

Developing a computer algebra system

G14PJS

Mathematics 3rd Year Project

Spring 2019/20

School of Mathematical Sciences

University of Nottingham

Dominic Broadbent

Supervisor: Dr. Edward Hall

Project code: EH P1

Assessment type: Review

I have read and understood the School and University guidelines on plagiarism. I confirm that this work is my own, apart from the acknowledged references.

Abstract

A computer algebra system (CAS) is software that can manipulate mathematical expressions symbolically. One such system was developed in Python 3.7.4 without the use of extensive external modules. We call it CASTle. The development process involved implementing Edsger W. Dijkstra's Shunting-Yard algorithm and various other purpose-built algorithms. These are used to compute the answer to numerical expressions with many supported functions and to simplify compound linear expressions in multiple variables. Within these compound linear expressions coefficients can be composite (e.g. $(4*(2+3))$) and non-linear “variable” expressions can be entered in an unoptimised form (e.g. y^2xxwy^{-1}).



Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 5 |
| 2 | Data Structures and Types | 7 |
| 2.1 | Queues | 7 |
| 2.2 | Stacks | 9 |
| 2.3 | Data Types | 11 |
| 2.4 | Lists and Dictionaries | 14 |
| 3 | Operators and Operands | 15 |
| 3.1 | Definitions and Examples | 15 |
| 3.2 | Order of Operations | 16 |
| 4 | Parsing | 20 |
| 4.1 | Issues in Parsing | 21 |
| 4.2 | Notation | 23 |
| 4.3 | The Shunting-Yard Algorithm | 24 |
| 4.4 | Postfix to Infix Algorithm | 30 |
| 5 | Evaluating Numerical Expressions | 31 |
| 5.1 | Computation Algorithm | 31 |
| 5.2 | Capabilities | 32 |
| 5.3 | Limitations | 35 |
| 6 | Simplifying Compound Linear Expressions | 37 |
| 6.1 | Simplification Algorithms | 37 |
| 6.2 | Capabilities | 47 |
| 6.3 | Limitations | 49 |
| 7 | Commercial Computer Algebra Systems | 49 |
| 7.1 | Applications of a CAS | 50 |
| 8 | Conclusions | 52 |

| | |
|----------------------|-----------|
| 9 Raw Code | 53 |
| 10 References | 55 |

1 Introduction

A computer algebra system (CAS) is a piece of software that mimics the paper and pencil symbolic manipulation of mathematical expressions. This distinguishes them from traditional calculators that deal with equations numerically. In general, the chief objective of a CAS is to replace the need for hand computation of arduous or tedious algebraic tasks. However, the scope of a CAS within mathematics is unlimited with many programs supporting number theory, group theory, probability and more.

The very first CAS, named ‘Schoonschip’ after the Dutch expression “schoon schip maken”; ‘to make a clean sweep’, was developed in 1963 by Nobel laureate Martinus J.G. Veltman. He conceived the project in order to compute the quadrupole moment of the W boson, which required “a monstrous expression involving in the order of 50,000 terms in intermediate stages” [1]. More systems spawned throughout the 1960s to meet the needs of theoretical physicists and AI researchers. By 1987 they had made their way to hand-held calculators [2] and in the present day there are many general-purpose systems both free and commercially available; perhaps the most widely recognised being Wolfram Mathematica.

Computer algebra systems tend to differ in scope and focus but all include the ability to simplify expressions. Indeed, the aim of this project was to develop a CAS from the ground up capable of simplifying expressions. This was a very general goal, and was later refined to the simplification of compound linear expressions in multiple variables. Compound linear expressions are a class of expressions we define in Section 6. In order to accomplish this goal, two data structures and various algorithms, largely original, were implemented in the programming language Python.

The development of CASTle began with a focus on numerical computation; this was in anticipation of calculating composite coefficients of variables. These could be very large and include a range of functions. The first hurdle to overcome was to parse the numerical expression, that is to process it into its constituent operators and operands (See Section

4). Once this parser had been devised, the next step was to implement the Shunting-Yard algorithm. This converts the expression from *infix* notation, the notation we write mathematics in, to *postfix* notation, a much more convenient form for computation (See Sections 4.2 and 4.3). The Shunting-Yard algorithm requires two data structures; the stack and the queue which we implemented in Python (See Section 2). It also relies on the enforcement of order of operation and associativity rules (See Section 3). The final step was to evaluate this postfix expression and output an answer (See Section 5.1). The full numerical capabilities and limitations can be seen in Section 5.2 and Section 5.3 respectively.

Once the numerical functionality was complete we moved onto simplification. This required the development of a new parser, which we call a *standardiser*, that converts compound linear expressions to a standard form (See Section 6.1.1). Once an expression is in standard form, variables are simplified and the composite coefficients are computed using the numerical functionality of CASTle (See Sections 6.1.2 and 6.1.3). Finally, alike terms are collected and the simplified compound linear expression is outputted (See Section 6.1.4). Full simplification capabilities and limitations can be seen in Section 6.2 and Section 6.3 respectively.

We also discuss the applications of fully developed computer algebra systems in Section 7. Concluding comments can be seen in Section 8 and the interested reader can find the structure of the raw code in Section 9.

2 Data Structures and Types

Throughout the back-end code of the CAS various basic data structures have been implemented for use in algorithms or custom Python functions.

2.1 Queues

A queue in computer science is a data structure that imitates a real life line or queue. When we enter the queue we do so at the back, then as people at the front of the queue leave, one moves forward. The next person who enters the queue will do so behind us, this process repeats until we are now at the front of the queue and are served. This is a First-In-First-Out (FIFO) system i.e. the person that is being served is the person who has spent the longest in the queue at that time [3].

Definition 2.1. In computer science, a *queue* is an abstract data structure in which items are kept in order. We have the principal operations; *enqueue*, the addition of entities to the rear, and *dequeue*, the removal of entities from the front. This makes the queue a First-In-First-Out (FIFO) data structure [4].

The queue has been implemented as a *class* in Python. A class provides a means of building objects that are comprised of data and associated functions, called *methods*. Methods associated with the queue class include the following:

- *Enqueue* - Add an item to the back of the queue
- *Dequeue* - Return the item at the front of the queue *and* remove it
- *Peek* - Return the item at the front of the queue but *do not* remove it
- *Check Empty* - Check if the queue is empty
- *Size* - Return the length of the queue
- *View* - Return the queue in the form of a Python list

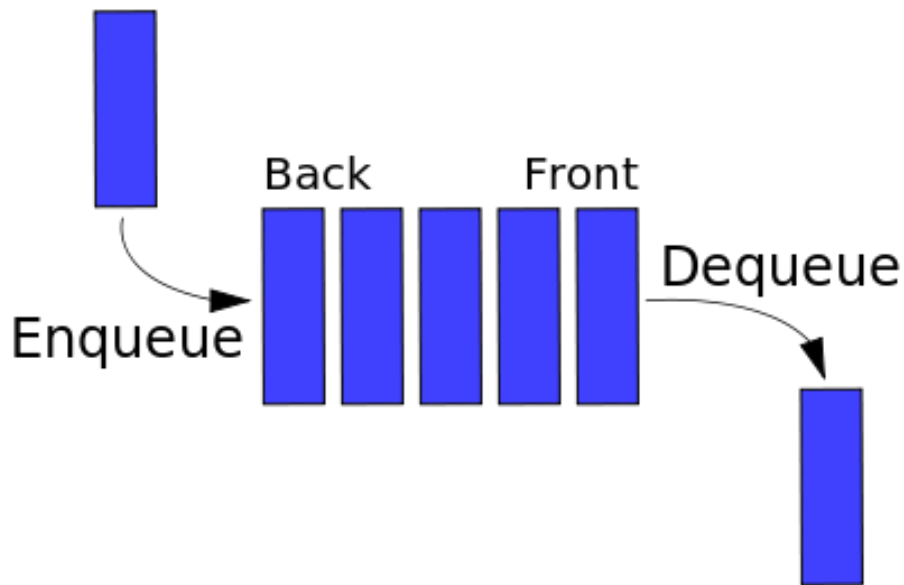


Figure 1: Illustration of queue data structure [5]

The Python class implementation of a queue is as follows:

```
class Queue:
    """A class that represents the first-in-first-out (FIFO) queue data
    structure and associated functions
    """
    def __init__(self):
        """Initialises the queue class"""
        self.items = []

    def enqueue(self, item):
        """Adds an item to the back of the queue"""
        self.items.append(item)

    def dequeue(self):
        """Returns the item at the front of the queue and removes it"""
        return self.items.pop(0)

    def peek(self):
        """Returns the item at the front of the queue but does not
        remove it"""
        return self.items[0]
```



```

def checkempty(self):
    """Checks if the queue is empty and returns True if it is"""
    if self.items == []:
        return True
    else:
        return False

def size(self):
    """Returns the length of the queue"""
    return len(self.items)

def view(self):
    """Returns the entire queue as a list with the first element
        representing the front of the queue
    """
    return(self.items)

```

2.2 Stacks

In computer science, a stack is an abstract data structure that acts similarly to a pile of books. That is, we can only add or remove an item from the top of the stack (or risk it tumbling). This is a Last-in-First-Out (LIFO) system i.e. the last item added to the stack is the only item that can be removed at that time [3].

Definition 2.2. In computer science, a *stack* is an abstract data structure in which items are kept in order. We have the principal operations; *push*, the addition of entities to the top, and *pop*, the removal of entities from the top. This makes the queue a Last-In-First-Out (LIFO) data structure [6].

The stack has been implemented as a class in Python with the following methods:

- *Push* - Push an item to the top of the stack
- *Pop* - Remove an item from the top of the stack *and* return it for future use

- *Peek* - Return the item at the top of the stack but *do not* remove it
- *Check Empty* - Check if the stack is empty
- *Size* - Return the length of the stack
- *View* - Return the stack in the form of a Python list

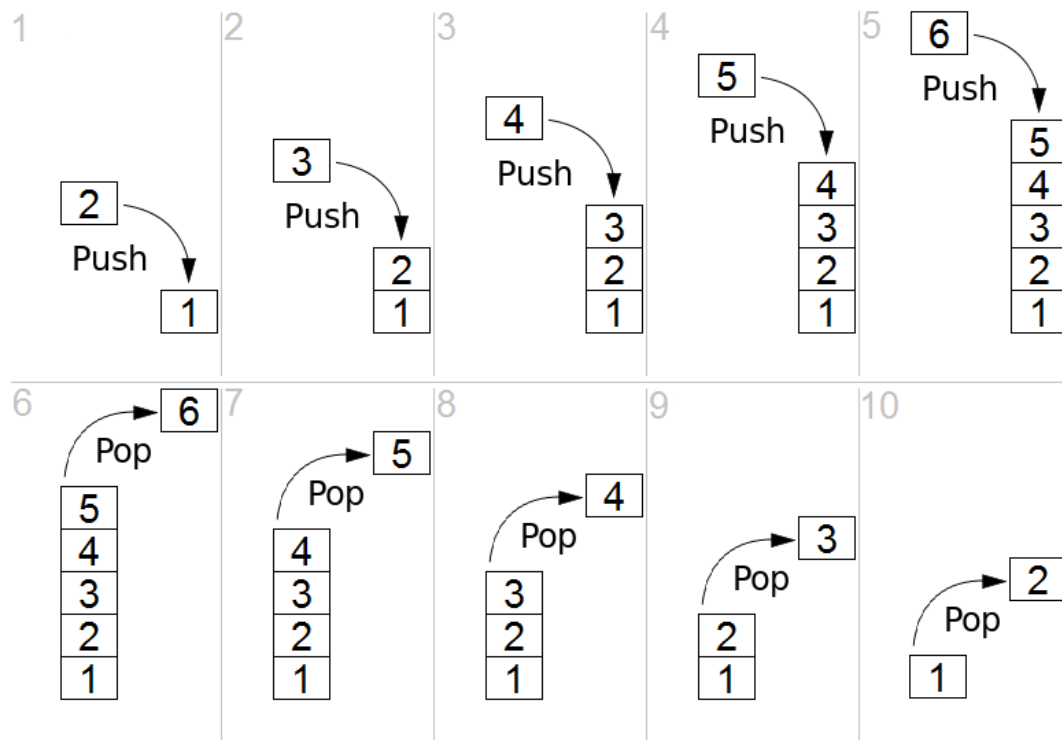


Figure 2: Illustration of stack data structure [7]

The Python class implementation of a stack is as follows:

```
class Stack:
    """A class that represents the last-in-first-out (LIFO) stack data
    structure and associated functions
    """
    def __init__(self):
        """Initialises the stack class"""
        self.items = []
```

```

def push(self, item):
    """Appends an item to the top of the stack"""
    self.items.append(item)

def pop(self):
    """Returns the at item at the top of the stack and removes it"""
    return self.items.pop()

def peek(self):
    """Returns the item at the top of the stack but does not remove
        it"""
    return self.items[-1]

def checkempty(self):
    """Checks if the stack is empty and returns True if it is"""
    if self.items == []:
        return True
    else:
        return False

def size(self):
    """Returns the length of the stack"""
    return len(self.items)

def view(self):
    """Returns the entire stack as a list with the last element
        representing the top of the stack"""
    return(self.items)

```

2.3 Data Types

In computer science, data is categorised into various types, these data types tell the programming language how the data is intended to be used.

2.3.1 Strings

Definition 2.3. In computer science, a *string* is a series of characters [8].

In Python, anything held inside quotes is considered a string [9]. Examples would be “I love maths” or “ $2 + 2 = 4$ ”. Even though these examples are composed of objects themselves, letters in the former and operands/operators in the latter, they are treated as **one** string by Python. In the CAS the mathematical expressions entered by the user are inputted as a string. This string needs to first be *parsed* by the CAS in order to be computed or simplified (See Section 4).

2.3.2 Booleans

Definition 2.4. In computer science, a *boolean* is a data type that can take only two values; true or false [10].

Booleans are generally used to test for a condition and we use them frequently throughout the back-end code of the CAS in the implementation of algorithms.

2.3.3 Integers and Floats

Definition 2.5. In computer science, a *float*, short for floating-point number, is a formulaic representation of a real number. They take the form ‘significand \times base^{-exponent}’, where the significand, base and exponent are integers.

Definition 2.6. In computer science, an *integer* is a positive or negative whole number with no decimal point [12].

Python converts decimals to floats. Examples of a float in decimal form include ‘3.1416’ or ‘0.05’. Integers are as usual with the subtlety that Python treats ‘1.0’ as a float despite its fractional component having no magnitude. The CAS was built to be capable of handling floating-point arithmetic in user input, evaluation and in the use of irrational constants such as π or e .

Note that while the CAS allows for the use of π and e , in actuality it is only using an approximation to 15 decimal places of the actual values. This is because floats are just

approximations of real numbers which cannot be stored exactly in binary. As a result, operations involving them can have strange outcomes. We see an example of such an outcome in Figure 3.

```
Enter your expression:
0.1+0.2

The expression has been interpreted as (0.1+0.2)

0.1+0.2 = 0.30000000000000004
```

Figure 3: Floating-point error in expression calculation

In most cases involving numerical computation small errors and approximations such as these will not be important. They will tend to be hidden within a longer calculation and have very little effect on the final answer. Nevertheless the CAS will spot most float errors, raise the error to the user's attention and then fix them. An example of this in practice can be seen in Figure 4.

```
Enter your expression:
0.1 + 0.2

The expression has been interpreted as (0.1+0.2)

0.1+0.2 = 0.30000000000000004
FLOAT ERROR: 0.30000000000000004 has been rounded to 0.3
0.3
```

Figure 4: Floating-point error fixed

Floating-point errors are most pressing in numerical computation when we are working with trigonometric functions. This is due to the approximation we use for π . The CAS also spots and fixes these errors, alerting them to the users attention in the process. See this below in Figure 5.

```

Enter your expression:
sin(pi)

The expression has been interpreted as sin(pi)

sin(3.141592653589793) = 1.2246467991473532e-16
FLOAT ERROR: 1.2246467991473532e-16 has been rounded to 0

```

Figure 5: Floating-point error in trigonometric functions

In simplification, float errors represent much more of a visible problem. This is because, unlike in numerical computation, float-errors do not disappear into the background of a calculation. We see this in Figure 6.

```

Enter your expression:
0.1x^2 + 0.2x^2

x^2 term: 0.1x^2 + 0.2x^2 = 0.30000000000000004x^2

```

Figure 6: Floating-point error in simplification

This is clearly unacceptable, we avert this issue by rounding the coefficients to a high number of significant figures. See this in Figure 7. This fixes the problem at the expense of accuracy in a case where the user wishes to work with very small or very large coefficients.

```

Enter your expression:
0.1x^2 + 0.2x^2

x^2 term: 0.1x^2 + 0.2x^2 = 0.30000000000000004x^2
FLOAT ERROR: 0.30000000000000004 has been rounded to 0.3
0.3x^2

```

Figure 7: Floating-point error fixed

Not all float errors are easy to fix; it is generally a case of management, that is balancing when to round versus what loss of accuracy is acceptable (See Section 5.3).

2.4 Lists and Dictionaries

Strings, integers, floats and booleans are all examples of data types that can be held within a Python *list* or *dictionary*. These data structures are a type of container, ordered

and unordered respectively. We use them frequently throughout the back-end code of the CAS in custom functions and implementations of algorithms.

A list, as we would expect, represents a list of elements that are referred to as *tokens*. These lists, denoted by square brackets as such; [token0 , token1], are ordered and indexed, with the first token having index 0. A queue and a stack are specific types of list with special methods associated.

A dictionary works slightly differently to a list; they are an unordered set of pairs of *keys* and *values* with the condition that the keys must be unique. They are denoted by curly brackets as such; {key0:value0, key1:value1}. These are excellent for representing entities as we can set attributes as keys, such as a person's name, and then associate values to them. Within the CAS they are used in the simplification of compound linear expressions (See Section 6).

3 Operators and Operands

The CAS works with numerical and variable expressions. These are composed of numerical or variable terms called *operands* which are linked via *operators*. In order to compute answers and simplify expressions the CAS needs to be able to interpret these expressions according to rules that are consistent and clear.

3.1 Definitions and Examples

Definition 3.1. An operator is a function that acts upon one or more inputs, sending them to a well-defined output. Operators with one input are called *unary* operators while operators with two inputs are called *binary* operators [13].

Definition 3.2. An operand is the object or quantity that is being operated on.

Common examples of binary operators are addition and multiplication, represented by '+' and '×' respectively. Note that the CAS represents multiplication by the '*' symbol and so all expressions involving multiplication will use this henceforth for consistency.

Unary operators come in different forms, we use the following in the CAS; prefix notation (e.g. -2 or $-x$) and functional notation (e.g. $\cos(2)$ or $\text{fac}(4)$).

An operand can be more complex than just a single number or algebraic term. For example, an expression made up of operands and operators can itself be an operand. This can be seen in the numerical expression ' $4 * (2 + 2)$ '; the first operand for the multiplication operator is ' 4 ' and ' $(2 + 2)$ ' the second. Here the operand ' $(2 + 2)$ ' is itself a numerical expression composed of operands and an operator.

3.2 Order of Operations

When evaluating a numerical expression, the order in which operations take place is important to avoid confusion. There are a collection of rules that dictate which operations should take precedence over others. The order is defined as follows, from highest precedence to lowest: (Zazkis and Rouleau [14])

- Exponentiation
- Multiplication and Division
- Addition and Subtraction

Parentheses can be used to suggest alterations to the above order i.e. ' $(2 + 2) * 4$ ' forces the addition to occur first despite multiplication appearing higher in the order of operations. There are subtleties to the order of operations that require further rules, such as how we handle operator associativity and when to implement unary operators.

3.2.1 Operator Associativity

The associativity of an operator dictates how operators of the same precedence are implemented in expressions without parentheses [15].

Definition 3.3. An operator, $\#$, is *left-associative* if ' $x \# y \# z$ ' evaluates to ' $(x \# y) \# z$ ' and *right-associative* if ' $x \# y \# z$ ' evaluates to ' $x \# (y \# z)$ '.

For example, in the case of $'1 - 2 + 3'$, with our above order of operations it is not clear if we have $'(1 - 2) + 3 = 2'$ or $'1 - (2 + 3) = -4'$. Similarly, the expression $'8 \div 4 \div 2'$ could return $'(8 \div 4) \div 2 = 1'$ or $'8 \div (4 \div 2) = 4'$. Exponents are also affected; we could interpret the user input $'4 \wedge 3 \wedge 2'?$ as $'(4^3)^2'$ or as $'4^{(3^2)}'$. Clearly there is ambiguity in expressions where operators of the same precedence follow after each other.

In these cases we have to choose the associativity of our operators to best represent natural usage. Hence, the addition, subtraction, multiplication and division operators are taken to be left-associative. There is no widely agreed upon standard for exponentiation in programming languages, however Python uses right-associativity [16]. Therefore exponentiation is taken to be right-associative. The above three examples are thus resolved by the CAS in the following fashion; see Figure 8.

```

Enter your expression:
1 - 2 + 3

The expression has been interpreted as ((1-2)+3)

1-2 = -1
Updated expression: (-1+3)
-1+3 = 2

```

(a) Subtraction/addition left-associativity

```

Enter your expression:
8/4/2

The expression has been interpreted as ((8/4)/2)

8/4 = 2.0
Updated expression: (2.0/2)
2.0/2 = 1.0

```

(b) Division left-associativity

```

Enter your expression:
4^3^2

The expression has been interpreted as (4^(3^2))

3^2 = 9
Updated expression: (4^9)
4^9 = 262144

```

(c) Exponentiation right-associativity

Figure 8: Operator associativity in the CAS

The properties of operators are assigned in the CAS as such:

```

OPS = {
  '^': Op(precedence=4, associativity= RIGHT),
  '*': Op(precedence=3, associativity= LEFT),
  '/': Op(precedence=3, associativity= LEFT),
  '+': Op(precedence=2, associativity= LEFT),
  '-': Op(precedence=2, associativity= LEFT)
}

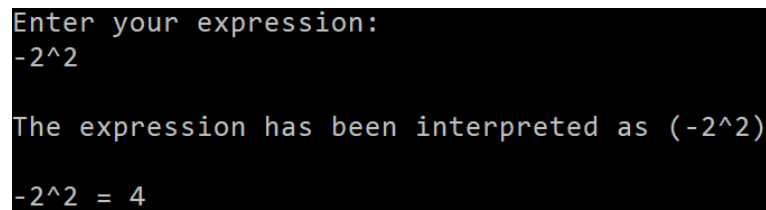
```

Note that these values can be changed at will, if so desired this allows us to work in a mathematical space where addition has higher precedence than multiplication, for example. Also note that associativity is redundant for unary operators such as negation,

however in interaction with other operators their own issues arise.

3.2.2 Unary Operator Associativity

Similar to the issues with associativity above, how the unary negative operator and exponentiation interact can differ by interpretation. For example, -2^2 can be implemented as $(-2)^2$ or $-(2^2)$. Once again there is no widely agreed upon standard here among programming languages [16] and so the former interpretation was chosen to avoid the user having to write brackets around the -2 term to get the more intuitive result of 4 . We see this below in Figure 9.



```
Enter your expression:
-2^2

The expression has been interpreted as (-2^2)
-2^2 = 4
```

Figure 9: Interaction of unary negative and exponentiation

3.2.3 Alternative Unary Negative Operator

Lots of problems arise due to using the same operator to represent the unary negative and binary subtraction. Python does not have functionality to easily distinguish between the two operators and so within the CAS the operator \sim can also be used to represent negation, this has been implemented in a manner such that it has the highest precedence among the traditional operators.

```
OPS = {
    '~': Op(precedence=5, associativity= LEFT),
    '^': Op(precedence=4, associativity= RIGHT),
    '*': Op(precedence=3, associativity= LEFT),
    '/': Op(precedence=3, associativity= LEFT),
    '+': Op(precedence=2, associativity= LEFT),
    '-': Op(precedence=2, associativity= LEFT)
```

}

This solution is uncommon in other computer algebra systems as they have more elaborate parsers (See Section 4). In most cases involving negating simple operands it will be unnecessary to use this operator ‘~’ over the traditional ‘-’. However, when working with operands more complicated than a single variable or number (e.g. $-(2 + 3)$), the CAS requires the use of the alternate unary negative operator to avoid errors. Thus expressions such as ‘ $-(2+2)^2$ ’ must be written as ‘ $\sim(2 + 2)^2$ ’. This can be seen in Figure 10 below.

```
Enter your expression:
~(2+2)^2

The expression has been interpreted as (-(2+2)^2)

2+2 = 4
Updated expression: (-4^2)
-4^2 = 16
```

(a) Example of alternate unary negative operator in use

```
Enter your expression:
-(2+2)^2
ERROR:Use the negation operator '~' when negating complicated operands
```

(b) CAS flags an error for use of incorrect operator

Figure 10: Explicit vs implicit unary negative

The aforementioned errors occur due to the use of an algorithm (see section 4.3) that does not treat unary negation as different to binary subtraction. This results in operations occurring in a nonsensical order. The problem is averted for negating simple operands like in ‘-2’ or ‘-x’ as the unary negative is assigned to the operand *before* the algorithm is implemented (see section 4). For more complicated operands this is not feasible.

4 Parsing

Parsing is the process of interpreting user-inputted strings, mathematical expressions in our case. The CAS asks for the expression the user wishes to compute or simplify as a string. In this form it is one object; not a combination of operands and operators. The

parser takes this string, breaks it into its constituent tokens and outputs a list representing the expression. An example of the parser in action can be seen in Figure 11.

```
Enter your expression:
(2+2)-12*fac(4)
['(', '2', '+', '2', ')', '-', '12', '*', 'fac', '(', '4', ')']
```

Figure 11: Parsed expression as a list of tokens

The parser has been built such that it recognises decimals and negative numbers. It also collects alphabetic characters together to build functions such as ‘sin’ and irrational constants like ‘pi’ or ‘e’. We can see that the parser does this correctly in Figure 12.

```
Enter your expression:
-2.2*sin(pi)
['-2.2', '*', 'sin', '(', 'pi', ')']
```

Figure 12: Parser capabilities

4.1 Issues in Parsing

There are various issues that arise in the development of a parser capable of analysing mathematical expressions. A big problem is user-error; the program must be capable of spotting issues with an inputted mathematical expression and then flagging them clearly to the user. A frequent example of user-error can be found in mismatched parentheses; the program cannot proceed if parentheses are not correctly handled due to the ambiguity it causes in evaluating an expression. Hence the issue is flagged to the user so they can re-type the expression more carefully. See Figure 13.

```
Enter your expression:
12/(2+4
ERROR: mismatched parenthesis
```

Figure 13: Parser flagging parentheses error

Another case of user-error is the arbitrary use of spaces within an expression. The parser must be able to interpret $(2+2)*7$ and $(2 + 2) * 7$ in the same manner. That is, with the irrelevant spaces removed so the end result is just a list of operands and operators that the rest of the program can then use. Using the above example, in Figure 14 we can see that the parser deals with spaces correctly:

```
Enter your expression:
(2+2)*7
['(', '2', '+', '2', ')', '*', '7']
```

(a) An expression with no spaces

```
Enter your expression:
( 2 + 2 ) * 7
['(', '2', '+', '2', ')', '*', '7']
```

(b) An expression with spaces

Figure 14: An example of how the parser deals with arbitrary spaces

The most pressing non-user error is how we treat the operator '-', i.e. as unary or binary. For example, when performing the subtraction $2 - 4$ the parser must be capable of *first* assigning the unary operator to the '4' string and *then* implementing the binary operator to get the correct interpretation of the expression; $2 - (-4)$. Similarly this is a problem when working with negative functions such as in the case of $2 - \sin(4)$. We can see the parser deals with these cases correctly in Figure 15.¹

```
Enter your expression:
2--4
['2', '-', '-4']
```

(a) Subtracting a negative number

```
Enter your expression:
2--sin(4)
['2', '-', '-sin', '(', '4', ')']
```

(b) Subtracting a negative function

Figure 15: An example of how the parser deals with unary operators

¹The interested reader should see the file *cas-parser.py* for the full code that accomplishes this

Note that operands that themselves take the form of an expression such as in ‘ $-(2 + 2)$ ’ are not automatically negated by the parser like ‘ -2 ’ or ‘ $-\sin$ ’ are. When using more complicated operands such as these we require the use of the additional unary negation operator ‘ \sim ’ from section 3.2.3. This is done to avoid errors that occur due to limitations in the Shunting-Yard algorithm (see section 4.3).

4.2 Notation

We interpret and write mathematics in a notation called *infix*. For example, ‘ $12 * (2 + 3)$ ’ is in *infix* form as the operators are *inside* the operands. This form is difficult to work with in a computer program due to issues with order of operations. Simply looping through the parsed expression from the beginning and operating as we go will not work. For the example above ‘ $2 + 3$ ’ must be calculated *before* multiplying it by 12. This is intuitive to us in this form as we are familiar with traditional order of operations but must be explicitly coded into the program.

One way of implementing correct order of operations is to convert the infix expression to *postfix*, or equivalently, *Reverse Polish* notation, named after the nationality of its inventor, the logician Jan Łukasiewicz (Hamblin [17]). It is called *postfix* as the operators follow after the operands. For example, the postfix form of ‘ $2 + 3$ ’ is ‘ $2\ 3\ +$ ’.

An expression in postfix notation is interpreted by scanning, from left to right, until one hits an operator. If the operator is binary then the two operands immediately preceding the operator are operated on. If the operator is unary then only the operand immediately preceding the operator is operated on. The result of this operation is then inserted back into the postfix expression in the place of the previous operand(s) and operator. This procedure is then repeated until the expression has been fully evaluated.

Example:

The postfix form of ‘ $12 * (2 + 3)$ ’ is ‘ $12\ 2\ 3\ +\ *$ ’. We scan the expression until we reach the operator $+$, then we operate on the two operands immediately preceding; 3 and 2.

This gives $3 + 2 = 5$ and we place this back into the postfix expression as $12\ 5\ *$. Once again we scan the expression from where we left off until we reach the operator $*$, then take the preceding operands 5 and 12 to perform the multiplication $12 * 5 = 60$. Thus we have evaluated the postfix expression to get 60, as expected.

One big advantage postfix notation has over infix notation is the lack of parentheses; they are not required as the order in which we make operations is obvious from the order of the expression. Another is that the expression can be computed only working from left to right, this makes postfix notation perfect for looping over in the back-end code of the CAS.

Prefix, or equivalently, *Polish* notation, once again named after the nationality of its inventor, Jan Łukasiewicz, (Hamblin [17]) is another way of writing mathematical expressions that solves the order of operations issue. Instead of operators appearing *after* their operands, as in postfix notation, they appear *before* their operands. Using the same example as above; $12 * (2 + 3)$, in prefix form we have $* + 3\ 2\ 12$. This is evaluated in a similar fashion to postfix notation. It is clear that *reversing* this prefix expression shifts it to postfix notation, hence their respective names, *Polish* and *Reverse Polish*.

4.3 The Shunting-Yard Algorithm

Converting from infix to postfix notation is performed via an algorithm first developed by Edsger W. Dijkstra in 1960 (Dijkstra [18]); the Shunting-Yard Algorithm, named as such due to its operation resembling a rail-road shunting yard. The CAS implementation of the algorithm takes a parsed infix expression and returns a parsed postfix expression. The output is built as a queue while operators waiting to be added to the queue are held in a stack.

A simple conversion from infix to postfix of the expression $5 * 6$ goes as follows:

- Input parsed expression $[5, *, 6]$
- Push 5 to the output queue \implies Output queue $= [5]$

- Push '*' to the operator stack \implies Operator stack = ['*']
- Push '6' to the output queue \implies Output queue = ['5', '6']
- The expression has been fully read so pop '*' from the operator stack to the output queue \implies Output queue = ['5', '6', '*']
- Return the postfix expression ['5', '6', '*']

The above is a simple example, however it does show a few rules already; namely that all operands are immediately pushed to the output queue upon being read. Also, after the expression has been fully read we pop all the remaining operators from the operator stack to the output queue.

A more complicated example can be found in converting 'A + B * C - D' to postfix notation:

- Input parsed expression ['A', '+', 'B', '*', 'C', '-', 'D']
- Push 'A' to the output queue \implies Output queue = ['A']
- Push '+' to the operator stack \implies Operator stack = ['+']
- Push 'B' to the output queue \implies Output queue = ['A', 'B']
- Push '*' to the operator stack \implies Operator stack = ['+', '*']
- Push 'C' to the output queue \implies Output queue = ['A', 'B', 'C']
- The top of the operator stack is '*' and we are considering the token '-'. Now, multiplication has higher precedence than subtraction, hence we pop '*' from the operator stack to the output queue \implies Output queue = ['A', 'B', 'C', '*'] and Operator stack = ['+']
- The top of the operator stack is now '+' and we are still considering the token '-'. Subtraction is left associative and addition has equal precedence to it, hence we pop

'+' from the operator stack to the output queue \implies Output queue = ['A', 'B', 'C', '*', '+'] and Operator stack = []

- Push '-' to the operator stack \implies Operator stack = ['-']
- Push 'D' to the output queue \implies Output queue = ['A', 'B', 'C', '*', '+', 'D']
- The expression has been fully read so pop '-' from the operator stack to the output queue \implies Output queue = ['A', 'B', 'C', '*', '+', 'D', '-']
- Return the postfix expression ['A', 'B', 'C', '*', '+', 'D', '-']

This example shows how the order of operations and operator associativity are handled by the Shunting-Yard algorithm. A graphical illustration of this procedure can be seen below:

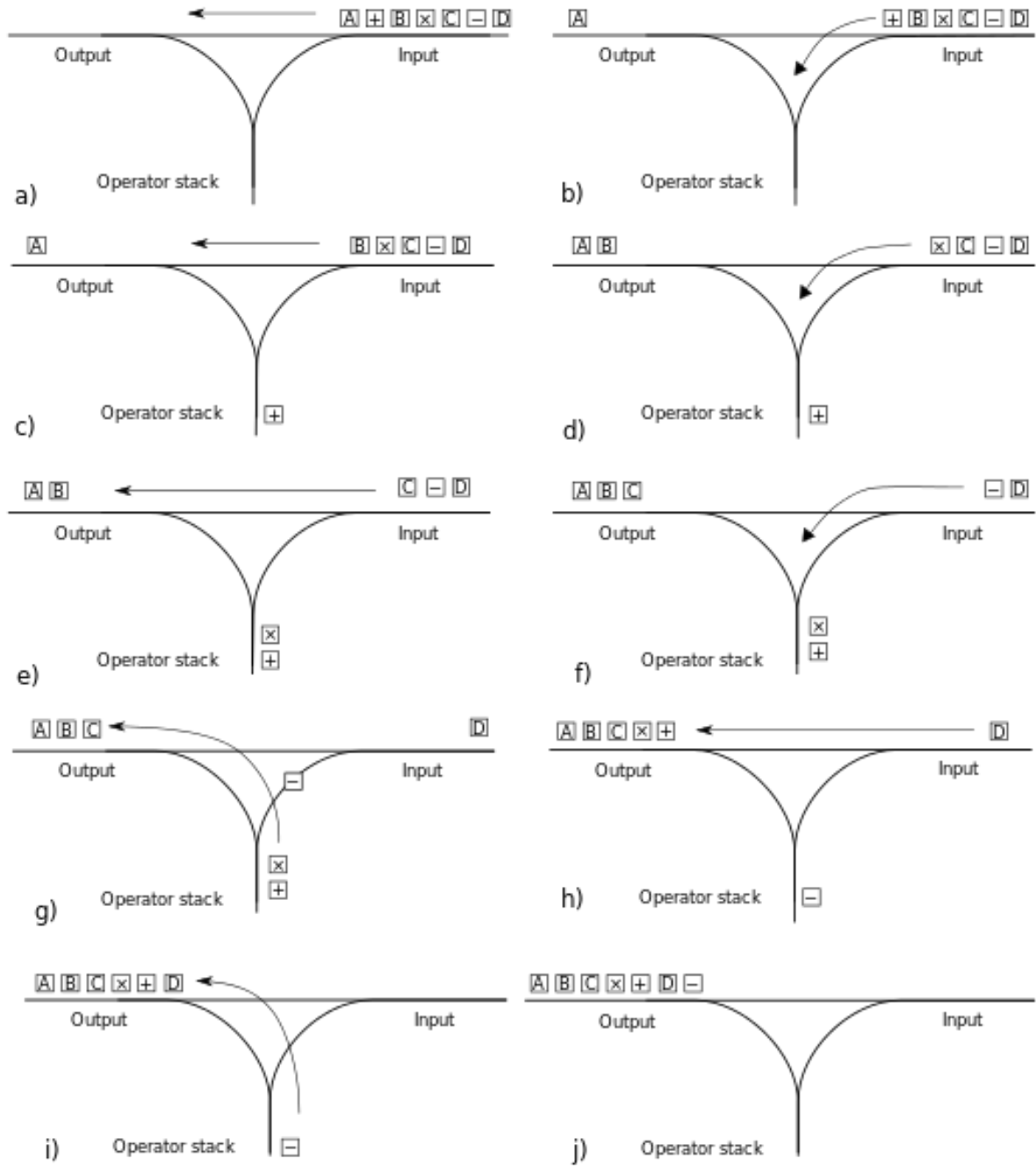


Figure 16: Illustration of Shunting-Yard algorithm implementation [19]

CASTle has the capability to output a table with step-by-step instructions on how the Shunting-Yard algorithm was implemented. Taking the example 'A + B * C - D' from above, the program outputs the table seen in Figure 17.

Enter your expression:
A+B*C-D

In this case the Shunting-Yard Algorithm is implemented as follows:

| TOKEN | ACTION | OUTPUT QUEUE | OPERATOR STACK | NOTES |
|-------|---|---------------|----------------|--|
| A | Enqueue A to output | A | | |
| + | Push operator '+' token to stack | A | + | |
| B | Enqueue B to output | A B | + | |
| * | Push operator '*' token to stack | A B | + * | |
| C | Enqueue C to output | A B C | + * | |
| - | Pop operator token '*' from stack to output | A B C * | + | '*' has equal or greater precedence than '-' |
| - | Pop operator token '+' from stack to output | A B C * + | | '+' has equal or greater precedence than '-' |
| - | Push operator '-' token to stack | A B C * + | - | |
| D | Enqueue D to output | A B C * + D | - | |
| | Pop operator token '-' from stack to output | A B C * + D - | | Expression has been fully read so drain operator stack |

The final output in postfix notation is:

A B C * + D -

Figure 17: Shunting-Yard step-by-step instructions adapted from [20]

The full pseudo-code of the algorithm, including how parentheses are managed, is as follows: [21]

```

Initialise an operator stack
Initialise an output queue
While there are tokens to be read:
    Read a token
    If it is a number:
        Add it to queue
    If it is an operator:
        While there is an operator on the top of the stack with greater
        precedence:
            Pop operators from the stack onto the queue
        Push the current operator onto the stack
    If it is a left bracket push it onto the stack:
    If it is a right bracket:
        While there is not a left bracket at the top of the stack:
            Pop operators from the stack onto the output queue
        Pop the left bracket from the stack and discard it
While there are operators on the stack, pop them to the queue

```

Note that an operator 'a' is said to have greater precedence over another operator 'b' if it appears higher in the order of operations **or** if they appear on the same level in the

order of operations *and* ‘b’ is left-associative. This is implemented into the CAS via the custom Python function *has_precedence*:

```
def has_precedence(a,b):  
    """Returns true if operator a has precedence over operator b taking  
    note of associativity of operations"""  
    if (OPS[b].associativity == RIGHT and  
        OPS[a].precedence > OPS[b].precedence):  
        return True  
    elif (OPS[b].associativity == LEFT and  
          OPS[a].precedence >= OPS[b].precedence):  
        return True
```

4.3.1 Limitations of the Shunting-Yard Algorithm

The Shunting-Yard algorithm is not adapted for unary operators and so various workarounds have been implemented into the CAS to treat them correctly. A limited solution is found by assigning the unary negative ‘-’ to a number, variable or function *before* implementing the algorithm, however this is not feasible for more complicated operands (see section 4.1). A more comprehensive solution is the addition of a new explicit unary negative operator ‘~’ that solves the issue but is less intuitive to use (see section 3.2.3).

The Shunting-Yard algorithm also does not account for composite functions. Our implementation successfully and intuitively handles composite functions with one input but requires extra parentheses for composite functions with two inputs. For example, in the CAS the logarithm function takes two inputs, the exponent and then the base. In order to use functions inside the logarithm they must be bracketed so they have correct precedence. The same rule applies for the modulus and greatest common divisor functions. See this in Figure 18.

```

Enter your expression:
log((fac(3)),2)

The expression has been interpreted as log(fac(3),2)

fac(3) = 6
Updated expression: log(6,2)
log(6,2) = 2.584962500721156

```

(a) Correct use of composite functions

```

Enter your expression:
log(fac(3),2)

The expression has been interpreted as log(3,fac(2))

fac(2) = 2
Updated expression: log(3,2)
log(3,2) = 1.5849625007211563

```

(b) Incorrect use of composite functions

```

Enter your expression:
fac(fac(sqrt(9)))

The expression has been interpreted as fac(fac(sqrt(9)))

sqrt(9) = 3.0
Updated expression: fac(fac(3.0))
fac(3.0) = 6
Updated expression: fac(6)
fac(6) = 720

```

(c) Single input composite functions

Figure 18: How to use composite functions in the CAS

4.4 Postfix to Infix Algorithm

It is useful to transfer an expression from postfix notation back to infix notation. In order to perform this conversion an algorithm was adapted from [22] to allow for both binary and unary operators. The algorithm works on a parsed postfix expression and outputs an infix expression as one string. The full pseudo-code is as follows:

```

Initialise a stack
While there are tokens to be read:

```

```

Read a token

If it is an operand:
    Push it to the stack

If it is a binary operator:
    Pop two operands from the stack
    Create a string of the operation
    Push the operation string to the stack

If it is a unary operator:
    Pop one operand from the stack
    Create a string of the operation
    Push the operation string to the stack

Pop the full infix expression from the stack and return it

```

An example of this algorithm applied to the expression `'(2 + 2) - (12 * sqrt(4))'` can be seen in Figure 19 below.

```

Enter your expression:
(2 + 2) - (12 * sqrt(4))
The expression has been parsed in the following way:
['(', '2', '+', '2', ')', '-', '(', '12', '*', 'sqrt', '(', '4', ')', ')']
The Shunting-yard Algorithm has been implemented to receive:
['2', '2', '+', '12', '4', 'sqrt', '*', '-']
The postfix to infix Algorithm has been implemented to receive:
((2+2)-(12*sqrt(4)))

```

Figure 19: Infix to postfix and back again

5 Evaluating Numerical Expressions

Now that we have implemented the Shunting-Yard algorithm and developed a parser capable of processing numerical expressions, we are ready to begin computation.

5.1 Computation Algorithm

Once the expression is in postfix form it is trivial to compute the answer; we saw one simple example in Section 4.2. The algorithm that achieves this takes a parsed postfix expression and outputs the answer. The full pseudo-code is as follows:

```
Initialise a stack
While there are tokens to be read:
    Read a token
    If the token is a number:
        Push it to the stack
    If the token is a binary operator:
        Pop two operands from the stack
        Perform the operation
        Insert the result back into the expression list
    If the token is a unary operator:
        Pop one operand from the stack
        Perform the operation
        Insert the result back into the expression list
```

We also developed an accessory algorithm that is functionally identical to the above but suppresses the output of the answer and instead outputs specific error messages for each operator, if an error is encountered.

5.2 Capabilities

The CAS has numerical functionality similar to and in some areas exceeding a scientific calculator. A full list of the features associated with numerical expressions is as follows:

- Compute numerical expressions with various functions implemented
- Round the answer to a user specified number of significant figures
- Output relevant error messages
- Output step-by-step instructions on request
- Output the expression in parsed and postfix form on request with step-by-step Shunting-Yard instructions

The program can perform most numerical calculations with the basic operators, most trigonometric functions, and a large range of other functions including the factorial, square

root and more. The full list of implemented functions and operators, and how to use them can be seen in Figure 20.

```
The following operators are used as such:

x to the power of y = x^y
x multiplied by y = x*y
x divided by y = x\y
x plus y = x+y
x minus y = x-y
Negative x = ~x (IMPORTANT: See section 2.2.3)

The following functions are used as such:

x mod y = mod(x,y) (See section 4.2.1)
Greatest Common Divisor of x and y = gcd(x,y) (See section 4.2.1)
Log base y of x = log(x,y) (See section 4.2.1)
Square root of x = sqrt(x)
Exponential of x = exp(x)
Absolute value of x = abs(x)
Factorial of x = fac(x)
Sine of x = sin(x)
Cosine of x = cos(x)
Tangent of x = tan(x)
Arcsin of x = asin(x)
Arccos of x = acos(x)
Arctan of x = atan(x)
Sinh of x = sinh(x)
Cosh of x = cosh(x)
Tanh of x = tanh(x)
Arcsinh of x = asinh(x)
Arccosh of x = acosh(x)
Arctanh of x = atanh(x)
```

Figure 20: Numerical capabilities of the CAS

The CAS also spots and makes the user aware of errors in their expression. These include parentheses and function errors that are triggered when the input is invalid. Figure 21 shows some examples.

```

Enter your expression:
fac(3.5)

3.5! = Undefined
ERROR: The factorial function only accepts non-negative integers

```

(a) Factorial error message

```

Enter your expression:
asin(2)

arcsin(2) = Undefined
ERROR: Inverse sine only accepts values on the interval [-1,1]

```

(b) Inverse sine error message

```

Enter your expression:
log(-2,4)

log(-2) (base 4) = Undefined
ERROR: The logarithm function only accepts positive numbers

```

(c) Logarithm error message

Figure 21: Error message examples

Step-by-step instructions are provided, on request. This can be useful for long winded calculations. It also allows errors in the program to be discovered much more easily as we can spot exactly which operation caused the issue. These instructions are created by the CAS by implementing the algorithm in Section 4.4 at each stage of the calculation in order to rebuild the infix expression from its postfix form. An example of this in action can be seen in Figure 22 below.

```

Enter your expression:
((12*2)/3)-fac(3)^2

The expression has been interpreted as (((12*2)/3)-(fac(3)^2))

12*2 = 24
Updated expression: ((24/3)-(fac(3)^2))
24/3 = 8.0
Updated expression: (8.0-(fac(3)^2))
fac(3) = 6
Updated expression: (8.0-(6^2))
6^2 = 36
Updated expression: (8.0-36)
8.0-36 = -28.0

```

Figure 22: Step-by-step instructions of solving an example expression

Finally, the CAS can round answers to an amount of significant figures the user inputs as Figure 23 shows.

```
Enter your expression:
log(2,e)

Do you want to round your answer?: [y/n]
y

How many significant figures do you want your answer to be rounded to?:
3

Do you want step-by-step instructions?: [y/n]
y

-----

The expression has been interpreted as log(2,e)
log(2,2.718281828459045) = 0.6931471805599453

Evaluating your expression to 3 significant figures gives the following answer:
0.693
```

Figure 23: CAS rounding an answer

5.3 Limitations

The limitations of the CAS are generally a result of floating-point arithmetic and the associated approximations (see section 2.3.3). Not all floating-point errors can be fixed. This is due to the difficulty of balancing rounding such that float errors are caught whilst maintaining an acceptable level of accuracy. Figure 24 shows one example that falls through the cracks.

```

Enter your expression:
1.2*3

The expression has been interpreted as (1.2*3)

1.2*3 = 3.5999999999999996
3.5999999999999996

```

Figure 24: Unresolved floating-point errors

Python, like other programming languages, can only work with very large or very small numbers up to a point. The largest possible float Python has access to is ‘1.7976931348623157e+308’, any larger and Python outputs ‘inf’ in its stead. The smallest possible float is ‘2.2250738585072014e-308’. It is unlikely that any expressions in the CAS will require larger/smaller numbers than these. We can see an example of this in figure 25.

```

Enter your expression:
1/10^-320

The expression has been interpreted as (1/(10^-320))

10^-320 = 1e-320
Updated expression: (1/1e-320)
1/1e-320 = inf

```

Figure 25: Floating-point infinite representation

Floating-point arithmetic is also only accurate up to a certain point. The upper bound on the error of a floating-point number in Python, called the *machine epsilon*, is equal to ‘2.220446049250313e-16’. More formally, this is the difference between one and the next largest float.

Float errors also make operations involving complex numbers difficult to implement. In theory they are possible with CASTle, as they are supported by Python, however in application float errors make their use very unwieldy. As a result many of the algorithms would have to be adjusted significantly to spot when the CAS is dealing with a complex number. Thus we exclude them from the list of capabilities.

Finally, the list of available functions is not complete; most of the very common or simple functions are implemented, however some are missing such as the floor and ceiling functions. Less frequently used functions such as the gamma function or partition function, for example, are not in CASTle. Adding functions to the program is relatively simple, and in theory most functions that deal with real numbers are possible.

6 Simplifying Compound Linear Expressions

The main function of our computer algebra system is the simplification of compound linear expressions in multiple variables. Limiting ourselves to these expressions streamlines the process of simplification, this is because we can now always be sure what form our expression will come in. This reduces the back-end logic required by orders of magnitude.

Definition 6.1. A real-valued linear expression is an equation that can be manipulated into the form $a_0 + a_1x_1 + \dots + a_nx_n = 0$ where x_1, \dots, x_n are the variables and a_0, a_1, \dots, a_n are the real coefficients.

A *compound* linear expression is equivalent to a linear expression where the ‘variables’ can be simple non-linear expressions without operations or coefficients such as ‘ xyz ’ or ‘ $z^{10}x^{-5}$ ’. An example of a compound linear expression is ‘ $12z^2xy^{-3} + 4xyz - 7wz wz$ ’. Henceforth we refer to compound linear expressions as *linear expressions* and the above simple non-linear expressions as *variables*. We do this for readability. The numerical component of the CAS was developed to service the simplification function.

6.1 Simplification Algorithms

There are various hurdles that must be overcome in the development of a CAS that can simplify expressions.

6.1.1 Parsing Linear Expressions

The first issue is how we parse a linear expression. The parser developed for this purpose builds upon the numerical parser from Section 4. We refer to it as a *standardiser*.

By restricting ourselves to linear expressions we ensured that the user input, which is a single string, will always be predictably structured. The standardiser takes this string, and converts it into a list in *standard form*. This form separates operators, variables and both simple and composite coefficients/constants. Composite here just means composed of operations. For example, we can see how the expression $(4 + 2) + 3x^2 - (5 * 3)y^3$ is standardised in Figure 26.

```
Enter your expression:
(4+2) + 3x^2 - (5*3)y^3
['(4+2)', '+', '3', 'x^2', '-', '(5*3)', 'y^3']
```

Figure 26: Linear expression in standard form

This is accomplished by using an algorithm we developed. Firstly the numerical parser is applied to the user’s expression to output a list representing it (See Section 4). Then we use the following algorithm to output the expression as a standard list. The logic relies on two booleans; these are *b_run* and *v_run*. If *b_run* is *True* then the algorithm is currently building a composite coefficient string, if it is *False* then we are not. The boolean *v_run* is similar except that it detects when the program is building a non-linear term such as x^2yz . We refer to these terms as *variables* for simplicities sake. This stops the program from terminating the building of a complicated variable term early. The full pseudo-code is below:

```
Parse the linear expression into a list using the numerical parser
Initialise the number of expected close brackets b_num = 0
Initialise two booleans b_run and v_run = False
Initialise an empty list to hold the standardised expression

While there are tokens to be read:
    Read a token
    If the token is equal to (:
        If b_num is equal to 0:
            Initialise coefficient b_co as an empty string
```

```

        Set b_run as True
    If b_run is True:
        Combine b_co with the token into one string
        If the token is equal to (:
            Increment b_num by 1
        If the token is equal to ):
            Reduce b_num by 1
        If b_num is now equal to 0:
            Append the finished composite coefficient to the
            standard list
            Set b_run as false
    If the token is a number and we are not building a variable or
    coefficient:
        Append the token to the standard list
    If the token is an operator and we are not building a variable or
    coefficient:
        Append the token to the standard list
    If the token is not a number or operator and we are not building a
    variable or coefficient and we are not at the end of the list:
        Initialise variable v as an empty string
        Set v_run as True
    If v_run is True:
        If the token is an operator or we are at the end of the list:
            Append the finished variable to the standard list
        Otherwise:
            Combine v with the token into one string

```

To best understand this algorithm it is helpful to work through a simple example; we will use the expression $(1 * (2 + 3))x^2y^2 + 2x^3$. Firstly we parse it into the list $[(, '1', *, (, '2', '+', '3', ')', ')', 'x', '^', '2', 'y', '^', '2', '+', '2', 'x', '^', '3']$ using the numerical parser. The initial state of our two lists is as follows:

- Expression list = $[(, '1', *, (, '2', '+', '3', ')', ')', 'x', '^', '2', 'y', '^', '2', '+', '2', 'x', '^', '3']$
- Standard list = $[]$

Now we begin to work through the algorithm logic. Reading tokens left to right from the expression list we get the token ‘(’. This is an open bracket and so we know we are going to be building a composite coefficient, thus we set *b_run* as *True* and initialise an empty string *b_co* = ‘’ to hold this coefficient. We also know to expect a close bracket token sometime in the future, hence we increment *b_num* by one to 1.

Then we add the token ‘(’ and the empty string together to receive *b_co* = ‘(’. Now we read more tokens from the expression list; if the token is **not** a close or open bracket we combine the token with our *b_co* string. The first two instances of this loop results in the string *b_co* = ‘(1*’. The next token is an open bracket ‘(’. We now know to expect **another** close bracket string sometime in the future, thus we increment *b_num* by one to 2 and combine the token with our *b_co* string. The next few iterations of this result in the string *b_co* = ‘(1*(2+3’. Now the next token is a close bracket ‘)’, we thus reduce the number of expected close brackets *b_num* by one to 1 and add it to the coefficient string. Repeating this process with the next token, which is a close bracket, results in the composite coefficient string *b_co* = ‘(1*(2+3))’. Now the algorithm checks the value of *b_num*; it is equal to zero and thus we know to stop building the coefficient. Hence we set *b_run* as *False*. Now we append this string to our empty standard list.

- Expression list = [(‘(’, ‘1’, ‘*’, ‘(’, ‘2’, ‘+’, ‘3’, ‘)’, ‘)’, ‘x’, ‘^’, ‘2’, ‘y’, ‘^’, ‘2’, ‘+’, ‘2’, ‘x’, ‘^’, ‘3’]
- Standard list = [(‘(1*(2+3))’)]

The next token in the expression list is ‘x’. This token is **not** a number *or* operator *and* we are not currently building a variable or coefficient string *and* we are not at the end of the list. This tells us that we should begin building a variable string, thus we initialise an empty string *v* = ‘’ to hold it and set *v_run* as *True*. The logic of building a variable is simple due to our constraints on variable terms. All exponents in the variable must be simple, i.e. no operations, thus we know to stop building the variable either when we are at the end of the list or we come to an operator. Thus looping through the tokens in the above expression list, adding them to the variable string and stopping when we hit ‘+’

results in $v = 'x^2y^2'$. We now set v_run as *False* and append the finished string to the standard list.

- Expression list = $[('(', '1', '*', ('(', '2', '+', '3', ')'), ')', 'x', '^', '2', 'y', '^', '2', '+', '2', 'x', '^', '3']$
- Standard list = $[(1*(2+3)), 'x^2y^2']$

The next token in our expression list to be considered is '+'; we are not currently building a composite coefficient, hence this is just a simple operator and we append it to the standard list. The following token is '2'; for the same reason as above this is just a simple coefficient and we can append it to the standard list.

- Expression list = $[('(', '1', '*', ('(', '2', '+', '3', ')'), ')', 'x', '^', '2', 'y', '^', '2', '+', '2', 'x', '^', '3']$
- Standard list = $[(1*(2+3)), 'x^2y^2', '+', '2']$

The remaining few loops run through the process of building another variable string with the subtlety that the operation stops because we are at the end of the list and not due to hitting an operator. Hence the final state of our two lists is as follows:

- Expression list = $[('(', '1', '*', ('(', '2', '+', '3', ')'), ')', 'x', '^', '2', 'y', '^', '2', '+', '2', 'x', '^', '3']$
- Standard list = $[(1*(2+3)), 'x^2y^2', '+', '2', 'x^3']$

We have successfully parsed the compound linear expression into a *standard* form. By doing this we now have the advantage of knowing what to expect from our expression, i.e. when we come across a variable term the preceding term will almost always be a coefficient and the following term will either not exist or always be an operator. We have also combined the composite coefficients into one string, ready to be computed by the numerical wing of CASTle. This is a strong foundation for the rest of CASTle's simplification logic to build on.

6.1.2 Commutativity in Variables

Definition 6.2. A binary operation ‘ $a \# b$ ’ is commutative if ‘ $a \# b$ ’ = ‘ $b \# a$ ’

Now that the CAS has parsed the linear expression, we have to ensure that variables that are equivalent, but inputted in different forms, are converted to one standard form. In most cases this will be an issue of commutativity, that is to treat the variable ‘ xy ’ as equivalent to ‘ yx ’. However it also includes accounting for unorthodox user inputs such as converting ‘ x^0 ’ to ‘1’ or ‘ y^1 ’ to ‘ y ’.

In Python, the variables ‘ xy ’ and ‘ yx ’ are strings and thus distinct objects; there is no innate sense of commutativity and so it must be explicitly implemented. This is important if we want to deal with expressions that have complicated variables such as in ‘ $2zyx + 3xyz$ ’, for example. In order to achieve this we devised an algorithm that takes a variable string from the user input and outputs it with exponents simplified and variables appearing in alphabetic order; this ensures that commutative variables are equivalent.

The algorithm has two functions, it can spot that ‘ xy ’ = ‘ yx ’ and it can take ‘ $xxxx$ ’ and output ‘ x^4 ’. These two features combined allow the CAS to convert very complicated and unoptimised variables such as ‘ $xx^2yy^0z^3xyzx^{-2}$ ’ to the much simplified form ‘ $x^2y^2z^4$ ’. The main logic of the algorithm relies on runs.

Definition 6.3. A run is a sequence of consecutive equal objects. The *length* of a run is equal to the amount of objects in the run.

The variable ‘ $xxxx$ ’ contains a single run of length four whilst the variable ‘ $zzzxxyxww$ ’ consists of five runs. We explain the algorithm using the example above; ‘ $xx^2yy^0z^3xyzx^{-2}$ ’. Firstly we break the expression into a list using the numerical parser and some further simple logic to receive `['x', 'x', '^', '2', 'y', 'y', '^', '0', 'z', '^', '3', 'x', 'y', 'z', 'z', 'x', '^', '-2']`. This list now represents our variable; however the notation is clunky so we move to string notation for readability.

Once we have the variable term sufficiently parsed, we must deal with the exponents. We

refer to the components of the variable term such as ‘ x ’ as *sub-variables*. We remove all sub-variables in the expression with exponents and replace them with a run of the sub-variable with length equal to the exponent. If the exponent is negative, we replace the sub-variable with a run of its upper-case form. That is, we send the list ‘ $xx^2yy^0z^3xyzx^{-2}$ ’ to ‘ $xxxyzzzxyzXX$ ’. Then we sort this list alphabetically; ‘ $XXxxxxyyzzzz$ ’.

Now that the variable is sorted alphabetically and has had all exponents removed, we initialise a Python dictionary to represent it. This dictionary works as such {key = sub-variable : value = exponent}. If the sub-variable is only present in its upper case form we initialise its value as -1, if it is present only in its lower case form we initialise the value as 1 and if the variable is present in **both** its upper and lower case form we initialise its value as 0. The dictionary for our example variable is initialised as {‘ x ’:‘0’, ‘ y ’:‘1’, ‘ z ’:‘1’}.

The runs in the list ‘ $XXxxxxyyzzzz$ ’ are now used to update the dictionary values. We loop through the list and compare each token to the token immediately in front of it. If they are upper-case and equal we reduce the variables dictionary value by 1 and if they are lower-case and equal we increase the variables dictionary value by 1. Applying this to the list ‘ $XXxxxxyyzzzz$ ’ results in the dictionary {‘ x ’:‘2’, ‘ y ’:‘2’, ‘ z ’:‘4’}. This process is equivalent to initialising the dictionary as {‘ x ’:‘0’, ‘ y ’:‘0’, ‘ z ’:‘0’} and using the length of the runs instead. That is, increasing the dictionary value by the length of a lower-case run and decreasing the dictionary value by the length of an upper-case run. While more elegant in theory, it is simpler to implement the former algorithm in Python.

We now have a dictionary that represents our variable, the final step is to build a string of the variable using this dictionary. For nice notation we make sure not to include any sub-variables with exponent zero and avoid including the ‘ $^$ ’ operator for sub-variables of exponent one. Implementing this outputs ‘ $x^2y^2z^4$ ’ from {‘ x ’:‘2’, ‘ y ’:‘2’, ‘ z ’:‘4’}. We have successfully simplified ‘ $xx^2yy^0z^3xyzx^{-2}$ ’ to ‘ $x^2y^2z^4$ ’. This is now in a form we refer to as *standard*, i.e. variables appear in alphabetic order from left to right. This means that ‘ $xx^2yy^0z^3xyzx^{-2}$ ’ and ‘ $z^3x^{-3}yzxyx^3$ ’; two variable terms that look nothing alike, both

evaluate to ' $x^2y^2z^4$ ' and can then be equated when collecting alike terms later on in the simplification process. We can see this in Figure 27.

```
Enter your expression:
xx^2yy^0z^3xyzx^-2
The variable xx^2yy^0z^3xyzx^-2 is equal to: x^2y^2z^4
```

(a) First form of the variable

```
Enter your expression:
z^3x^-3yzxxyx^3
The variable z^3x^-3yzxxyx^3 is equal to: x^2y^2z^4
```

(b) Second form of the variable

Figure 27: Equivalent variable expressed in two different forms, evaluated successfully

This algorithm only works with variables represented by lower-case alphabetic characters as we need access to an upper-case form to work with negative exponents. Powers also must be integers and cannot be composite, i.e. no operations in the exponent. The full pseudo-code of the algorithm is as follows:

```
Parse the string into a list
While there are tokens to be read:
    Read a token
    If the token represents a variable with an exponent:
        Pop the associated exponent, variable and ^ token
        from the list
        If the exponent is non-negative:
            Insert the variable in lower-case to the list a
            number of times equal to the exponent
        If the exponent is negative:
            Insert the variable in upper-case to the list a
            number of times equal to the exponent
Sort the list alphabetically

Initialise a dictionary with keys = sub-variables and
```

```

values = zero

If the variable appears in the list in upper-case:
    Reduce the dictionary value by 1
If the variable appears in the list in lower-case:
    Increase the dictionary value by 1

While there are tokens to be read:
    Read a token
    If it is equal to the token immediately in front of it:
        If the tokens are upper-case:
            Reduce the dictionary value by 1
        If the tokens are lower-case:
            Increase the dictionary value by 1
Build the variable from the dictionary into a string

```

6.1.3 Computing Composite Coefficients

We now have our linear expression as a list with all variables simplified to a standard form. The next step is to use the numerical functionality of the CAS to compute the composite coefficients. This process uses a very simple algorithm wherein composite coefficients are extracted from the list, run through the Shunting-Yard algorithm, computed and then inserted back into the list. The full pseudo-code is as follows:

```

Standardise the expression into a list
Simplify the variables in the list

While there are tokens to be read:
    Read a token:
        If the token is a composite coefficient:
            Pop the token
            Apply the Shunting-Yard Algorithm
            Compute the answer
            Insert the answer back into the list

```

Applying this to the expression $(4 + 2) + 3x^2 + (4 * (2 + 3))x^2$ results in the list seen in Figure 28.

```
Enter your expression:
(4+2) + 3x^2 + (4*(2+3))x^2
['6', '+', '3', 'x^2', '+', '20', 'x^2']
```

Figure 28: Computing composite coefficients

6.1.4 Collecting Alike Terms

Now that we have computed the composite coefficients, simplified the variables and standardised the linear expression into a list, we are ready for the final step in the simplification process; collecting alike terms. This is achieved by an algorithm that represents the expression via a dictionary with keys for each variable and coefficients for values, i.e. $\{\text{key} = \text{variable} : \text{value} = \text{coefficient}\}$.

We loop over the standardised list looking for variables, if this is the first time we have seen the variable we initialise it into the dictionary with its corresponding coefficient. If we have seen the variable before, then it will have already been initialised and we increase or reduce the corresponding dictionary value by its coefficient. In this step we treat the constant terms as their own ‘variable’. This simple process outputs a dictionary containing every variable in the linear expression with their coefficients collected together. The pseudo-code is as follows:

```
Initialise empty dictionary
While there are tokens to be read:
    Read a token
    If the token represents a variable:
        If we have not seen the variable previously:
            Initialise the variable as a key in the dictionary with its
            coefficient as the value
        If we have seen the variable previously:
```

```
Add or subtract the coefficient of the variable to its  
dictionary value
```

Now that we have created a dictionary that represents the simplified linear expression, the last step is to process it into a string that can be outputted to the user. As a part of this process we take the unordered dictionary and convert it into a form that can be ordered. This is done to achieve nice notation where constant terms are placed at the front of the expression. Also, variable terms appear in the expression in ascending order by the sum of their exponents, i.e. ' $3xyz + 4x^2 + 12 + 4x^{-1}$ ' is outputted as ' $12 + 4x^{-1} + 4x^2 + 3xyz$ '. We have now achieved simplification functionality for linear expressions. As an example, applying all the above algorithms to the expression ' $(4 + 2) + 12zyx + (4 * 3)xyz - 2 + 3y^2$ ' outputs ' $6 + 3y^2 + 24xyz$ '. See this in Figure 29.

```
Enter your expression:  
(4+2) + 12zyx + (4*3)xyz - 2 + 3y^2  
  
CASTle has simplified your expression to:  
4 + 3y^2 + 24xyz
```

Figure 29: Simplification example

6.2 Capabilities

The CAS is capable of simplifying a class of compound linear expressions using the algorithms detailed in the previous section. We developed it with natural notation in mind, this means the user can input their expressions in the exact same form that they would write them on paper. That is, there are no unnecessary multiplication signs, exponents such as in ' $2x^0$ ' or ' $4x^1$ ' are not required and terms with a coefficient of one can be written without the coefficient, as in ' x^3 '. These are all steps outside of the standardised form we saw in Section 6.1.1 and so lots of extra logic is required in the application of the above algorithms to account for these various cases.

The CAS is also capable of outputting contextual step-by-step instructions. The genera-

tion of these instructions makes the application of the algorithms much more involved as we have to spot when we are dealing with constants versus variable terms. An example that details the full capabilities of the simplification function can be seen in Figure 30.

```

Enter your expression:
3z^0 + (sqrt(16)*(5+2.5)) + x^2x^-3 + 3.5xyzxyz + 4zyxzyx - (2^4)y^2 + (sin(pi))

Do you want step-by-step instructions?: [y/n]
y

-----

The variable z^0 is equal to: 1
The variable x^2x^-3 is equal to: x^-1
The variable xyzxyz is equal to: x^2y^2z^2
The variable zyxzyx is equal to: x^2y^2z^2
The expression has been simplified to: 3 + (sqrt(16)*(5+2.5)) + x^-1 + 3.5x^2y^2z^2 + 4x^2y^2z^2 - (2^4)y^2 + (sin(pi))
The constant term (sqrt(16)*(5+2.5)) has been interpreted as: (sqrt(16)*(5+2.5))

sqrt(16) = 4.0
Updated expression: (4*(5+2.5))
5+2.5 = 7.5
Updated expression: (4*7.5)
4*7.5 = 30.0

The expression has been simplified to: 3 + 30.0 + x^-1 + 3.5x^2y^2z^2 + 4x^2y^2z^2 - (2^4)y^2 + (sin(pi))
The coefficient of (2^4)y^2 has been interpreted as: (2^4)

2^4 = 16

The expression has been simplified to: 3 + 30.0 + x^-1 + 3.5x^2y^2z^2 + 4x^2y^2z^2 - 16y^2 + (sin(pi))
The constant term (sin(pi)) has been interpreted as: sin(pi)

sin(3.141592653589793) = 1.2246467991473532e-16
FLOAT ERROR: 1.2246467991473532e-16 has been rounded to 0

The expression has been simplified to: 3 + 30.0 + x^-1 + 3.5x^2y^2z^2 + 4x^2y^2z^2 - 16y^2 + 0
Constant term: 3 + 30 = 33
x^2y^2z^2 term: 3.5x^2y^2z^2 + 4x^2y^2z^2 = 7.5x^2y^2z^2

-----

Simplifying your expression gives the following answer:

33 + x^-1 - 16y^2 + 7.5x^2y^2z^2

```

Figure 30: Full simplification algorithm applied to an example

6.3 Limitations

The simplification functionality has some limitations, a full list is below:

- Variables must be lower-case alphabetic characters
- Exponents of variables must be integers
- Exponents of variables must be simple, i.e. no operations within the exponent
- Compound linear expressions only

Allowing for variables other than lower-case alphabetic characters would require the development of a new variable simplification algorithm as the current one relies on the existence of an upper-case form to represent negative powers. Similarly, requiring integer powers is a result of runs requiring integer lengths (See Section [6.1.2](#)).

Composite exponents are very possible, however a new standardiser would have to be created to correctly build variables. This is because the current algorithm assumes brackets appear only in coefficients or constants. The more pressing problem is that without allowing for decimal exponents we predict the user would be frustrated with the restriction of their composite exponents having to evaluate to an integer. Errors would be very frequent (See Section [6.1.1](#)).

Finally, the restriction to compound linear expressions was the basis for the entire development of the simplification function and lifting this restriction would require starting from scratch with much more general algorithms.

7 Commercial Computer Algebra Systems

There are various fully-fledged general-purpose computer algebra systems currently available, both free and paid. The most commonly used are Mathematica [\[23\]](#), Maple [\[24\]](#), SageMath [\[25\]](#), Maxima [\[26\]](#) and SymPy [\[27\]](#). Each tends to focus on different areas; for example, SageMath has extensive number theory capability, Maple is excellent for data

analysis and visualisation while Mathematica has machine learning and neural network features. All the above computer algebra systems have strong simplification ability including Mathematica, the underlying CAS of Wolfram Alpha. In Figure 31 we can see that Wolfram simplifies expressions into various different forms.



WolframAlpha computational intelligence.

Simplify $x^2 + 2x + 3x + (x+1)^2 + 2$

Extended Keyboard Upload Examples Random

Input interpretation:

simplify $x^2 + 2x + 3x + (x+1)^2 + 2$

Results: More forms ☒ Step-by-step solution

$(2x+1)(x+3)$

$2x^2 + 7x + 3$

$\frac{1}{8}(4x+7)^2 - \frac{25}{8}$

Figure 31: Wolfram Alpha simplifying a complicated algebraic expression [28]

7.1 Applications of a CAS

Commercial computer algebra systems are used frequently in education and research.

7.1.1 In Education

Those advocating for more CAS use in the classroom have drawn comparison to the introduction of the calculator (Koepp et al [29]). For example, when working with transcendental functions it was common to perform paper and pencil computation, now with the widespread use of calculators the need for these methods has gone and performing the same computation takes seconds. It is possible to imagine a future classroom where the same thing has occurred to paper and pencil symbolic algebra. For example, when performing arduous integration through the use of partial fractions. This in fact is already taking place in some schools around the world. In recent years computer algebra

systems are seeing more use within the classroom at both primary and secondary level [30].

In higher education computer algebra systems are used much more extensively, with many universities integrating their use directly into their courses. One study states that more than half of participating mathematicians reported using a CAS in teaching, with one-sixth expressing frequent use (Lavicza [31]). An example can be found at the University of Nottingham where MATLAB is either taught or used in various modules.

CASTle has educational value in the form of step-by-step instructions for both linear expression simplification and numerical computation. In particular this is useful for showing the order in which algebraic/computational steps should be taken.

7.1.2 In Research

Computer algebra systems have seen extensive use in mathematical research. One study found that more than two-thirds of participating mathematicians indicated at least occasional use of CAS in research with one one-third indicating frequent use (Lavicza [31]). An example can be seen in Houston and Sime [32], where symbolic algebra was exploited to approximate systems of partial differential equations. SymPy and FEniCS were used, the latter of which is a PDE equation solver package with an inbuilt CAS written in Python.

8 Conclusions

We have achieved the goal of simplifying compound linear expressions in multiple variables. This capability was built upon the successful implementation of numerical computation functionality. The limitations of the CAS, aside from the restriction to linear expressions, are generally a result of floating-point arithmetic. These cause errors in computation and complicate the simplification of variables. For future development within the scope of the systems current focus on linear expressions, I would endeavour to build a class representing rational numbers. This would fix many of the issues surround floats as their use would be avoided almost entirely. However this would require a complete rework of the application of most of the algorithms in the back-end of CASTle.

CASTle was written in Python; this comes with lots of benefits. Unlike many other programming languages, Python was developed with object-oriented design as a pillar of the language. This means that data structures such as the stack and queue can be developed easily using Python's class system. Other structures such as the list and dictionary come built-in. Python is also very easy to learn, with syntax that is often much simpler and more readable than other languages; this results in code that is far easier to maintain and write for a beginner. It is feasible however that one day the capabilities of CASTle could be limited by our choice of language as relatively few general-purpose computer algebra systems are written in Python. Instead we would opt for *C++* or *Wolfram Language*, the proprietary language of Wolfram Mathematica.

Whilst keeping the scope of the CAS relatively restrained, future development could include building functionality to expand brackets and factorise expressions. Or we could be more ambitious and move in a pure direction and attempt to include support for Group or Number theory capabilities.

9 Raw Code

The back-end of the CAS consists of over 2000 lines of code split into 9 files and bundled into an executable. If the reader is interested in the code, the following is a short description of each file discussed in the most logical order:

- **cas_data_structures.py** - This file has classes representing the queue and stack data structures with their associated methods.
- **cas_parser.py** - This file contains code responsible for parsing mathematical expressions, including functions that split strings in various ways, check the data type of an input and ultimately the parser function itself; *parse*.
- **cas_shunt.py** - This file is responsible for implementing the Shunting-Yard algorithm on infix expressions; there are two main functions, one that outputs the postfix expression and another that outputs tabular instructions. These rely on a dictionary called *OPS* which contains every operator and their precedence level/associativity properties. There is also a function responsible for checking which of two operators is of higher precedence according to the above properties.
- **cas_deshunt.py** - This file has code capable of converting postfix expressions back to infix notation. It relies on dictionaries which split the operators into four different categories. These are the *UNARY_OPS* representing single-input functions, the *BINARY_OPS* representing the normal five binary operators, *SPECIAL_BINARY_OPS* representing double-input functions and *SPECIAL_UNARY_OPS* which represents our \sim operator.
- **cas_modify.py** - This file has two functions, one rounds a numerical answer to a specified number of significant figures and the other turns integer floats into integers.
- **cas_evaluate_num.py** - This file contains the code that computes the answer to numerical expressions in postfix form. The responsible function takes two arguments, the expression itself, and the answer to the question of whether the user desires step-by-step instructions.

- **cas_error_message.py** - This file consists of one function responsible for generating error messages.
- **cas_simplify_linear.py** - This file is responsible for simplifying linear expressions in different variables. The main function, called *simplify_linear*, implements an algorithm on a linear variable expression in order to collect alike terms. Another function, called *commutative*, takes unsimplified variables and simplifies them (e.g. $xyyx$ to x^3y^2). The function *standardise* is a type of parser that takes already parsed expressions and standardises them into a form the rest of the file can use.
- **CAStle.py** - This file contains the main logic responsible for using the systems various features. The main function; *CAStle*, allows the user to input their expressions, request simplification or computation, round their answer and finally attain access to the parsed and shunted form of their expression.
- **CAStle.exe** - This is the executable.

10 References

- [1] M. J. G. VELTMAN (1999), *Nobel Lecture*, Aula Magna, Stockholm University, accessed 28th March 2020, [<https://www.nobelprize.org/prizes/physics/1999/veltman/lecture/>]
- [2] R. NELSON (No date), *Hewlett-Packard Calculator Firsts*, www2.hp.com, accessed 28th March 2020, archived at [<https://web.archive.org/web/20100703031935/http://h20331.www2.hp.com/Hpsub/cache/392617-0-0-225-121.html>]
- [3] S. K. CHANG (2003), *Data Structures and Algorithms*, World Scientific Publishing, University of Pittsburgh
- [4] COMPUTERSCIENCEWIKI (No date), *Queue*, computersciencewiki.org, accessed 1st March 2020, [<https://computersciencewiki.org/index.php/Queue>]
- [5] WIKIPEDIA USER VEGPUFF (2009), *Diagram representing Data Queues*, wikipedia.com, accessed 1st March 2020, [https://en.wikipedia.org/wiki/File:Data_Queue.svg]
- [6] COMPUTERSCIENCEWIKI (No date), *Stack*, computersciencewiki.org, accessed 1st March 2020, [<https://computersciencewiki.org/index.php/Stack>]
- [7] WIKIPEDIA USER MAXTREMUS (2015), *A LIFO (Last In First Out) stack, abstract data type, in a simple representation of a stack runtime with push and pop operations.*, wikipedia.com, accessed 1st March 2020, [https://commons.wikimedia.org/wiki/File:Lifo_stack.png]
- [8] COMPUTERSCIENCEWIKI (No date), *String*, computersciencewiki.org, accessed 2nd March 2020, [<https://computersciencewiki.org/index.php/String>]
- [9] E. MATTHES (2016) *Python Crash Course, A Hands-On, Project-Based Introduction to Programming*, no starch press, San Francisco
- [10] COMPUTERSCIENCEWIKI (No date), *Boolean*, computersciencewiki.org, accessed 26th March 2020, [<https://computersciencewiki.org/index.php/Boolean>]

- [11] COMPUTERSCIENCEWIKI (No date), *Float*, computersciencewiki.org, accessed 2nd March 2020, [<https://computersciencewiki.org/index.php/Float>]
- [12] COMPUTERSCIENCEWIKI (No date), *Int*, computersciencewiki.org, accessed 2nd March 2020, [<https://computersciencewiki.org/index.php/Int>]
- [13] MATH VAULT (No date) , *The Definitive Glossary of Higher Mathematical Jargon*, mathvault.ca, accessed 25th February 2020, [<https://mathvault.ca/math-glossary>]
- [14] R. ZAKIS, A. ROLEAU (2017), *Order of operations: On convention and met-before acronyms*, Educational Studies in Mathematics, retrieved 25th February 2020, [<https://link-springer-com.ezproxy.nottingham.ac.uk/article/10.1007%2Fs10649-017-9789-9>]
- [15] CHEMNITZ UNIVERSITY OF TECHNOLOGY (No date), *Priority rules and associativity*, accessed 26th February 2020, Original webpage [<https://www.tu-chemnitz.de/urz/archiv/kursunterlagen/C/kap2/vorrang.htm>], Archived translation [<https://archive.is/232CE#selection-1433.1-1491.38>]
- [16] CODEPLEA (2016), *Exponentiation Associativity and Standard Math Notation*, codeplea.com, accessed 25th February 2020, [<https://codeplea.com/exponentiation-associativity-options>]
- [17] C. L. HAMBLIN (1962), *Translation to and from Polish Notation*, The Computer Journal, retrieved 26th February 2020, [<https://academic.oup.com/comjnl/article/5/3/210/424386>]
- [18] E. W. DIJKSTRA (1961), *Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60*, Mathematisch Centrum, retrieved 26th February 2020, [<https://ir.cwi.nl/pub/9251>]
- [19] WIKIPEDIA USER SALIX ALBA (2010), *Illustration of the shunting yard algorithm*, wikipedia.com, accessed 26th February 2020, [https://commons.wikimedia.org/wiki/File:Shunting_yard.svg]

- [20] ROSETTACODE (No date), *Parsing/Shunting-yard algorithm*, rosettacode.org, accessed 15th March 2020, [https://rosettacode.org/wiki/Parsing/Shunting-yard_algorithm#Python]
- [21] B. TILIKESW, J. THELWALL, J. KHLIM, J. SILVERMAN (No date), *Shunting Yard Algorithm*, Brilliant.org, accessed 26th February 2020, [<https://brilliant.org/wiki/shunting-yard-algorithm/>]
- [22] P. SINGH (No date), *Postfix to Infix*, geeksforgeeks.org, accessed 3rd March 2020, [<https://www.geeksforgeeks.org/postfix-to-infix/>]
- [23] *Wolfram Mathematica*, wolfram.com, accessed 6th April 2020, [<https://www.wolfram.com/mathematica/>]
- [24] *Maplesoft - Software for Mathematics*, maplesoft.com, accessed 6th April 2020, [<https://www.maplesoft.com/>]
- [25] *SageMath - Open-Source Mathematical Software System*, sagemath.org, accessed 6th April 2020, [<https://www.sagemath.org/>]
- [26] *Maxima, a Computer Algebra System*, maxima.sourceforge, accessed 6th April 2020, [<http://maxima.sourceforge.net/>]
- [27] *SymPy*, sympy.org, accessed 6th April 2020, [<https://www.sympy.org/en/index.html>]
- [28] WOLFRAMALPHA (No date), *Wolfram Alpha, computational intelligence*, wolfram-alpha.com, accessed 10th March 2020, [<https://www.wolframalpha.com/>]
- [29] W. KOEPF (2003) *Computer Algebra in Education, Chapter: Computer Algebra in Education*, Springer, Berlin, accessed 12th March 2020, [<https://pdfs.semanticscholar.org/e28d/8cf4244125b4efc1a1a0ebf6562f87ef5617.pdf>]
- [30] MEDIUM (2017), *The Use of Computer Algebra Systems (CAS) in Senior Mathematics*, medium.com, accessed 12th March 2020, [<https://medium.com/tech-based-teaching/the-use-of-computer-algebra-systems-cas-in-senior-mathematics-799f105a1733/>]

- [31] Z. LAVICZA (2008), *A comparative analysis of academic mathematicians' conceptions and professional use of computer algebra systems in university mathematics*, accessed 25th March 2020, [https://www.researchgate.net/publication/298187352_A_comparative_analysis_of_academic_mathematicians'_conceptions_and_professional_use_of_computer_algebra_systems_in_university_mathematics]
- [32] P. HOUSTON, N. SIME (2018), *Automatic symbolic computation for discontinuous Galerkin finite element methods*, accessed 25th March 2020, [<https://arxiv.org/abs/1804.02338/>]