Assignment 2

Submission Date: March 15th 2023

Now that you know how to do syntax-checking using flex and bison, in this stage we shall dive deep into the heart of the compiler. When we come out of the other end, we shall only be left with the job of generating code for some machine. Here is an overview:

Overview of this stage: By now you know that a translation unit consists of one or more function declarations, each containg a declarative part and an non-declarative (or imperative) part. For example in the simple translation unit:

```
int main ()
{
   int x, y;
   y = x + 1;
}
```

the part in red is the declarative part (it declares properties such as types of variables and functions). Now, going forward, this is what the `iplC` compiler should do:

1. *Extra Grammatical Syntactic Checks:* The `iplC` grammar may produce programs that are not syntactically correct C programs. An example is: `x++++`. This situation is not as unusual as it seems. While such deviations can be rectified by changing the grammar, it might result in a large and unwieldy grammar. Therefore even the ANSI C grammar admits syntactically incorrect C programs, which are then rejected by C compilers after further analysis. In summary, you must ensure that every program accepted by `iplC` is also a valid GCC C program.
2. *Symboltable Construction:* As you process the program, the information contained in the declarations is stored in a structure called *symboltable*. As we shall see in the class, this consists of the types of variables and functions, and the sizes and offsets of variables.
3. *Scope checking, type checking, overloading resolution:* When you process the non-declarative part, two things happen: The information in the symboltable is used to ensure that the variables are used within the scope of their declarations and are used in a type-correct manner. If they are not, then the program is rejected. In addition, operloaded operators (such a `+`) are resolved to a specific type ($+_{\texttt{int}}$) or ($+_{\texttt{float}}$).
4. *AST Creation:* If the program is syntactically and semantically (type and scope) correct, a tree representation called an Abstract Syntax Tree (AST) is generated.
5. *Code Generation:* The AST, along with the information in the symboltable, will be used to generate code.

The road ahead: The way we shall proceed in the rest of the lab course is as follows:

1. In the first part of the second assignment, you will do step 4 first, i.e. you will construct ASTs for programs without constructing the symboltable or doing semantic checks. Note: This means that you may construct ASTs for even wrong programs. The AST will also be printed in a format that we shall describe below. *This is part of Assignment 2 and is not required to be submitted.* Suggestion: This is an easy part of the assignment, and you should keep for yourself an internal deadline of 15th February (latest before mid-semester exams).

2. In the second part of the second assignment, we shall construct the symboltable, do the semantic checks and then generate ASTs using the AST construction scheme that was developed in the first part. If an error is encountered, it will be reported and no AST will be generated. *This will complete Assignment 2. This part will have to be submitted by 15th March.*

3. In Assignment 3, we shall generate target code using the symboltable and the AST. *Tentative deadline for this is 30th April.*

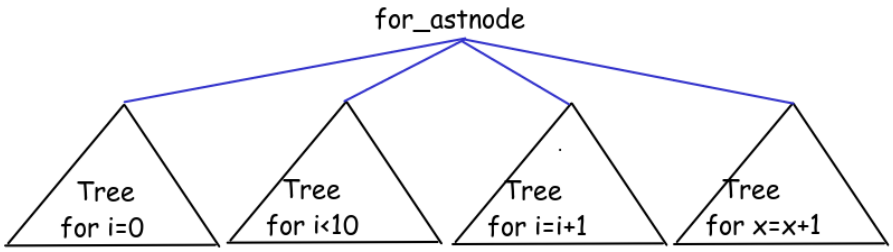
Assignment 2.1

First, make a slight change in the grammar. The grammar so far did not allow a `compound_statement` with just a `declaration_list`. Change the grammar to:

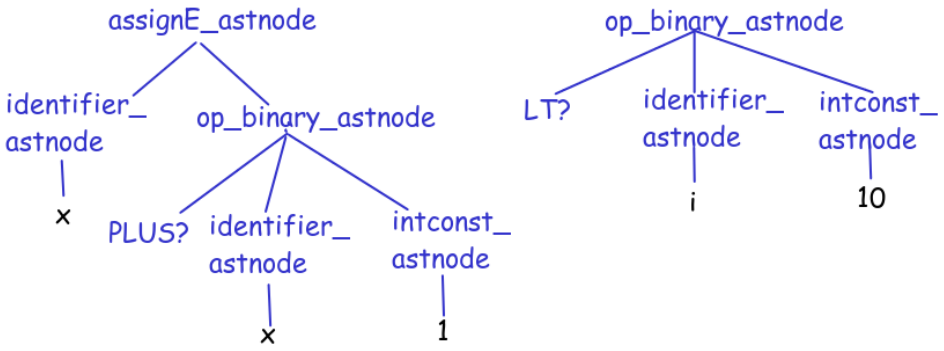
```
compound_statement:
'{' '}'
| '{' statement_list '}'
| '{' declaration_list '}'   // This is new
| '{' declaration_list statement_list '}'
;
```

and also remove all the earlier actions from the grammar.

AST: An AST represents the essential structure of the non-declarative part of the program. In an AST, we represent each construct (statement, expression etc.) as a tree consisting of a root node that represents an "operator" operating on some sub-trees. As an example, the AST for a `for` statement `for (i=0; i<10; i=i+1) x = x + 1;` can be picturized like this.

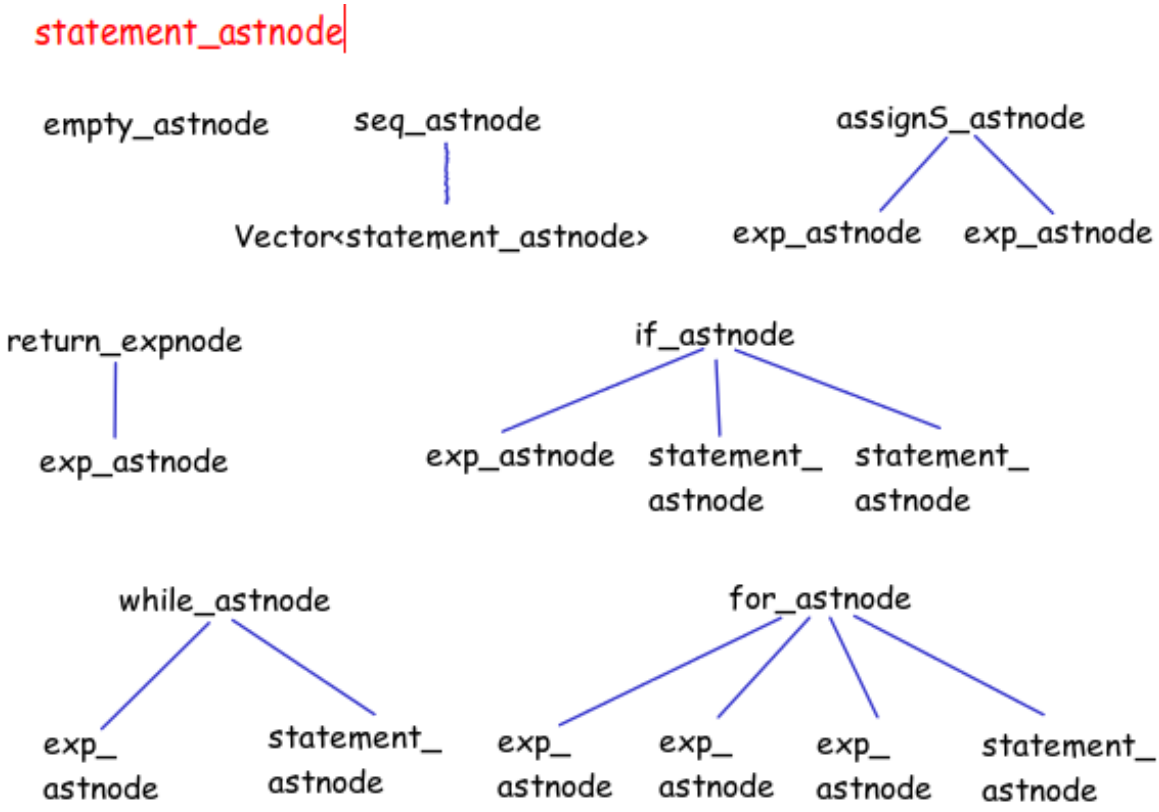


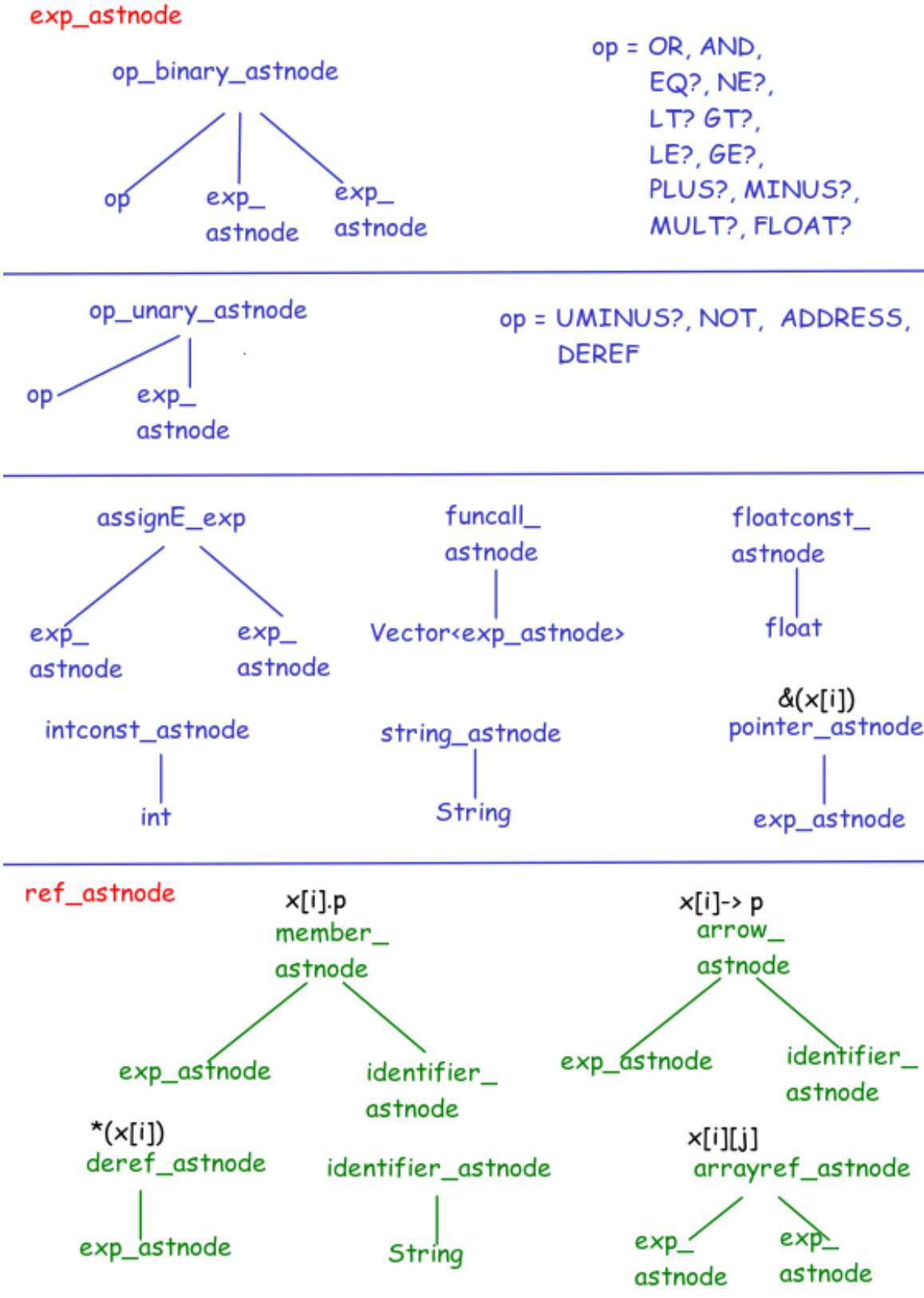
This says that the essential structure of a `for` statement consists of an *initilizer expression*, a *guard*, a *step expression*and a *body*.To complete the example, here are the ASTs for the body and the guard.



The `?` in `PLUS?` and `LT?` shows that these operators are are overloaded. Later, in Assignment 2.2, you would have to discover whethet these operators are supposed to be `int` or `float`. If `int`, the names will change to `PLUS_Int` and `LT_Int`.

Below, I give in pictures all the different types of AST nodes that would be required.




exp_astnode
op_binary_astnode
op
exp_ astnode
exp_ astnode
op = OR, AND,
EQ?, NE?,
LT? GT?,
LE?, GE?,
PLUS?, MINUS?,
MULT?, FLOAT?
op_unary_astnode
op
exp_ astnode
op = UMINUS?, NOT, ADDRESS,
DEREF
assignE_exp
exp_ astnode
exp_ astnode
funcall_ astnode
Vector<exp_astnode>
floatconst_ astnode
float
intconst_astnode
int
string_astnode
String
&(x[i])
pointer_astnode
exp_astnode
ref_astnode
x[i].p
member_ astnode
exp_astnode
identifier_ astnode
x[i]-> p
arrow_ astnode
exp_astnode
identifier_ astnode
*(x[i])
deref_astnode
exp_astnode
identifier_astnode
String
x[i][j]
arrayref_astnode
exp_ astnode
exp_ astnode

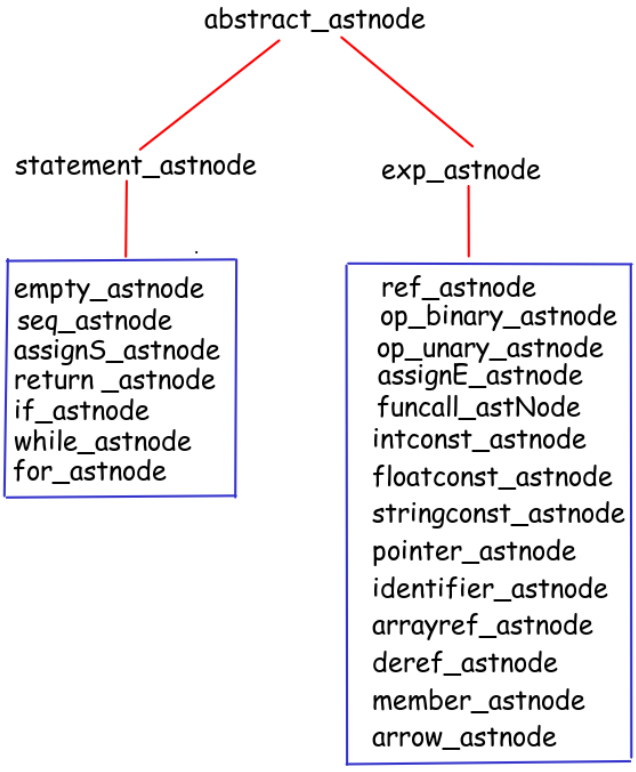
What you have to do in the first part of the assignment is:

1. Design classes to represent the ASTs.
2. Add actions to the bison script to create ASTs

The AST class hierarchy: To start off, here is a abstract class from which you can inherit other classes that describe the AST. `typExp` is a enum whose values identify the AST type of the inherited class. In this stage of Assignment 2, the only function that you would have to implement in the inherited classes is `print`, which will print a JSON representation of the AST. Feel free to add data and function members if you want.

```
class abstract_astnode
{
public:
virtual void print(int blanks) = 0;
enum typeExp astnode_type;
...
protected:...
};
```

I also suggest the following obvious inheritence hierchy. All the AST classes in a box inherit from the box's predecessor.



Output: Your output should be the AST for the program in the JSON format. The output should be in correct (but not necessarily beautified) JSON. To tie all things together, here is a program and the expected output at the end of the first part of the second assignment.

```
main()
{
    int x, y;
    for (x=0; x <10; x++);
    if (y >1) {x=x-1; y=y+1;} else ;
  }
```

The expected output is:

```
[
  {
      "seq": [
          {
              "for": {
                  "assignE": {
                      "identifier": "x",
                      "op_binary": {
                          "": "PLUS?",
                          "identifier": "x",
                          "intconst": 1
                      }
                  },
                  "op_binary": {
                      "": "LT?",
                      "identifier": "x",
                      "intconst": 10
                  },
                  "": "empty"
              }
          },
          {
              "if": {
                  "op_binary": {
                      "": "GT?",
                      "identifier": "y",
                      "intconst": 1
                  },
                  "seq": [
                      {
                          "assignS": {
                              "identifier": "x",
                              "op_binary": {
                                  "": "MINUS?",
                                  "identifier": "x",
                                  "intconst": 1
                              }
                          }
                      },
                      {
                          "assignS": {
                              "identifier": "y",
                              "op_binary": {
```

```
                                    "": "PLUS?",
                                    "identifier": "y",
                                    "intconst": 1
                                }
                            }
                        }
                    ],
                    "": "empty"
                }
            }
        ]
    }
]
```

Assignment 2.2

In this part of the assignment, we shall first construct the symbol table. We shall then perform checks for type correctness, scope correctness and other correctness that go beyond syntax. We shall collectively call such correctness checks as *semantic checks*. Along with this we shall also perform resolution of overloaded operators. Finally, we shall generate ASTs using the AST construction scheme that was developed in the Assignment 2.1.

Symboltable: I suggest that you create the following two level symbol table structure.

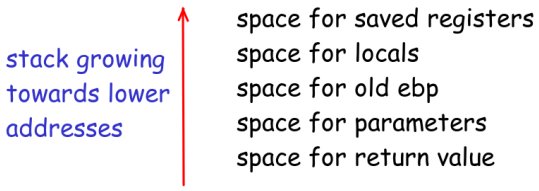
1. A global symbol table (`gst`) that maps function and `struct` names to their local symbol tables.
2. A local symbol table for every function that contains the relevant information for the parameters and the local variables of the function. You have to decide and defend which information is relevant. Here is something that you should keep in mind: *The AST and the information in the symbol table should be enough to generate code in the third assignment.*

Semantic Checks and Overloading Resolution: Having constructed the symbol table, you will now process the non-declarative part of the program once again to create an AST, re-using the code that you had developed during the third assignment. However, you will now perform semantic checks to ensure that semantically incorrect programs are rejected. While doing this, you try to mimic as closely as possible the behaviour of GCC–any program that GCC rejects, the `ipl`$\mathcal{C}$ compiler should also reject.

1. For an expression involving an overloaded operator such as `x + y`, do the following:
   (a) If `x` and `y` are both ints, resolve the `+` to $+_{int}$. The AST (when printed) will be `(PLUS_int x y)`.
   (b) If `x` and `y` are both `float`s, resolve the `+` to $+_{float}$. The AST (when printed) will be `(PLUS_float x y)`.
   (c) If `x` is `int` and `y` is `float`, cast `x` to a `float` and resolve the `+` to $+_{float}$. The AST (when printed) will be `(PLUS_float (TO_float x) y)`.

2. For `x && y`, No casting is required. The result is 1 if both operands are non-zero, and 0 otherwise. The type of the expression is `integer`.
3. For `x < y`, if `x` is `int` and `y` is `float`, cast `x` to a `float` and resolve the `<` to $<_{float}$. The AST (when printed) will be `(LT_float (TO_float x) y)`. The result will be `integer`.

Activation record layout and Offset Calculation: Some of the information in the symbol-table will require you to know the structure of the activation record. Here it is:



However, don't bother about saving registers right now. Assume that all registers will be dead just before a function call and do not have to be saved. How to lay out local variables and parameters is your choice. You need not do it in the same way as GCC.

Errors: At the first error, print a message reporting the cause of the error and location and exit. Here is an (incomplete) list of errors that you have to look out for:

1. All form of type errors. Examples: Array index not being integer. Variables being declared as being the void type.
2. All form of scoping errors.
3. Restrictions on the language that cannot be captured by a context-free grammar. For example, array indexed with less indices than its dimension and functions being passed with lesser than required number of parameters.
4. For bad programs, `iplC` should mention the error that caused the program to be rejected. Examples of error messages are: "Type mismatch in line 34", or "Undeclared variable in line 78".

Output: The output of this stage will be:

1. A dump of the global symboltable in JSON format.
2. For each function or struct, a dump of its local symbol table in JSON format,
3. For each function, a dump of its AST in JSON format.
4. For bad programs, `iplC` should mention the error that caused the program to be rejected and its location.