University of Toronto
CSC467F Compilers and Interpreters, Fall 2018

Assignment 3: AST Construction and Semantic Analysis

Introduction

The Assignment 3 is about constructing the abstract syntax tree (AST) and making three AST visitors: one that does semantic checking, one that deallocates the entire AST, and one that dumps the AST.

You need to start Assignment 3 on the top of your previous assignments. However, you need to first update the existing `Makefile` and `compiler467.c` files with the newer version we provide. This is because you will be adding new files in this assignment, which needs to be included in the source code and compilation script.

To help you build your AST, we provide two skeleton files of `ast.h` and `ast.c`. You are free to use these two files, or choose your own design. More details will be described in the next section.

This assignment includes several 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, you need to write a document and state exactly which bonuses you have done in your submission. Your mark in this assignment will not exceed 100%.

Note: starting from this assignment, you do not need to support the `while` control flow statement. This means that we will not test your compiler against any program that contains a `while` loop.

Project Overview

Before you start this lab, please carefully read this section and make sure you setup the structure of your project properly.

- `Makefile` and `compiler467.c`
  Because we are adding new source code files in this assignment, the `Makefile` and `compiler467.c` file need to be updated. You will need to download the updated version of these two files on the course website and overwrite the old version. Remember to fill in your group information in the `compiler467.c` file.

- `ast.h/c`
  This is where you will define your AST structures. There is already a dummy implementation provided on the course website. You could choose to start from the provided implementation, or start your own design from scratch. Please make sure to name your AST nodes self-descriptive, which will make your code easier to debug, and easier for the TA to understand your program. You need to create these two files in your project.

- `semantic.c`
  This is where you should put your code for traversing the AST and checking types, argument counts, etc. In your `semantic.c` file you need to implement the function `semantic_check()`, which is declared in the provided `ast.h` file. See the AST visitors section and semantic check section below. You need to create this files in your project.

- `symbol.h/c`
  This is where you would put your code for the 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. You need to create these two files in your project.

The following functions must be implemented for this assignment. You have to implement the same function interfaces even if you are choosing to design your own AST structure.

- AST construction
  The AST construction feature should be implemented in the `ast_allocate()` function in `ast.c`.

- AST tear-down
  The AST construction feature should be implemented in the `ast_free()` function in `ast.c`.

- AST printing (`-Da`) The AST construction feature should be implemented in the `ast_print()` function in `ast.c`.

- Semantic checking
  The semantic checking feature should be implemented in the `semantic_check()` function in `semantic.c`.

AST

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

AST 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);
```

Listing 1: Allocating a AND AST node

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

```
...
%union {
    int as_int;
    ...
}
...
%type <as_int> sum
%type <as_int> product
%type <as_int> factor
%type <as_int> INT
```

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

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

Listing 2: Hypothetical calculator parser

Visitor Pattern

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 (as in C). 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);
};
```

Listing 3: 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);
}
```

Listing 4: Example AST visitor in C

AST Printing

You must implement the `ast_print()` function to print out the AST. Your AST printer must print the AST according to the following recursively defined forms. You can look up the term "`S-expression`"[1] to know more about this printed form of AST. Note: your AST printer can add arbitrary spaces for readability.

- **Statement Forms**
  - `(SCOPE (DECLARATIONS ...)  (STATEMENTS ...))`
    The `...` are zero or more `DECLARATION` and `STATEMENT` forms.
  - `(DECLARATIONS ...)`
    where `...` is zero or more `DECLARATION` forms.
  - `(DECLARATION variable-name type-name initial-value?)`
    where `initial-value?` is present only if an initial value is assigned to the variable. The initial value should be in an expression form.
  - `(STATEMENTS ...)`
    where `...` is zero or more statement forms (`ASSIGN, IF, SCOPE`).

[1]https://en.wikipedia.org/wiki/S-expression

  - `(ASSIGN type variable-name new-value)`
    where `type` is the type form of the variable, `variable-name` is the name of the variable, `new-value` is an expression form.
  - `(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 type op expr)`
    where `op` is either `-` or `!` and `expr` is an expression form, and `type` is the type form corresponding to the type of the resulting unary expression.
  - `(BINARY type op left right)`
    where `op` is one of the binary operators defined in the language, `left` and `right` are both expression forms. `type` is the type form corresponding to the type of the resulting binary operation.
  - `(INDEX type id index)`
    where `id` is the identifier of the vector variable, `index` is an expression form, and `type` is the type form corresponding to the type of the result value of the index operation.
  - `(CALL name ...)`
    where `name` could be the name of a function in the case of a function call, or a type in the case of a constructor call. `...` are one or more expression forms representing the arguments, respectively.
  - *<literal>*
    where *<literal>* is just the literal value. If the value has type `bool` then either `true` or `false` should be printed. If the value is an integer, then a decimal integer number should be printed. If the value is a floating point number then a decimal floating point number should be printed.
  - *<identifier>*
    where *<identifier>* the exact name of the variable.

- **Type Forms**
  Print out the exact name of the type. E.g. `int`, `bool`, `vec3`, etc.

Printing your AST is a critical step in this lab, because your assignment will be graded based on what you printed. So make sure your output conforms to the format described above. You should write a top-down visitor to print the AST in the given format.

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, except those that are explicitly marked as bonus.

- **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 operands of a binary operator must have exactly the same *base* type (e.g. it is valid to multiple an `int` and an `ivec3`). See below table for an outline of which classes of types can be operated on simultaneously. You may consider a basic type to be a vector of size 1 as a way to dealing with 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*	Arithmetic
*	*ss, vv, sv, vs*	Arithmetic
/, ^	*ss*	Arithmetic
!	*s, v*	Logical
&&, ||	*ss, vv*	Logical
<, <=, >, >=	*ss*	Comparison
==, !=	*ss, vv*	Comparison

Table 1: Operators and their valid operands

- **Conditions**
  The expression that determines which branch of an `if` statement should be taken must have the type `bool` (not `bvec`).

- **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);
    float dp3(ivec4, ivec4);
    float 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, `bvec2(true,false)` is valid , whereas `bvec2(1,true)` is invalid.

- **Vector Indexing**
  - The index into a vector (e.g. `1` in `foo[1]`) must be in the range $[0, i1]$ if the vector has type *veci*. 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, 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 in the inner scope can shadow the same-named variable in the outer scope. 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.

Report 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 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 + operator, 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.

Symbol Table

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

One way to implement a symbol table is to have a stack of tables, where each table corresponds to one scope. Each time you enter an AST scope node, you push a table onto 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.

Submission

First, don't forget to fill in your group information in the updated `compiler467.c` source code file. Second, pack up your code and submit your assignment by typing the following command on one of the UG EECG machines:

```
tar czvf lab3.tar.gz compiler467
submitcsc467f 3 lab3.tar.gz
```

Submit only one file per two person group from either one of your two accounts. Otherwise, a random partner's submission will be selected.