PROJECT 1
IMPLEMENTATION GUIDE

CSE 340 FALL 2025

Rida Bazzi

This document is provided to help you with the implementation

The first step is to write your parser. You should finish writing the parser completely and you should test it thoroughly 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 second step is to write the semantic error checking functionality: do not work on "execution" functionality (Task 2) and the "polynomial combining and reordering" functionalities (Tasks 3, 4, and 5) before you are done with semantic error checking!

The third step is to write "monomial combining and reordering" functionality (Task 3): This will provide a good foundations for the other tasks.

After that, it is up to you how to proceed with the other functionalities. If you are done with Task 3, you are in a good position to tackle the remaining tasks (Tasks 2, 4 and 5)

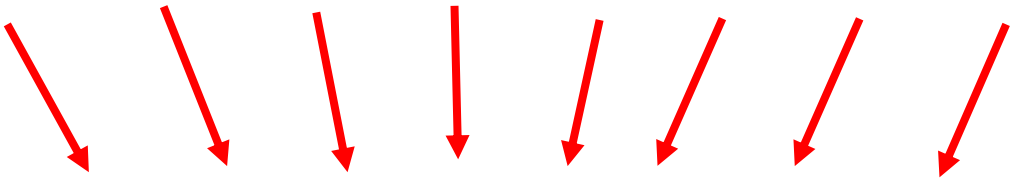
That said, you should have a clear idea about what the various parts entail before you start coding.

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

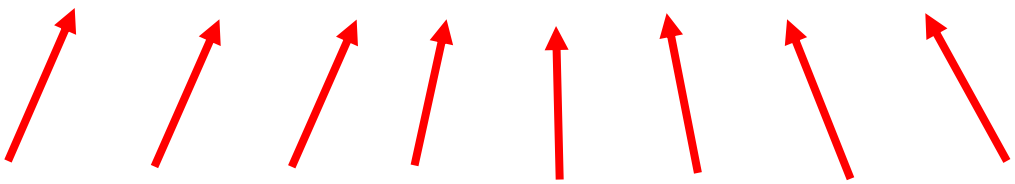
You should treat all code fragments in this document as pseudocode and NOT as code. You should not cut and paste code fragments from this document into your program. I did not compile the code fragments!

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 systematically and test thoroughly before attempting to write anything else. I cannot emphasize this strongly enough.



The first step is to write your parser COMPLETELY and TEST IT thoroughly.



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

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

NOTE: You should treat all code fragments in this document as pseudocode. You should not cut and paste code fragments from this document into your program. I did not compile the code fragments!

Content

Major Components

The major components of the project are the following (note that the total is 115%, so there are 15% bonus points)

1. Error Checking
    1. Syntax checking (30%): 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 carefully as a first step.
    2. Semantic error checking (30%): This part has quite a few grade points allocated to it, but it is not the most involved part. If you understand what needs to be done and you read the suggestions on how to do it, you should be able to finish it in a relatively short time. That said, start early!

2. Program Execution (15%): To implement the program execution functionality, you need some supporting functionality:

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

    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 that parses a polynomial declaration and stores a representation of the polynomial in a data structure that can be executed later.

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

    2.5 OUTPUT Statements. implement the code that will transform an OUTPUT statement into an intermediate representation that can be executed later

    2.6 Assignment Statements. implement the code that will transform an assignment statement into an intermediate representation that can be executed later.

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

    2.8 Execution. implement an `execute_program()` function that takes the data structure generated in 2.7 as an argument to simulate the execution of the program and generate the output of the program

3. Polynomial rewriting (40%) There are three polynomial rewriting tasks. The simplest asks that you rewrite monomial lists, the next asks that you combine monomial lists and the last asks that you fully expand the polynomial based on the previous two tasks.

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!

Parsing (1/2)

The easiest way to write the parser is to write one parsing function for each non-terminal and make sure that when that function is called, it consumes part of the input that corresponds to the non-terminal. This was covered in the lectures. Make sure you understand the lecture material before starting. I will repeat some of the information covered in the lectures by showing how to write the parse functions for some of the non-terminals of the project's grammar.

Remember that code in this document has not been compiled. You should not treat it as code that you can cut and paste into your program.

```
void parse_input()
{  // the input consists of a program followed by nothing

   parse_program();
   expect(END_OF_FILE);
}

void parse_program()
{  // program -> tasks_section poly_section execute_section inputs_section

   parse_tasks_section();
   parse_poly_section()
   parse_execute_section();
   parse_inputs_section();
}
```

The non-terminal `program` has only one rule. To parse the program, we need to consume a sequence of tokens that corresponds to `program`. Remember that we are writing our parsing functions so that when parse_X() is called, it consumes part of the input that corresponds to X. The rule for `program` is of the form

A -> B C D E

For such a rule, the sequence of tokens corresponding to `A` would consist of a section corresponding to `B` followed by a sequence corresponding to `C` followed by a sequence corresponding to `D` followed by a sequence corresponding to `E`. So, to parse `A`, we need to consume a sequence corresponding to B, then consume a sequence corresponding to `C`, then consume a sequence corresponding to `D` and finally consume a sequence corresponding to `E`. This is achieved by calling

```
parse_B();
parse_C();
parse_D();
parse_E();
```

This is the same situation we have for the non-terminal `program`.

Let us look at a couple more example on the next page.

Parsing (2/2)

Next, we will consider the rules for the non-terminal `poly_decl`

```
void parse_poly_decl()
{ // poly_decl -> poly_header  EQUAL poly_body SEMICOLON

  parse_poly_header();
  expect(EQUAL);
  parse_poly_body();
  expect(SEMICOLON);
}
```

This is similar to the case of `program`. We have only one rule for `poly_decl`, so to parse `poly_decl`, we need to consume a sequence of tokens corresponding to the right-hand side of the single rule for `poly_decl`. Unlike the example of `program`, here the right-hand side has terminals (tokens). We have already seen that to consume part of the input corresponding to a non-terminal `X`, we call `parse_X()`. To consume a particular terminal (token), we use the `expect()` function (provided). The code for `parse_poly_decl()` is color coded and shows how each call corresponds to the right-hand side of the rule for `poly_decl`.

Note: you must have noticed that I include the rules for the non-terminals as a comment in the code for the non-terminal. This is not done only to explain how parsing is done. This is done to avoid mistakes and is good practice in general. I recommend that you start by writing one parse function for each non-terminal and include in it the rules for that non-terminal. Then you start writing the bodies of these functions one by one by following the rules.

I am going to look at one more example before finishing the discussion of parsing. We will consider the non-terminal `term`. This non-terminals has four rules. To decide which rule to follow to do the parsing, we call `peek(1)` which returns the upcoming token but does not consume it. In general, `peek(k)` will return the `k`'th token ahead.

```
void parse_primary()
{  // term -> monomial_list
   // term -> coefficient monomial_list
   // term -> coefficient
   // term -> parenthesized_list

   Token t1 = lexer.peek(1);
  if (t.token_type == ID)
       parse_monomial_list();
   else if (t.token_type == NUM) {
       parse_coefficient();
       Token t2 = lexer.peek(1);
       if ( . . . )
            . . .
       else
           . . .
   } else if (t.token_type == LPAREN) {
       parse_parenthesized_list();
   } else
       syntax_error();
}
```

Notice how this code proceeds. First, it peeks at the next token. If it is `ID`, then it determines that the rule for primary must be `primary -> monomial_list` because only `monomial_list` amongst the four righthand sides can correspond to a sequence of tokens starting with `ID`. So it parses accordingly by calling `parse_monomial_list()`. If `peek(1)` returns `NUM`, then it determines that it must be either `term -> coefficient` or `term -> coefficient monomial list`. So, we start by calling `parse_coefficient()` which will consume `coefficient`. Then, we need to call `peek(1)` again to determine if there is a `monomial_list` after the coefficient. I leave that for you to finish. The last case to consider is when `t1.token_type` is `LPAREN`. In that case, we must parse according to the rule `term -> parenthesized_list` and we call `parse_parenthesized_list()`. If the token is not `ID`, `NUM` or `LPAREN`, then `syntax_error()` is called.

Note. For some rules, we might need to peek for two tokens ahead to determine which right-hand side to parse. For this project's grammar, this is the case for some non-terminals.

Semantic Error Checking (1/3)

The semantic error checking functionality asks you to determine if

1. A polynomial is declared more than once
2. A name (ID) appears in the body of a polynomial declaration but not in the list of parameters to the polynomial.
3. A polynomial evaluation uses a name of a polynomial that was not declared
4. A polynomial evaluation does not have the correct number of arguments.

For the first case, we have *conflicting declarations*. For cases 2,3 and 4, we have *uses* of names that do not match declarations of names. To support these functionalities, we need to store information about the declarations, as they are parsed and, when a particular name is parsed, we check it against the stored information. I will explain in detail how to check for polynomial declared more than once and give a high-level explanation on how to check for the other semantic errors.

Read this section fully before using any of the pseudocode. The presentation is meant to explain how and why we represent the data structures as we do. The initial choices on page 2/3 are modified on page 3/3 !

Let us consider the rules for the polynomial declarations section (`decl` in the names stands for declaration)

```
poly_section  -> POLY poly_decl_list
poly_dec_list -> poly_decl
poly_dec_list -> poly_decl poly_dec_list
poly_decl     -> poly_header EQUAL poly_body SEMICOL
poly_header   -> poly_name
poly_header   -> poly_name LPAREN id_list RPAREN
id_list       -> ID
id_list       -> Id COMMA id_list
```

The code for `parse_poly_decl_list()` looks as follows

```
void parse_poly_decl_list()
{  // poly_dec_list -> poly_decl
   // poly_dec_list -> poly_decl poly_dec_list

   parse_poly_decl(); // parses one declaration
   Token t = lexer.peek(1);
   if (t.token_type == ID) { // ID is the start of a new poly_decl_list
       parse_poly_decl_list();
   } else if (t.token_type == EXECUTE) // polynomial declaration list
       return;                         // is followed by EXECUTE
   else
       syntax_error();
}
```

Note that the part highlighted in yellow can be omitted for this function and replaced with `return`. because after the call to `parse_poly_section()` returns, `parse_execute_section()` will be called by `parse_program()` and the first call that `parse_execute_section()` makes is `expect(EXECUTE)`.

You should be able to see that the polynomials are listed one after another in the `poly_section`. When a particular call to `parse_poly_decl()` is active, it must be handling the next polynomial in the list.

I continue the explanation on the next page.

Semantic Error Checking (2/3)

Now, let us consider the rules for `poly_decl` and `poly_header`

```
poly_decl     -> poly_header EQUAL poly_body SEMICOLON
poly_header   -> poly_name
poly_header   -> poly_name LPAREN id_list RPAREN
```

The code for `parse_poly_decl()` looks as follows

```
void parse_poly_decl()
{  //poly_decl -> poly_header EQUAL poly_body SEMICOLON

   parse_poly_header();
   expect(EQUAL);
   parse_poly_body();
   expect(SEMICOLON);
}
```

This does not look different from what we have seen so far, but it only parses the input and does not store information about what is parsed. We need to store the polynomial header information for the list of polynomials. So, we need `parse_poly_header()` to return a data structure that represents the header. The following data structure could be used

```
struct poly_header_t {
     Token name;
     vector<Token> id_list;
}
```

This representation is somewhat redundant, but it will do (we don't really need the `token_type` field in the `Token` struct). `parse_poly_header()` will look like this

```
struct poly_header_t parse_poly_header()
{
    struct poly_header_t header;
    header.name = parse_poly_name();
    Token t = lexer.peek(1);
    if (t.token_type == LPAREN)
    {   expect(LPAREN);
        header.id_list = parse_id_list();
       expect(RPAREN);
    } else
        header.id_list = make_list("x");
    return header;
}
```

Now, we can write

```
struct poly_decl_t parse_poly_decl ()
{  //poly_decl -> poly_header EQUAL poly_body SEMICOLON

   struct poly_decl_t decl;
   decl.header = parse_poly_header();
   expect(EQUAL);
   decl.body = parse_poly_body();
   expect(SEMICOLON);
   return decl;
}
```

Semantic Error Checking (3/3)

The code on the previous page used `struct poly_decl_t` but did not declare it. The struct will contain a header which is of type `poly_header_t` and a `poly_body` which we will discuss later.

Going back to `parse_poly_decl_list()`, we have

```
void parse_poly_decl_list()
{  // poly_dec_list -> poly_decl
   // poly_dec_list -> poly_decl poly_dec_list

   struct poly_dec = parse_poly_decl(); // parses one declaration
   . . . ? . . .
   Token t = lexer.peek(1);
   if (t.token_type == ID) // ID is the start of a new poly_decl_list
   {
       parse_poly_decl_list();
       . . . ? . . .
   }
}
```

We need to store these declarations in one list or vector of polynomial declarations. In general, when we are constructing a data structure recursively with a recursive descent parser, we need to create the data structure dynamically. That will be needed for creating the representation of the polynomial body for execution, but it is not really needed for the sequence of polynomial declarations. Since the declarations are listed one after the other and there are no nested declarations, we can have a global vector to store all the declarations. The code can be:

```
vector<poly_decl_t> polynomials; // global data structure

void parse_poly_decl_list()
{  // poly_dec_list -> poly_decl
   // poly_dec_list -> poly_decl poly_dec_list

   struct poly_decl = parse_poly_decl(); // parses one declaration
   polynomials.push_back(poly_decl);
   Token t = lexer.peek(1);
   if (t.token_type == ID) // ID is the start of a new poly_decl_list
   {
       parse_poly_decl_list();
   }
}
```

The highlighted code pushes the declaration into the global `polynomials` vector. Actually, this single line of highlighted code will have the effect of pushing ALL the declarations onto the `polynomials` vector! You should try to trace the code by hand to get a better understanding of what is happening. If you don't have a good understanding, you will not be able to do the other parts of the project.

Pushing the the polynomial declaration to the vector easily allows us to check for duplicate polynomial names. Before adding `poly_decl` to the vector, we can check if the polynomial name in the `poly_header` of the `poly_decl` struct is already in the vector `polynomials`. You will need to write the code to check for duplicates.

The code as written would not be adequate to support "undeclared monomial" error because when `parse_poly_body()` is called from `parse_poly_decl()` (see previous page), the `poly_decl` struct is not available in the global vector. One way to get around this is to pass the `header` (see previous page) to `parse_poly_body()`. Alternatively, you could push the `header` info to the global vector directly in `parse_poly_decl()` (pushing a `poly_decl` with only the header part filled and with the `poly_body` field empty).

The data structures discussed here should also be enough to check for undeclared polynomial name and for wrong number of arguments (the number of arguments would need to be calculated when `parse_id_list()` is called, and the number stored in the `poly_decl` struct.

Program Execution

Supporting the program execution functionality (Task 2) as well as the "polynomial rewriting tasks" requires more involved data structures that is more involved than semantic error checking because you need to be able to represent the body of the polynomial in a recursive data structure. In addition, for Task 2, you need to represent the EXECUTE section in a data structure that will be *executed* when parsing is completed.

For Task 2, you will need to:

1. Represent the polynomial body in a data structure that can be used later for execution
2. Allocate locations to variables (in the EXECUTE section) so that each variable has a fixed locations to store its value (for INPUT or assignment statements) and to read its value (for OUTPUT statements).
3. Represent the EXECUTE section in a data structure (linked list of statements)
4. Call an `execute_program()` function that will execute the data structure from item 3 above

In what follows, I explain how to support various aspects of the execution functionality.

Allocating memory for variables

All variables in the EXECUTE section need memory to be allocated to them. In this section I explain how 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 "executed" later.

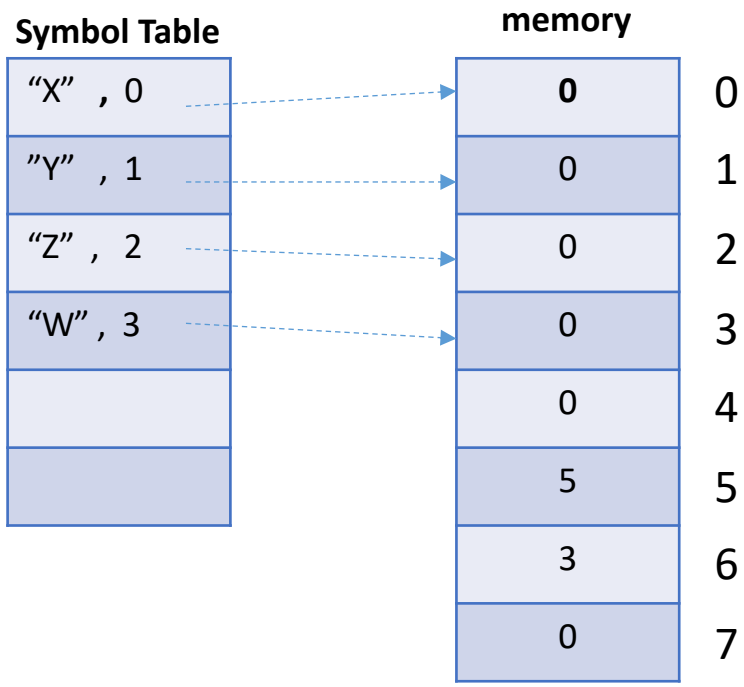
```
struct stmt_t * parse_input_statement()
{       // input_statement -> INPUT ID SEMICOLON

        Token t;


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


        // at this point t.lexeme 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 of the variable. If we find it, nothing needs
        // to be done as far as memory allocation is concerned. If we don't find the
        // name, 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, we will have:
        //
        //        st->stmt_type = INPUT; and
        //        st->op1 = loc;
        //
        // st is returned by the function:

        return st;
}
```

Memory allocation is not only done in INPUT statements. Memory allocation is done any time a variable name appears for the first time in the EXECUTE section. This can be on the left-hand side of an assignment, as an argument to a polynomial evaluation or even in an OUTPUT statement.

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

Allocating memory for variables

```
TASKS 1 2
POLY F(x,y) = x+y;
EXECUTE
  INPUT X;
  INPUT Y;
  X = F(X,Y);
  INPUT Z;
  INPUT Y;
  Y = F(X,W);
  INPUT X;
  INPUT W;
  W = F(X,W);
  OUTPUT W;
INPUTS    1 4 17 18 19 13 14
```



The sample program above has 4 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 (can be implemented as a map in C++).

Notice that the entries are initialized to zero.

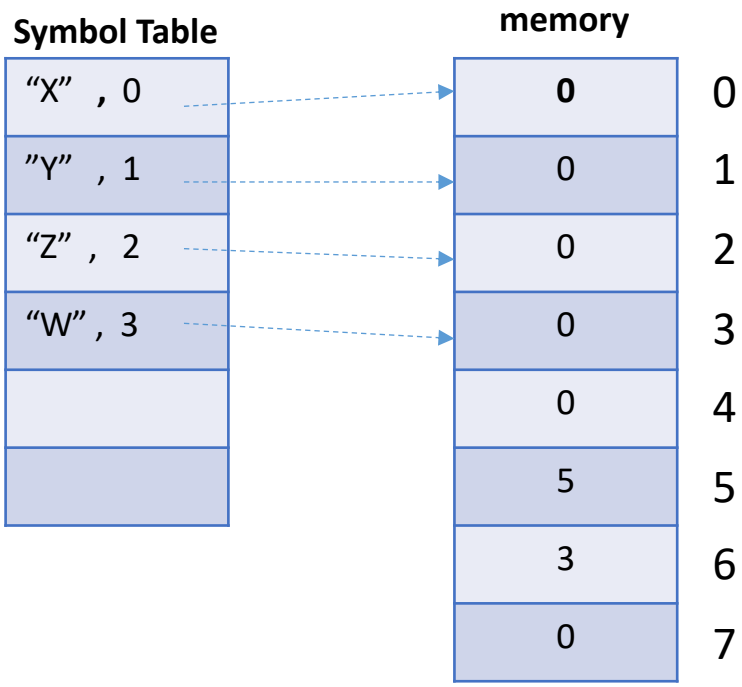
Now, that we know what should go in the symbol table, the question is: where is that functionality implemented in the parser? We have already seen where that functionality should go for an `input_statement` (see previous page). You should also have that functionality in `assign_statement` and in `input_statement`.

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

```
TASKS 1 2
POLY F(x,y) = x+y;
EXECUTE
  INPUT X;
  INPUT Y;
  X = F(X,Y);
  INPUT Z;
  INPUT Y;
  Y = F(X,W);
  INPUT X;
  INPUT W;
  W = F(X,W);
  OUTPUT W;
INPUTS    1 4 17 18 19 13 14
```



On the previous page, we have shown how to allocate memory to variables.

We also need 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 be achieved by storing the input values successively in a vector as shown to the right

The C++ std::stoi() function (lookup its documentation) is used to obtain the integer values that will be stored in the vector

Representing Polynomial Declarations (1/3)

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, the polynomial name and 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
6. When a monomial is parsed, a data structure consisting of two fields is returned. The first field is the `ID` Token and the second field is the exponent (default is 1 is missing).
7. When a monomial list is parsed, we need to build the vector representation of the monomial list (see project specification document). This is achieved by passing the vector by reference to the function and each time a monomial is parsed, the corresponding entry is appropriately updated.
8. When a term is parsed, the coefficient and the monomial list (if any) is returned in a struct
9. 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 std::stoi(t.lexeme); // you need to lookup information
                                  // about std::stoi() and how to use it
}

struct monomial_t parse_monomial()
{
      // A monomial consists of an ID and an optional exponent
      //
      //          struct monomial_t {
      //                  Token ID;
      //                  int exponent;  // equal to 1 if there is no specified
      //                                 // exponent
      //           }

 }
```

A monomial list can simply be a vector of integers as we discussed in the project specification document. A term can be defined as a struct with five fields: (1) A kind field that specifies if the term is a monomial list or a parenthesized list, (2) and operator field to represent the operator to be applied to the term, (3) a coefficient field, (4) a vector of integer to represent the monomial list. This field will be an empty vector if the term is a parenthesized list and (5) a vector of terms lists if the term is a parenthesized list, so a vector of vectors of term nodes. This field is an empty vector if the term is a monomial list.

```
struct TermNode {
   Kind kind;                                  // parenthesized list or monomial list
   Operator op;                                // this applies to parenthesized lists and
                                               // to monomial lists.
   int coefficient;                            // this applies only to monomial lists
   vector<int> monomial_list;                  // this obviously applies only to monomial lists
   vector<vector<TermNode>> parenthesized_list; // this applies only to parenthesized lists
};
```

Note that in the project specification document, the highlighted part was mistakingly omitted.

Representing Polynomial Declarations (2/3)

```
void parse_monomial_list(int<vector> & monomial_list_ptr)
{
          // This function takes as input a vector of integers that represents
          // the monomial list (see project specification document)
          // (passed by reference so that it can be modified)


          // The first thing to do is to parse a monomial and store
          // its information in the vector as explained in the project
          // specification document. To that end, you need to determine the
          // position of the ID so that the appropriate entry in the
          // vector is updated. To determine the position, you need access to the
          // list of parameters.

          // if there is more in the monomial list, then make the recursive
          // call and make sure to pass the vector so that it can be updated.
}

void parse_term(struct Term_t &term )
{
          // parse the term and store the result in the "term" variable
          // provided as an argument
          // Term_t will contain the coefficient and the monomial list vector.
}

void parse_polynomial_body(struct TermList_t &termList)
{
          // parse the polynomial body and store the result in the termList passed
          // by reference
}
```

Representing Polynomial Declarations (3/3)

At the top level, we need to maintain a list of polynomial. The list can be kept in a table (vector) 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.

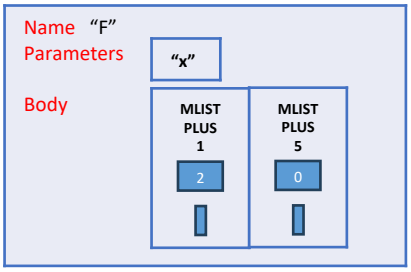
These ideas are best illustrated by an example. Consider the declarations

```
F = x^2 + 5;
G(x,y,z) = 2 y^3 x^4 + 3 x z^2;
H(x,yz) = (1+2x^3)(4+ 5z^6)+7y^8;
```

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. This is just an illustration. In the implementation, you will store the line numbers (full tokens) and as discussed earlier, you can have a separate structures for the header which will contain the polynomial name and the parameter information.

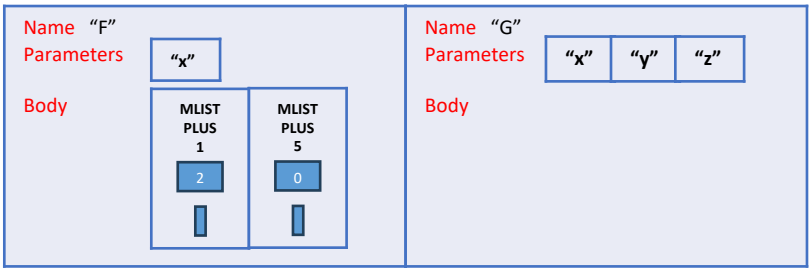
After x^2 + 5 is parsed, the new polynomial representation becomes the following



This requires some explanation

- The body is a vector of terms. In our example, there are two terms, each of which is a monomial list (MLIST).
- Each term has a plus operator before it. Even though the first term x^2 does not have an explicit + before it, its operator is PLUS, which is only meaningful value for the field.
- The first term x^2 has the coefficient 1 by default (there is no explicit coefficient) and the second term has the coefficient 5
- Since there is only one argument, the monomial lists are represented as vectors of size 1. For x^2, the entry contains 2 which is the power of x. For 5, the vector contains 0 which represents x^0 = 1
- Finally, since these are MLIST terms, the final entry for parenthesized list is empty.

After parsing G(x,yz), the representation becomes as follows

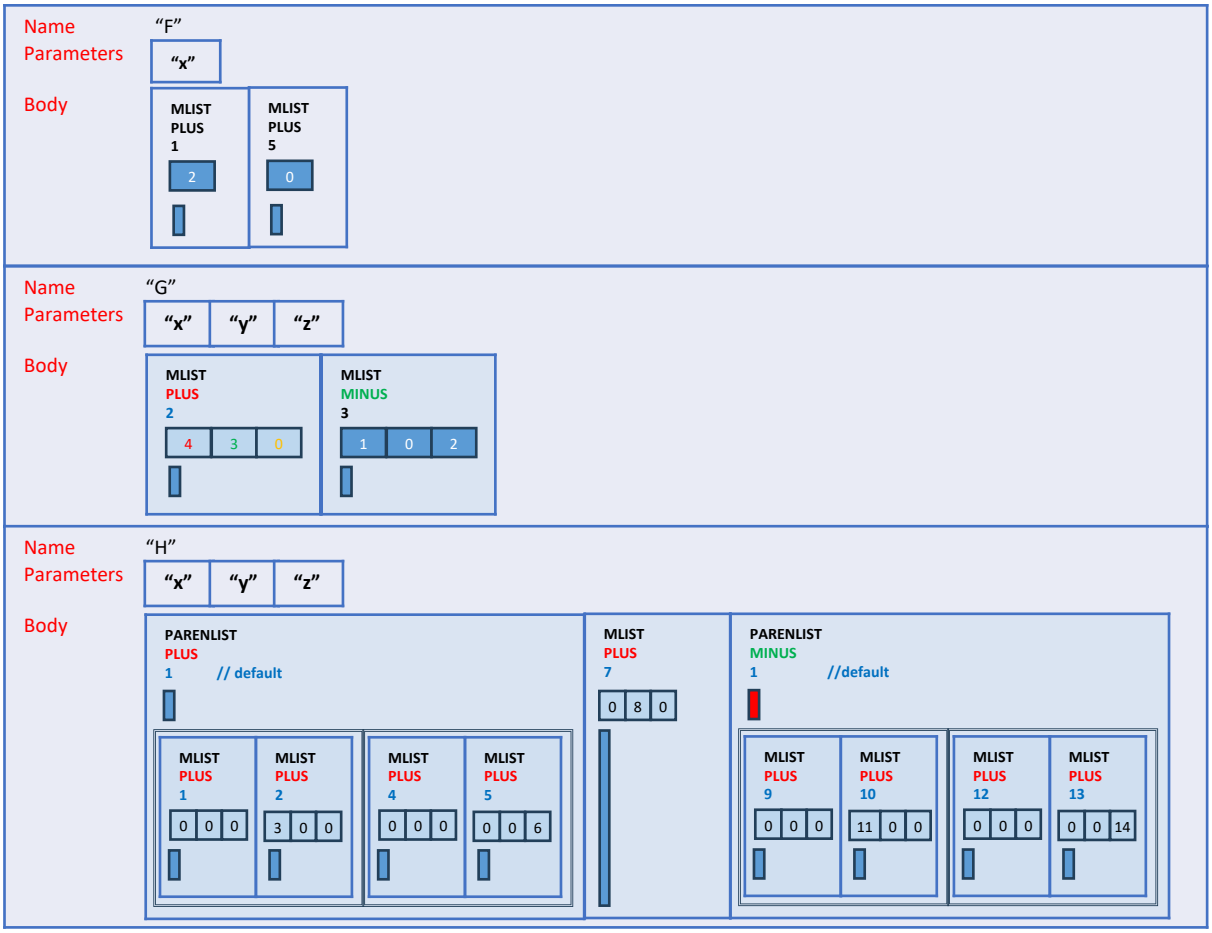


Representing Polynomial Declarations

We continue with our example:

```
F = x^2 + 5;
G(x,y,z) = 2 y^3 x^4 - 3 x z^2;
H(x,yz) = (1+2x^3)(4+ 5z^6) + 7y^8 - (9+10x^11)(12+ 13z^14);
```

After parsing the body of G(x,y,z) and H(x,y,z) are parsed, the representation becomes as follows



I already explained the representation for F's body. I explain here the representation of G's and H's bodies.

G has three parameters, x, y and z, so monomial lists will be represented as vectors of three elements.

The first term is a monomial list and represents + 2 x^4 y^3 z^0. You should be able to understand on your own the second term of the body of G.

The body of H is a term list with three terms. The first and the last terms are parenthesized lists. The default coefficient is 1. The monomial_list field is an empty vector (and). The bodies of the two parenthesized lists are vectors of two term lists (each term list has a double lines rectangle around it in the picture). In this example each of the term lists is simply a vector of terms which happen to be monomial lists. If you understand the representation, you should be able to see that we could also represent nested parenthesized lists.

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. The data structure that I suggest is a recursive data structure. The program will be a vector of statements. For basic statements such as INPUT and OUTPUT all the information about the statement can be stored in the statement node itself. For assign statements, we need to represent polynomial evaluations. The representation of polynomial evaluation is done recursively because the arguments of a polynomial evaluation can themselves be polynomial evaluations. I show what the data structure looks like for an example program

```
TASKS 1 2

POLY
    F = x^2 + 5;

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

EXECUTE

  INPUT X;

  INPUT Y;

  Y = F(X,W);

  W = F(X,W);

  OUTPUT W;

INPUTS    1 4 17
```

stmt_type the statement type is an integer that indicates the type of the statement. It can be INPUT, OUTPUT or ASSIGN (you can use whatever names you want as long as you distinguish between the cases)

LHS This field is only relevant for assignment statements. It is an integer which is equal to the index of the location associated with the variable on the LHS of an assignment statement.

poly_eval This field is relevant only if the statement type is an assignment statement. The value is a pointer to a data structure that represents the polynomial evaluation statement.

var This field is relevant for INPUT and OUTPUT statements. It is an integer which is equal to the index of the location associated with the variable in INPUT and OUTPUT statements.

data structure for a statement

```
struct Stmt_Node_t {
    int stmt_type;
    int LHS;
    poly_eval_t *  poly_eval;
    int var;
};
```

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 for Evaluation

A polynomial evaluation needs the following information

1. A way to specify which polynomial is to be evaluated. 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 locations associated with the variable must be stored in the argument
    3. Polynomial Evaluation. If the argument is a polynomial evaluation, then we need a field to represent the polynomial evaluation (recursive representation).

It should be clear that, 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 (you need to define them). I illustrate these ideas with an example program on the next page for the following program

```
TASKS 1 2
POLY
  F = x^2 + 5;
  G(x,y,z) = 2 y^3 x^4 - 3 x z^2;
EXECUTE
  INPUT X;
  INPUT Y;
  Y = F(1);
  W = G( G(4,X,Y) , 2 , 3)
  OUTPUT W;
INPUTS    1 4 17
```

A `poly_eval_t` struct can be declared as follows

```
struct poly_eval_t {
        int polyIndex;                          // index of polynomial
        vector<poly_argument_t> polyArgs;       // vector of polynomial arguments, with
                                                // one entry  for each argument of the
                                                // polynomial
};
```

and a `poly_argument_t` struct can be declared as follows

```
struct poly_argument_t {
        int kind;                   // NUM, ID, or POLYEVAL
        int value;                  // Value of NUM
        int varIndex;               // index of ID when kind is ID
        poly_eval_t *polyEval;      // pointer to poly_eval_t struct when the
                                    // argument kind is POLYEVAL}
};
```

Since these data structures are mutually recursive, you need to have a declaration

```
struct poly_argument_t;
```

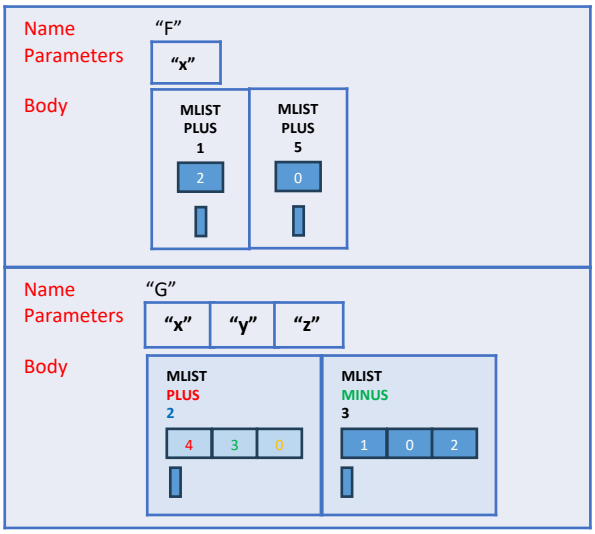
before the declaration of `poly_eval_t`

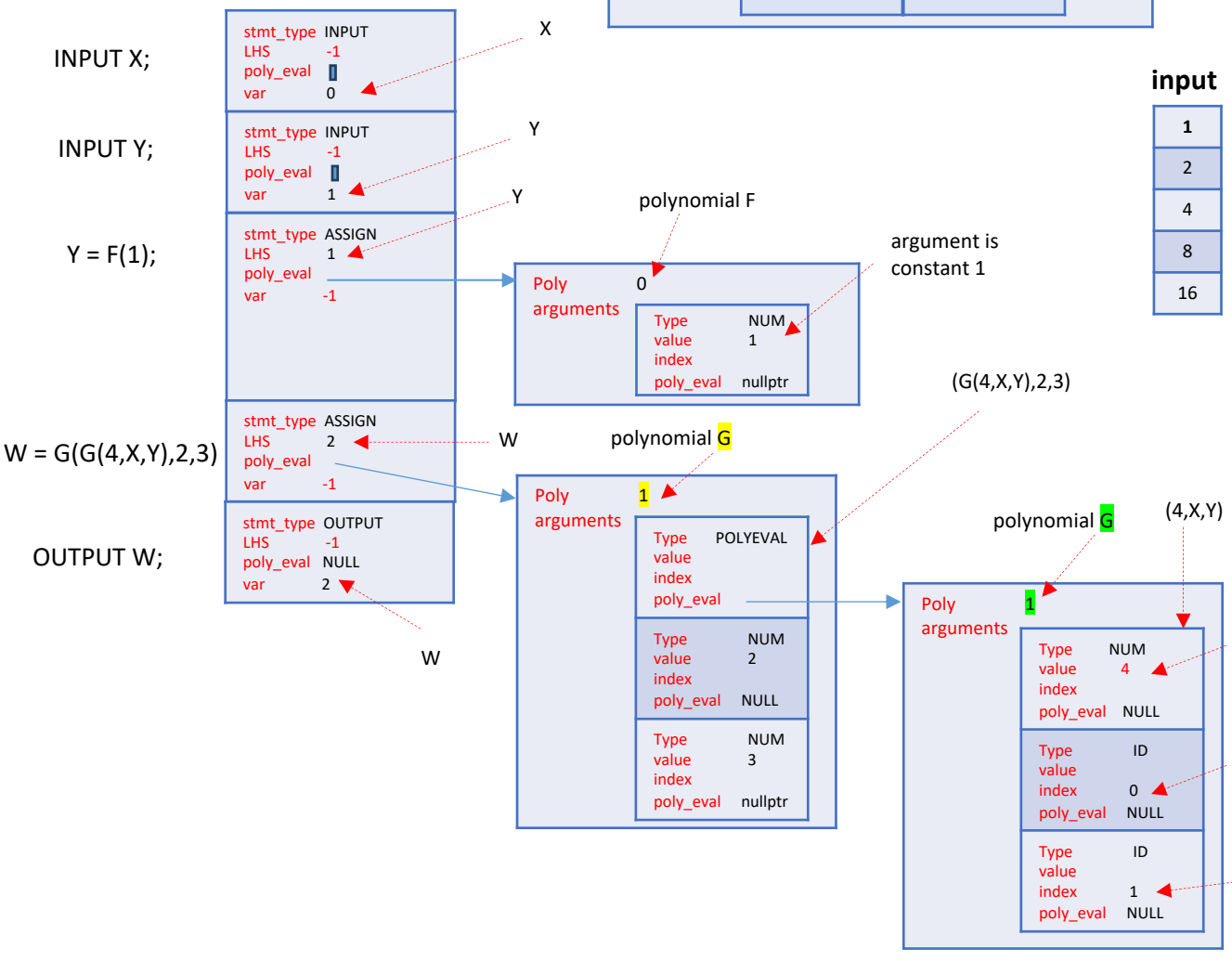
Representing Polynomial for Evaluation: Example

Consider the following program

```
TASKS 1 2
POLY
  F = x^2 + 5;
  G(x,y,z) = 2 y^3 x^4 - 3 x z^2;
EXECUTE
  INPUT X;
  INPUT Y;
  Y = F(1);
  W = G(G(4,X,Y),2,3)
  OUTPUT W;
INPUTS    1 4 17
```

This 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

```
void parse_statement_list(struct vector<Stmt_Node_t> stmtlist)
{
        struct Stmt_Node_t st;
        ...
        st = parse_stmt();

        // push st on stmtList

        // if there is more input
        parse_statement_list(stmtList);
}
```

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 vector of statements 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()` functiom.

The `execute_program()` function is a function iterates 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.

```
void execute_program(vector<Stmt_Node_t> program)
{
     int v;
    for (auto stmt : program) {
            switch (stmt.stmt_type) {
               case POLYEVAL:      v = evaluate_polynomial(stmt.poly_eval);
                                   cout << v << endl;                          // endl is std::endl
               case INPUT:         mem[stmt.var] = inputs[next_input];
                                   next_input++;
                                   break;
               /// ....
            }
    }
}
```

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 are 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. 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 use the representation of the polynomial that we already discussed 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.