**my-ast.png**

**Hierarchy**

**Description**

It was attempted to make the tree balanced in breadth and depth, and as a result the hierarchy does not have more than four layers. All classes fundamentally inherit from one base class, used as the API for sub-classes. Constants (e.g. integers) are only stored in basic tokens, allowing the code generation to be consistent with pointers.

Every class in the AST links back to a base structure and can access the tokens, context and registers/stacks. Expressions require two member functions that statements do not, *constant fold* for global variables, and the ability to retrieve ID. Each subclass is then derived from the *Expression* and *Statement*, split according to their functionality, requiring it to have only the necessary functions. Where possible, another hierarchy of an abstract class has been made, enabling neater code for inheriting classes (such as maths operations).

Statement classes depend on the expression classes, and their correct evaluation, however they do not inherit directly from them. Although the diagram does not show, most expressions also depend on the code generation of other expressions.

## Strengths

### Strength 1

Allowing multiple abstract classes within the hierarchy of the tree has enabled the actual assembly generation code relatively short (abstraction). Code reusability is improved as classes that can be categorised into the same group are grouped together. This also improves the code readability.

### Strength 2

Through ensuring the hierarchy is not too deep, the AST structure achieves a balanced trade-off between short, readable codes and modularity. Thus, debugging specific classes is also made easier. Having register and stack allocation managed by a separate class also achieves shorter and easier to debug/extend code.

## Limitations

### Limitation 1

Unless constant folding is performed, at any point in time the context in not aware of the value of any register nor the memory. This makes extending the compiler into an optimising one, such as performing dead branch removal, difficult.

### Limitation 2

As expressions and statements are derived from different abstract classes, a certain disconnection between the two exists. The addition of a class that is neither or that requires both would require inheritance that is not supported by the AST structure produced.

List classes, etc..

# Variable binding

## General approach

The basic API involves the use of a context class and memory class. Each class can thereafter generate their own code, without concerning about the previous or next type.

Context is used to keep track of the scoping, as well as mapping of the variable. The context knows the current scope, and therefore all the declarations within the scope and above it. Essentially forming a separate symbol table for each scope. Subroutines can be called from the context class, as it keeps track of the current function. Allocation of appropriate amounts of memory is evaluated using a separate member function.

Registers and stack allocation are handled by a separate class, passed in the same way as the context. A vector of structs is used for this purpose, with the current value being a string (as they do not have to be initialised). This allows the obtaining of free space (as stack or registers), writing to it, and freeing it as and when necessary. Whether or not a register or the stack is required depends on the member function using it.

## Strengths

It allows for complex input programs. By obtaining a temporary register or memory space and freeing it when not required, there is no need to consider which register may overwrite another as the file is parsed. Variables are stored in the stack, allowing the input programs to have more than thirty-one declarations, for example.

Binding variables is simple for each member function, as the member function only needs to insert symbols and indicate when a new scope should be created. This makes the program easier to read, debug and further extend.

## Limitations

A vector of (pointers to) symbol tables takes a lot of memory, as each symbol table has its own bindings. The context is not flushed of variables, as the program assumes it may be required, thus using considerably more stack memory.

Handling of the memory is performed separately to binding the variables, which allows for more errors. This can be prevented by keeping one class for both that always writes to memory when a variable is declared.

# Reflection

### Strength

For debugging purposes, every non-abstract class has a pretty print function (which works on a particular stream). The output is much easier to read than debugging assembly (which also has comments for debugging upon enabling) and provides the ability to know which member function is currently executing the input.

Classes are included in files (hpp or cpp) according to their purpose, rather than having one class per file. This improves the usability/readability of the compiler.

## Scope for Improvement

The grammar restricts the AST structure, which has a further disconnection between *Expressions* and *Statements* in the (e.g. declarations and initialisers). Although this could be achieved using global variables, it is not a scalable solution. The AST could be made deeper, or another class could be used to connect them.

Although the types are stored – they are not treated differently. To support more than just characters and integers, checking the type would need to performed and accordingly the generation of assembly changed.