Compilers Principle Project 2 Report

Basic Design

Basic part is straightforward, I followed the visitor pattern mentioned in our lab section. By design a name space called `visitor`, and design the individual visit function for each semantic block.

```
struct visitor {
    static void program(Node* self) {...}
    static void ext_def_list(Node* self) {...}
    static void ext_def(Node* self) {...}
    ...
}
```

Hence it can recursively search the entire semantic tree and find its error.

As for symbol table design, I uses a symbol table stack to distinguish each scope. The bottom is the global scope, the top is the current scope.

> Scope Stack:
> Current Scope - Top
> ...
> ...
> ...
> Global Scope - Bottom

Bonus Design

Removal of assumptions

For the assumptions listed in the document, my implementation can remove all but 4th one.

The removal of **Assumption 1, 2, 3** is straightforward, just check for operand's type when perform certain operation.

Here I will focus on the last three assumptions.

Assumption 5

This assumption means there is no overlapping in structure definitions. It is removed by not storing member info of a structure inside symbol table instead, member info can be stored inside structure definition and not pushed into symbol table.

```
struct Struct {
    std::string name;
    std::vector<Field*> fields;
};
```

If member are stored inside a structure, they will not occupy the places in the symbol table.

This leads to another problem, what if there is one structure that contains two member with the **same** name. I do a check for this situation and set it as the 16th error.

```
if (self->rule == 1) {
   structure->fields = def_list(self->children[3]);
   for (auto repeat_f : _repeat_struct_field(structure->fields)) {
        add_err(ErrorType::MEM_REDEF, repeat_f->lineno, "Repeated field name inside
structure",
                        repeat_f->name.c_str());
       }
}
```

As for the overlapping of **structure name** and **variable name**, this is taken care of by the design of symbol table entry. Each symbol table entry is actually a three length tuple for storing **function name**, **structure name** and **variable name** separately. Hence they will not interference each other.

```
if (iter == scope->end()) {
    array<Entry*, 3> entry_arr{0, 0, 0}; // Multiple item in the same entry,
distinguished by                                    // their type
    entry_arr[(int)entry->entry_type] = entry;
    scope->insert({entry->name(), entry_arr});
}
```

Assumption 6

This is removed by the **scope stack** mentioned in basic design.

Assumption 7

This is removed by the **recursive type checking**.

```
    static bool _is_equivalent(Type* s1, Type* s2) {
        if (s1->category == Category::PRIMITIVE &&
            s2->category == Category::PRIMITIVE) {
            return s1->primitive == s2->primitive;
        } else if (s1->category == Category::STRUCT &&
                   s2->category == Category::STRUCT) {
            auto fields_1 = s1->structure->fields;
            auto fields_2 = s2->structure->fields;
            if (fields_1.size() == fields_2.size()) {
                int s = fields_1.size();
                for (int i = 0; i < s; i++) {
                    if (!_is_equivalent(fields_1[i]->type, fields_2[i]->type))
                        return false;
                }
            } else
                return false;
        } else if (s1->category == Category::ARRAY &&
                   s2->category == Category::ARRAY)
            return s1->array->size == s2->array->size &&
                   _is_equivalent(s1->array->base, s2->array->base);
```

```
        else
            return false;
        return true;
    }
```

THIS MAY CAUSE CONFLICTS WITH THE NAME EQUIVALENCE TEST CASES

Detection of undefined structure

As my implementation will push structure definitions into stack too, hence my semantic analyzer is capable of detect whether a structure is defined or not. I've given it a error label 17th

```
    static Type* _type_exist(Type* type, int lineno) {
        if (type->category == Category::STRUCT) {
            Type* t = lookup(type->structure->name, EntryType::TYPE)->type;
            if (t == NULL)
                add_err(ErrorType::STRUCT_NO_DEF, lineno, "Type not exist",
                        type->structure->name.c_str());
            return t;
        }
        return NULL;
    }
```