# CSC467 Lab 3

AST Construction and Semantic Checking

**Overview** This lab is about constructing abstract syntax trees (ASTs) and making three AST visitors: one that does semantic checking, one that de-allocates an AST, and one that prints an AST. These steps will be described in this document.

**Note:** This lab does not include the `while` control-flow statement. Be sure to adjust your scanners/-parsers accordingly.

**Note:** This lab includes many bonus opportunities. Some of these bonuses are challenging and should not be taken lightly. If you decide to do one or more of the bonus questions, then state exactly which bonuses you have done in your report. Your mark in this report will not exceed 100%.

**Deliverables** The following must be implemented:

- AST construction.
- AST tear-down.
- AST printing (`-Da`).
- Semantic checking.
- Report: approach to type/semantic checking, approach to AST structure, breakdown of work by teammates, challenges faced, novelties, etc.

## 1 Starter Files

This assignment includes the following starter files. As before, you are free to use these starter files, use your own files, or mix and match.

The following is a list of interesting starter files:

compiler467.c Revised main module for Lab 3.

Makefile Revised Makefile for Lab 3.

scanner.l The working Flex scanner.

parser.y The working Bison parser with full Lab 2 functionality (excluding `while` loops).

semantic.h/c This is where you should put your code for traversing the AST and checking types, argument counts, etc. See the AST visitors section and semantic checking section below.

symbol.h/c This is where you would likely put your symbol table. You will need to build some form of symbol table abstraction wherein you will know the mappings of symbols to types within each scope. The symbol table will allow you to determine if a variable has been defined, and if so, what it's type is.

ast.h/c This is where you will define your AST structures/classes/etc. There is already an example of one approach; however, this approach is not the end-all-be-all. Consider finding your own approach. Please be sure to make your AST nodes self descriptive; it will make understanding your programs easier for me, and it will make debugging your programs easier for you.

**Note:** the starter files are provided "as is" and should represent a complete starting point for assignment 3; however, there might be bugs, or bug fixes. If you discover errors in the starter files then please *politely* report them on the course group.

## 2 ASTs

An AST is a tree that represents the high-level structure of a program. The AST should omit unnecessary complexities/particularities of the grammar. That is, you will likely have fewer AST node types than you will non-terminals, because some non-terminals exist for the sake of getting a specific kind of parse (e.g. operator precedence).

### 2.1 Construction

Below is a snippet of parser action code that constructs an AST node for the logical AND binary operator. The exampled code is valid according to the starter files (see `ast.h` and `ast.c`); however, there are many ways to approach AST construction.

```
$$ = ast_allocate(BINARY_EXPRESSION_NODE, AND, $1, $3);
```

Like parsing, AST construction will happen bottom-up, because the parser operates bottom-up. This is very convenient because it implies that when construction some AST node, you have already constructed that node's children, and so you need only connect them in. Hopefully the following hypothetical calculator will help you to understand the "flow" of data through the parser:

```
...

%union {
        int as_int;
        ...
}

...

%type <as_int> sum
%type <as_int> product
%type <as_int> factor
%token <as_int> INT

...

sum
        : sum '+' product { $$ = $1 + $3; }
        | product { $$ = $1; }
        ;

product
        : product '*' factor { $$ = $1 * $3; }
        | factor { $$ = $1; }
        ;

factor
        : INT { $$ = $1; }
        ;
```

Fig. 1: Hypothetical calculator parser

## 2.2 Visitors

An AST visitor is a function that is recursively called on each node of an AST. The traversal order depends on what the visitor is doing. For example, printing an AST would involve a pre-order traversal, whereas de-allocating an AST would involve a post-order traversal.

There are two typical and analogous ways of implementing visitors. One way is using virtual function dispatch (as in C++), and another way is to manually switch and dispatch. Below are examples of each:

```
class Node {
public:
        virtual void visit(Visitor &visitor) = 0;
};

class Expression : public Node {
public:
        virtual void visit(Visitor &visitor) {
                visitor.visit(this);
        }
};

class Statement : public Node {
public:
        virtual void visit(Visitor &visitor) {
                visitor.visit(this);
        }
};

class Visitor {
public:
        void visit(Expression *expr);
        void visit(Statement *stmt);
}
```

Fig. 2: Example AST visitor in C++

```
struct Node {
        enum {
                STATEMENT, EXPRESSION
        } kind;
        union {
                struct {
                        struct Node *next;
                } stmt;
                struct {
                        ...
                } expr;
        };
};

struct Visitor {
        void (*visit_expression)(Visitor *, Node *);
        void (*visit_statement)(Visitor *, Node *);
}

void visit(Visitor *visitor, Node *root) {
        if(NULL == root) return;
        switch(root->kind) {
        case STATEMENT: visitor->visit_statement(root); break;
        case EXPRESSION: visitor->visit_expression(root); break;
        }
};

void my_visit_expression(Visitor *visitor, Node *expr);
void my_visit_statement(Visitor *visitor, Node *stmt) {
        ...
        visit(visitor, stmt->next)
}
```

Fig. 3: Example AST visitor in C

## 2.3 Printing

You must print out the AST. Your AST printer must print the AST according to the following recursively defined forms. Note: your AST printer can add arbitrary spaces for readability.

AST printing is done with the function `ast_print`, in `ast.c`.

- **Statement Forms**

  `SCOPE` `(SCOPE (DECLARATIONS ...) (STATEMENTS ...))`

  `DECLARATIONS` `(DECLARATIONS ...)` where `...` is zero or more `DECLARATION`s

  `DECLARATION` `(DECLARATION` *variable-name type-name initial-value*?`)` where *initial*-value? is present only if an initial value is assigned to the variable. The value printed should be an expression form.

  `STATEMENTS` `(STATEMENTS ...)` where `...` is zero or more statements (`ASSIGN`, `IF`, `SCOPE`).

  `ASSIGN` `(ASSIGN` *type variable*-name new-value`)` where *new-value* is an expression form, and *type* is the type form corresponding to the type of the variable.

  `IF` `(IF` *cond then-stmt else-stmt*?`)` where *cond* is an expression form, *then-stmt* is a statement form, and *else-stmt*? is an optional statement form.

- **Expression Forms**

  `UNARY` `(UNARY` *type op expr*`)` where *op* is either of `-` or `!` and *expr* is an expression form, and *type* is the type form corresponding to the type of the resulting unary expression.

  `BINARY` `(BINARY` *type op left right*`)` where *op* is one of the binary operators of the language, *left* and *right* are both expression forms, and *type* is the type form corresponding to the type of the resulting binary expression.

`INDEX` `(INDEX` *type id index*`)` where *id* is the identifier of the variable, *index* is an expression form, and *type* is the type form corresponding to the type of the resulting value.

`CALL` `(CALL` *name* `...)`, where *name* is the name of a function (in the case of a function call) or type (in the cast of a constructor call) and `...` are one or more expression forms representing the arguments, respectively.

⟨*literal*⟩ A literal value. If the value has type `bool` then `true` or `false` should be printed. If the value is an integer, then a decimal number should be printed. If the value is a floating point number then a decimal floating point number should be printed.

⟨*identifier*⟩ The exact name of the variable.

- **Type Forms**

  Print out the exact name of the type (e.g. `int`, `bool`, `vec3`, etc.).

**Hint:** Printing your AST is a crucial step in this lab because your assignment will be graded according to what is printed.

**Hint:** Printing should be performed top-down.

## 3 Semantic Checking

Semantic checking must be automatically performed after AST construction (**Hint:** parser action of scope). Technically, you can do it during AST construction; however, I strongly suggest doing it after!

You must implement the following semantic checks. If you feel that some are missing, then please implement more:

- **Implicit Type Conversions**

  No implicit type conversions are supported. This means, for example, that one cannot use an `int` where a `float` is expected.

- **Operators**

  - All operands to logical operators must have boolean types.
  - All operands to arithmetic operators must have arithmetic types.
  - All operands to comparison operators must have arithmetic types.
  - Both opeands of a binary operator must have exactly the same *base* type (e.g. it is valid to multiple an `int` and an `ivec3`). See the below table for an outline of which classes of types can be operated on simultaneously.

    **Hint:** you might find it convenient to consider a basic type to be a vector of size 1 as a way of treating types in a uniform way.
  - If both arguments to a binary operator are vectors, then they must be vectors of the same *order*. For example, it is not valid to add an `ivec2` with an `ivec3`.
  - The table documents each operator and the classes of operands which that operator accepts. In the table, $s$ represents a scalar operand and $v$ represents a vector operand:

| Operator | Operand Classes | Category |
|---|---|---|
| `-` | $s$, $v$ | Arithmetic |
| `+`, `-` | $ss$, $vv$ | |
| `*` | $ss$, $vv$, $sv$, $vs$ | |
| `/`, `^` | $ss$ | |
| `!` | $s$, $v$ | Logical |
| `&&`, `\|\|` | $ss$, $vv$ | Logical |
| `<`, `<=`, `>`, `>=` | $ss$ | Comparison |
| `==`, `!=` | $ss$, $vv$ | |

- **Conditions**

  The expression that determins which branch of an `if` statement should be taken must have the type `bool` (not `bvec`$i$).

- **Function Calls**

  The following functions must be type-checked according to the following C declarations.

```
float rsq(float);
float rsq(int);

float dp3(vec4, vec4);
float dp3(vec3, vec3);
int dp3(ivec4, ivec4);
int dp3(ivec3, ivec3);

vec4 lit(vec4);
```

- **Constructor Calls**

  Constructors for basic types (`bool`, `int`, `float`) must have one argument that exactly matches that type. Constructors for vector types must have as many arguments as there are elements in the vector, and the types of each argument must be exactly the type of the basic type corresponding to the vector type. For example, the following is valid `bvec2(true,false)`, whereas `bvec2(1,true)` is invalid.

- **Vector Indexing**

  - The index into a vector (e.g. 1 in `foo[1]`) must be in the range $[0, i-1]$ if the vector has type `vec`$i$. For example, the maximum index into a variable of type `vec2` is `1`.
  - The result type of indexing into a vector is the base type associated with that vector's type. For example, if `v` has type `bvec2` then `v[0]` has type `bool`.

- **Initialization**

  - `const` qualified variables must be initialized with a literal value or with a uniform variable (see pre-defined variables below), and not an expression.
  - **Bonus:** If you choose to do this bonus then the previous requirement is subsumed by this requirement. `const` qualified variables must be initialized with constant expressions. A constant expression is an expression where every operand is either a literal or `const` qualified. If you do this bonus, then the value printed for the *initial-value*? field of the `DECLARATION` must be the *value* of the expression, and not the AST-printed expression. This is challenging because you will need to do minimal constant folding and constant propagation.

    **Hint:** Store the constant value of a variable/expression in a field in each AST node, as well as in your symbol table. Add an extra field/bit to your types to distinguish constant (i.e. compile-time) expressions from non-constant expressions. Interpret constant expressions while checking types.

- **Assignment**
  - The value assigned to a variable (this includes variables declared with an initial value) must have the same type as that variable, or a type that can be widened to the correct type.
- **Variables**
  - Every variable must be declared before it is used. A variable's declaration must appear either within the current scope or an enclosing scope.
  - A variable can only be declared once within the same scope. That is, the following is invalid:

```
{
        int a = 1;
        int a = 2;
}
```

  - If a variable is `const` qualified then it's value cannot be re-assigned.
  - One variable can shadow another variable. That is, the following is valid:

```
{
        int a = 1;
        {
                int a = 2;
                /* in here, a == 2 */
        }
        /* down here, a == 1 */
}
```

  - **Bonus**: Ensure that every variable has been assigned a value before being read. This is challenging because it requires checking potentially all control flow paths that lead to a read of a variable.
- **Pre-Defined Variables**

  There are several pre-defined variables. Each variable fits into one of the following type classes:

  **Attribute** Read-only, non-constant.

  **Uniform** Read-only, constant. These can be assigned to `const` qualified variables.

  **Result** Write-only, cannot be assigned anywhere in the scope of an `if` or `else` statement.

  Below is a list of the pre-defined variables. and their types. Each has been annotated with its type class.

```
result vec4 gl_FragColor;
result bool gl_FragDepth;
result vec4 gl_FragCoord;

attribute vec4 gl_TexCoord;
attribute vec4 gl_Color;
attribute vec4 gl_Secondary;
attribute vec4 gl_FogFragCoord;

uniform vec4 gl_Light_Half;
uniform vec4 gl_Light_Ambient;
uniform vec4 gl_Material_Shininess;

uniform vec4 env1;
uniform vec4 env2;
uniform vec4 env3;
```

**Hint:** type/semantic checking is usually done bottom-to-top so that you can determine the type of an expression by looking at the types of its sub-expressions.

## 3.1 Reporting Semantic Errors

Report as many semantic errors as possible, and `fprintf` each one to `errorFile` (as declared in `common.h`). If an error is found, change `errorOccurred` to `1`. Your errors should be descibe what the issue encountered was.

Reporting as many errors as possible implies that you have some mechanism for partially recovering from an error. Here are two example approaches of how to recover from an error that can easily be generalized:

- Use of an undefined variable: Declare the variable with an `any` type. If the variable is later declared in the same scope, then overwrite the fake definition.
- Bad operand type: If an `int` and a `float` are used as operands to a `+`, then report the error, and set the expression to have the `any` type.

We say an expression with an `any` type can be used anywhere. In this sense, assigning an erroneous expression to have the `any` type allows the compiler to continue performing type/semantic checks.

**Bonus:** Report the line number on which the semantic error occured.

**Bonus:** Report the column of the line, with the line number, on which the error occured.

**Bonus:** Provide a contextual message that gives extra meaning to the error to help the programmer debug the error. For example, if the error involves writing to a `const` qualified variable, then report the line number where that variable was declared.

# 4 Symbol Table

The symbol table maps variables to information about those variables (e.g. their type, etc.). The symbol table must have a notion of scope/context so that two same-named variables in different scopes do not clobber eachother within the table.

It is suggested that you opt for a simple symbol table implementation. You will not be graded on the efficiency/cleverness of your implementation.

**Hint:** One way to implement a symbol table is to have a cactus stack of tables, where each table corresponds to one scope. Each time you enter a scope, you push a table on to the stack and each time you exit a scope, you pop the scope table from the stack and insert it into the corresponding scope AST node.