GPL Phase 5

- In this phase, the students implement an *expression* tree that will be part of the parse tree.
- The *expression* tree is a tree that represents an expression in the input gpl code. For example,

```
3 + 2
  +
 / \
3  2
```

```
i + k - 3
    -
  /   \
 +     3
/ \
i   k
```

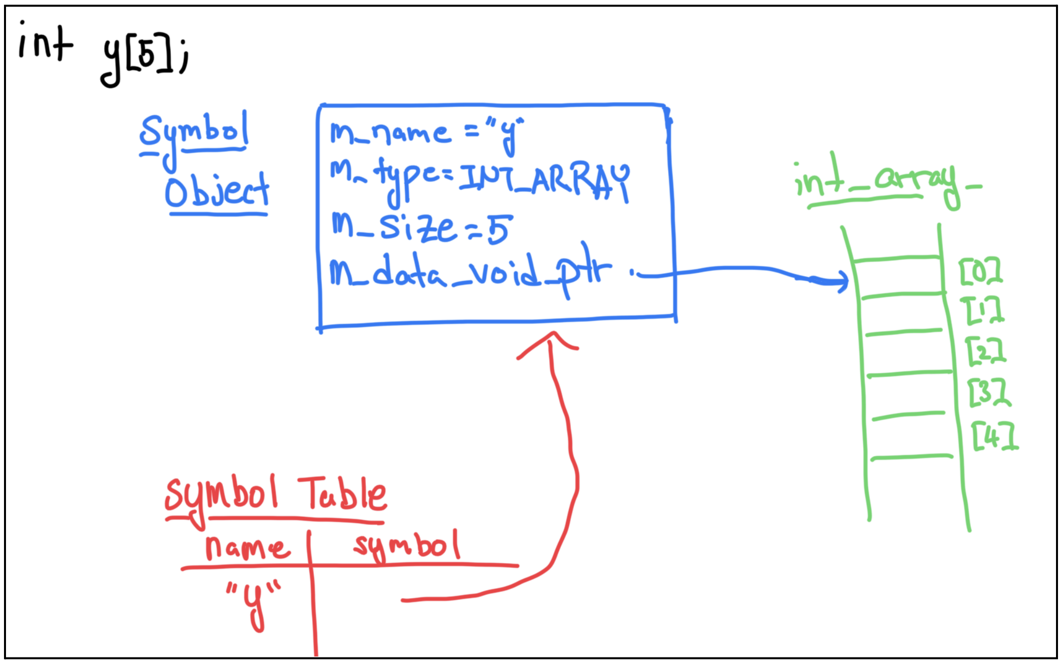
- In this phase, the students only have to handle expressions with simple types: *integer*, *double*, and *string*.
- The reason we have to handle (i.e., create trees with our data structure and add to the parse tree) is because of the semantic rules.
    - Remember that the syntax analysis does not handle (care) about the semantics of the input, e.g., int k = 1.0 + “hello”; it is an error, but an error caught by the semantic check, not by the syntactic check.
- In P4, the students only handled a new variable declaration with default values (42, 3.14159, and “Hello world”). For example,

```
// The following will create a new symbol with an initial value of 42, then
add it to the symbol table.
int x;
// The following will create a new symbol with an initial value 3.14159,
then add it to the symbol table.
double k = 1.1;
// The following will create a new symbol with every index position
initialized to 42, then add it to the symbol table.
int y[5];
```

- In this phase, the students will change the code so the parser will parse the input properly to assign either the given value or default values (0, 0.0, or “”).
    - Array positions will all be initialized to appropriate default values depending on the type of the array (e.g., int, double, or string).
    - The initialization of array positions needs to be handled using the Symbol class constructor for the array.

    - The following shows an example of creating an array on a heap, initializing it with a default value, and making m_data_void_ptr point to the array.

```
int *int_array = new int[m_size];
for (int i = 0; i < m_size; i++) {
    Int_array[i] = 0;
}
M_data_void_ptr = (void *) int_array;
```



- The initialization can be an expression. For example,

```
int k = 5;
int m = k + 6;
```

    - 5, k, +, and 6 are all individual expression objects
    - IMPORTANT: The expression object must not cache the constant value directly. The expression object holds a pointer(s) to a constant object, variable object, or other expression object.
    - This is because Expression nodes are non-terminal/internal nodes. Remember how the parser parses the tree. The inputs match with the leaf nodes.

- Let’s look at examples of how Expression trees look like:
  int k = 5;

```
Expression
     |
Constant
```

int i = k;

```
Expression
     |
Variable
```

int j = i + 6;

```
                   Expression
                   /         \
          Expression      Expression
               |               |
           Variable        Constant
```

- Expression Object Structure

```
Gpl_type           m_type = NO_TYPE;
Operator_type    m_op = NO_OP;
Expression       *m_rhs = NULL;
Expression       *m_lhs = NULL;
Constant          *m_constant = NULL;
Variable          *m_variable = NULL;
```

- If **m_constant** is **not** NULL, m_rhs, m_lhs, and m_variable must be NULL, and m_op must be NO_OP.
- If **m_variable** is **not** NULL, m_rhs, m_lhs, and m_constant must be NULL, and m_op must be NO_OP.
- Constructor for Binary
    - If **m_lhs** and **m_rhs** are **not** NULL, m_variable and m_constant must be NULL. m_op must be given an appropriate type.
    - For example,

```
if (lhs->m_type == STRING || rhs->m_type == STRING)
{
    // adding two strings results in a string (concatenation)
    if (op == PLUS) {
      m_type = STRING;
    }
    // comparing two strings results in an INT (0 or 1)
    else if (op == EQUAL || op == NOT_EQUAL || op == LESS_THAN
            || op == LESS_EQUAL || op == GREATER_THAN
            || op == GREATER_EQUAL
           ) {
      m_type = INT;
     }
     else assert(false);
}
```

- If Constructor for Unary
    - If **m_lhs** is **not** NULL (the operand, e.g., e in -e, gets assigned to m_lhs), m_rhs, m_variable, and m_constant must be NULL. m_op must be given an appropriate type.
    - For example,

```
if (op == NOT || op == RANDOM || … ) m_type = INT;
if (op == SIN || op == COS || … ) m_type = DOUBLE;
if (op == UNIARY_MINUS || …) m_type = operaand->m_type
```

- Constant Object Structure

```
Constant_union     m_union_value;
Gpl_type           m_type;
```

- Variable Object Structure

```
Symbol *m_symbol = NULL;
Gpl_type m_type = NO_TYPE;
Expression *m_expression = NULL;
```

* m_expression points to the Expression object when the variable is an array, i.e., Expression object for the array index.

```
// Assume we have an array of size 5.
x[3]
```

```
 Variable
      |
Expression
      |
 Constant
```

IMPORTANT: We do not create a new Variable object when we are declaring the variable (e.g., int x;). We create the Variable object when we are using it (e.g., in the expression or array index), i.e., we will call the Variable class's constructors under the action blocks for variable non-terminal.

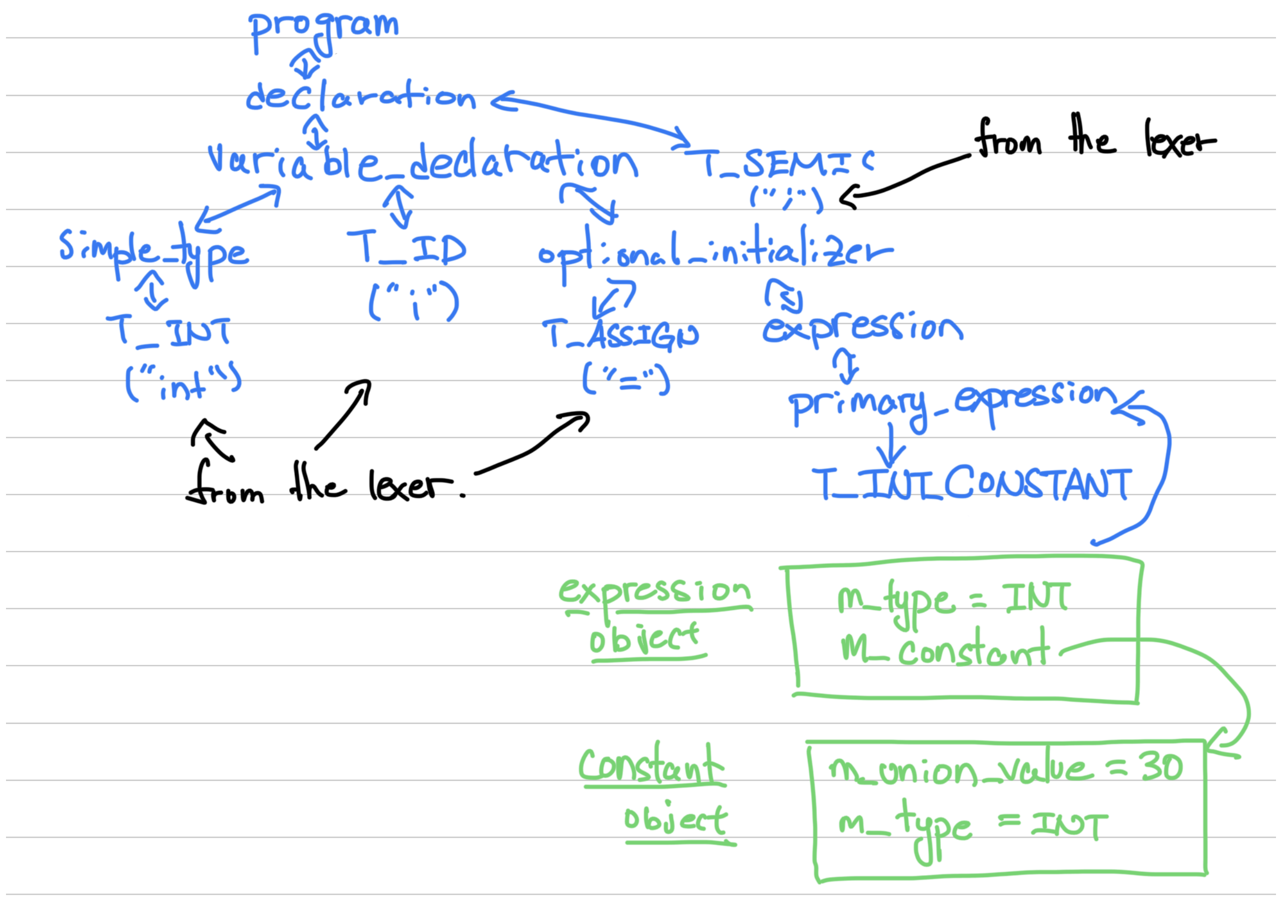
```
variable:
     T_ID
     | T_ID T_LBRACKET expression T_RBRACKET
     | T_ID T_PERIOD T_ID
     | T_ID T_LBRACKET expression T_RBRACKET T_PERIOD T_ID
```

* The last two will be handled in the future phase.

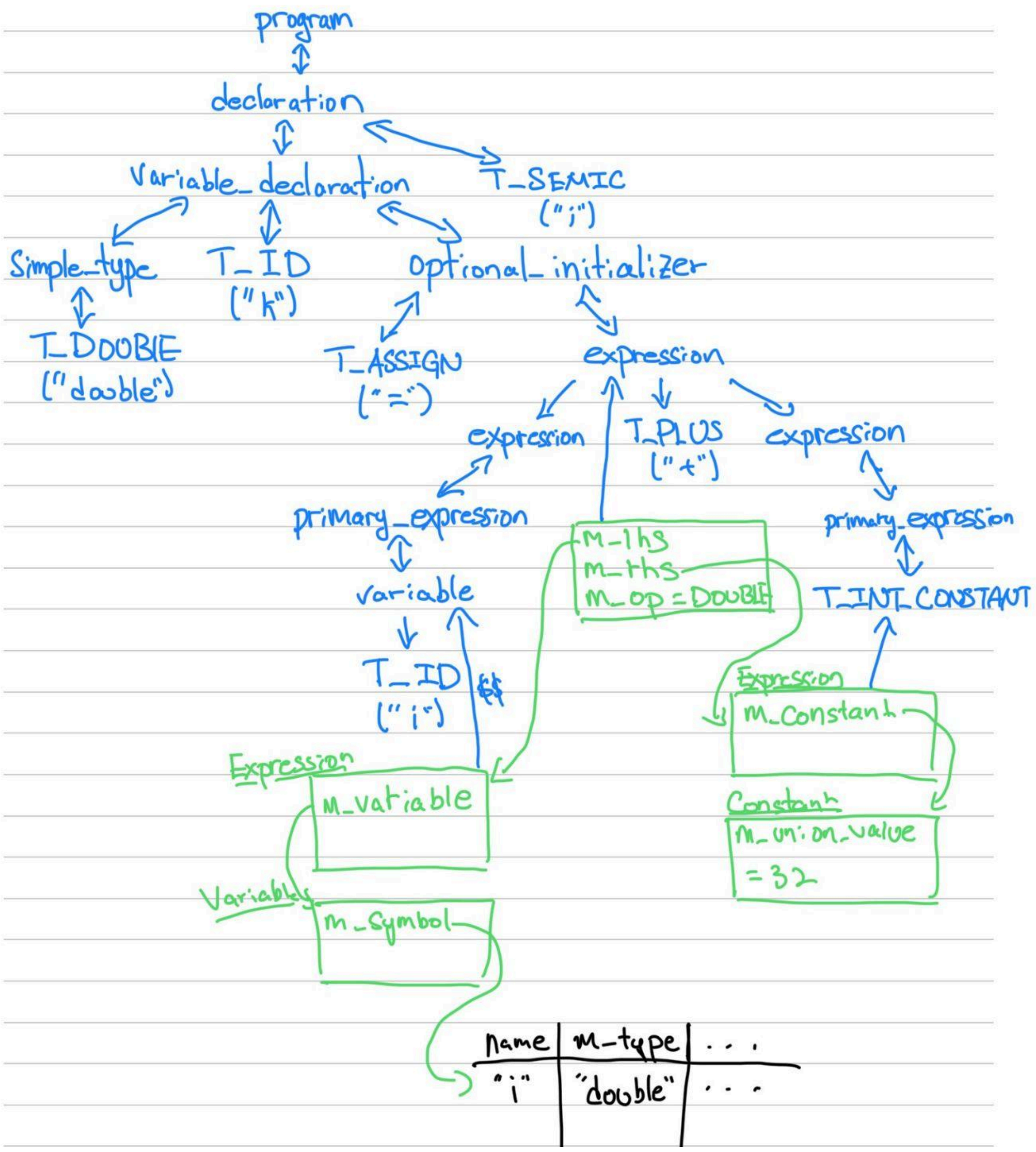
Examples

Note a parser generates ONLY a single parse tree. These examples are for the simplified explanation. For example, I didn't add symbols like "declaration_list", which will hold multiple "declaration" children nodes.

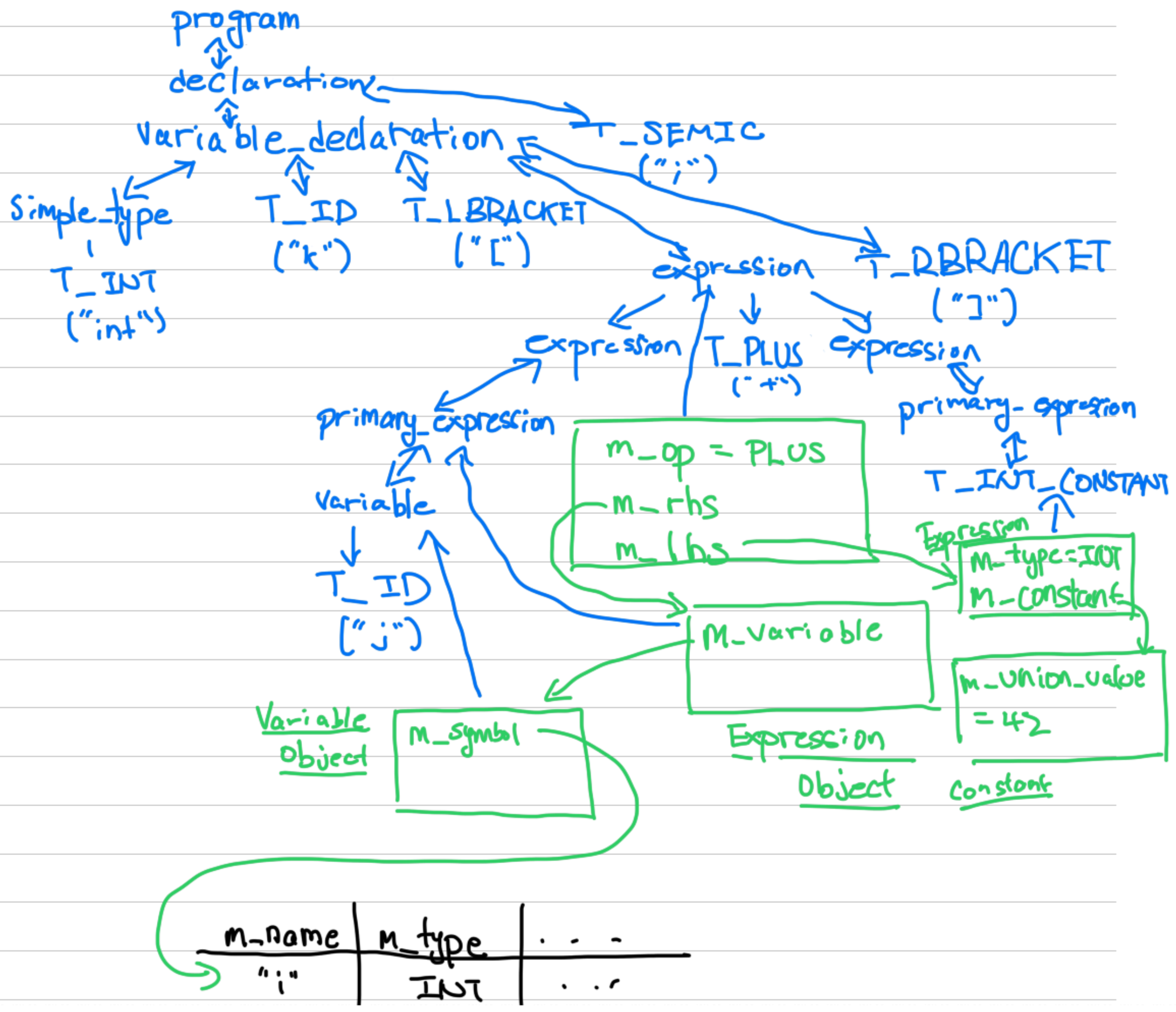
int i = 30;



int k = i + 32;


program
declaration
Variable_declaration
T_SEMIC
(";")
Simple_type
T_ID
("k")
optional_initializer
T_DOUBLE
("double")
T_ASSIGN
("=")
expression
expression
T_PLUS
("+")
expression
primary_expression
primary_expression
variable
T_INT_CONSTANT
T_ID
("i")
$$
m_lhs
m_rhs
m_op = DOUBLE
Expression
m_constant
Constant
m_union_value
= 32
Expression
m_variable
Variable
m_symbol
name
m_type
. . .
"i"
"double"
. . .

int k[j+42];


program
declaration
T_SEMIC
(";")
variable_declaration
simple_type
T_INT
("int")
T_ID
("k")
T_LBRACKET
("[")
expression
T_RBRACKET
("]")
expression
T_PLUS
("+")
expression
primary_expression
primary_expression
variable
T_INT_CONSTANT
T_ID
("j")
m_op = PLUS
m_rhs
m_lhs
Expression
m_type=INT
m_constant
m_variable
m_union_value
= 42
Variable
Object
m_symbol
Expression
Object
Constant
m_name
m_type
. . .
"j"
INT
. . .