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 `ipl`$\mathcal{C}$ compiler should do:

1. *Extra Grammatical Syntactic Checks:* The `ipl`$\mathcal{C}$ 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 `ipl`$\mathcal{C}$ 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.

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.

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.

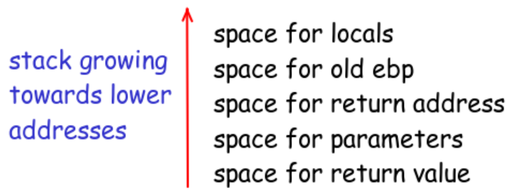
Symboltable: I suggest that you create the following two level symbol table structure.

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. However, you will also 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 `iplC` compiler should also reject (or conversely, any program accepted by `iplC` should also be accepted by GCC).

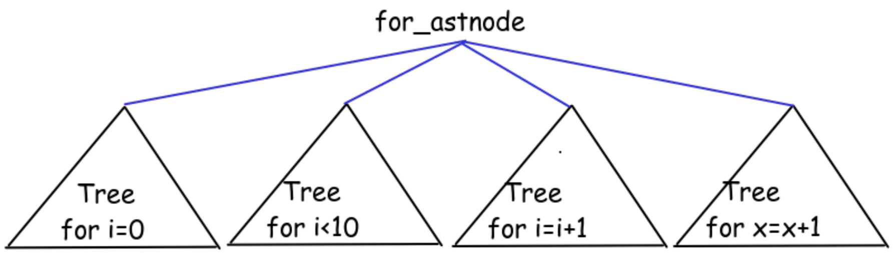
1. For an expression involving an overloaded operator such as `x + y`, do the following:
   (a) If `x` and `y` are both ints, resolve the `+` to $+_{int}$. The AST (when printed) will be `(PLUS_INT x y)`.
   (b) If `x` and `y` are both `float`s, resolve the `+` to $+_{float}$. The AST will be `(PLUS_FLOAT 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 `int`.
3. For `x < y`, if `x` is `int` and `y` is `float`, cast `x` to a `float` and resolve the `<` to $<_{float}$. The AST will be `(LT_OP_FLOAT (TO_FLOAT x) y)`. The result will be `int`.

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:

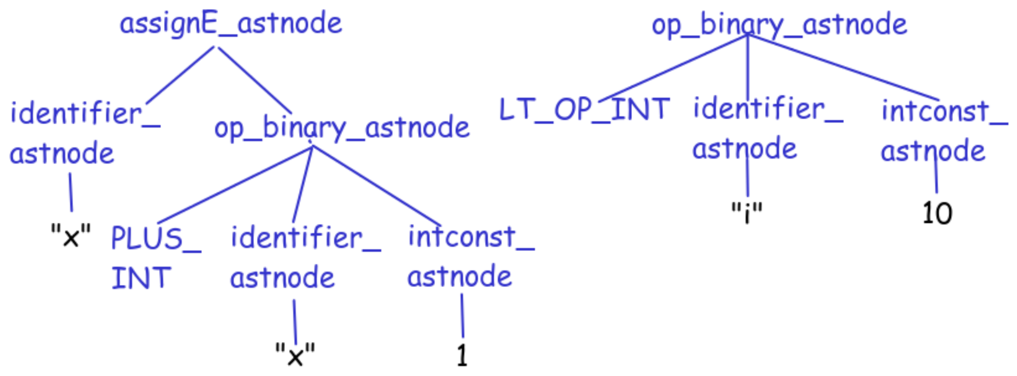


We shall adopt the following principle for laying out local variables and parameters: Variables get addresses from high to low in the order of declaration. The variable declared first gets the highest address.

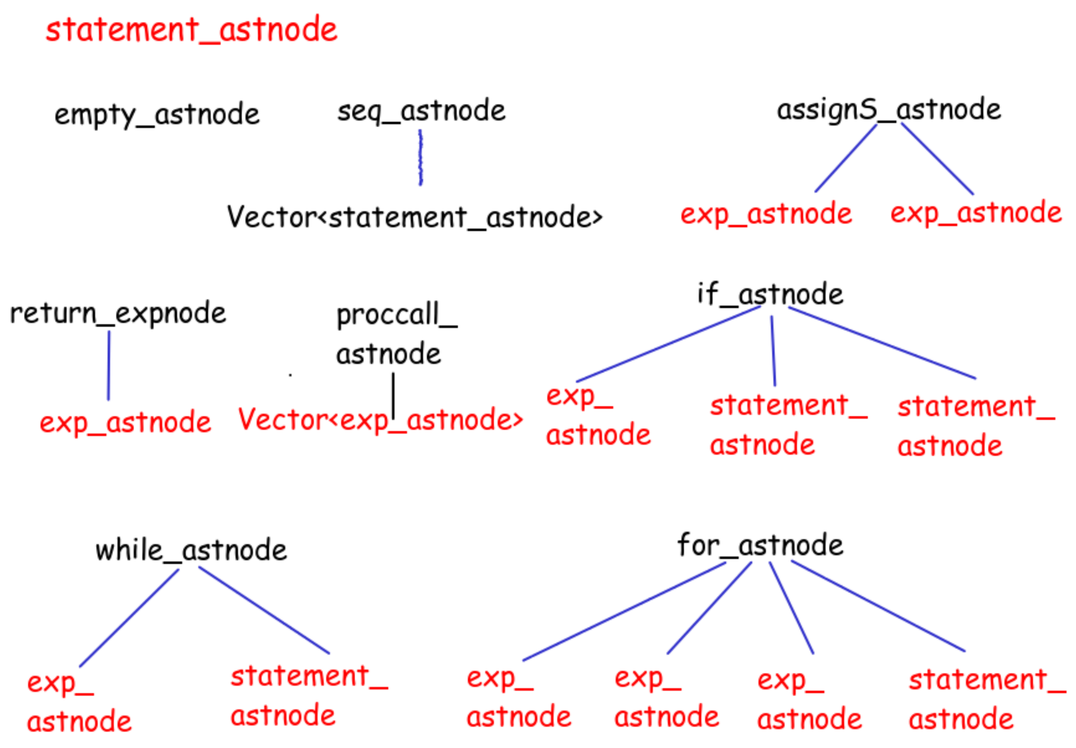
AST: An AST represents the essential structure of the non-declarative part of the program. In an AST, we represent each construct (statement, expression etc.) as a tree consisting of a root node that represents an "operator" operating on some sub-trees. As an example, the AST for a `for` statement `for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

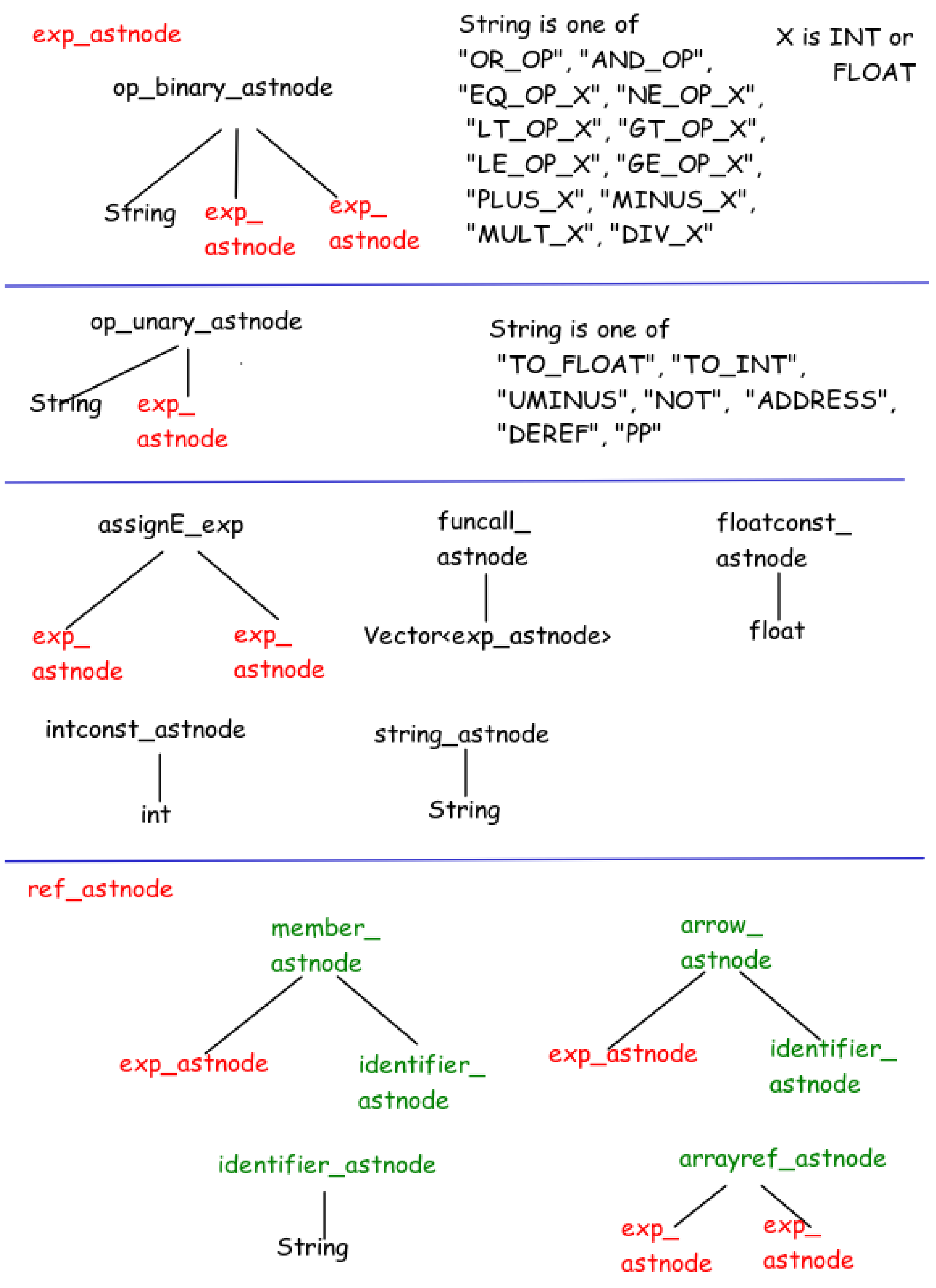


This says that the essential structure of a `for` statement consists of an *initilizer expression*, a *guard*, a *step expression*and a *body*.To complete the example, here are the ASTs for the body and the guard. The assumption is that `i` and `x` are both `int`s.



The `_INT` shows that overloading in operators `PLUS` `LT_INT` have been resolved to `int`. Below, I give in pictures all the different types of AST nodes that would be required.


statement_astnode
empty_astnode
seq_astnode
Vector<statement_astnode>
assignS_astnode
exp_astnode
exp_astnode
return_expnode
exp_astnode
proccall_astnode
Vector<exp_astnode>
if_astnode
exp_astnode
statement_astnode
statement_astnode
while_astnode
exp_astnode
statement_astnode
for_astnode
exp_astnode
exp_astnode
exp_astnode
statement_astnode



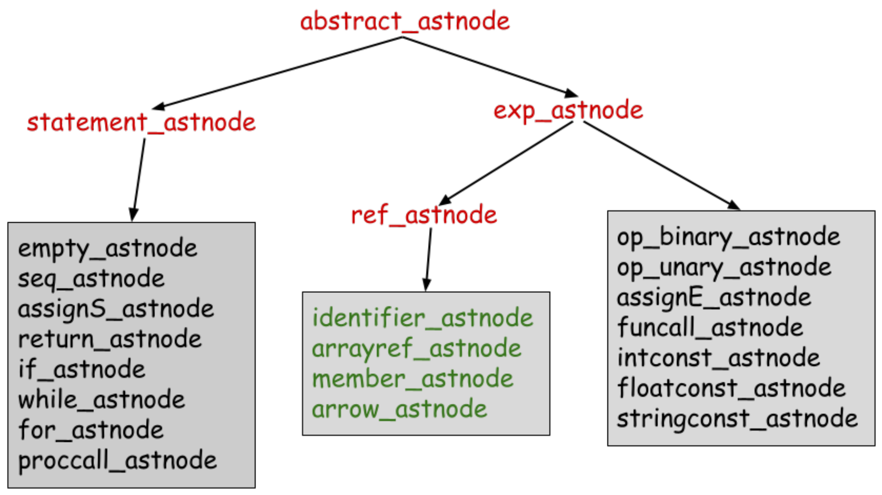
What you have to do as the next step is:

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 inheritence hierchy. All the AST classes in a box inherit from the box's predecessor. AST classes in red are abstract. Classes under `ref_astnode` have a l-value.



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 an integer. A variable declared as having 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, 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". *Note: The current version of* `iplC` *does not track line numbers. I shall soon give you a way of doing it.*

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.

For the example program shown below:

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

The expected output is:

```
{
  "globalST": [
    [
      "main",
      "fun",
      "global",
      0,
      0,
      "int"
    ]
  ],
  "structs": [],
  "functions": [
    {
      "name": "main",
      "localST": [
        [
          "x",
          "var",
          "local",
          4,
          -4,
          "int"
        ],
        [
          "y",
          "var",
          "local",
          4,
          -8,
          "int"
        ]
      ],
"ast": {
  "seq": [
    {
      "for": {
        "init": {
          "assignE": {
            "left": {
              "identifier": "x"
            },
            "right": {
              "intconst": 0
            }
          }
        },
        "guard": {
          "op_binary": {
            "op": "LT_OP_INT",
            "left": {
              "identifier": "x"
            },
            "right": {
              "intconst": 10
            }
          }
        },
        "step": {
          "assignE": {
            "left": {
              "identifier": "x"
            },
            "right": {
              "op_binary": {
                "op": "PLUS_INT",
                "left": {
                  "identifier": "x"
                },
                "right": {
                  "intconst": 1
                }
              }
            }
          }
        },
        "body": "empty"
      }
    },
    {
      "if": {
        "cond": {
          "op_binary": {
            "op": "GT_OP_INT",
            "left": {
              "identifier": "y"
            },
            "right": {
              "intconst": 1
            }
          }
        },
              "then": {
                "seq": [
                  {
                    "assignS": {
                      "left": {
                        "identifier": "x"
                      },
                      "right": {
                        "op_binary": {
                          "op": "MINUS_INT",
                          "left": {
                            "identifier": "x"
                          },
                          "right": {
                            "intconst": 1
                          }
                        }
                      }
                    }
                  },
                  {
                    "assignS": {
                      "left": {
                        "identifier": "y"
                      },
                      "right": {
                        "op_binary": {
                          "op": "PLUS_INT",
                          "left": {
                            "identifier": "y"
                          },
                          "right": {
                            "intconst": 1
                          }
                        }
                      }
                    }
                  }
                ]
              },
              "else": "empty"
            }
          }
        ]
      }
    }
  ]
}
```

The overall structure of the JSON output is:

```
{
  "globalST":[...],
  "structs": [
     {"name":"structname1,  "localST":[...]},
     {"name":"structname2,  "localST":[...]},
     ...],
  "functions:[
     {"name":"fun1", "localST":[...], "ast": {}},
     {"name":"fun2", "localST":[...], "ast": {}},
     ...]
}
```