


# Implementing Symbolic Algorithms as a part of ASR

for Google Summer of Code 23'

**Anutosh Surendra Bhat**

**IIT MADRAS (2019 - 2024)**

**Potential Project Mentors:**

@certik, @czgdp1807

# Contents

# Personal Information

### Points of contacts and other relevant links

- Name - Anutosh Surendra Bhat
- University - Indian Institute of Technology Madras (IIT M)
- Degree - Integrated Dual Degree (Bachelors and Masters)
- Major - Data Science and Engineering
- Residence - Madras/Chennai, India
- Timezone - Indian Standard Time (UTC + 5:30)
- Typical Working Hours - 6:00-10:00, 17:00-24:00 (IST)
- Github Account - GitHub
- Email -
    1. be19b037@smail.iitm.ac.in (Institute mail)
    2. andersonbhat491@gmail.com (Github linked mail)
    3. anutosh.bhat.21@gmail.com (Personal use)
- Language spoken - English

I am Anutosh Bhat , a fourth year undergraduate pursuing an integrated dual degree in the Data Science and Computational Biology Department of IIT Madras . I have been fortunate enough to study at IIT Madras since 2019 which holds the national ranking of **1**, currently in India ..... which is a country producing 15 lakh engineering graduates every year through thousands of institutions nationwide.

I am an Open Source and Software Development enthusiast and have contributed to some influential libraries like SymPy, SageMath, Networkx, Kyverno and a couple others in the past . My main interests revolve around domains like Symbolic and Numerical computations/algorithms and also some Cloud Native Computing based stuff. My goal is to explore a vast majority of Computer Science fields during my undergrad degree like Operating Systems, Database Management Systems and Compiler Design

and Architecture. Hence contributing to LPython now makes a lot of sense to me at this stage of my degree.

## Programming Background

- **General Programming Background -** I was introduced to python back in August of 2020 when I had registered for the following course on Programming, computing and graphics using Python at University . The course really showed me the power python carries and all its advantages . I really like it because of its easy to read syntax . I have been coding in python ever since.

  I am also decently fluent with CPP as I practise Data Structures and Algorithm questions in it and also Golang because I've contributed to a couple libraries which were primarily based on it.

- **Git and Github experience -**I have been using git for around 2 years now and I am quite familiar with all workflows involved with git and github . I understood the more tedious and error prone stuff like rebasing on other branches, hard and soft reset etc through making a decent number of mistakes on my Pr's while contributing to SymPy.
- **Platform Details -** I use Ubuntu 20.04 as my operating system on WSL 2, Windows 10 along with Visual Studio Code as my primary editor and debugger whenever I am working on a cpp/python project.
- **Open Source experience & GSoC 22' -** I have been involved in the Open Source community now for the past 2 years . As I mentioned above I've contributed to quite some influential libraries like SymPy, SageMath, Networkx, Kyverno and a couple others in the past. I was also a **Google Summer of Code** mentee with **SymPy** last year , where I worked on improving their limits and series module . A detailed report of my work can be found over here . After successful completion of my project, I was promoted and made a member of the core development team. Hence, I have been reviewing code and helping out new contributors actively since then. I will also be flying to Austin, Texas to deliver a talk on SymPy and my work there at the

annual SciPy Conference, 2023 . I was also actively involved in the CNCF community last year for around 6 months.

### Motivation behind participating in GSOC 23’

**General -** I am motivated to participate in GSoC 23' as a contributor, heavily based on my experiences from last year's GSoC. While I had the option to mentor projects from SymPy this year, I feel I still need to improve my software development skills before taking on that demanding task. Contributing allows me to do so. My past GSoC experience taught me communication ethicets, writing standard code, building and testing before review.

**LPython -** My Motivation behind contributing to LPython in particular is because unlike SymPy and some other orgs I’ve contributed to , it is currently in the development phase and hasn’t been established yet. Contributing to such a project and creating something impactful is what I look forward to. Also as I wrote above , my goal is to explore a vast majority of Computer Science fields during my undergrad degree and Compiler Design Is one of them. Hence contributing to LPython now makes a lot of sense to me at this stage of my academic life.

# Contributions to Lpython

I started contributing to LPython in the month of January, 2023 and have been contributing on a daily basis since then. Here are some of my contributions to Lpython in chronological order.

### Pull Requests

**(Merged) #1503** - Updated error message related to for loops being used on undeclared iterables

**(Merged) #1509** - Added empty visit_For function in the SymbolTableVisitor class

**(Merged)** **#1512** - Fixed compatibility of sep, end keyword with list/tuple containers

**(In Progress)** **#1529** - Extended escape sequence (tab and backspace) support

**(Merged)** **#1534** - Added support for negative constant indexing in tuples

**(Merged)** **#1543** - Fixing local increments for the target variable in for loops

**(In Progress)** **#1552** - Fixed printing for arrays as per CPython

**(Merged)** **#1570** - Fixes handling of while loop during pass

**(In Progress)** **#1616** - Added IntrinsicFunction in grammar

### Issues Raised

**(Open)** **#1533** - Negative variable indexing do not work for lists

Apart from this, I have also been involved in discussions with other developers on the github issues page and have also helped contributors test out their pull requests from time to time.

# Overview of the Project

The project **Implementing Symbolic Algorithms as a part of ASR**, is a project where no prior work has been done and has to be worked on from scratch. The idea here is to provide both runtime support (preferably using the SymEngine library) and compile time support (in ASR) to implement Symbolic Algorithms in LPython. The runtime operation could use SymEngine for LLVM, C and C++ backends. For the Python backend, we can simply use SymPy itself.

Runtime support is needed because the symbolic algorithms will need to be executed during the runtime of the program. This is where SymEngine

comes in - it provides the necessary runtime support for performing symbolic computations. It is preferred as it is quite fast and robust.

However, compile time support is also necessary because the symbolic algorithms need to be integrated into the larger program. This is where ASR comes in - it provides a way to represent the symbolic algorithms in a way that can be checked for correctness/semantics and optimised by the compiler.

The algorithms would be implemented in the ASR code, which would make them independent of any specific frontend. Any frontend that uses the ASR code could then make use of the symbolic algorithms without having to reimplement them. This would make it easier to create new frontends that support symbolic computation, and would also make it easier to maintain and improve the symbolic algorithms themselves. LPython, in this case, would not implement the symbolic algorithms themselves, but would instead parse the syntax used by the SymPy library, which already has a rich set of symbolic algorithms, and use the ASR representation of these expressions to perform various operations. This would allow LPython to have powerful symbolic manipulation capabilities without having to reinvent the wheel.

In general we also know that proprietary software like Mathematica is often faster than open source libraries like SymPy. This is because Mathematica is optimised for speed and has been developed over a longer period of time. Introducing symbolic algorithms as a part of ASR gives us a chance to make LPython like Mathematica, but better. This could eventually fix all computational speed and performance based issues with SymPy without hampering the nice syntax which it maintains as of now.

# Proposed Work

In this section , I will be explaining the details of my project . I expect this structure to be considerably improved under the guidance of my mentors and fellow contributors.

## The Goal

The end goal of this project is to enable the LPython compiler to process a patch framed using SymPy's syntax. For eg We would want to compute the **most rapidly varying term leadterm** of a function like the following

```
def mrv_leadterm(e, x):
    """Returns (c0, e0) for e."""
    Omega = SubsSet()
    if not e.has(x):
        return (e, S.Zero)
    if Omega == SubsSet():
        Omega, exps = mrv(e, x)
    if not Omega:
        return exps, S.Zero
    if x in Omega:
        Omega_up = moveup2(Omega, x)
        exps_up = moveup([exps], x)[0]
        Omega = Omega_up
        exps = exps_up
    w = Dummy("w", positive=True)
    f, logw = rewrite(exps, Omega, x, w)
    try:
        lt = f.leadterm(w, logx=logw)
    except (NotImplementedError, PoleError, ValueError):
        n0 = 1
        _series = Order(1)
        incr = S.One
        while _series.is_Order:
            _series = f._eval_nseries(w, n=n0+incr, logx=logw)
            incr *= 2
        series = _series.expand().removeO()
        try:
            lt = series.leadterm(w, logx=logw)
        except (NotImplementedError, PoleError, ValueError):
            lt = f.as_coeff_exponent(w)
            if lt[0].has(w):
                base = f.as_base_exp()[0].as_coeff_exponent(w)
                ex = f.as_base_exp()[1]
                lt = (base[0]**ex, base[1]*ex)
    return (lt[0].subs(log(w), logw), lt[1])
```

The LPython compiler must be able to comprehend the diverse symbolic terminologies utilised in the aforementioned patch. One possible approach is to deconstruct the patch into its individual components and identify areas where additional support may be required. Something like the following

- **Different types of Symbols -** Symbol, Dummy, Wild etc.
- **Symbolic Unary/Binary Operators** - Add, Mul, Pow, Mod etc.
- **Elementary/ Special Functions** - Sqrt, Abs, Modulo, Sin, Cos, LegendreP, BesselJ, Hypergeometric functions etc.
- **Symbolic Algorithms** - Limit, D (differentiate), N (numerical evaluation), Integrate, Sum, LeadingTerm, Series etc.
- **Query Methods/ Assumption checks** - is_algebraic, is_real, is_prime, is_nonnegative, is_Order, has etc
- **Algebraic manipulations** - expand(), rewrite(), as_base_exp(), simplify(), as_coeff_exponent(), subs() etc
- **Python builtins** - type(), is_instance() etc.

An example of a typical computation being answered by the LPython compiler based on the above use cases would be as follows

```
(lf) anutosh491@spbhat68:$ cat examples/expr2.py
from lpython import S
from sympy import Symbol

def main0():
    x: S = Symbol('x')
    y: S = Symbol('y')
    print(sin(x + y).diff(x, 2))

main0()

# Not implemented yet in LPython:
#if __name__ == "__main__":
#    main()
(lf) anutosh491@spbhat68:$ ./src/bin/lpython examples/expr2.py
-sin(x + y)
```

## The Roadmap

In this section , I will try to give a breakdown on how we plan on achieving the end goal of the project . This is mostly based on the issue Design of SymbolicExpression types and the discussions I've had with **Ondřej**, **Thirumalai, Gagandeep** and **Smit** over the past couple of weeks.

I will also be presenting and discussing several topics that require careful consideration from our side under the label of **Design Decisions**. I intend to allocate a significant amount of time to discuss these topics with my mentors, in order to finalise their design during the Application Review phase.

## 1. Introducing IntrinsicFunction in grammar

Intrinsic Functions were introduced in LFortran through #1371 and would also form a fundamental building block of this project. An Intrinsic function is a function which the compiler implements directly when possible, rather than linking to a library-provided implementation of the function for eg strcpy(). Similar to what's been done in LFortran, we would need to add an **expr** node called **IntrinsicFunction** which contains the **arguments**, a **function ID** (for eg "sin"), and an **overload ID** (a particular generic overload of sin based on the signature, 0-10, or so: integer, real, complex, symbolic, whatever combinations we will support for a given special function).

The **IntrinsicFunction expr** node is being introduced in LPython through a Pull Request of mine #1616, which is being assisted by Thirumalai. This would help us represent any math based special function and all symbolic algorithms . For registering any of our functions as an intrinsic function we would need to do the following

To accomplish this task, it will be necessary to generate a namespace for the function in the **intrinsic_function_registry.h** file and produce three functions (**eval***, **create***, **instantiate***) within that namespace. **create*** will be used in the AST→ASR to validate the semantics and establish the IntrinsicFunction node. Subsequently, in the intrinsic_function pass (**instantiate***), we will extract the semantic information contained within the IntrinsicFunction and establish all requisite functions and function calls (for the moment, C is employed to compute the value, such as lfortran_ssin). **eval*** utilises cmath to compute the compile-time value of the function call currently, but for cases that involve symbolic expressions, the SymEngine library will be employed for compile-time values.

## 2. Supporting imports from SymPy library

We should start with adding support for imports from the SymPy library.

```
(lf) anutosh491@spbhat68:$ cat examples/expr2.py
from sympy import Symbol

(lf) anutosh491@spbhat68:$ ./src/bin/lpython --show-asr examples/expr2.py
semantic error: Could not find the module 'sympy'. If an import path is available,
please use the `-I` option to specify it
 --> examples/expr2.py:1:1
  |
1 | from sympy import Symbol
  | ^^^^^^^^^^^^^^^^^^^^^^^
```

An import such as the above doesn't mean much to LPython as we don't expect it to process imports from SymPy. The Symbol variable would anyways be introduced as a **SymbolicSymbo**l intrinsic function as discussed later. The import is just required for CPython to run the program without any errors. Unlike an import statement like **from numpy import array** , where we would end up creating an **ArrayConstant** expr in the **python_ast_to_asr.cpp** file , here we need to check whether the class/function being imported is registered as an **InstrinsicFunction** in **visit_call** using the **is_intrinsic_function(call_name)** function and if that's the case create an Intrinsic Function node in the ASR out of it.

Though anyways we would like Lpython to parse/understand if we are importing sympy as a module or importing some class or function from sympy, hence we would need to add support for the same through the **load_module** and the **import_from_module** functions in the **python_ast_to_asr.cpp file** .

## 3. Introducing SymbolicExpression ttype and related subtypes

In order to accurately represent a wide range of Symbolic expressions, spanning from Universal Scientific Constants such as the Golden Ratio to any Special Function like the Elliptical Integral in the ASR, we need to introduce a novel data type within the grammar. This new data type shall

be referred to as **SymbolicExpression**, with the corresponding annotation designated as "**S**" in Python code to facilitate ease of usage.

```
ttype
    = Integer(int kind, dimension* dims)
    .....
    | SymbolicExpression()
    .....
```

Later we would need to add more subtypes of the "**SymbolicExpression**" type, such as "**Symbol**", "**SAdd**", "**STimes**", "**SInteger**", etc. These subtypes can be used to add more information to the symbolic expressions and perform different operations on them. For example, "Symbol" could be used to represent a single symbol, while "SAdd" could be used to represent an addition of multiple symbols.For subtypes we could create a new node and pass it as an argument: something like, binop, case_stmt, ... nodes in ASR.asdl

```
ttype
    = Integer(int kind, dimension* dims)
    .....
    | SymbolicExpression(subtype type)
    .....

subtype = Symbol | SAdd | STimes ....
```

Later, we would also want to support casting operations from S to some of the subtypes which we would have introduced. The main reason for introducing casting operations is to allow for more specific **type checking** and **error handling**. For example, if you have a function that only accepts Symbol types, you could use a casting operation to ensure that the input SymbolicExpression is actually a Symbol. If it is not, then an error could be raised.

## 4. Introducing Different Symbols, Operators, Casting Functions & Elementary Math Functions as intrinsic functions

Consider the following program. At the outset, it would be necessary to introduce various intrinsic functions such as Symbols, Operators, Casting Functions, and Elementary Math Functions. Although some examples of

these functions have been mentioned in the preceding section, a minimalistic illustration is provided below.

```
def f():
    x: S = Symbol("x")
    y: S = Symbol("y")
    z: S = x + 2*y
    s: S = sin(z)
    print(s)

f()
```

Here are some examples of the additions to consider

- **SymbolicSymbol**(str) → SymbolicExpression
- **SymbolicAdd**(SymbolicExpression, SymbolicExpression) → SymbolicExpression
- **SymbolicInteger**(int) → SymbolicExpression
- **SymbolicReal**(float) → SymbolicExpression
- **SymbolicSin**(SymbolicExpression) → SymbolicExpression

### Design Decision 1

**i)** How should a Symbolic variant of an already introduced math based IntrinsicFunction be defined ?

For eg Functions like Abs, Sin would already be defined as intrinsic functions. Eventually there would be a lot of intrinsic functions whose symbolic variants need to be handled. We could either

1) Introduce **Symbolicfunc** like **SymbolicAbs** and **SymbolicSin** as intrinsic functions which would take in a SymbolicExpression ttype as argument and return a SymbolicExpression argument.
2) We could just end up using the already implemented functions of Abs and Sin and overload SymbolicExpression as another **overload**, next to i32, f32 and f64 etc . A way to approach this would be how we handle the overload for complex types in sin as shown here

Introducing Symbolicfunc like SymbolicAbs and SymbolicSin as intrinsic functions has the advantage of making it clear which functions have symbolic variants, but may become unwieldy if many symbolic variants are introduced. Overloading the existing functions with a symbolic variant

simplifies the codebase, but may become unclear which functions have symbolic variants.

**ii)** Should Binop be introduced as an intrinsic function ?

As discussed in the ASR-Design Document, it may be beneficial to consider introducing **Binop** as an intrinsic function. Currently, we keep it outside. At present, we have the **Add** binary operator in place, and we plan to introduce **SymbolicAdd** as an intrinsic function for handling additions between symbolic expressions. However, as discussed earlier, the overloading approach may simplify the whole implementation. Therefore, if we introduce BinOp as an intrinsic function, we could overload accordingly when we use operands of type SymbolicExpression. Even if one of the operands is of type SymbolicExpression, we could use a casting operation or another intrinsic function like **SymbolicInteger(int)** for the other operand.

A very obvious advantage here is the ensuring consistency between ASR's of expressions like **(2 + 3)** and **(x + 3)** where x is a Symbol, which won't be possible otherwise.

```
// For 2 + 3
[(IntegerBinOp
    (IntegerConstant 2 (Integer 4 []))
    Add
    (IntegerConstant 4 (Integer 4 []))
    (Integer 4 [])
    (IntegerConstant 6 (Integer 4 []))
)]


// For x + 3
[(IntrinsicFunction
       SymbolicAdd
       [(IntrinsicFunction
              .....
              // Operand 1
       )
       (IntrinsicFunction
              .....
              // Operand 2
       )]
       // Overload Id
       (SymbolicExpression)
       (SymbolicExpressionConstant
              "x + 3"
              (SymbolicExpression)
```

```
        )
)]
```

Another thing we should be considering here is we would like to maintain consistency with the types of Binary Operators which SymPy supports . For example SymPy only supports the Add and the Mul type.

```
>>> x, y = symbols('x y')
>>> type(x - y)
<class 'sympy.core.add.Add'>
>>> type(x / y)
<class 'sympy.core.add.Mul'>
```

Hence if we introduce **SymbolicAdd** and **SymbolicMul** as intrinsic functions , we wouldn't really need to add support for **SymbolSub** and **SymbolicDiv** . **Also if we go for the overloading approach , we want to redirect the overloading calls of the Sub and Div operators to the Add and Mul operators respectively whenever we are dealing with the SymbolicExpression type.**

After establishing a successful approach for the aforementioned minimalistic case, we may proceed with introducing a comprehensive range of Elementary functions including **Complexes** (7 in number), **Trigonometric** (12 in number), **Hyperbolic** (12 in number), and **Exponential** (3 in number) functions. Subsequently, we should go on to introduce a set of **Special functions** like the **Bessel** (4 in number), **Beta** (3 in number), **Gamma** (8 in number) and the **Error** (21 in number) functions etc. Numbers have been referred from SymPy's Docs.

## 5. Introducing Query Methods, Assumptions and few python builtins

Before proceeding with the main objective of this project, it would be beneficial to introduce query methods. Consider any algorithm, we might want to implement for eg **leading term of function asinh**

```
def _eval_as_leading_term(self, x, logx=None, cdir=0):  # asinh
    arg = self.args[0]
    x0 = arg.subs(x, 0).cancel()
    if x0.is_zero:
        return arg.as_leading_term(x)
    # Handling branch points
    if x0 in (-I, I, S.ComplexInfinity):
```

```
        return self.rewrite(log)._eval_as_leading_term(x, logx=logx, cdir=cdir)
    # Handling points lying on branch cuts (-I*oo, -I) U (I, I*oo)
    if (1 + x0**2).is_negative:
        ndir = arg.dir(x, cdir if cdir else 1)
        if re(ndir).is_positive:
            if im(x0).is_negative:
                return -self.func(x0) - I*pi
        elif re(ndir).is_negative:
            if im(x0).is_positive:
                return -self.func(x0) + I*pi
        else:
            return self.rewrite(log)._eval_as_leading_term(x, logx=logx, cdir=cdir)
    return self.func(x0)
```

We will in almost all certainty encounter query methods like **is_positive**, **is_negative**, **is_zero**, **is_finite**, **is_infinite** in any algorithm we might want LPython to process. Hence we should add support for minimalistic examples as follows

```
def main():
    x: S = Symbol('x', real=True, positive=True)
    print((x**2 + 5).is_real)
    print((-x).is_positive)

main()

(lf) anutosh491@spbhat68:$ ./src/bin/lpython examples/expr2.py
True
False
```

For supporting query methods we would also need to support assumptions on Symbolic expressions and the Symbol subtype in particular.

## Design Decision 2

**i)** How should assumption attributes be supported for symbolic objects?

At present, SymPy supports a total of 27 assumptions, and linking these assumptions to the Symbol subtype may not necessarily be the most effective approach. To gain greater clarity on the optimal approach to handling these assumptions, it may be useful to examine how SymPy currently manages them. The method employed by SymPy is outlined below.

In SymPy, assumptions are handled using an object-oriented approach. Each symbol and expression in SymPy has an **AssumptionsContext** object associated with it, which stores its assumptions. The AssumptionsContext object has various methods for checking and modifying assumptions, such as is_positive, is_negative, assume, unassume, etc.

When you define a symbol with assumptions in SymPy, like x = Symbol('x', positive=True), SymPy creates a new Symbol object with an associated AssumptionsContext object that has the positive=True assumption set.

When you perform an operation on these symbols, such as (x + y), SymPy creates a new SymbolicExpression object to represent the expression. This object also has its own AssumptionsContext object that inherits the assumptions of its constituent symbols.

We can actually see that the assumptions dictionary of any Symbolic expression won't be populated initially . It's only after calling a query method and conducting a traversal through the assumptions imposed on its constituent arguments, that the assumptions dictionary is populated.

```
>>> x = Symbol('x', positive=True)
>>> y = Symbol('y', positive=True)
>>> (x + y).args
(x, y)
>>> (x + y)._assumptions
{}
>>> (x + y).is_real
True
>>> (x + y)._assumptions
{'real': True, 'commutative': True, 'complex': True, 'extended_real': True,
'imaginary': False, 'finite': True, 'infinite': False, 'hermitian': True}
```

A straightforward approach here is to tie up the assumption attributes with the Symbol subtype and proceed along the similar lines as to what SymPy is doing (If we query **(x**2 + 5).is_real**, we would be traversing the AST and checking the assumptions associated with the **Symbol** objects involved in the expression). But if we don't go for that , we might have to design something to handle assumption attributes on symbolic expressions.

## Implementing python builtin methods

Being an active contributor to SymPy, I would say that there are quite some python builtin functions which are used quite frequently throughout SymPy's codebase, especially in the algorithms that have been noted above. A couple examples of these would be the `**type()**` and the `**isinstance()**` function.

```
print(type("abc"))
semantic error: Function 'type' is not declared and not intrinsic

print(isinstance(1, i32))
semantic error: Function 'isinstance' is not declared and not intrinsic
```

We would like to introduce these as intrinsic functions too and frame **eval***, **create*** and **instantiate*** functions for the same .

## 6. Implementing Symbolic algorithms and Algebraic manipulations

Once we've accomplished the things above, we could move onto implementing symbolic algorithms and some algebraic manipulations. Surfing through SymPy's docs, we can see an extensive list of operations that could be implemented under this topic.

| | |
|---|---|
| **Algebraic manipulation** | simplify, expand, factor, collect, together, apart, cancel, trigonometric simplification, hyperbolic simplification, power simplification, logarithmic simplification, etc. |
| **Calculus** | differentiation, integration (indefinite, definite, improper), limits, series expansion, ODE solvers, PDE solvers, Laplace transforms, Fourier transforms, Z-transforms, etc. |
| **Number theory** | prime numbers, factorization, modular arithmetic, gcd, lcm, Diophantine equations, continued fractions, number partitions, Riemann zeta function, etc. |
| **Statistics** | probability distributions, random variables, moments, characteristic functions, hypothesis testing, confidence intervals, Bayesian inference, etc. |
| **Sets** | union, intersection, complement, symmetric_difference, cartesian_product, direct_product, contains, is_subset, |

| | |
|---|---|
| | is_proper_subset, is_superset etc |
| **Physics** | mechanics, optics, electromagnetism, quantum mechanics, special relativity, general relativity, etc. |

To get started with, we would like to prioritise the modules and related operations we would like to implement . To begin with I would like to cover algorithms from the **Calculus**, **Algebraic Manipulations** module and some operations from the **Physics** modules.

### Understanding Compile time and Runtime operations

```
// compile-time
diff(sin(2*x), x, 2)
series(sin(x), x, 0)
((x + y)**2).expand()

// runtime
n :i32 = read_from_somewhere()
diff(sin(2*x), x, n)
series(sin(x), x, 0, n)
((x + y)**n).expand()
```

The first set of operations can be fully evaluated using symbolic algebra and do not require runtime computation, thus qualifying as **compile-time** operations. To represent them in the ASR, we can use an intrinsic function node and populate its value field with a compile-time value. This approach is similar to how we handle operations like sin(0.5), where we use the cmath library for the value. However, for the operations under consideration, we would use the **SymEngine** library to reduce computation in the backend.
For runtime operations we would be passing a **nullptr** in the value field. Hence we would have to define symbolic math based logic for each of these algorithms and operations in their **eval*** function to decipher which operations would be compile time and which ones would be runtime . For eg

- **diff(f(x), n)** - Here f(x) should be a simple mathematical expression without any external dependencies or user inputs, and n should be a known constant.

- **integrate(f(x), (x, a, b))** - Similar constraints on f(x) and a and b should be known constants

### Design Decision 3

SymPy supports algorithms and other operations as both Class implemented methods and attributes of SymPy objects. For eg

```
>>> limit(sin(x), x, 0)
0
>>> sin(x).limit(x, 0)
0
>>> simplify(sin(x)**2 + cos(x)**2)
1
>>> (sin(x)**2 + cos(x)**2).simplify()
1
```

Using this analogy, we find ourselves in a position where we can introduce algorithms in two ways

1) As **intrinsic functions** itself - For eg **SymbolicLimit**(SymbolicExpression, Symbol, SymbolicReal/ SymbolicInteger etc) ➜ SymbolicExpression
2) As **attributes of SymbolicExpression** ttype as a part of **stmt** in grammar as done for attributes of other types like **List, Dict** etc

```
stmt
    =
    . . .
    | SymbolicExpressionDiff(expr a, expr var, expr times)
    | SymbolicExpressionSubs(expr a, expr expr1, expr expr2)
    . . .
```

What can be done here is, we could add support for one of these methods and redirect the call for the other method to the one which has been implemented . For example if we go forward with the first method and implement SymbolicLimit as an intrinsic function , we could get **sin(x).limit(x, 0)** converted into **limit(sin(x), x, 0)** . This would be done similar to how we approach something like **str.startswith()** . We pass a string as an argument to the function **_Ipython_str_startswith** as shown here . We might have to

introduce a method like **handle_symbolic_expr_attributes**, similar to what we have for strings i.e. **handle_string_attributes** .

## 7. Framing a pass to call into C API of SymEngine

Let's consider the following program

```
from lpython import S
from sympy import Symbol

def f():
    x: S = Symbol("x")
    y: S = Symbol("y")
    z: S = x + y
    print(z)

f()
```

Once we have established a reliable ASR for this program, our next step would be to convert it into explicit calls to the C interface of SymEngine. This could be achieved through code similar to the following, where functions such as "**basic**", "**symbol**", "**add**" and "**str**" are implemented by SymEngine, and "basic" is also implemented within the library.

```
basic x = symbol("x")
basic y = symbol("y")
basic z = add(x, y)
printf(str(z))
free(z)
free(x)
free(y)
```

The pass would be implemented in the **pass/intrinsic_function.cpp** file, where we would want to replace the **IntrinsicFunction** node by a **FunctionCall** (this creates a new function in the scope and uses that symbol in FunctionCall). This FunctionCall would basically be calling into C's API of SymEngine.

The process of initialising variables, assigning values, passing arguments, and printing results should be carried out step by step in the program. Initially, the variable **basic x** is initialised as a **symbol("x")** and assigned a

value. Subsequently, variable y is initialised and assigned a value, after which both x and y are passed as arguments to the add function call. Next, variable z is initialised and assigned a value. The value of z is then printed, and finally all variables are freed. This entire process is executed via **C** function calls in **lfortran_intrinsics.c**

## 8. Working with backend

We could choose any backend here to get started with . The idea here is that we are generating C function calls to SymEngine's C interface in our ASR pass. To actually execute these calls, we need to link against the SymEngine library, which provides the implementation of these functions. To do this, we can use the LLVM backend to generate an object file containing our program's LLVM IR code. This object file will have references to the C functions in the SymEngine library that we need to link against. This linking step can be done similarly to how we link **lfortran_intrinsics.c** in LLVM. We compile **lfortran_intrinsics.c** and generate a binary using **CMake** and then link this to the **LLVM** and use the function from the binary generated previously.

If we link statically, the code from the SymEngine library will be included in our executable binary. If we link dynamically, our executable binary will depend on the SymEngine library, which must be installed on the system where the binary is run.

In addition to linking against the SymEngine library, we also need to ensure that the C++ runtime library that SymEngine depends on is properly linked. This is because SymEngine is implemented in C++, and uses features of the C++ language and standard library. The specifics of how to do this will depend on one's platform and build system, but generally it involves specifying the correct linker flags and library paths.

## 9. Implementing few ASR optimizations

As the final part of the project, it would be worthwhile to focus on implementing few **ASR optimizations**. With a functional frontend interface

and a robust ASR in place, there exists the possibility of implementing numerous symbolic optimizations at the ASR level, using pass-rewrites. These optimizations have the potential to significantly reduce resource consumption in the backend, as well as improve runtime performance.

These go as follows

1) Rewrite **x / 1** or **x * 1 → x**
2) Rewrite **x - 0** or **x + 0 → x**
3) If we plan on introducing **SymbolicSub** (for the "**-**" operator) and **SymbolicDiv** (for the "**/**" operator) , we would need a pass to replace these nodes by **SymbolicAdd** and the **SymbolicMul,** intrinsic function respectively. **This would just involve extracting the value from the right operand followed by inverting the value and the operator.**

```
>>> x - y == x + (-y)
True
>>> x / y == x * (1 / y)
True
```

4) We could also work towards replacing expensive operations with cheaper ones . For eg **x * 2** with **x << 1**
5) Algebraic simplifications like clubbing similar types of Symbolic expressions. For eg **(a + b) - b → a** or **(a + b) + 2a → 3a + b**

# Proposed Timeline and Milestones

This section provides a basic overview regarding how I plan to utilise the Pre-Gsoc, Gsoc contributing and the Post Gsoc period .

- ## Pre Gsoc /Application Review Period -

1. I would try to get a better grasp of the codebase , brush up on some C++ & Python essentials. As I am not too well versed with how the LLVM backend works, I would like to surf through LLVM's documentation for the same.
2. Keep contributing to LPython in general and get my open PR's merged.

- ## May 4 - May 28 (Community Bonding Period) -

1) This is a newly introduced project within LPython that requires the development of its design and implementation details from scratch. The project aims to address a range of new ideas that LPython wishes to incorporate, however, a well-defined implementation plan is yet to be concretized. Regular communication with the mentors will be maintained throughout this duration to formulate a comprehensive document that outlines the implementation plan. This will be uploaded to the GitHub wiki for reference.
2) I would set up a blog post which would then be updated on a weekly basis.
3) I also intend to begin working on the project ahead of schedule, potentially during the final week of the community bonding period, to gain a head start.

- ## Week 1 - 3

1) Coding period starts. Once my mentors and I have finalised on a somewhat concrete approach for dealing with symbolic expressions , I

would like to start with a minimalistic case and get that working first. This would be the following

```
from lpython import S
from sympy import Symbol

def main0():
    x: S = Symbol('x')
    print(x)

main0()
```

2) This would involve supporting imports from SymPy, introducing the SymbolicExpression ttype, introducing SymbolicSymbol as an intrinsic function, making calls to SymEngine's C interface and getting the backend (preferrably LLVM) to work .
3) Once we can get a working output through LPython for printing a Symbol, I would like to get all binary operators compatible with symbolic expressions along with some casting functions (for converting primitive data types into their symbolic forms)

## Week 4- 6

1) I would be introducing a set of essential elementary functions next, including trigonometric, hyperbolic, and exponential functions. To ensure that these functions can handle symbolic and real values from both inside and outside their domains, I would create dedicated test files for each type of function.
2) Introduce a set of special functions like the beta, gamma, bessel and the error functions
3) Write the report for the first evaluation and complete any pending blogs/ progress blogs.

## Week 7 - 9

1) At this point, we would have introduced Symbols, Unary/Binary Operators, and various classes of Mathematical Functions. Therefore, it would be appropriate to introduce specific subtypes for these entities to enhance the type safety and make the code more organized.These

subtypes would inherit from the SymbolicExpression class and provide specialised functionality and additional safety checks. This would be accompanied by casting operations from the SymbolicExpression parent type to all subtypes

2) Introduce a few essential assumption attributes for symbols and Symbolic expressions.
3) Once we've accomplished the above step, I would start with introducing a few important query methods like is_positive, is_negative, is_real, is_finite, is_infinite etc. It is crucial to perform thorough testing at this stage, as there are many areas where we could find potential breaches through assumption attributes.
4) If time permits, I would also like to start with introducing a few Symbolic Algorithms like differentiation , series expansions and limits here .

- ## Week 10 - 12

1) Introduce all (around 10 in number) Symbolic Algorithms from the Calculus module.
2) Introduce a few algebraic and other misc operations like subs(), expand(), simplify() etc.
3) If time permits, I would like to get a few basic ASR optimizations down through some ASR passes .
4) I would start framing a document for the final GSoC submission and complete any pending blogs from past weeks.

- ## Post GSoC Period - I would be doing the following things once my tenure of contribution is over .

1) Complete any unfinished work on Pull Requests associated with the proposed ideas that may have been delayed due to obstacles encountered during the process.
2) Create an issue report that provides an overview of the current project status, including completed tasks, and outlines the roadmap for future development. This issue will serve as a reference for potential contributors interested in working on the project.

3) Upon the successful completion of my GSoC project, **I would also like to join LPython/LFortran as a full-time compiler developer for a minimum of one year**. In the event of project completion, I would like to undertake additional projects that align with my interests, such as compiling benchmarking code in Python for LPython.

- ## Time Commitment -

I can positively dedicate 30 hours to my project on a weekly basis. On a need basis I wouldn't mind scaling the input hours up. I will inform my mentor in advance if any personal or miscellaneous work causes me to miss a deadline or weekly meeting. I will also provide a timeframe during which I will dedicate extra time to complete the work.

I plan to finish the Gsoc project with almost a week of buffer . This will give me more time to address anything left or give more importance to any issue that demands more time and debugging. This extra time will also enable me to thoroughly document my work and test the newly added features in their entirety.

I would also like to take this opportunity to express my gratitude to those who have provided me with tremendous support and guidance during the past few months. In particular, I would like to thank Ondřej Čertík (@certik), Thirumalai Shaktivel (@Thirumalai-Shaktiv), Smit Lunagariya (@Smit-create) and Gagandeep Singh (@czgdp1807), and for their invaluable contributions in reviewing my PRs and providing assistance whenever I encountered difficulties.

# References and Links


- ASR Design
- Intrinsic Functions Design
- LFortran Design
- SymPy Documentation
- SymEngine's C wrapper
- Design of SymbolicExpression types