

Phase 5 Table Summary

Symbol Table

The symbol table (`sym_table`) that we've been using throughout the project is a linked list of the symbols (key words and variable names) found in our program. Each element in the linked list has the form:

```
typedef struct sym_entry {
    char *name;           // symbol name
    int val;              // a unique value for identifying entry
    enum token toktype;   // type of token
    struct sym_entry *link; // link to next element in linked list
} SYM_ENTRY;
```

Definition Table (New to phase 5)

Created by the **scoping** routine uses a preorder (node-left-right) tree traversal.

```
extern DEFINITION def_table[256];
extern int def_cnt; /* initialize it to zero */

typedef struct def {
    enum type_t type; // TYPE_ILLEGAL, TYPE_INT, TYPE_FLOAT, TYPE_ERROR, TYPE_NONE
    enum v_kind kind; // PARAMETER, VARIABLE
} DEFINITION;
```

Node Structure (used previously and added to in phase 5)

We add two new items to the Node structure:

defval - added by the scoping routine. This is an index into the definition table (`def_table`)

valtype - added by the typing routine. Sets values (`node_ty`) from the new type list.

```
typedef struct node {
    enum node_ty nodetype;
    struct node *child[3]; /* used for tree structure */
    struct node *link;     /* used for linked lists */
    union
    {
        int intval;
        int symval;
        float floatval;
    } val;
    int defval; /* set by scoping routine */
    enum type_t valtype; /* set by typing routine */
    int line, column;
} NODE;
```

Typing routine types (used to modify `nodetype` in the Node structure)

This list has been expanded to reflect identifying the "type" of operators to apply.

```
enum node_ty {
    /* use this for initializing */
    ILLEGAL_NODE,

    /* this is the top node of the tree */
    SOURCE,

    /* these are the two types in C-- */
    INTTYPE,    FLOATTYPE,

    /* this is a declaration */
    DECL,
```

```

/* these are leaves with values */
INTNUM,    FLOATNUM,    IDENT,

/* these are statements */
COMPOUND,  CONDITIONAL, ITERATION,    BREAK,
CONTINUE,  RETURN,     COMPUTATION,  EMPTY,

/* these are expressions */
ASSIGN,    CONDITION,  OR_OP,    AND_OP,    NOT_OP,    EQUAL_OP,    GT_OP,
LT_OP,    ADD_OP,     SUB_OP,    MULT_OP,   DIV_OP,    MOD_OP,     NEG_OP,

/* new nodes needed after overloading has been resolved */
ADD_OP_INT,  ADD_OP_FLOAT,  SUB_OP_INT,  SUB_OP_FLOAT,  MULT_OP_INT,
MULT_OP_FLOAT,  DIV_OP_INT,  DIV_OP_FLOAT,  NEG_OP_INT,  NEG_OP_FLOAT,
GT_OP_INT,    GT_OP_FLOAT,  LT_OP_INT,  LT_OP_FLOAT,  EQUAL_OP_INT,
EQUAL_OP_FLOAT,  INT_TO_FLOAT,  TRUNCATE    };

```

Stack Implementation

```

#ifndef SNODE_H
#define SNODE_H

#include "compile.h"

struct sNode {
    int symVal;    // the symbol number from struct node
    int defVal;    // definition number from def_table
    NODE * parent; // a node pointer to the parent (compound or source)
};

#endif

#ifndef PARSERSTACK_H
#define PARSERSTACK_H

#include "sNode.h"

const int stackNodeListSize = 1000;

class parserStack {
public:
    parserStack();
    ~parserStack();
    void push(int symVal, int defVal, NODE * nodePtr);
    void pop();
    sNode * find(int symVal);
    sNode * peek();
    void dump(char *);
    int empty();
private:
    sNode * stackList[stackNodeListSize];
    int stackCount;
};

#endif

#include "parserStack.h"
#include "compile.h"
#include <assert.h>
#include "sNode.h"
#include <stdio.h>

parserStack::parserStack() {
    stackCount = 0;
}

parserStack::~parserStack() {}

void parserStack::push(int symVal, int defVal, NODE * nodePtr) {
    assert(stackCount < stackNodeListSize);

```

```

    stackList[stackCount] = new sNode;
    stackList[stackCount]->symVal = symVal;
    stackList[stackCount]->defVal = defVal;
    stackList[stackCount]->parent = nodePtr;
    stackCount++;
}

void parserStack::pop() {
    assert(stackCount != 0);
    delete stackList[stackCount--];
}

sNode * parserStack::find(int symVal) {
    for(int i = stackCount - 1; i >= 0; i--)
        if(symVal == stackList[i]->symVal) return stackList[i];
    return NULL;
}

sNode * parserStack::peek() {
    if(stackCount == 0) return NULL;
    else return stackList[stackCount-1];
}

int parserStack::empty() {
    if(stackCount == 0) return 1;
    else return 0;
}

void parserStack::dump(char * message) {

    char * node_str[] =
        {
            "ILLEGAL",    "SOURCE",    "INTTYPE",    "FLOATTYPE",    "DECL",
            "INTNUM",    "FLOATNUM",
            "IDENT",    "COMPOUND",    "CONDITIONAL",    "ITERATION",    "BREAK",
            "CONTINUE",    "RETURN",
            "COMPUTATION",    "EMPTY",    "ASSIGN",    "CONDITION",    "OR_OP",
            "AND_OP",    "NOT_OP",
            "EQUAL_OP",    "GT_OP",    "LT_OP",    "ADD_OP",    "SUB_OP",
            "MULT_OP",    "DIV_OP",
            "MOD_OP",    "NEG_OP",    "ADD_OP_INT",    "ADD_OP_FLOAT",
            "SUB_OP_INT",    "SUB_OP_FLOAT",
            "MULT_OP_INT",    "MULT_OP_FLOAT",    "DIV_OP_INT",    "DIV_OP_FLOAT",
            "NEG_OP_INT",
            "NEG_OP_FLOAT",    "GT_OP_INT",    "GT_OP_FLOAT",    "LT_OP_INT",
            "LT_OP_FLOAT",    "EQUAL_OP_INT",
            "EQUAL_OP_FLOAT",    "INT_TO_FLOAT",    "TRUNCATE"
        };

    char * type_str[] =
        {
            "(type ILLEGAL)",    "(type INT)",    "(type FLOAT)",    "(type ERROR)",
            "(type NONE)"    };

    char * v_kind_str[] = { "PARAMETER", "VARIABLE" };

    printf(" current stack contents (%s):\n",message);

    if(stackCount == 0) printf(" stack empty\n");

    for(int i = 0; i < stackCount; i++) {
        printf("entry = %3d symVal = %3d defVal = %3d ",
            i, stackList[i]->symVal, stackList[i]->defVal);
        if(stackList[i]->parent == NULL) printf(" parent null");
        else printf("parent address %p",stackList[i]->parent);
        printf("\n");
    }
}

```