

PROJECT 3 IMPLEMENTATION GUIDE

CSE 340 FALL 2021

Rida Bazzi

Steps to implement the project

These steps are not exhaustive but they cover many of the important implementation points. I will describe each of these steps in more details in the remainder of this document.

As a general rule, you can save yourself a lot of time if you read everything carefully.

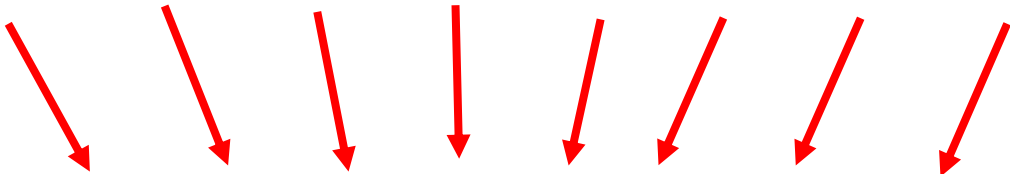
1. The first step is to write your parser. You should finish writing the parser completely and test it by **passing ALL test cases on the submission site before attempting to implement anything else**. I cannot emphasize this strongly enough. I have seen many projects in the past that suffered because the parser was not written first.

NOTE You should read this document at least a couple of times to get a good understanding.

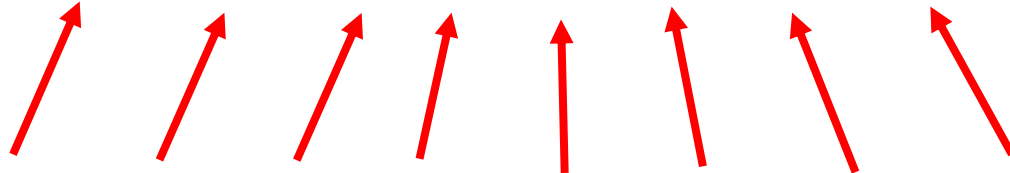
NOTE You should consider all code fragments in this document as pseudocode and not as code that you can cut and paste into your program. I did not compile the code fragments. **If you ignore this advice, you are living dangerously!**

FIRST STEP

In case you missed it on the previous page, The first step is to write your parser. You should finish writing the parser completely and test it by **passing ALL test cases on the submission site before attempting to implement anything else.** I cannot emphasize this strongly enough. I have seen many projects in the past that suffered because the parser was not written first.



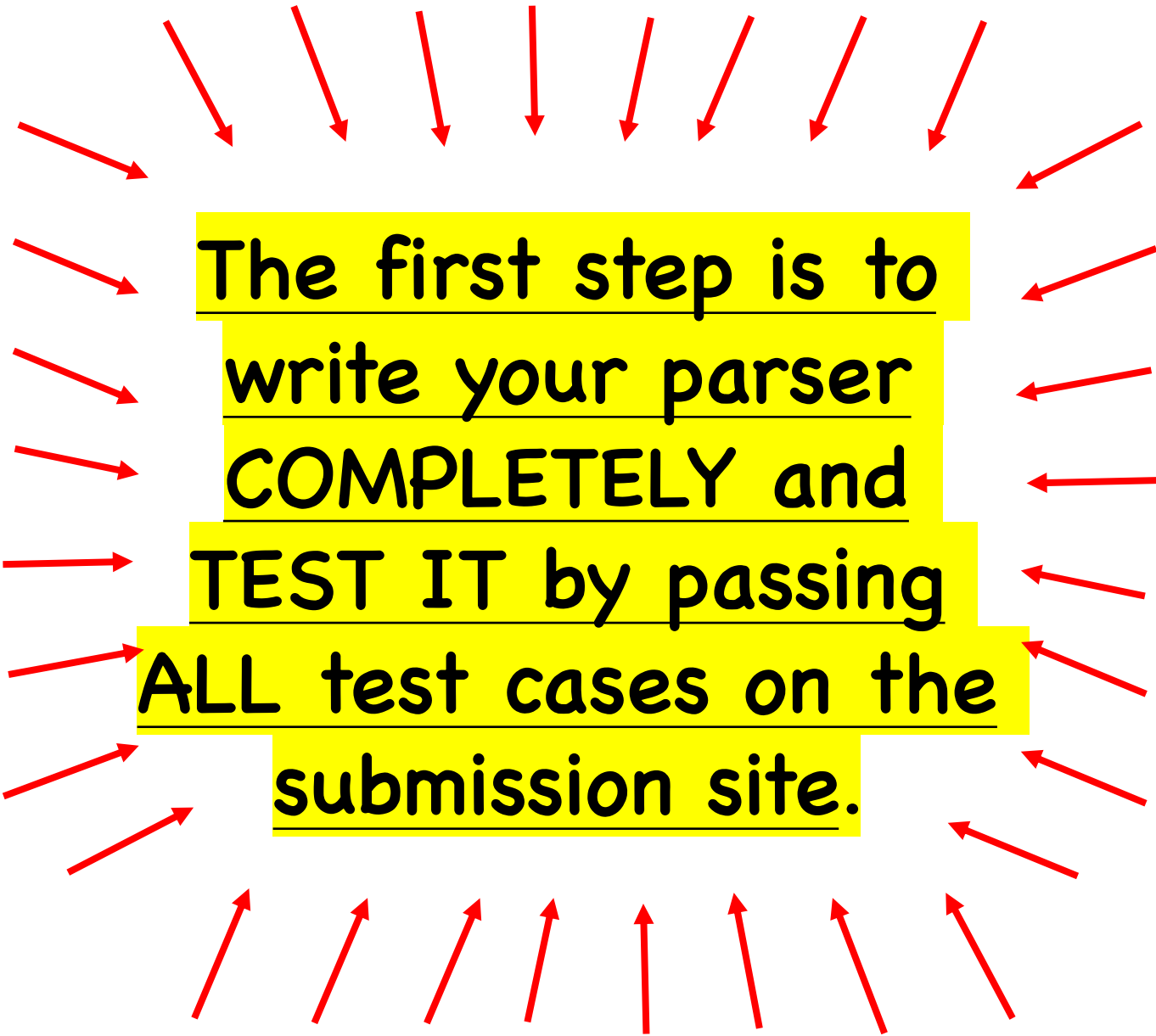
**The first step is to write your parser
COMPLETELY and TEST IT by passing
ALL test cases on the submission site.**



I have seen many projects in the past that suffered because the parser was not written first.

NOTE You should read this document at least a couple of times to get a good understanding.

NOTE You should consider all code fragments in this document as pseudocode and not as code that you can cut and paste into your program. I did not compile the code fragments. **If you ignore this advice, you are living dangerously!**



The first step is to
write your parser
COMPLETELY and
TEST IT by passing
ALL test cases on the
submission site.

ABOUT WRITING THE PARSER

- Unlike the first two projects, the grammar for this project is involved.
- The parser will require two lookaheads for some rules. So for some parts you need `t1 = lexer.peek(1)` and `t2 = lexer.peek(2)` to determine which righthand side to parse
- Remember that there is no partial credit for parsing. It is either 30 points or 0.
- The best advice I can give you is to write the parser systematically like we learned in class
- You will encounter a lot of issues if you try to take shortcuts
- If you feel that you have not properly learned predictive recursive descent parsing, you should study the notes before writing your parser
- Remember, when you are deciding which righthand side to parse, you should check the value that the tokens should be equal to not the values they are different from. For example, you write

```
if (t.tokenType == TYPE1)
```

```
.....
```

```
else if (t.tokenType == TYPE2) | (t.tokenType == TYPE3)
```

```
....
```

do not write

```
if (t.tokenType == TYPE1)
```

```
.....
```

```
else if (t.tokenType != TYPE4) & (t.tokenType != TYPE5)
```

```
....
```

Steps to implement the project

These steps are not exhaustive but they cover many of the important implementation points. I will describe each of these steps in more details in the remainder of this document.

1. **The first step is to write your parser.** You should finish writing the parser before attempting to do anything else. I cannot emphasize this strongly enough. I have seen many projects in the past that suffered because the parser was not written first.
2. The second step is to implement semantic error checking. This is not discussed in this document
3. **The third step is to implement functionality for supporting polynomial declarations and the START section.**

2.1 Memory Allocation. After writing the parser, you should implement a function to allocate memory for variables, this will be useful for other functionality.

2.2 Storing Inputs. You should implement functionality to store the inputs internally in a data structure that can be accessed later when the program is executed

2.3 Polynomial Declaration. Implement code stores a representation of the polynomial declaration in a data structure that can be executed later. This is done while parsing.

2.4 INPUT Statements. implement the code that will transform an INPUT statement to an intermediate representation that can be executed later

2.5 Statement List. Implement the code that will transform a list of statements into a linked list that can be executed later.

2.6 Execution. implement an execute function that takes the data structure generated in 2.3 and 2.4 as a parameter and accesses the stored "inputs" to simulate the execution of the program and generate the output of the program

NOTE You should read this document at least a couple of times to get a good understanding.

NOTE You should consider all code fragments in this document as pseudocode and not as code that you can cut and paste into your program. I did not compile the code fragments!

Allocating Memory for Variables

Variables in polynomial declarations are formal variables and do not need memory allocated for them. On the other hand, all variables in the START section need memory to be allocated to them. In this section I explain how the memory allocation should be done.

Let us consider the parse function for INPUT statements. The function will return a data structure which is a representation of the statement that can be later "executed".

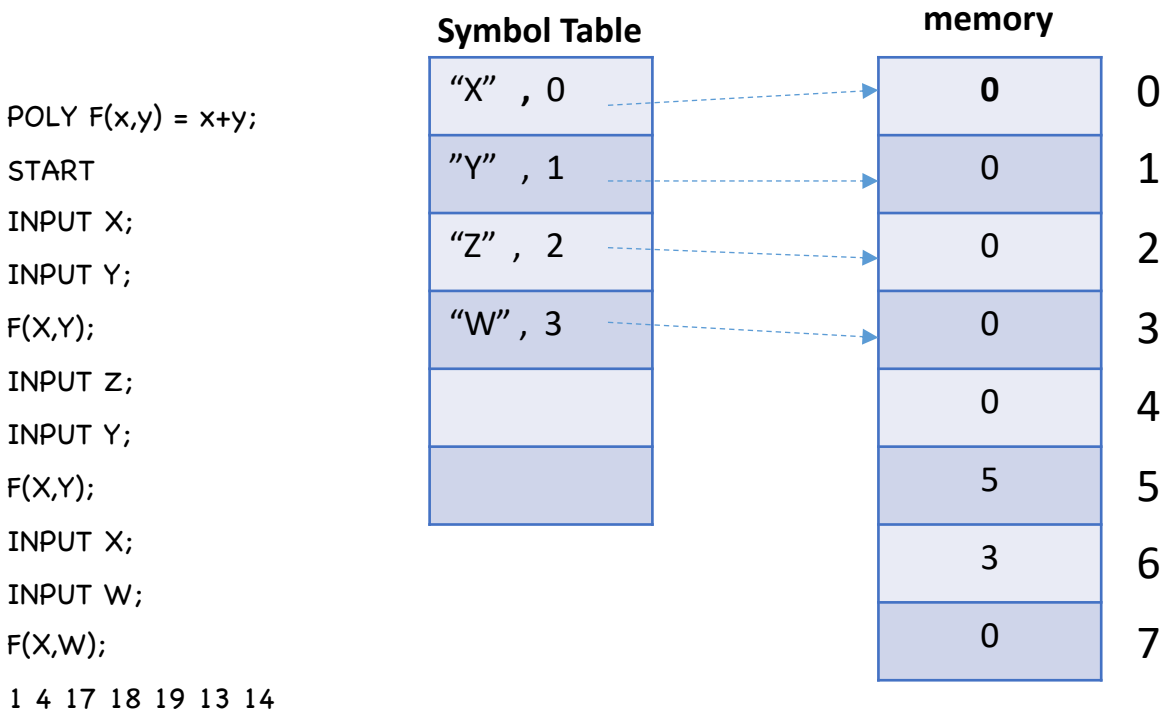
```
struct stmt * parse_input_statement()
{
    Token t;

    struct stmt* st = new stmt;
    expect(INPUT);
    t = expect(ID);

    // at this point t contains the name of a variable.
    // There are two possibilities
    // 1. the name has previously appeared during parsing in which
    //    case memory has been allocated to the variable, or
    // 2. this is the first time we see the name, in which case we
    //    need to allocate memory for it
    // so, we need to lookup the name. If we find it, nothing needs
    // to be done as far as memory allocation is concerned, otherwise
    // memory needs to be reserved for the variable remember
    // that the actual name of the variable is in t.lexeme, so, your
    // program will lookup if t.lexeme is already in the symbol table
    // (see next page for details on the symbol table) if loc is the
    // location associated with t.lexeme, then the statement
    // node will have st->stmt_type = INPUT and st->op1 = loc
    // st is returned by the function (see later)
    expect(SEMICOLON);
}
```

So, how do we allocate memory for variables? See next page!

Allocating memory for variables



The illustration on the right shows variables x, y, z, and w with locations 0, 1, 2, and 3 allocated to them. The allocation is noted in the symbol table.

Notice that the entries are initialized to zero but initialization is not really needed because a variable cannot be used in a polynomial evaluation if there is no preceding INPUT statement for the variable (see "Error Codes").

Now, that we know what should go in the symbol table, the question is: where is that functionality implemented in the parser? We have already seen where that functionality should go for an input_statement (see previous page)

How to do allocation? The allocation itself is straightforward. You keep a global counter variable in your parser called `next_available`. When you need to allocate space you insert (`t.lexeme`, `next_available`) in the symbol table and you increment `next_available` so that the next allocation will be to the next location in memory.

Note. There is a difference between program variables that have memory allocated to them and between polynomial parameters that do not have memory allocated to them.

Storing inputs

POLY $F(x,y) = x+y$;

START

INPUT X;

INPUT Y;

$F(X,Y)$;

INPUT Z;

INPUT Y;

$F(X,Y)$;

INPUT X;

INPUT W;

$F(X,W)$;

1 4 17 18 19 13 14

Symbol Table

"X" , 0
"Y" , 1
"Z" , 2
"W" , 3

memory

0	0
0	1
0	2
0	3
0	4
5	5
3	6
0	7

In the example above, we have shown how to allocate memory to variables.

One other task that we need to do is to store the sequence of input values in an array or vector so that they can be accessed later when the program is "executed"

This can simply be achieved by storing the input values successively in an array or a vector. Here I illustrate the inputs stored in an array.

The C++ `atoi()` function (lookup its documentation) can be used to obtain the integer values that will be stored in the array (it gives the integer value corresponding to a string).

inputs

1
4
17
18
19
13
14

Representing Polynomial Declarations

As your parser parses a polynomial declaration, it needs to represent it internally as a data structure that can be used later to help in evaluating the polynomial. In what follows, I explain how this data structure can be build piece by piece. The approach will be as follows:

1. When a polynomial header is parsed, a list containing the names of the parameters is returned
2. When a coefficient is parsed, the integer value for the coefficient is returned
3. When an exponent is parsed, an integer representing the exponent is returned
4. When a monomial is parsed, a struct consisting of two fields is returned: the first field is the order of the monomial variable in the list of variables for the polynomial and the second field is the power to which it is raised
5. When a monomial list is parsed, a list containing the information of each monomial in the list is returned
6. When a term is parsed, the coefficient and the monomial list (if any) is returned in a struct
7. When a polynomial body is parsed, a data structure containing information about each term and the operator of the polynomial (PLUS/MINUS) is returned

```
int parse_coefficient()
{
    Token t;
    t = expect(NUM);
    return atoi(t.lexeme);    // you need to lookup information
                             // about atoi() and how to use it
                             // this might not work as written
}
```

```
struct monomial * parse_monomial()
{
    // A monomial consists of a variable name (parameter) and an optional exponent. This function
    // parses a monomial and return a pointer to monomial struct that contains two integers:
    // one integer representing the order in which the variable appears in the list of parameters
    // and one integer representing the exponent. To transform variable names to integers,
    // this function will need to have access to the list of parameter. This is done through
    // the polynomial data structure (see next slide).
    //
    // Note that you do not need to store the name of the parameter in the representation of the
    // monomial. It is enough to store the order in which it appears because this is all the information
    // that is needed in order to determine how a particular argument contributes to the evaluation of
    // the polynomial
}
```

Representing Polynomial Declarations

```
struct monomial_list * parse_monomial_list()
{
    // parse the monomial list and return pointer to the list
}

struct term * parse_term()
{
    // parse the term and return a structure containing the coefficient and
    // the monomial list
}

struct polynomial_body * parse_polynomial_body()
{
    // parse the polynomial body and return a list representing the polynomial and
    // its operators.
}
```

Representing Polynomial Declarations

At the top level, we need to maintain a list of polynomial. The list can be kept in a table in which each entry will have the polynomial name and the list of parameters (the polynomial header) and the representation of the polynomial body

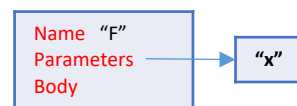
Initially, the first thing that is parsed is the polynomial header: polynomial name and parameter list. When the header is parsed, information about it is added to a polynomial declaration table. The table is a simple vector to which you should keep track of the last added entry. This information can be used by the `parse_monomial()` function to determine the order in which a variable appears in the list of parameters of the polynomial.

These ideas are best illustrated by an example. Consider the declarations

$$F = x^2 + 5;$$

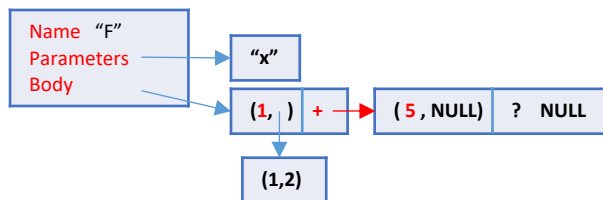
$$G(x,y,z) = 2 y^3 x^4 + 3 x z^2;$$

When F is parsed, we have the following in the polynomial table



Notice how the list of parameters is given as "x". This is the default when there is no explicit parameter list given.

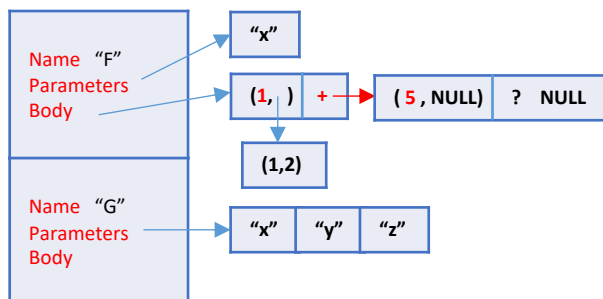
After $x^2 + 1$ is parsed, the new polynomial representation becomes the following



This requires some explanation

- $(1, [(1,2)], +, \rightarrow)$: The 1 is the coefficient. The monomial list has only one element representing the monomial "x". The 1 is the order of "x" in the list of parameters since "x" is the first parameter. The 2 is the exponent of "x". The + represents the + symbol in the + term_list for this polynomial and the is a pointer to a term list which represents the term list representing the constant 5 in the polynomial (the constant 5 is a term list containing one element).
- $(5, [], ?, NULL)$: The 5 is the coefficient. Since there is no monomial list, the monomial list is empty. The ? represents the value of the operator field which is not set because there is no next term list (next term list is NULL)

After parsing $G(x,y,z)$, the representation becomes as follows



Note: this representation assumes that variables are counted starting from index 1. for instance, "x" is the first parameter and is represented with 1. In your implementation, you will probably start from index 0 and you should take into account.

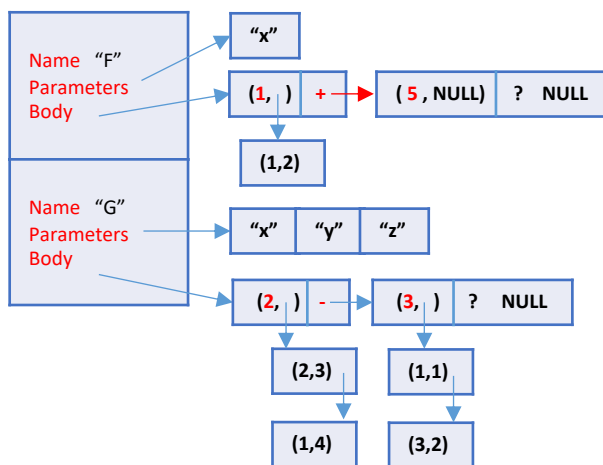
Representing Polynomial Declarations

We continue with our example:

$$F = x^2 + 5;$$

$$G(x,y,z) = 2 y^3 x^4 - 3 x z^2;$$

After parsing the body of $G(x,y,z)$, is parsed, the representation becomes as follows



Let us examine each part of the body

- $(2, [(2,3),(1,4)], -,)$: The **2** is the coefficient in front of $y^3 x^4$. The first element in the list of monomials is (2,3) and represents y^3 because y is the second parameter in the list of parameters and it is raised to the power 3. (1,4) represents x^4 because x is the first parameter in the list of parameters and it is raised to the power 4. The **-** represents the minus sign in **- term_list** for this polynomial.
- $(3, [(1,1),(3,2)], ?, NULL)$: The **3** is the coefficient in front of $x z^2$. The first element in the list of monomials is (1,1) and represents x which is the first parameter in the list of parameters and whose (implicit) exponent is 1. The second element in the list of monomial is (3,2) and represents z^2 . The second part **?, NULL** is as in the previous example

Note: this representation assumes that variables are counted starting from index 1. for instance, "x" is the first parameter and is represented with the number 1, "y" is the second parameter and is represented with the number 2 and so on. In your implementation, you will probably start from index 0 and you should take that into consideration.

Representing the program as a linked list

The goal of this step is to represent the program as a data structure that can be easily “executed” later. I first show what the data structure looks like for our example program

POLY F(x,y) = x+y;

START

INPUT X;

INPUT Y;

F(X,Y);

INPUT Z;

INPUT Y;

F(X,Y);

INPUT X;

INPUT W;

F(X,W);

1 4 17 18 19 13 14

stmt_type the statement type is an integer that indicates the type of the statement. It can be INPUT or POLYEVAL (you can use whatever names you want as long as you distinguish between the two case)

poly_eval This field is relevant only if the statement type is POLYEVAL. The value is a data structure that represents the polynomial evaluation statement which I have already described.

variable This field is relevant only if the statement type is INPUT. It is an integer which is equal to the index of the variable in the INPUT statement.

data structure for a statement

```
int stmt_type
struct poly_eval *
int variable
struct stmt * next;
```

You should declare the data structures for statements yourself in your program. It is not provided with the provided code.

Now that we know how a statement is represented, we can look on the next page at the representation of the program above

Representing Polynomial Evaluation

Polynomial evaluation statements are the most involved data structure in the program, but they are not really hard to work with. A polynomial evaluation needs the following information

1. A reference to the polynomial. This can be the index of the polynomial entry in the polynomial declaration table
2. The list of arguments. An argument to a polynomial evaluation can be either a NUM, or an ID or a polynomial evaluation. We discuss each case
 1. NUM. If the argument is NUM, then the actual value of the argument needs to be computed at compile time and stored in the polynomial evaluation statement
 2. ID. If the argument is the name of a variable, then the index of the memory entry for the variable must be stored in the argument
 3. Polynomial Evaluation. If the argument is a polynomial evaluation, then the a pointer to a polynomial evaluation data structure (recursive representation).

It should be clear that, in order to distinguish between the various kinds of arguments, we will need a flag that specifies the argument type. I will assume that the flag takes the values NUM, ID, and POLYEVAL (not defined and you will need to add it in the parser). I illustrate these ideas with an example program on the next page for the following program

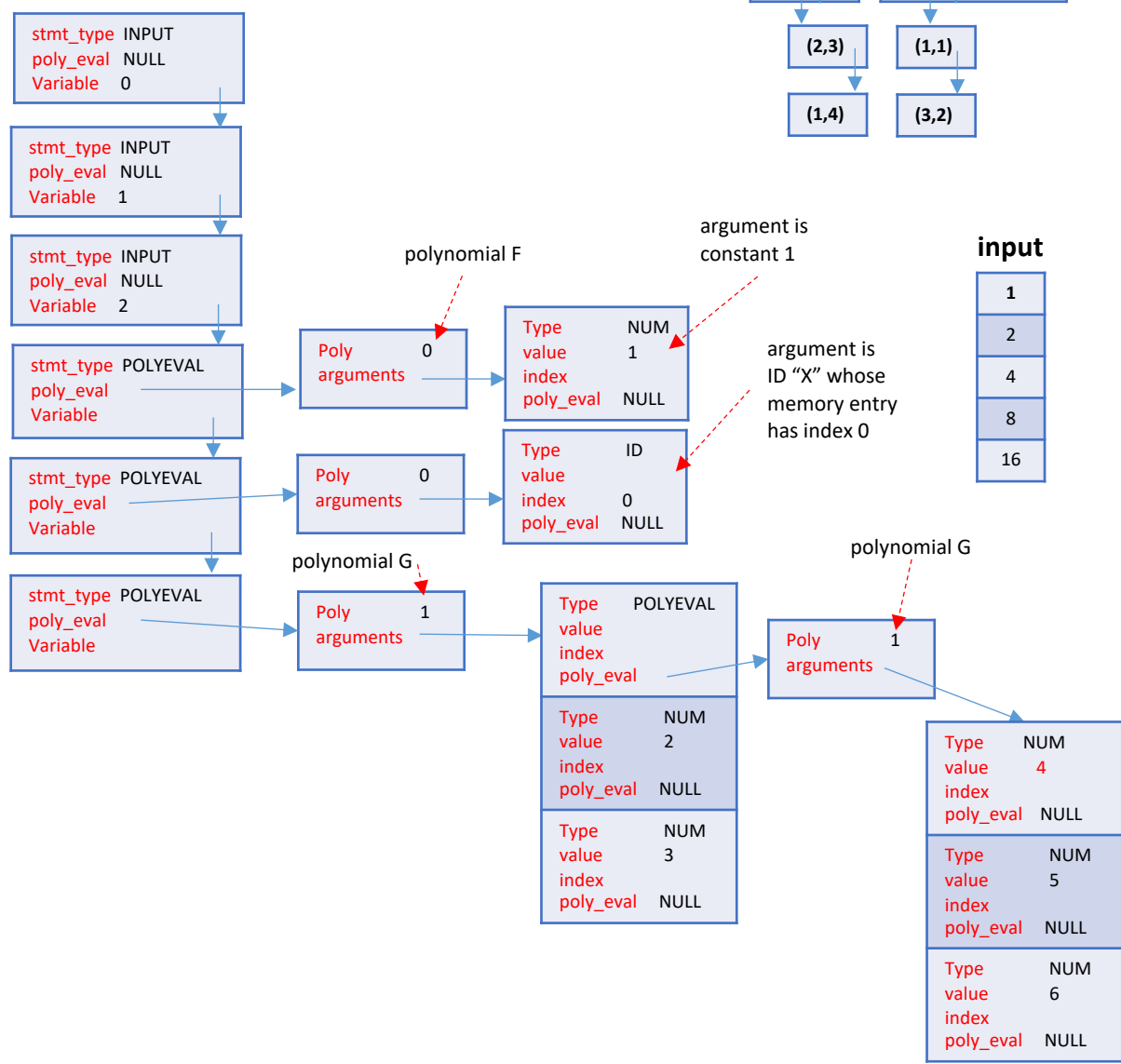
```
F = x^2 + 1;  
G(x,y,z) = 2 y^3 x^4 + 3 x z^2;  
START  
INPUT X;  
INPUT Y;  
INPUT Z;  
F(1);  
F(X);  
G(G(1,2,3),2,3);  
1 2 4 8 16
```

Representing Polynomial Evaluation: Example

Consider the following program

```
F = x^2 + 1;  
G(x,y,z) = 2 y^3 x^4 + 3 x z^2;  
START  
INPUT X;  
INPUT Y;  
INPUT Z;  
F(1);  
F(X);  
G(G(4,5,6),2,3);  
1 2 4 8 16
```

The program will be represented as follows



Generating a statement list

So far, we discussed how to generate the representation of individual statements. One issue that remains is how to generate the list of statements. This is explained in the following code fragment

```
struct stmt* parse_statement_list()
{
    struct stmt * st;
    struct stmt * stl;
    ...
    st = parse_stmt();

    // if there no more input for the statement list
    return st;

    // if there is more input
    stl = parse_statement_list();
    // append stl to st
    // return st
}
```

Executing the program

So far, we have discussed what the parser should do. Here are the things that the parser should do:

1. build the symbol table
2. build the polynomials tables
3. generate the linked list that represents the program.

The actual “execution” of the input program happens after all these steps are done. For that, we introduce the `execute_program()` function.

The `execute_program()` function is a function that takes as argument a pointer to the first statement node of the program. Then, `execute_program()` will iterate over the nodes one at a time and “execute” the node. The following is an illustration of the high-level loop of `execute_program()` which also shows how some statements are executed.

```
execute_program(struct stmt * start)
{
    struct stmt * pc;
    int v;
    pc = start;
    while (pc != NULL) {
        switch (pc->stmt_type) {
            case POLYEVAL:    v = evaluate_polynomial(pc->poly_eval);
                               cout << v << endl;                // endl is std::endl
            case INPUT:      mem[pc->variable] = inputs[next_input];
                               next_input++;
                               break;
            ....
        }
        pc = pc->next;
    }
}
```

This code fragment captures the essential of `execute_program()`. One variable to highlight is the `next_input` variable which keeps track of how many inputs have been consumed. Note how `next_input` is incremented after an input is consumed (by executing an INPUT statement).

Evaluating Polynomials

What remains to be done is to evaluate the polynomial statements. In order to evaluate the polynomial, we need to first evaluate the arguments. If the argument is NUM or ID, the evaluation is straightforward. If the argument is POLYEVAL, then the evaluation function is called recursively to get the value of the argument*. So, the only remaining part is to explain how to evaluate a polynomial once all arguments have been evaluated.

The evaluation function will have all the argument values stored in a vector and will go to the representation of the polynomial to be evaluated and evaluate each component of the polynomial using the arguments already calculated. I omit the details which you should work on figuring out.

* In a real compiler, procedures are not executed recursively. They are executed iteratively. Also, they have parameters. This makes procedure execution significantly more involved than what we are able to do in a first project.