NAME:	Jinyi Xia
STUDENT ID:	2021212057
CLASS NUMBER:	2021211802
CONTAINER NUMBER:	df492137f3eca5844e1f8c7dce8b55741165f498417aafe88d30c2be6816f3d1

REPORT ON PROJECT 2

1 Definitions of Data Structures

The implementation of symbol table's data structure is in `symbol.h`. It has two main parts: `type_t` to describe a type, and `table_t` to describe a symbol table.

1.1 Auxiliary Definitions

Before the definition of these parts, there are some enumerations and macros to help us describe the type.

```
typedef void *HANDLE;

enum Category {
    MASK_EXP   = 0x00f,
    MASK_VAR   = 0x0f0,
    MASK_DEC   = 0xf00,
    FLAG_ATOM  = 0x001,
    FLAG_SUB   = 0x010,
    FLAG_TAB   = 0x100,
    /***************************************************************************
    CATEGORY              VISIBILITY     DESCRIPTION
    ***************************************************************************/
    E_RVAL     = 0x003, /* atom          Exp */
    V_PRIM     = 0x025, /* atom          Var & Param of primitive types */
    V_CMPLX    = 0x030, /* sub           Var & Param of struct types
                                             take sub as struct type */
    V_ARRAY    = 0x043, /* atom & sub    Var of array types
                                             take atom as elem type
                                                 if atom != NOT_T
                                                 (stores primitive)
                                             take sub as elem type
                                                 if atom == NOT_T
                                                 (stores struct)*/
    D_STRUCT   = 0x300, /* tab           Dec of struct type */
    D_FUNC     = 0x303, /* atom & tab    Dec of function type
                                             take atom as retval
                                             take tab as a var_table */
    /***************************************************************************/
};

enum Primitive {
```

```
    NOT_T   = 0b0000,
    FLOAT_T = 0b0001,
    INT_T   = 0b0010,
    CHAR_T  = 0b0100,
};
```

Category is an enumeration of all possible categories of a type.

- **FLAG_ATOM** indicates that the type can visit atom field.
- **FLAG_SUB** indicates that the type can visit sub field.
- **FLAG_TAB** indicates that the type can visit tab field.
- **MASK_EXP** is a mask to verify the category of an expression.
- **MASK_VAR** is a mask to verify the category of a variable.
- **MASK_DEC** is a mask to verify the category of a declaration.
- **E_RVAL** is the category of an expression that can not be resolved as a left value.
- **V_PRIM** is the category of a variable that is an primitive type.
- **V_CMPLX** is the category of a variable that is a struct type.
- **V_ARRAY** is the category of a variable that is an array type.
- **D_STRUCT** is the category of a declaration that is a struct type.
- **D_FUNC** is the category of a declaration that is a function type.

Primitive is an enumeration of all possible primitive types.

- **NOT_T** indicates that the type is not a primitive type.
- **FLOAT_T** indicates that the type is a float type.
- **INT_T** indicates that the type is an integer type.
- **CHAR_T** indicates that the type is a character type.

1.2 Type Definition

With previous definitions, we can define the type `type_t`. Its data structure and functions are as follows. It is noted that instead of storing all the fields in the structure directly, we store a handle to the real data which can be automatically allocated and freed (implementation details can be found in `symbol.cpp`), thus passing variables by reference more easily, and avoiding the trouble of memory management.

```
struct type_t {
    HANDLE handle;
};

struct type_t     type_new          (const char *name, enum Category cat);
struct type_t     type_new_rval     (const char *name, enum Primitive atom);
```

```
struct type_t     type_new_prim     (const char *name, enum Primitive atom);
struct type_t     type_new_cmplx    (const char *name, struct type_t sub);
struct type_t     type_new_array    (const char *name, enum Primitive atom);
struct type_t     type_clone        (struct type_t type);
const char *      type_set_name     (struct type_t type, const char *name);
enum Category     type_set_cat      (struct type_t type, enum Category cat);
enum Primitive    type_set_atom     (struct type_t type, enum Primitive atom);
struct type_t     type_set_sub      (struct type_t type, struct type_t sub);
struct table_t    type_set_tab      (struct type_t type, struct table_t tab);
const char *      type_name         (struct type_t type);
enum Category     type_cat          (struct type_t type);
enum Primitive    type_atom         (struct type_t type);
struct type_t     type_sub          (struct type_t type);
struct table_t    type_tab          (struct type_t type);
```

type_new family creates a new type with the given information.

- **type_new** creates a new type with the given name and category.
- **type_new_rval** creates a new type of right value with the given name and primitive type.
- **type_new_prim** creates a new type of primitive type with the given name and primitive type.
- **type_new_cmplx** creates a new type of struct type with the given name and sub type.
- **type_new_array** creates a new type of array type with the given name and primitive type.

type_set_field family sets the certain field of the type.

type_field family gets the certain field of the type.

type_clone clones a type.

The implementation of symbol table can be found in `symbol.cpp`.

1.3 Table Definition

Table's definition also uses the handle to refer to the real data.

```
struct table_t {
    HANDLE handle;
};

struct table_t    tab_new       ();
int               tab_add       (struct table_t tab, struct type_t type);
struct type_t     tab_get       (struct table_t tab, const char *name);
int               tab_size      (struct table_t tab);
struct table_t    tab_clone     (struct table_t tab);
const char *      tab_join      (struct table_t dst, struct table_t src);
struct type_t     tab_next      (struct table_t tab);
#define           tab_traverse  (tab, type) \
    for (struct type_t type = tab_next(tab); type.handle != NULL; type =
        tab_next(tab))
```

tab_new creates a new table.

tab_add adds a type to the table.

tab_get gets a type from the table.

tab_size gets the size of the table.

tab_clone clones a table.

tab_join joins two tables.

tab_next provides a way to traverse the table by returning the next member.

tab_traverse traverses the table with a loop.

The implementation of symbol table is in `symbol.cpp`.

```
class Table {
    public:
        list<type_t> type_list;
        map<string, list<type_t>::iterator> type_map;

        Table() {}
        int add(type_t type);
        type_t get(string name);
        table_t dump();
        static Table &load(table_t table);
};
```

To take both the efficiency and the flexibility into consideration, A red-black tree and a linked list are used to implement the symbol table. The red-black tree is used to store the map between the name and the pointer to the type's location in the linked list, and the linked list is used to store the types themselves.

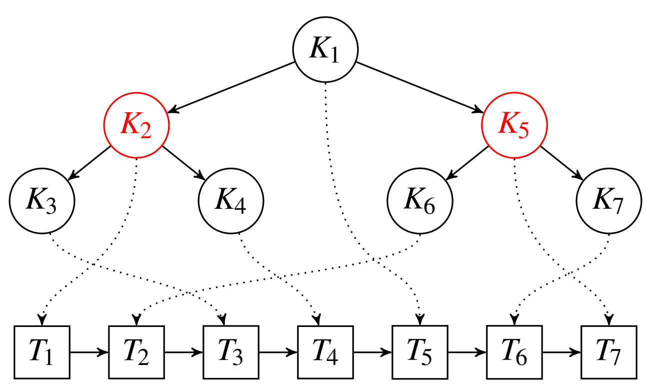

Figure 1: The relationship between the red-black tree and the linked list.

When adding a type to the table, its name is firstly checked to see if it has been occupied. If not, the new type will be added to the linked list, and the iterator of the new symbol will be added to the red-black tree.

2 Strategies of Semantic Analysis

2.1 Strategy for S Attributes

S attributes in this project are used to store the information of expressions' types. They can be easily analyzed because Bison works in a bottom-up way. When the parser is reducing a production, the semantic action can be executed after all the attributes of the production's symbols are calculated.

2.2 Strategy for L Attributes

L attributes in this project are used to store the information of types in declarations since they always appear on the left side of identifiers. To record the information of types for further use, a stack is used to store the types of the current declaration. When the parser reduces a production for a specifier, the type of the specifier will be pushed into the stack. Further identifiers' types will be set to the same as the top of the stack. When the parser reduces a production for a declaration, the type of the declaration will be popped from the stack.

2.3 Strategy for Context Switching

Context switching is used to handle the situation that the same identifier is used in different scopes. To implement this, a stack is used to store the current scope. When the parser reduces a production for a compound statement, a new scope will be pushed into the stack. When the parser reduces a production for a compound statement, the scope will be popped from the stack.

3 Build and Test

To build the project, run `make bplc` or merely run `make` in the root directory of the project. `test.sh` provides a way to test through all the cases, which can be used by running `make test` or `sh test.sh`. Some results of the given test cases are shown as follows.

Figure 2: Test cases.