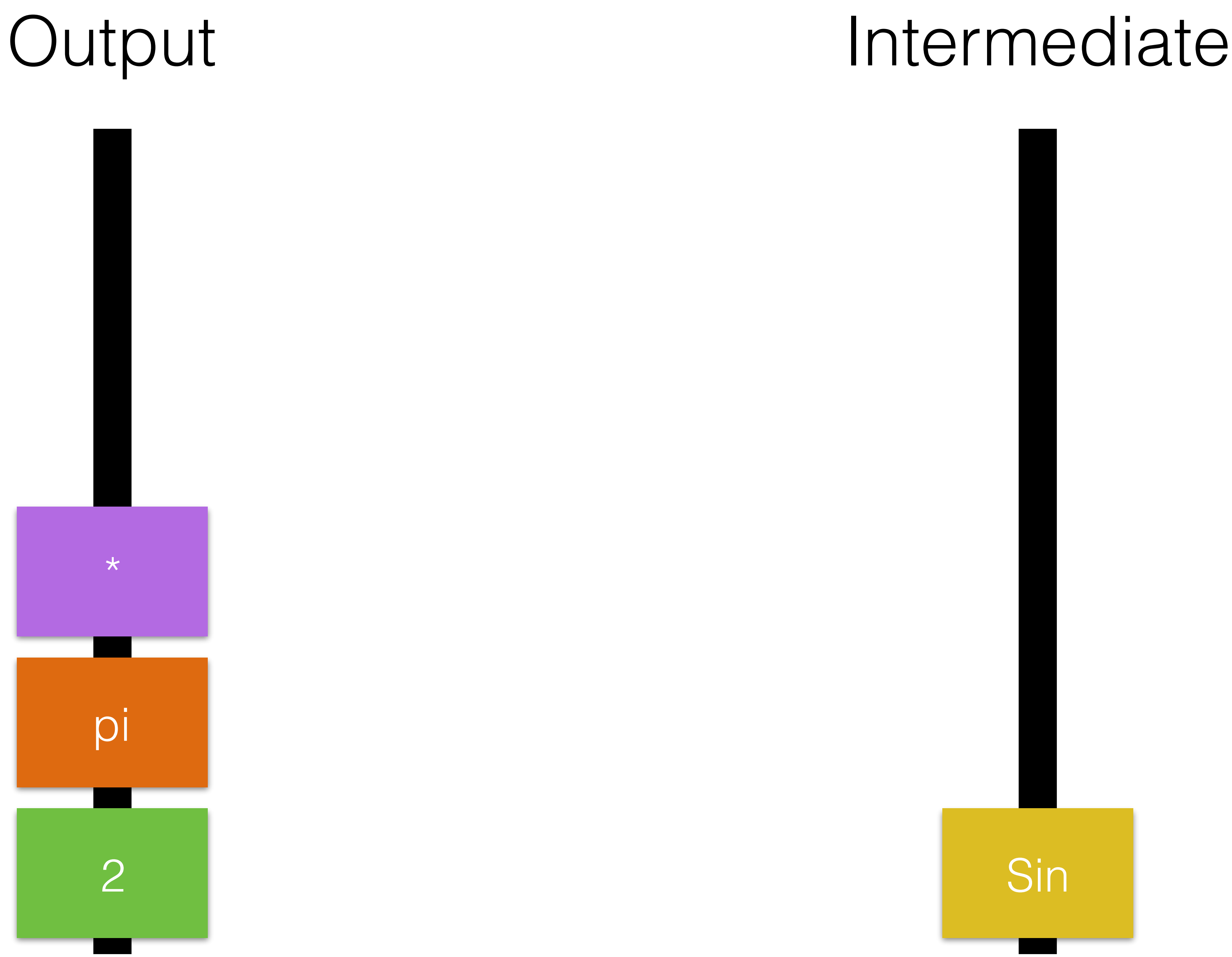
EIS-2 Maths Parser

```
sin(2 * pi)
        ^
```



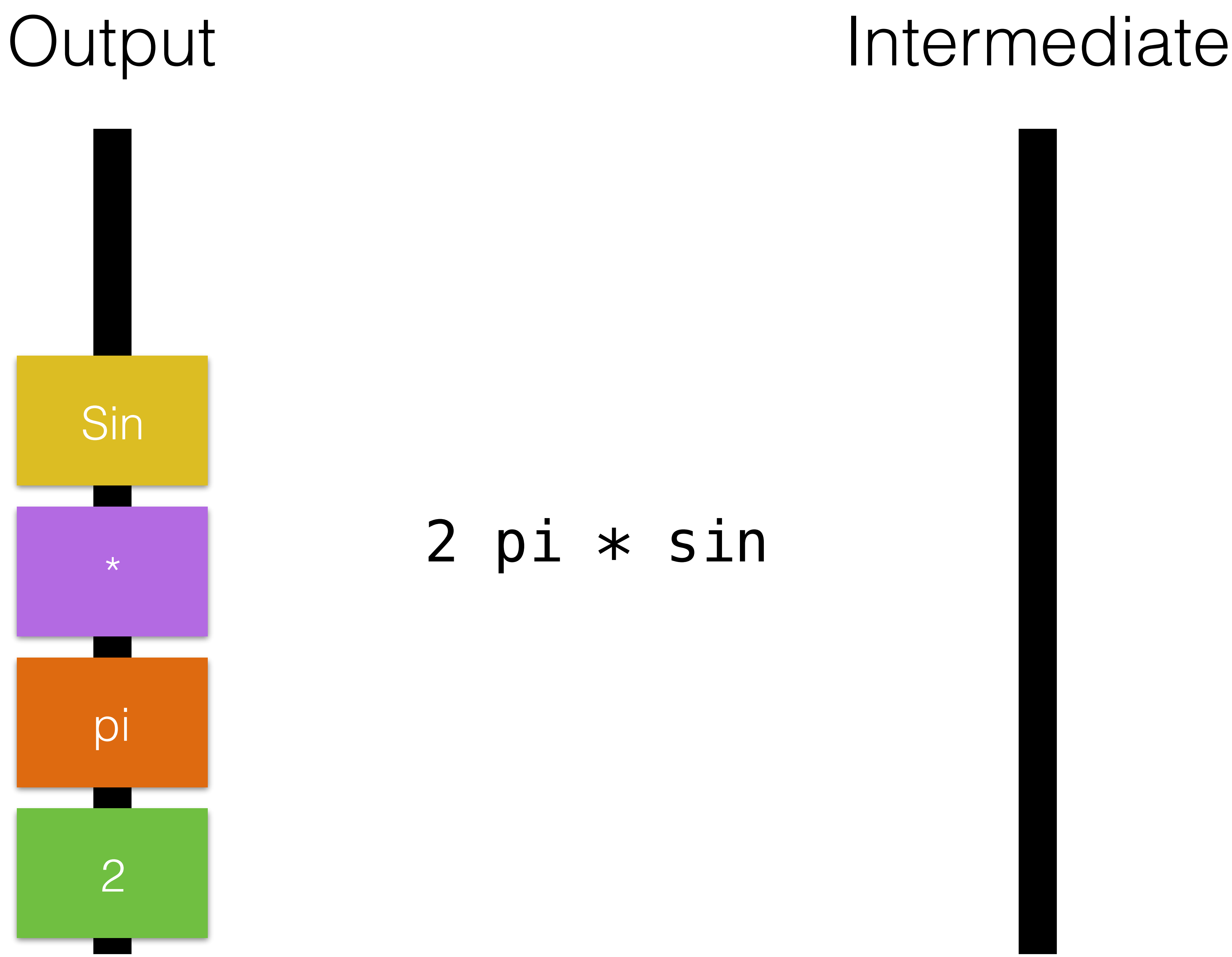
EIS-2 Maths Parser

`sin(2 * pi)`



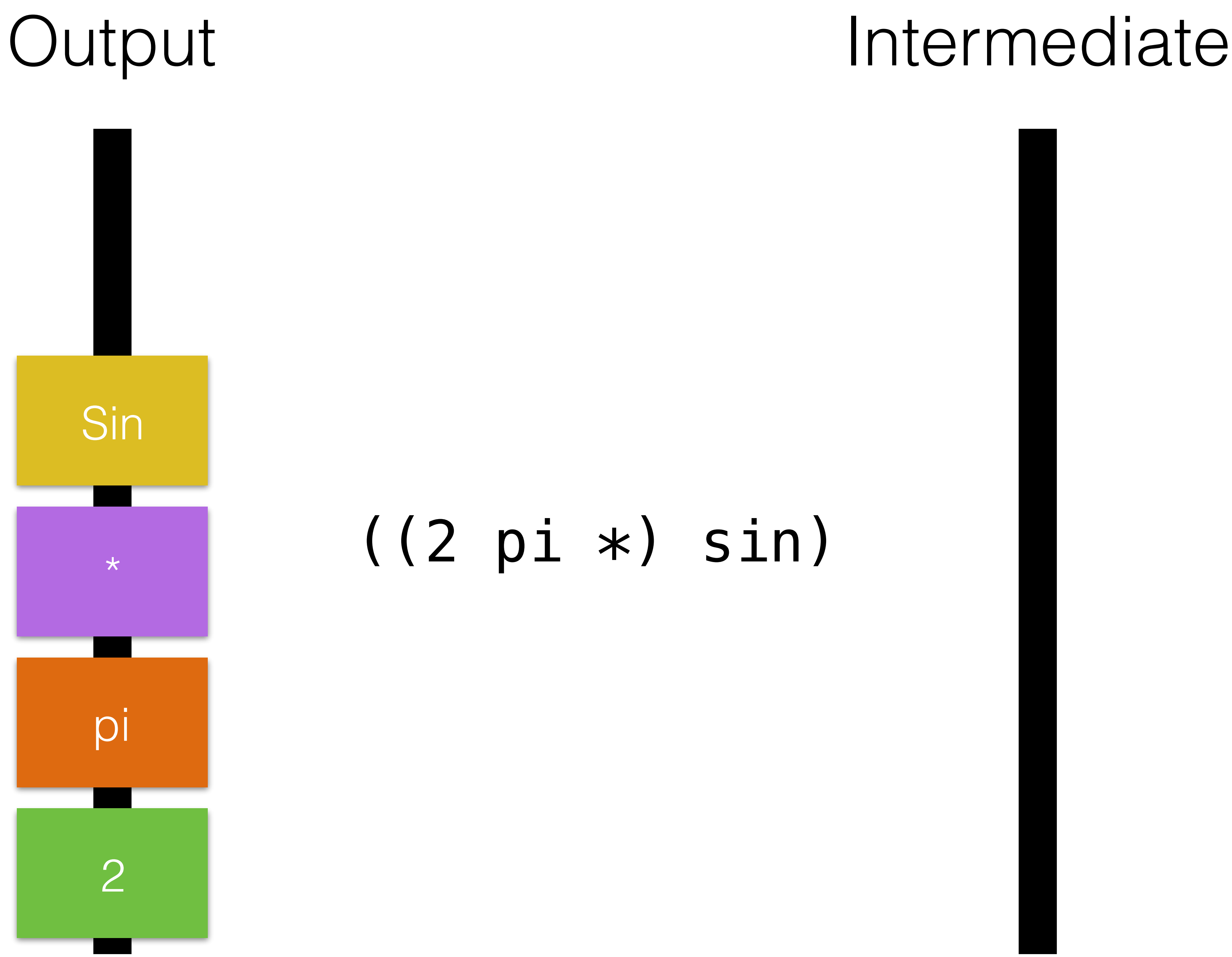
EIS-2 Maths Parser

`sin(2 * pi)`



EIS-2 Maths Parser

`sin(2 * pi)`



EIS-2 Maths Parser

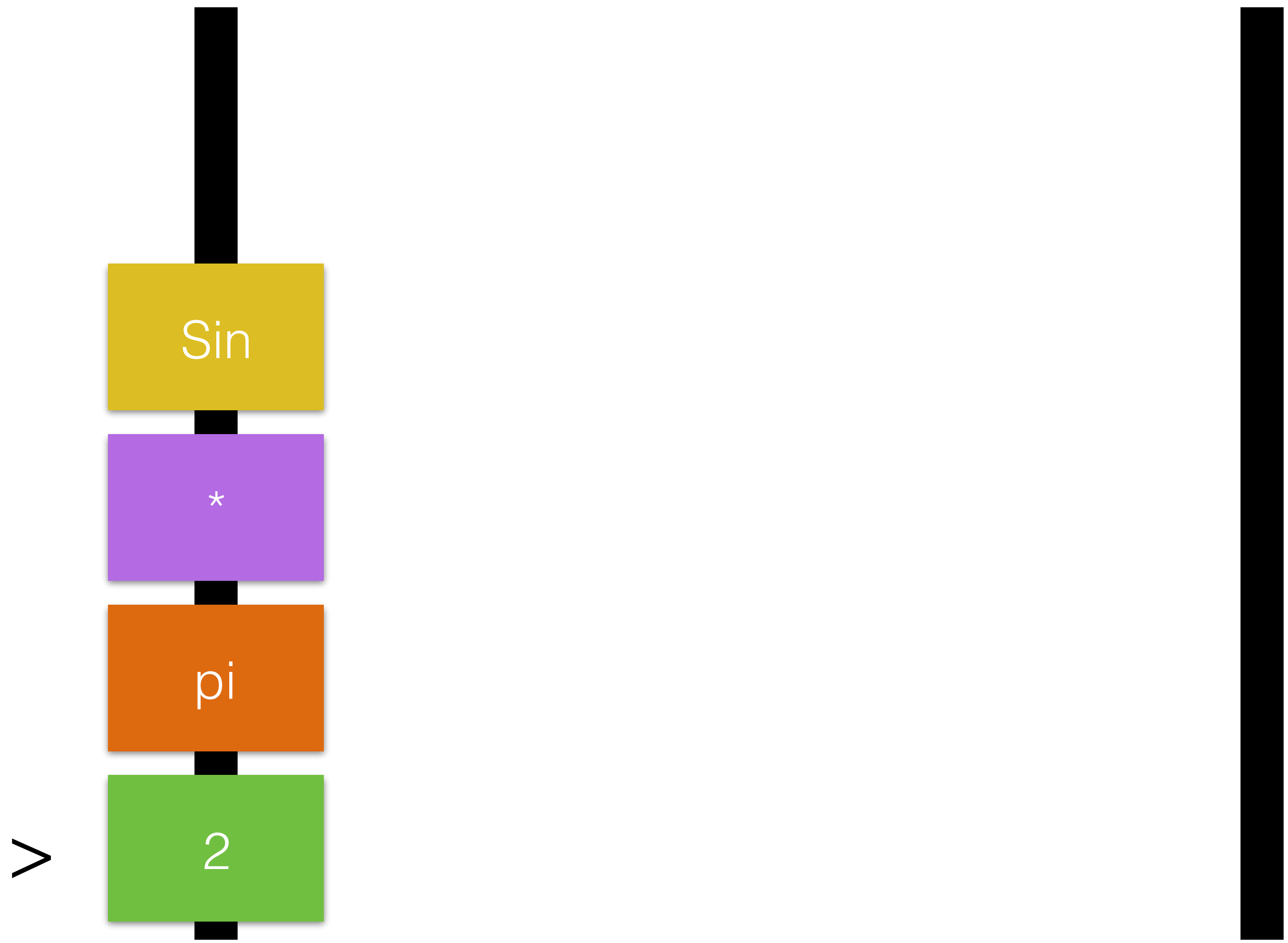
- This basically just converts from infix maths to postfix or RPN form
- **BUT** the computer can now simply evaluate this expression
- Just start at the bottom and work your way up

EIS-2 Maths Evaluator

`2 pi * sin`

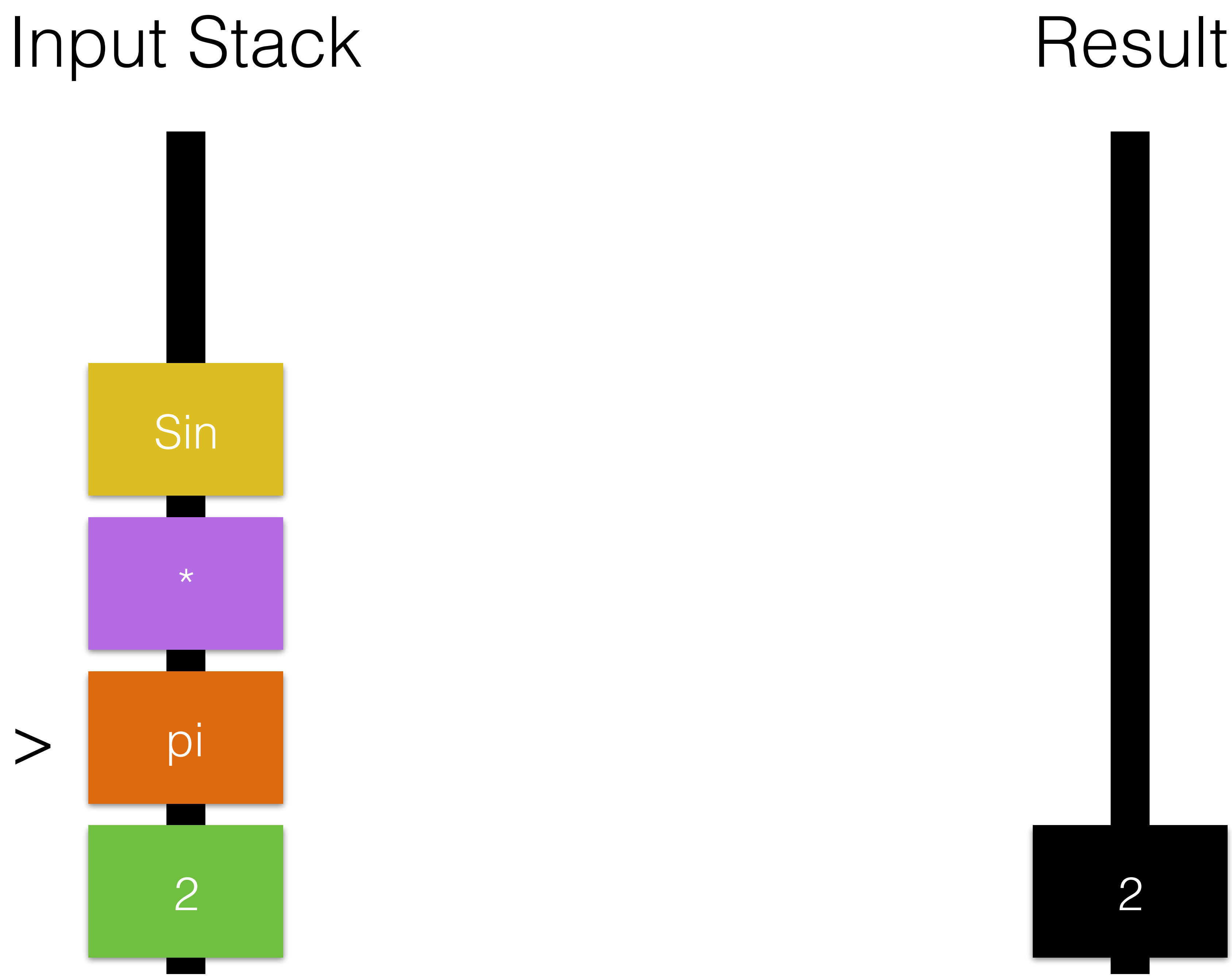
Input Stack

Result



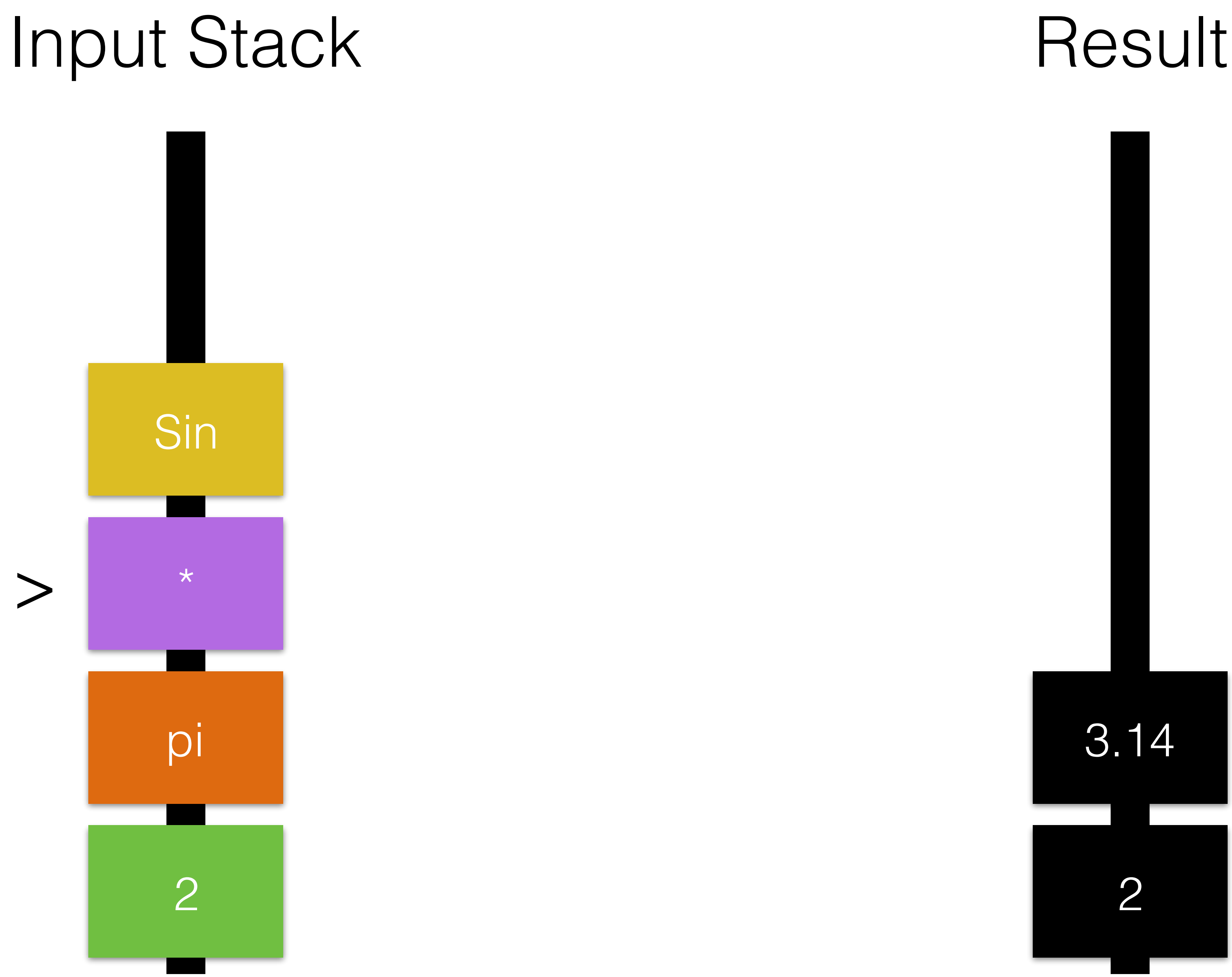
EIS-2 Maths Evaluator

`2 pi * sin`



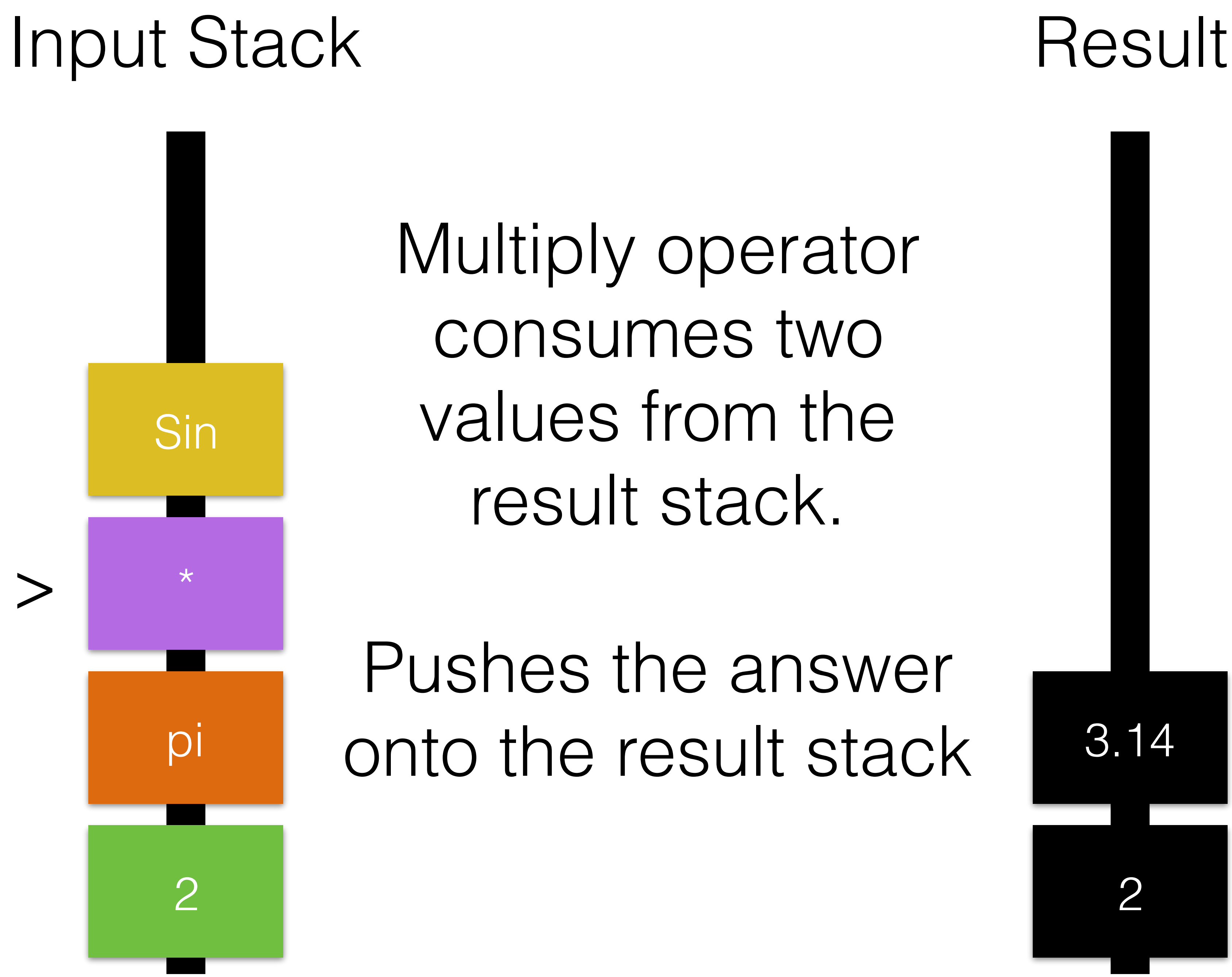
EIS-2 Maths Evaluator

```
2 pi * sin
```



EIS-2 Maths Evaluator

`2 pi * sin`

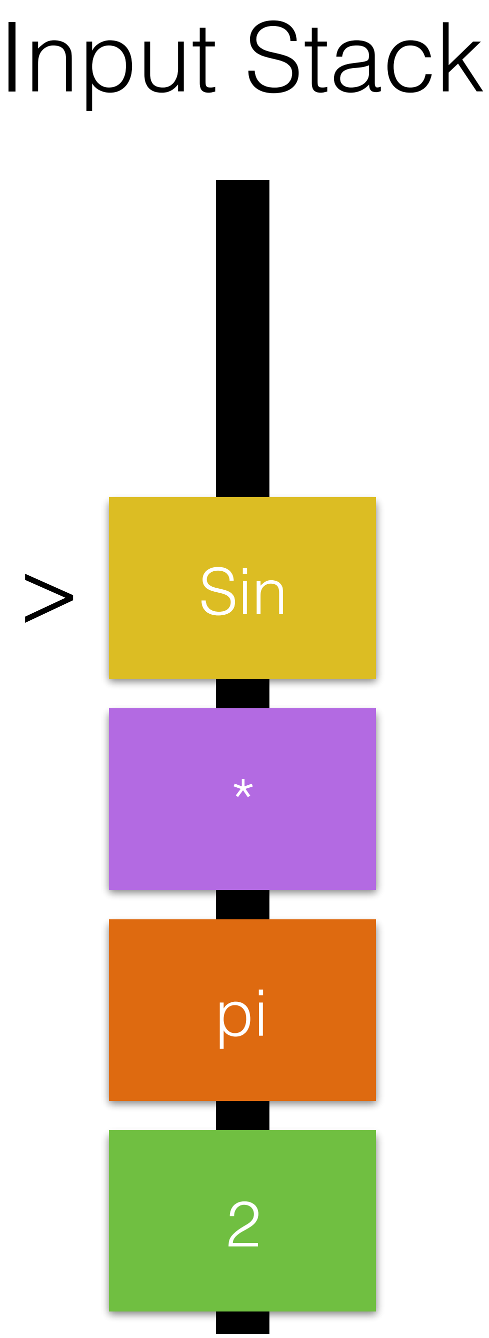


Multiply operator consumes two values from the result stack.

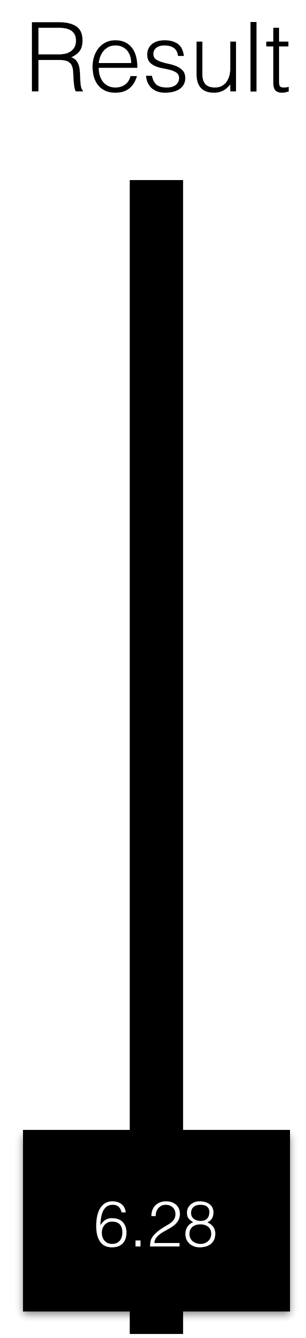
Pushes the answer onto the result stack

EIS-2 Maths Evaluator

`2 pi * sin`



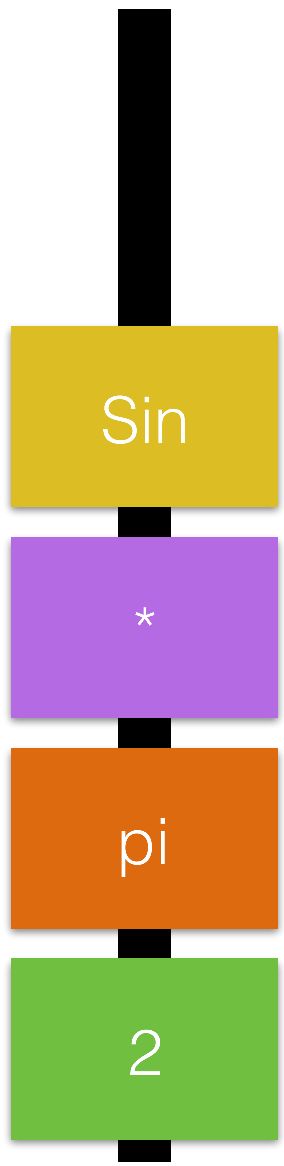
Sin consumes
one value
and pushes
on the result



EIS-2 Maths Evaluator

`2 pi * sin`

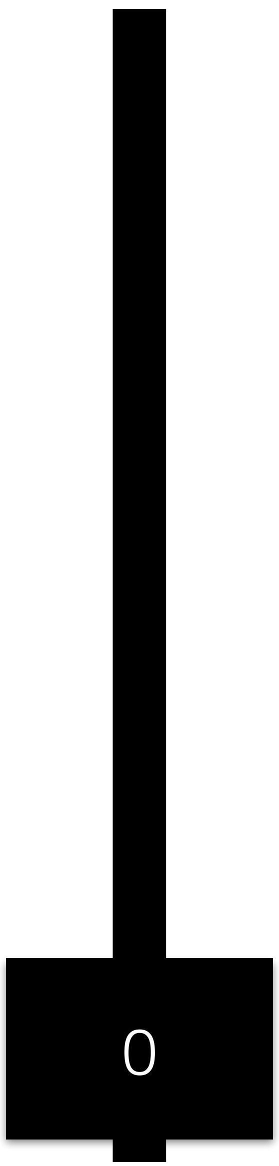
Input Stack



Now just get the result from the result stack

Input stack can be evaluated again if needed

Result



EIS-2 Maths Parser

- All of this mechanical work is hidden away by EIS-2
    - It is helpful to know because it explains some of how the code works
- How do you actually use the EIS-2 parser?
    - Not going to try to do a tutorial but will show the simplest bits

Actual code

```
PROGRAM test

  USE eis_parser_header
  TYPE(eis_parser) :: parser
  CHARACTER(LEN=1000) :: input
  INTEGER(eis_error) :: errcode
  REAL(eis_num), DIMENSION(:), ALLOCATABLE :: result
  INTEGER :: ct

  DO WHILE(.TRUE.)
    WRITE(*,'(A)', ADVANCE = 'NO') "Please input a mathematical expression :"
    READ(*,'(A)') input
    ct = parser%evaluate(input, result, errcode)
    IF (errcode == eis_err_none) THEN
      PRINT *,'Result is ', result(1:ct)
    ELSE
      CALL parser%print_errors()
    END IF
  END DO

END PROGRAM test
```

EIS-2 Maths Parser

- Fortran "**eis_parser**" object does the heavy lifting
- Call the "**evaluate**" method with a string containing a valid mathematical expression
  - Result is an array containing all the values left on the "**result**" stack after evaluation
  - Allows evaluation of vector valued expressions as well as single values
- There is also a "**tokenize**" method to produce a stack that you can keep and re-evaluate
- If there are errors in the expression then you get an error code returned and can examine the errors produced

EIS-2 Maths Parser

```
Please input a mathematical expression :sin(10,20)
================================================================================
The wrong number of parameters was used in a function call

1 : sin(10,20)
    ^
    1
================================================================================
```

EIS-2 Maths Parser

```
Please input a mathematical expression :10+20*wibble(2*pi)
================================================================================
Unknown value or function

1 : 10+20*wibble(2*pi...
          ^
          7
================================================================================
```

EIS-2 Maths Parser

```
Please input a mathematical expression :1+log10(-1)
================================================================================
A mathematically invalid operation was requested

1 : 1+log10(-1)
      ^
      3
================================================================================
```

EIS-2 Maths Parser

```
Please input a mathematical expression :1+log10(-1)
================================================================================
A mathematically invalid operation was requested

1 : 1+log10(-1)
      ^
      3
================================================================================
```

- Error messages are built into the EIS-2 code but can be overridden from an external file for localisation purposes

- Support for Unicode where your Fortran compiler supports it (none of them do very well on this yet)

Parser Objects

- **Literals** - Put in by users as number
- **Operators** - both unary operators and binary operator. No ternary operators
- **Constants** - Name mapped to a constant value
- **Functions** - Take parameters and return a value. May optionally be given a number of expected parameters or be *variadic* and check their parameter count themselves
- **Variable** - Name mapped to a result function (like a **function**) but takes no parameters
- **Functor** - Object that behaves like a function in the deck but carries state with it

Adding a constant

```
PROGRAM test

  USE eis_parser_header
  TYPE(eis_parser) :: parser
  CHARACTER(LEN=1000) :: input
  INTEGER(eis_error) :: errcode
  REAL(eis_num), DIMENSION(:), ALLOCATABLE :: result
  INTEGER :: ct

  CALL parser%add_constant('myconstant', 1.2345_eis_num, errcode)
  IF (errcode /= eis_err_none) CALL parser%print_errors()

  DO WHILE(.TRUE.)
    WRITE(*,'(A)', ADVANCE = 'NO') "Please input a mathematical expression :"
    READ(*,'(A)') input
    ct = parser%evaluate(input, result, errcode)
    IF (errcode == eis_err_none) THEN
      PRINT *,'Result is ', result(1:ct)
    ELSE
      CALL parser%print_errors()
    END IF
  END DO

END PROGRAM test
```

Adding a function

```
PROGRAM test

  USE eis_parser_header
  TYPE(eis_parser) :: parser
  CHARACTER(LEN=1000) :: input
  INTEGER(eis_error) :: errcode
  REAL(eis_num), DIMENSION(:), ALLOCATABLE :: result
  INTEGER :: ct

  CALL parser%add_function('cauchy', cauchy_dist, errcode, expected_params = 3)
  IF (errcode /= eis_err_none) CALL parser%print_errors()

  DO WHILE(.TRUE.)
    WRITE(*,'(A)', ADVANCE = 'NO') "Please input a mathematical expression :"
    READ(*,'(A)') input
    ct = parser%evaluate(input, result, errcode)
    IF (errcode == eis_err_none) THEN
      PRINT *,'Result is ', result(1:ct)
    ELSE
      CALL parser%print_errors()
    END IF
  END DO

END PROGRAM test
```

Adding a function

```fortran
!Function to implement the Cauchy distribution
!https://en.wikipedia.org/wiki/Cauchy_distribution
FUNCTION cauchy_dist(nparams, params, host_params, status_code, errcode) &
    RESULT(res) BIND(C)
  INTEGER(eis_i4), VALUE, INTENT(IN) :: nparams
  REAL(eis_num), DIMENSION(nparams), INTENT(IN) :: params
  TYPE(C_PTR), VALUE, INTENT(IN) :: host_params
  INTEGER(eis_status), INTENT(INOUT) :: status_code
  INTEGER(eis_error), INTENT(INOUT) :: errcode
  REAL(eis_num) :: res
  REAL(eis_num), PARAMETER :: pi = 4.0_eis_num * ATAN(1.0_eis_num)

  !params(1) - x, dependent variable
  !params(2) - x0, location parameter
  !params(3) - gamma, scale parameter

  res = 1.0/(pi * params(3)) * (params(3)**2 / (params(1) - params(2))**2 &
      + params(3)**2)

END FUNCTION cauchy_dist
```

Functions, Variable and Functors

- Functions and variables all have the same “getter function” as shown for the Cauchy function
  - Variables may optionally be specified by a Fortran or C pointer
- Functors are implemented as Fortran types derived from the “eis_functor” type and implement almost exactly the same function as an “operate” method but have a “this” parameter that refers to the functor itself

EIS-2 Interoperability

- EIS-2 is a Fortran library since it's main purpose is to work with EPOCH
- Can create all Fortran objects through a C interface. Become integer handles in C
- Either uses BIND(C) functions in Fortran (as shown above) or has both C and Fortran function interfaces
- C Functors work by capturing a "**void***" pointer at the time they are created and having an extra parameter to the getter function in C that returns that pointer
- Interoperability interface nearly complete for maths parser
- About 50% complete for whole library

Host parameters

- Generally don't want a parser that is entirely context independent
    - Want to specify context for what value a variable or function should return
- Can do this in various ways but **host parameters** is one
- C void pointer to anything you like. Taken when you evaluate an expression and passed to all the evaluation functions
- EPOCH uses host parameters to pass space and time information to parameters that users then use in the deck

Host parameters

```
TYPE, BIND(C) :: data_item
  REAL(eis_num) :: x = 0.0_eis_num
  REAL(eis_num) :: y = 0.0_eis_num
END TYPE data_item

CONTAINS

FUNCTION get_x(nparams, params, host_params, status_code, errcode) &
    RESULT(res) BIND(C)
  INTEGER(eis_i4), VALUE, INTENT(IN) :: nparams
  REAL(eis_num), DIMENSION(nparams), INTENT(IN) :: params
  TYPE(C_PTR), VALUE, INTENT(IN) :: host_params
  INTEGER(eis_status), INTENT(INOUT) :: status_code
  INTEGER(eis_error), INTENT(INOUT) :: errcode
  REAL(eis_num) :: res
  TYPE(data_item), POINTER :: dat

  IF (.NOT. C_ASSOCIATED(host_params)) RETURN
  CALL C_F_POINTER(host_params, dat)
  res = dat%x

END FUNCTION get_x
```

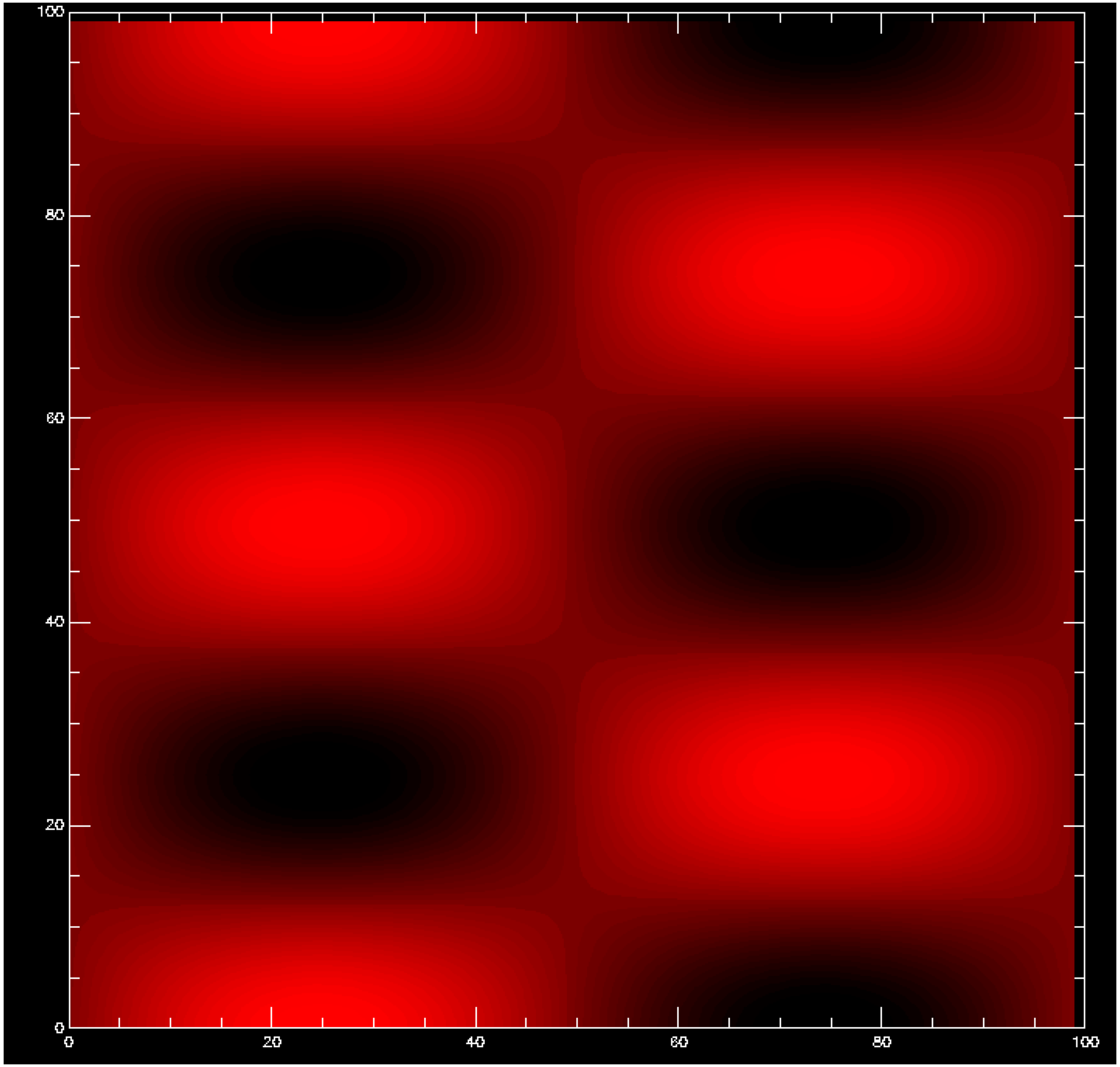
Host parameters

```
TYPE(data_item), TARGET :: item
CALL parser%add_variable('x', get_x, errcode)
CALL parser%add_variable('y', get_y, errcode)

WRITE(*,'(A)', ADVANCE = 'NO') "Please input a mathematical expression :"
READ(*,'(A)') input
CALL parser%tokenize(input, stack, errcode)
DO iy = 1, 100
  item%y = REAL(iy-1, eis_num)/99.0_eis_num
  DO ix = 1 , 100
    item%x = REAL(ix-1, eis_num)/99.0_eis_num
    ct = parser%evaluate(stack, result, errcode, host_params = C_LOC(item))
    WRITE(10,*) result(1)
  END DO
END DO
```

Host parameters

- Can put in any function of X and Y that you want
- Will be evaluated between 0->1 in both X and Y and written to file
- sin(x*2*pi)*cos(y*4*pi)



Advanced Parser Objects

- **Stack variables** - Variables that are created from a stack rather than from a numerical value. If variables, functions or functors behave differently when they are called with different **host parameters** they will continue to do so when they are referred to through a stack variable

```
begin:constant
  v0 = 0.05 * c
  p0 = v0 * me * (1.0 + 4.0 * x/x_max)
end:constant
```

p0 varies in space

Advanced Parser Objects

```
begin:species
  name = proton
  number_density = number_density(Electron)
  identify:proton
end:species
```

- **Emplaced functions** - Generalisation of stack variables. Takes parameters like a function but returns a stack rather than a value which retains all of it's time and space varying properties as well.

  - Have a different getter function to a normal function

- You tell the parser when to actually call the getter function so you can use emplaced functions to choose exactly what a given stack does when it is evaluated

Simplifier

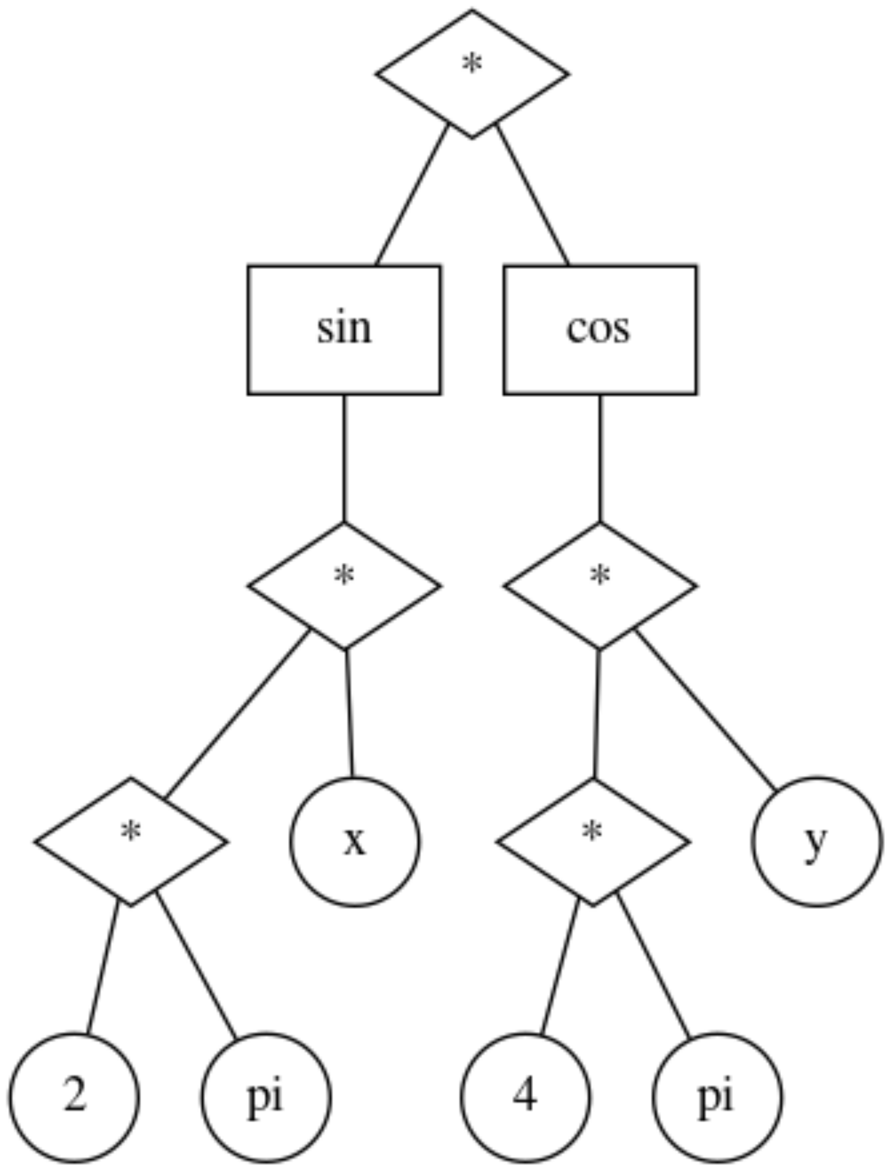
- The logic shown above for the parser ignores one obvious question - simplification
- Many of the tokens can be combined immediately but some have to be kept because they use host parameters or other external data source to change their results
- All constants **are** simplifiable
- Functions and functors where all parameters are simplifiable **are** simplifiable (unless the developer overrides)
- Variables are not simplifiable unless the developer overrides

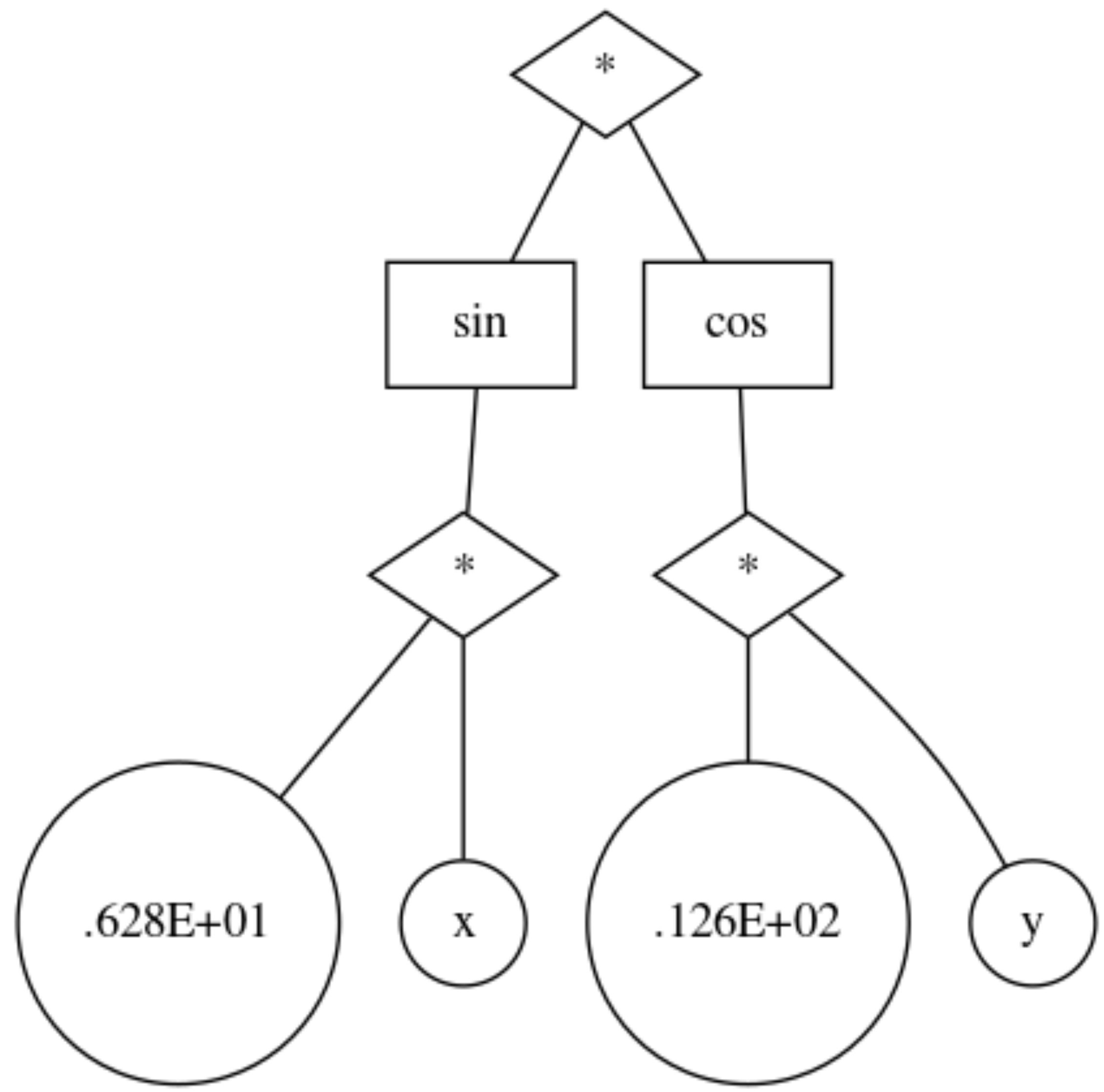
Simplifier

- Simplification works by forming an **abstract syntax tree** and replacing any branches that are simplifiable
- Alternative data structure to the stacks that EIS-2 uses. Many other parsers use ASTs to store their main data
  - Tends to be slower to evaluate
- Can massively simplify expressions

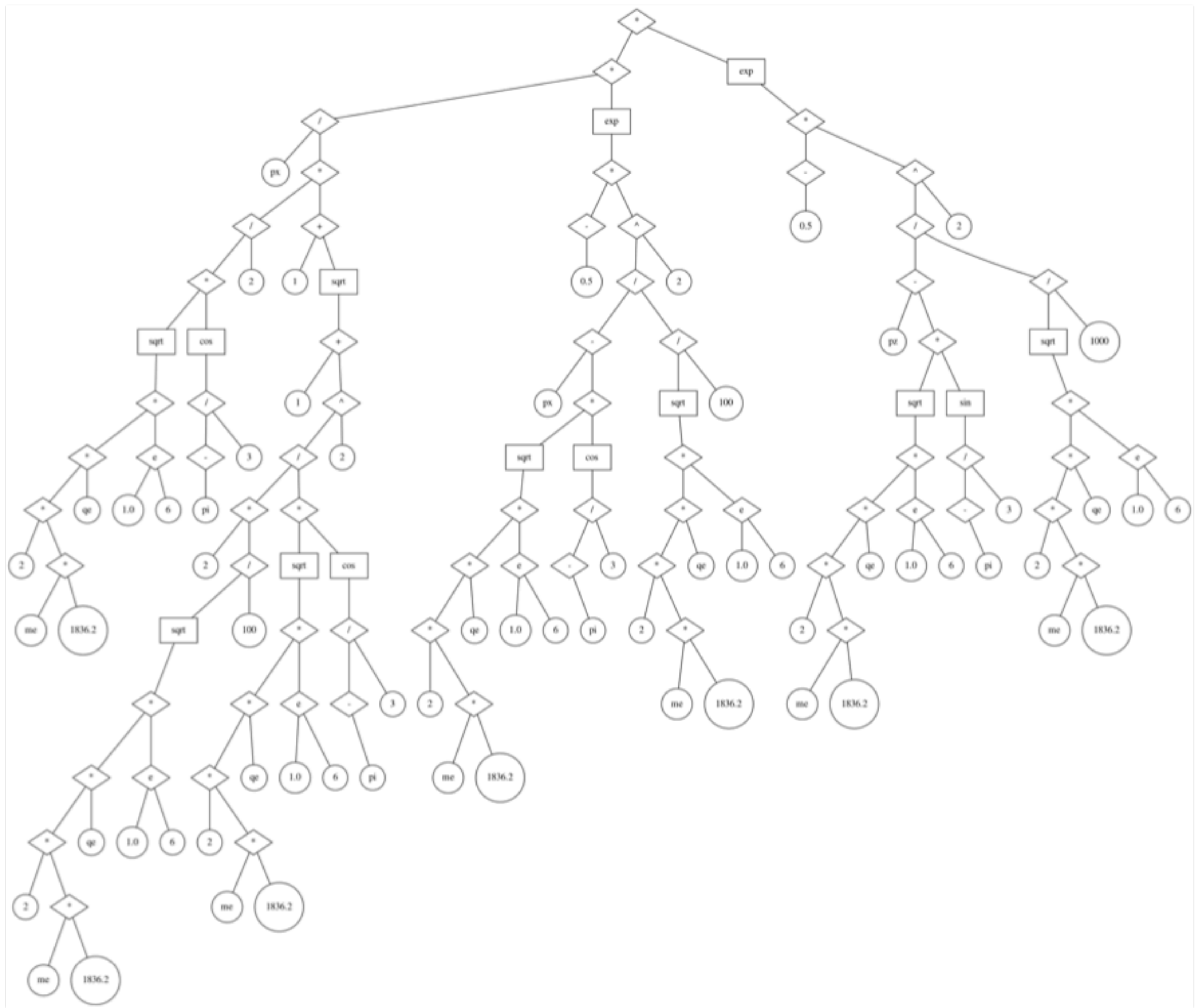
Simplifier

sin(2 * pi * x) * cos(4 * pi * y)

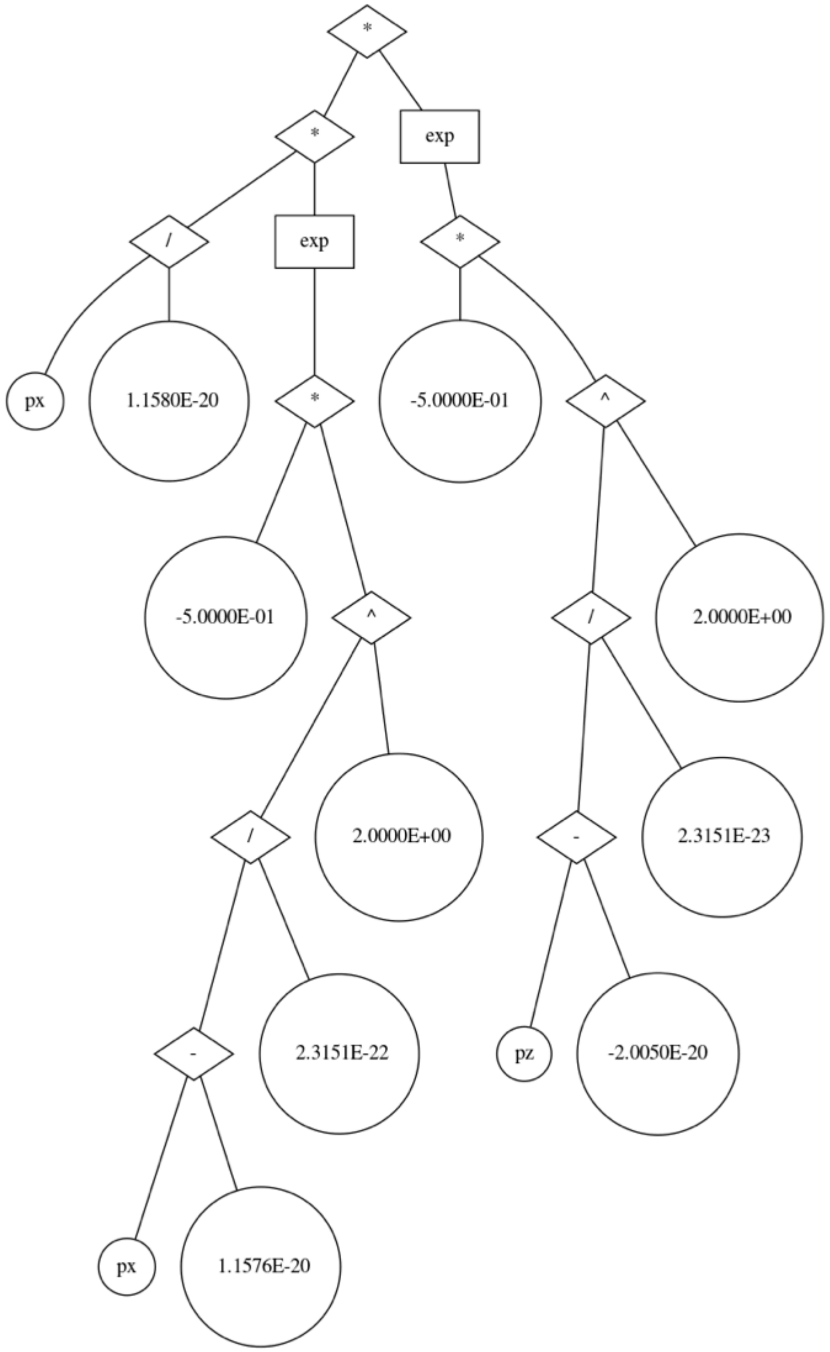




Simplifier



Simplifier



*
*
exp
/
exp
*
px
1.1580E-20
*
-5.0000E-01
^
-5.0000E-01
^
/
2.0000E+00
/
2.0000E+00
-
2.3151E-23
-
2.3151E-22
pz
-2.0050E-20
px
1.1576E-20

Performance

- Can’t get to performance of native code
- Performance is still quite good
  - ~30-40 CPU cycles per stack element
  - Typically about 1/10th speed of native scalar floating point code
- About 25 times faster than using a Python interpreter with numpy

Performance

- Stacks are opaque objects, the host code just **evaluate**s them to get a result
  - Can bind a **result function** to them for faster performance
  - Overhead of one function pointer ~ 4-5 CPU cycles
- Future work aims to offer an option to use libllvm to compile stack expressions
  - Only preliminary tests so far and user specified functions/constants will either be harder to produce or will still be external function calls

Advantages and Disadvantages

Advantages

- Static library in standards compliant F2003 (or optionally F2008) with no library dependencies
  - Easy to build on any platform and easy to link to your code
- Much faster than general purpose scripting languages like Python (~25x faster)
- Easier to add to your code than Python (or even Lua)

Advantages

- Gives users exactly the level of control that you want over your code
- No effects outside your code in the EIS-2 library
  - Decks are safe to run unless your code implements destructive features through the deck
- BSD 3 clause license, compatible with open and closed source software
  - Intended for shipping with your code
- Intended for HPC workflow

Disadvantages

- Only supports real and string datatypes (and strings are a bit limited)
- Simplifier has some limitations
    - Working on them!
- Is more restrictive than scripting language if you want to give users that much power
- Limited ecosystem since brand new

Deck parser

Deck parser

- Why deck?
  - Literally from a deck of punch cards back in the early computer days
- Why not?
  - As good a term as anything
- Means an input telling a program what to do without having to recompile the source code
- In EIS-2 connected to the maths parser since it is used to process some of the input

Deck Terms

- **Block** - collection of connected keys. May contain other blocks
  - **Type** - a definition of a block. Every block type has a unique ID number
  - **Instance** - an actual block in a deck. Each instance of a block has a unique ID that is not related to the unique ID for the block type
- **Key** - named item that is associated with a value
- **Value** - an input that the host code wants
- **Definition** - A definition of the possible blocks and keys in a deck. Done through and eis_deck_definition object
- **Pass** - A run of the deck parser over a deck. Decks may take multiple passes to be fully read

Deck Terms

- **Root** - The block instance that all other blocks are within
- **Parents** - The unique IDs of the block instances that are the parents of the current block instance. The last parent of a block is the block itself
- **Parent_kinds** - The unique IDs of the block types that are the parents of the current block instance. The last parent_kind of a block is the kind of the block itself

```
begin:laser
  boundary = x_min
  intensity_w_cm2 = 1.0e15
  lambda = 1 * micron
  t_profile = gauss(time,4*femto,4*femto)
  t_end = 14 * femto
end:laser
```

Deck definitions

- A deliberate design decision of the EIS-2 deck parser was to separate the *definition* of the structure of an input deck from the *instantiation* of a deck
  - Currently written to parse EPOCH style decks
  - Could easily write a parser for JSON, YAML, XML, Windows INI files etc. and would be a drop in replacement for a code
- Definition specifies **action functions** for when events occur
  - In order to keep interfaces natural for each language, separate C and Fortran versions of all action functions

Block action functions

- **init_block** - When a block *type* is first encountered in the first pass
- **start_pass** - When a block *type* is first encountered in a given pass
- **start_block** - When a block *instance* is started
- **end_block** - When a block *instance* is ended
- **end_pass** - When a pass ends and a block of this *type* has been encountered
- **final_block** - When parsing is finished and a block of this *type* has been encountered

Key action functions

- Key action functions are called when a key is encountered
- **key_value_fn** - Returns strings for key and value
- **key_numeric_value_fn** - Returns string for key and uses an eis_parser object to calculate a numeric value
- **key_stack_fn** - Returns string for key and an eis_stack object for the value
- Can also store the numeric value directly to a C or Fortran pointer to an integer or real variable
- Blocks have any_* versions of these for handling non-specific keys

EPOCH deck parsing

- EPOCH type decks are parsed using an **eis_text_deck_parser** object
- Loads data from file and processes it using a definition
  - Option to load data and store it to a character variable
    - Includes information on line number etc. that is used for error reporting
    - Used in EPOCH to load deck on rank 0 and broadcast to other processors

Example

```
TYPE(eis_text_deck_parser) :: deck
TYPE(eis_deck_definition) :: dfn
INTEGER(eis_error) :: errcode
TYPE(eis_deck_block_definition), POINTER :: root, block

errcode = eis_err_none
root => dfn%init()
block => root%add_block('block1')

CALL block%add_key('key1', key_value_fn = key_str_sub, &
    key_numeric_value_fn = key_val_sub)
CALL block%add_key('key2', key_value_fn = key_str_sub, &
    key_numeric_value_fn = key_val_sub)

block => root%add_block('block2')

CALL block%add_key('new_key', key_value_fn = key_str_sub, &
    key_numeric_value_fn = key_val_sub)

CALL deck%init()
CALL deck%parse_deck_file('demo.deck', dfn, errcode, &
    allow_empty_blocks = .TRUE.)
IF (errcode /= eis_err_none) THEN
  DO ierr = 1, deck%get_error_count()
    CALL deck%get_error_report(ierr, str)
    PRINT *, str
  END DO
  DEALLOCATE(str)
END IF
```

Example

```fortran
SUBROUTINE key_str_sub(key_text, key_value, pass_number, &
    parents, parent_kind, status_code, host_state, errcode)
  CHARACTER(LEN=*), INTENT(IN) :: key_text
  CHARACTER(LEN=*), INTENT(IN) :: key_value
  INTEGER, INTENT(IN) :: pass_number
  INTEGER, DIMENSION(:), INTENT(IN) :: parents
  INTEGER, DIMENSION(:), INTENT(IN) :: parent_kind
  INTEGER(eis_status), INTENT(INOUT) :: status_code
  INTEGER(eis_bitmask), INTENT(INOUT) :: host_state
  INTEGER(eis_error), INTENT(INOUT) :: errcode
  INTEGER :: lq, uq

  !If no quotes are present then this isn’t a string
  lq = INDEX(key_value, '"')
  IF (lq == 0) THEN
    status_code = eis_status_not_handled
    RETURN
  END IF

  PRINT *,'Found text key ', TRIM(key_text), '. Value is ', &
      key_value(lq+1:uq-1)

END SUBROUTINE key_str_sub
```

Example

```
SUBROUTINE key_val_sub(key_text, values, pass_number, cap_bits, parser, &
    parents, parent_kind, status_code, host_state, errcode)
  CHARACTER(LEN=*), INTENT(IN) :: key_text
  REAL(eis_num), DIMENSION(:), INTENT(IN) :: values
  INTEGER, INTENT(IN) :: pass_number
  INTEGER(eis_bitmask), INTENT(IN) :: cap_bits
  TYPE(eis_parser), INTENT(INOUT) :: parser
  INTEGER, DIMENSION(:), INTENT(IN) :: parents
  INTEGER, DIMENSION(:), INTENT(IN) :: parent_kind
  INTEGER(eis_status), INTENT(INOUT) :: status_code
  INTEGER(eis_bitmask), INTENT(INOUT) :: host_state
  INTEGER(eis_error), INTENT(INOUT) :: errcode

  PRINT *,'Found numerical key ', TRIM(key_text), '. Values are ', values

END SUBROUTINE key_val_sub
```

Example

```
begin:block1
  key2 = "my key"
end:block1

begin:block2
  new_key = 7+12
end:block2

begin:block1
  key1 = sin(pi/3)
end:block1
```

```
Found text key key2. Value is my key
Found numerical key new_key. Values are    19.000000000000000
Found numerical key key1. Values are   0.86602540378443860
```

Parser and stacks

- By default an eis_text_deck_parser object will create a maths parser for itself
- You can optionally specify a pointer to an eis_parser object to the init method to specify a custom parser that has your variables, functions etc. in it
- If you use the action functions that give you a stack you should copy the stack if you want to keep it
  - In Fortran just do “mystack = stack”
  - In C there is an ‘eis_copy_stack’ function

Other use of deck parser

- As well as `eis_text_deck_parser` there is another method for calling the bits of a deck definition. `eis_deck_caller`
- This is a programatic way of starting and ending blocks and calling keys
- Together with the ability to bind a result function to a stack this also means that you can use your deck definition to provide an external interface to your code

Other use of deck parser

```
FUNCTION epoch_start_block(block_name) BIND(C)

  IMPLICIT NONE

  TYPE(C_PTR), VALUE, INTENT(IN) :: block_name
  INTEGER(eis_error) :: epoch_start_block
  INTEGER :: uid

  CALL eis_c_f_string(block_name, f_blockname)
  uid = deck_caller%start_block(f_blockname, epoch_start_block, &
      pass_number = 1)

  DEALLOCATE(f_blockname)

END FUNCTION epoch_start_block
```

Other use of deck parser

```
FUNCTION epoch_call_key(key, value, value_function) BIND(C)

  TYPE(C_PTR), VALUE, INTENT(IN) :: key, value
  TYPE(C_FUNPTR), VALUE, INTENT(IN) :: value_function
  INTEGER(eis_error) :: epoch_call_key
  PROCEDURE(parser_result_function), POINTER :: value_function_f
  INTEGER :: uid
  CHARACTER(LEN=:), ALLOCATABLE :: key_f, value_f

  CALL eis_c_f_string(key, key_f)
  CALL eis_c_f_string(value, value_f)
  IF (C_ASSOCIATED(value_function)) THEN
    CALL C_F_PROCPOINTER(value_function, value_function_f)
    uid = deck_caller%call_key(key_f, epoch_call_key, &
        value_text = value_f, value_function = value_function_f, &
        pass_number = 1)
  ELSE
    uid = deck_caller%call_key(key_f, epoch_call_key, &
        value_text = value_f, pass_number = 1)
  END IF
```

Other use of deck parser

```
FUNCTION epoch_end_deck() BIND(C)

  INTEGER(eis_error) :: epoch_end_deck

  CALL deck_caller%end_pass(epoch_end_deck, pass_number = 1)
  IF (epoch_end_deck /= eis_err_none) RETURN

  ! This line replays the deck that you ran for pass 2
  CALL deck_caller%replay_deck(epoch_end_deck, pass_number = 2, &
      replay_control_blocks = .TRUE.)
  ! Now finalise all blocks and the deck has been parsed
  CALL deck_caller%finalise_all_blocks(epoch_end_deck, pass_number = 2)

END FUNCTION epoch_end_deck
```

Other use of deck parser

- That isn't everything that's needed even for that interface (what's missing is mostly EPOCH internals but there is also stuff on reporting lines and filenames for errors from external codes)
- Also in a practical code you will probably want to have more features than just being able to run your input deck from an external program
- But it is a very strong start to being able to mix a high performance text file input with either allowing linking of codes for multi scale simulations or using a scripting language to provide interactive operation of your code

Description and Literate Input

Descriptions

- Both parser elements (functions, constants, functors etc.) and deck blocks and keys can have text descriptions associated with them

- You can retrieve these descriptions element by element or you can get EIS-2 to produce you a markdown document for either parser or deck information

Descriptions

EPOCH deck

Functions

- `abs(a)` - Returns absolute value of a
- `floor(a)` - Returns the nearest integer to a rounding towards zero
- `ceil(a)` - Returns the nearest integer to a rounding away from zero
- `ceiling(a)` - Returns the nearest integer to a rounding away from zero
- `nint(a)` - Returns the nearest integer to a rounding to nearest integer
- `trunc(a)` - Returns the nearest integer to a by ignoring the non-integer part
- `truncate(a)` - Returns the nearest integer to a by ignoring the non-integer part
- `aint(a)` - Returns the nearest integer to a by ignoring the non-integer part
- `sqrt(a)` - Returns the square root of a
- `sin(a)` - Returns the sine of a, specifying a in radians
- `cos(a)` - Returns the cosine of a, specifying a in radians
- `tan(a)` - Returns the tangent of a, specifying a in radians
- `asin(a)` - Returns the inverse sin of a, specifying the resulting angle in radians

Parser Visualisations

- The maths parser can visualise a stack in various ways
  - You can get the RPN version of the current state of the stack (simplified or not as specified)
  - You can convert a stack back into infix maths in current state (even without simplification some changes to brackets will occur although the expression **will** be mathematically equivalent)
  - A graphviz .dot file view of the stack (see the diagrams from earlier)

Deck Visualisations

- You can also visualise deck definitions or actual decks as graphviz dot files
- Bit niche in general because most decks tend to be quite flat
- Can be useful for debugging

Conclusions

Future Work

- Finish the interoperability interface
- Add vector execution option - execute several values at once
    - Performance improvement only for complex individual getter functions
- Add LLVM compilation option (?)
    - Quite tricky and performance benefits will be limited without requiring host code functions be compiled to LLVM intermediate language
- More examples and documentation

Conclusions

- EIS-2 provides a complete, open source library for reading input files containing rich mathematical notation
- Currently fully featured for Fortran code but with a complete C interface coming soon for other languages
- Use of a statically linked library with no internal communications makes it well suited to large scale HPC
- Can also be used to provide a link from your code to other software drivers