env.c

```c
#include "env.h"


E_enventry E_VarEntry(Ty_ty ty) {
    E_enventry e = checked_malloc(sizeof(*e));
    e->kind = E_varEntry;
    e->u.var.ty = ty;
    return e;
}

E_enventry E_FunEntry(Ty_tyList formals, Ty_ty result) {
    E_enventry e = checked_malloc(sizeof(*e));
    e->kind = E_funEntry;
    e->u.fun.formals = formals;
    e->u.fun.result = result;
    return e;
}

S_table E_base_tenv(void) {
    S_table tenv = S_empty();
    // add pre-defined type int and string
    S_enter(tenv, S_Symbol("int"), Ty_Int());
    S_enter(tenv, S_Symbol("string"), Ty_String());
    return tenv;
}

S_table E_base_venv(void) {
    S_table venv = S_empty();
    // add standard library function
    S_enter(venv, S_Symbol("print"), E_FunEntry(Ty_TyList(Ty_String(), NULL), Ty_Void()));
    S_enter(venv, S_Symbol("flush"), E_FunEntry(NULL, Ty_Void()));
    S_enter(venv, S_Symbol("getchar"), E_FunEntry(NULL, Ty_String()));
    S_enter(venv, S_Symbol("ord"), E_FunEntry(Ty_TyList(Ty_String(), NULL), Ty_Int()));
    S_enter(venv, S_Symbol("chr"), E_FunEntry(Ty_TyList(Ty_Int(), NULL), Ty_String()));
    S_enter(venv, S_Symbol("size"), E_FunEntry(Ty_TyList(Ty_String(), NULL), Ty_Int()));
    S_enter(venv, S_Symbol("substring"), E_FunEntry(Ty_TyList(Ty_String(), Ty_TyList(Ty_Int(), Ty_TyList(Ty_Int(), NULL))), Ty_String()));
    S_enter(venv, S_Symbol("concat"), E_FunEntry(Ty_TyList(Ty_String(), Ty_TyList(Ty_String(), NULL)), Ty_String()));
    S_enter(venv, S_Symbol("not"), E_FunEntry(Ty_TyList(Ty_Int(), NULL), Ty_Int()));
    S_enter(venv, S_Symbol("exit"), E_FunEntry(Ty_TyList(Ty_Int(), NULL), Ty_Void()));
    return venv;
}
```

semant.h

```c
#ifndef SEMANT_H
#define SEMANT_H

#include <stdio.h>
#include "util.h"
#include "symbol.h"
#include "errormsg.h"
#include "types.h"
#include "absyn.h"
#include "env.h"
typedef void *Tr_exp;

typedef struct ty_list_ *ty_list;

struct ty_list_ {
    S_symbol head;
    ty_list tail;
};

ty_list Ty_List(S_symbol head, ty_list tail);

struct expty {Tr_exp exp; Ty_ty ty; } ;

struct expty expTy(Tr_exp exp, Ty_ty ty);

void SEM_transProg(A_exp exp);
struct expty transVar (S_table venv, S_table tenv, A_var v);
struct expty transExp (S_table venv, S_table tenv, A_exp a, bool inloop, S_symbol index);
void transDec (S_table venv, S_table tenv, A_dec d);
Ty_ty transTy (S_table tenv, A_ty a);

#endif
```

semant.c

```c
#include <stdio.h>
#include "semant.h"

struct expty expTy(Tr_exp exp, Ty_ty ty) {
    struct expty *e = checked_malloc(sizeof(*e));
    e->exp = exp;
    e->ty = ty;
    return *e;
}

ty_list Ty_List(S_symbol head, ty_list tail) {
    ty_list t = checked_malloc(sizeof(t));
    t->head = head;
    t->tail = tail;
    return t;
}

bool in_list(ty_list list, S_symbol ty) {
    if (list) {
        if (ty == list->head) {
            return TRUE;
        } else {
            return in_list(list->tail, ty);
        }
    } else {
        return FALSE;
    }
}

bool in_type_dec_list(A_nametyList types, S_symbol sym) {
    if (types) {
        if (types->head->name == sym) {
            return TRUE;
        } else {
            return in_type_dec_list(types->tail, sym);
        }
    } else {
        return FALSE;
    }
}

bool in_func_dec_list(A_fundecList functions, S_symbol sym) {
    if (functions) {
        if (functions->head->name == sym) {
            return TRUE;
        } else {
            return in_func_dec_list(functions->tail, sym);
        }
    } else {
        return FALSE;
    }
}

A_namety find_ty(A_nametyList types, S_symbol sym) {
    A_nametyList tt = types;
    while (tt) {
        if (tt->head->name == sym) {
            return tt->head;
        }
        tt = tt->tail;
    }
    return NULL;
}

ty_list add_to_list(ty_list list, S_symbol ty) {
    return Ty_List(ty, list);
}


Ty_ty actual_ty(Ty_ty ty) {
    if (!ty) {
        return NULL;
    }
    if (ty->kind == Ty_name) {
        Ty_ty temp = ty->u.name.ty;
        return actual_ty(temp);
    } else {
        return ty;
    }
}

// to match two ty
bool ty_match(Ty_ty tt, Ty_ty ee) {
        Ty_ty t = actual_ty(tt);
        Ty_ty e = actual_ty(ee);
```

```
        int tk = t->kind;
        int ek = e->kind;

        return (((tk == Ty_record || tk == Ty_array) && t == e) ||
                        (tk == Ty_record && ek == Ty_nil) ||
                        (ek == Ty_record && tk == Ty_nil) ||
                        (tk != Ty_record && tk != Ty_array && tk == ek));
}



// to tranfer a A_fieldList into a Ty_tyList recursively
Ty_tyList makeFormalTyList(S_table tenv, A_fieldList a_fieldList) {
    if (a_fieldList == NULL) {
        return NULL;
    } else {
        A_field a_field = a_fieldList->head;
        Ty_ty ty = S_look(tenv, a_field->typ);
        if (!ty) {
            EM_error(a_field->pos, "undefined type %s", S_name(a_field->typ));
            return NULL;
        }
        return Ty_TyList(ty, makeFormalTyList(tenv, a_fieldList->tail));
    }
}

void SEM_transProg(A_exp exp) {
    S_table venv = E_base_venv();
    S_table tenv = E_base_tenv();
    transExp(venv, tenv, exp, FALSE, NULL);
    return;
}

struct expty transVar(S_table venv, S_table tenv, A_var v) {
    switch (v->kind) {
        case A_simpleVar: {
            E_enventry entry = S_look(venv, v->u.simple);
            // check type
            if (!entry || entry->kind != E_varEntry) {
                EM_error(v->pos, "undefined variable %s", S_name(v->u.simple));
                return expTy(NULL, Ty_Int());
            }

            return expTy(NULL, actual_ty(entry->u.var.ty));
        } break;
        case A_fieldVar: {
            A_var var = v->u.field.var;
            S_symbol sym = v->u.field.sym;

            struct expty var_expty = transVar(venv, tenv, var);
            // var type must be record
            Ty_ty var_ty = var_expty.ty;
            if (var_ty->kind != Ty_record) {
                EM_error(var->pos, "record var required");
            }

            Ty_fieldList fields;
            // find the field with the name equals to sym
            for (fields = var_ty->u.record; fields; fields = fields->tail) {
                Ty_field field = fields->head;
                if (sym == field->name) {
                    // found, return the type the field
                    return expTy(NULL, actual_ty(field->ty));
                }
            }

            // not found
            EM_error(v->pos, "field %s doesn't exist", S_name(sym));
            return expTy(NULL, NULL);
        } break;
        case A_subscriptVar: {
            A_var var = v->u.subscript.var;
            A_exp exp = v->u.subscript.exp;

            struct expty var_expty = transVar(venv, tenv, var);
            struct expty exp_expty = transExp(venv, tenv, exp, FALSE, NULL);

            // exp must be int
            if (exp_expty.ty->kind != Ty_int) {
                EM_error(exp->pos, "#5");
                return expTy(NULL, Ty_Array(NULL));
            }
            // var must be array
            if (var_expty.ty->kind != Ty_array) {
                EM_error(var->pos, "array required");
                return expTy(NULL, Ty_Array(NULL));
            }
```

```
            Ty_ty ty = var_expty.ty;
            // return the type of the array
            return expTy(NULL, actual_ty(ty->u.array));
        } break;
        default: assert(0);
    }
}

struct expty transExp(S_table venv, S_table tenv, A_exp a, bool inloop, S_symbol index) {
    switch (a->kind) {
        case A_varExp: {
            A_var var = a->u.var;
            return transVar(venv, tenv, var);
        } break;
        case A_nilExp: {
            return expTy(NULL, Ty_Nil());
        } break;
        case A_intExp: {
            return expTy(NULL, Ty_Int());
        } break;
        case A_stringExp: {
            return expTy(NULL, Ty_String());
        } break;
        case A_callExp: {
            S_symbol func = a->u.call.func;

            // find symbol in the value environment
            E_enventry entry = S_look(venv, func);
            // entry must be function
            if (!entry || entry->kind != E_funEntry) {
                EM_error(a->pos, "undefined function %s", S_name(func));
                return expTy(NULL, Ty_Void());
            }

            A_expList args = a->u.call.args;
            Ty_tyList formals = entry->u.fun.formals;
            // check parameters
            A_expList el = args;
            Ty_tyList fl = formals;

            while (el && fl) {
                struct expty t = transExp(venv, tenv, el->head, inloop, index);
                if (!ty_match(t.ty, fl->head)){
                    EM_error(el->head->pos, "para type mismatched");
                }
                el = el->tail;
                fl = fl->tail;
            }
            if (el && !fl) {
                EM_error(el->head->pos,"para type mismatched");
                return expTy(NULL, Ty_Void());
            } else if (!el && fl) {
                EM_error(a->pos, "para type mismatched");
                return expTy(NULL, Ty_Void());
            } else if (entry->u.fun.result == NULL) {
                    // no return type
                    return expTy(NULL, Ty_Void());
            } else {
                    return expTy(NULL, actual_ty(entry->u.fun.result));
            }
        } break;
        case A_opExp: {
            A_oper oper = a->u.op.oper;
            A_exp left = a->u.op.left;
            A_exp right = a->u.op.right;

            Ty_ty left_ty = transExp(venv, tenv, left, inloop, index).ty;
            Ty_ty right_ty = transExp(venv, tenv, right, inloop, index).ty;

            // left and right must have value
            if (left_ty->kind == Ty_void) {
                EM_error(left->pos, "integer required");
                return expTy(NULL, Ty_Int());
            }
            if (right_ty->kind == Ty_void) {
                EM_error(right->pos, "integer required");
                return expTy(NULL, Ty_Int());
            }

            if (oper == A_plusOp || oper == A_minusOp || oper == A_timesOp || oper == A_divideOp) {
                // left and right must be integer
                if (left_ty->kind != Ty_int) {
                    EM_error(left->pos, "integer required");
                }
                if (right_ty->kind != Ty_int) {
                    EM_error(right->pos, "integer required");
```

```
            }
            return expTy(NULL, Ty_Int());
        } else if (oper == A_eqOp || oper == A_neqOp) {
            if (left_ty->kind == Ty_nil && right_ty->kind == Ty_nil) {
                // nil == nil, illegal
            } else if (left_ty->kind == Ty_nil || right_ty->kind == Ty_nil) {
                // nil == a, a == nil, legal
            } else if (left_ty->kind != right_ty->kind) {
                EM_error(a->pos, "same  type required");
            }
            return expTy(NULL, Ty_Int());
        } else if (oper == A_ltOp || oper == A_leOp || oper == A_gtOp || oper == A_geOp) {
            // must both int or both string
            if (left_ty->kind==Ty_int && right_ty->kind==Ty_int) {
                return expTy(NULL, Ty_Int());
            }
            if (left_ty->kind==Ty_string && right_ty->kind==Ty_string) {
                return expTy(NULL, Ty_Int());
            }
            EM_error(a->pos, "same  type required");
            return expTy(NULL, Ty_Int());
        }
    } break;
    case A_recordExp: {
        S_symbol typ = a->u.record.typ;

        Ty_ty ty = actual_ty(S_look(tenv, typ));
        // ty must be a record
        if (!ty || ty->kind != Ty_record) {
            EM_error(a->pos, "undefined record rectype");
            return expTy(NULL, Ty_Record(NULL));
        }
        // check record field
        Ty_fieldList ty_fields = ty->u.record;
        A_efieldList a_efields = a->u.record.fields;
        while (ty_fields && a_efields) {
            struct expty exp_expty = transExp(venv, tenv, a_efields->head->exp, inloop, index);
            if (!(ty_fields->head->name == a_efields->head->name) || !ty_match(ty_fields->head->ty, exp_expty.ty)){
                EM_error(a->pos, "efields not match");
                return expTy(NULL, Ty_Record(NULL));
            }
            ty_fields = ty_fields->tail;
            a_efields = a_efields->tail;
        }
        if (ty_fields && !a_efields) {
            EM_error(a->pos, "miss efields");
            return expTy(NULL, Ty_Record(NULL));
        } else if (!ty_fields && a_efields) {
            EM_error(a->pos, "redundant efields");
            return expTy(NULL, Ty_Record(NULL));
        }
        return expTy(NULL, ty);
    } break;
    case A_seqExp: {
        if (!a->u.seq) {
            return expTy(NULL, Ty_Void());
        }
        A_expList seq;
        for (seq = a->u.seq; seq->tail; seq = seq->tail) {
            // ignore these exp type
            transExp(venv, tenv, seq->head, inloop, index);
        }
        // the whole seq type is equal to the last exp type
        return transExp(venv, tenv, seq->head, inloop, index);
    } break;
    case A_assignExp: {
        A_var var = a->u.assign.var;
        A_exp exp = a->u.assign.exp;

        if (index != NULL && var->kind == A_simpleVar && index == var->u.simple) {
            // can not assign to index
            EM_error(exp->pos, "invalid assign to index");
            return expTy(NULL, Ty_Void());
        }

        struct expty var_expty = transVar(venv, tenv, var);
        struct expty exp_expty = transExp(venv, tenv, exp, inloop, index);

        // exp type should not be void
        if (exp_expty.ty->kind == Ty_void) {
            EM_error(a->u.assign.exp->pos, "#23");
            return expTy(NULL, Ty_Void());
        }

        // var type should be equal to exp type
        if (var_expty.ty && exp_expty.ty && !ty_match(var_expty.ty, exp_expty.ty)) {
            EM_error(a->pos, "type mismatch");
```

```
            expTy(NULL, Ty_Void());
        }

        return expTy(NULL, Ty_Void());
    } break;
    case A_ifExp: {
        A_exp test = a->u.iff.test;

        struct expty test_expty = transExp(venv, tenv, test, inloop, index);
        // test exp should be int
        if (test_expty.ty->kind != Ty_int) {
            EM_error(a->u.iff.test->pos, "#25");
            return expTy(NULL, Ty_Void());
        }

        A_exp then = a->u.iff.then;
        struct expty then_expty = transExp(venv, tenv, then, inloop, index);

        if (a->u.iff.elsee == NULL) {
            // if-then expression
            if (then_expty.ty->kind != Ty_void) {
                EM_error(a->u.iff.then->pos, "this exp must produce no value");
            }
            return expTy(NULL, Ty_Void());
        } else {
            // if-then-else expression
            A_exp elsee = a->u.iff.elsee;

            struct expty elsee_expty = transExp(venv, tenv, elsee, inloop, index);

            // then exp and else exp should be the same type
            if (ty_match(then_expty.ty, elsee_expty.ty)) {
                return expTy(NULL, actual_ty(then_expty.ty));
            } else {
                EM_error(a->u.iff.elsee->pos, "then exp and else exp type mismatch");
                return expTy(NULL, Ty_Void());
            }
        }
    } break;
    case A_whileExp: {
        A_exp test = a->u.whilee.test;
        struct expty test_expty = transExp(venv, tenv, test, inloop, index);
        // test exp should be int
        if (test_expty.ty->kind != Ty_int) {
            EM_error(a->u.whilee.test->pos, "#28");
        } else {
            A_exp body = a->u.whilee.body;
            struct expty body_expty = transExp(venv, tenv, body, TRUE, index);
            // body exp should be void
            if (body_expty.ty->kind != Ty_void) {
                EM_error(a->u.whilee.body->pos, "this exp must produce no value");
            }
        }
        return expTy(NULL, Ty_Void());
    } break;
    case A_forExp: {
        A_exp lo = a->u.forr.lo;
        A_exp hi = a->u.forr.hi;
        struct expty lo_expty = transExp(venv, tenv, lo, inloop, index);
        struct expty hi_expty = transExp(venv, tenv, hi, inloop, index);
        // lo and hi should be int
        if (lo_expty.ty->kind != Ty_int) {
            EM_error(lo->pos, "#30");
        }
        if (hi_expty.ty->kind != Ty_int) {
            EM_error(hi->pos, "integer type required");
        }
        // start a new scope
        S_beginScope(venv);
        S_enter(venv, a->u.forr.var, E_VarEntry(Ty_Int()));
        struct expty body_expty = transExp(venv, tenv, a->u.forr.body, TRUE, a->u.forr.var);
        S_endScope(venv);
        // body exp should be void
        if (body_expty.ty->kind != Ty_void) {
            EM_error(a->u.forr.body->pos, "#32");
        }
        return expTy(NULL, Ty_Void());
    } break;
    case A_breakExp: {
        // check whether this break is in a while or for loop
        if (!inloop) {
            EM_error(a->pos, "#33");
        }
        return expTy(NULL, Ty_Void());
    } break;
    case A_letExp: {
        // start a new scope
```

```
            S_beginScope(venv);
            S_beginScope(tenv);
            A_decList d;
            for (d = a->u.let.decs; d; d = d->tail) {
                transDec(venv, tenv, d->head);
            }
            struct expty body_expty = transExp(venv, tenv, a->u.let.body, inloop, index);
            S_endScope(venv);
            S_endScope(tenv);
            return body_expty;
        } break;
        case A_arrayExp: {
            Ty_ty ty = actual_ty(S_look(tenv, a->u.array.typ));
            if (!ty) {
                EM_error(a->pos, "undefined type");
                return expTy(NULL, Ty_Array(NULL));
            }
            // type must be array
            if (ty->kind != Ty_array) {
                EM_error(a->pos, "#34");
                return expTy(NULL, Ty_Array(NULL));
            }

            struct expty size_expty = transExp(venv, tenv, a->u.array.size, inloop, index);
            struct expty init_expty = transExp(venv, tenv, a->u.array.init, inloop, index);
            // size must be int
            if (size_expty.ty->kind != Ty_int) {
                EM_error(a->u.array.size->pos, "#35");
                return expTy(NULL, Ty_Array(NULL));
            }
            // init must be the same type as the array
            if (init_expty.ty != ty->u.array) {
                EM_error(a->u.array.init->pos, "type mismatched");
                return expTy(NULL, Ty_Array(NULL));
            }
            return expTy(NULL, actual_ty(ty));
        } break;
        default: EM_error(a->pos, "illegal kind"); assert(0);
    }
}

void transDec(S_table venv, S_table tenv, A_dec d) {
    switch (d->kind) {
        case A_functionDec: {
            A_fundecList functions;
            // loop the handle all the function decs
            // first pass add empty function into environment to handle mutally recursive call
            for (functions = d->u.function; functions; functions = functions->tail) {
                A_fundec function = functions->head;

                if (in_func_dec_list(functions->tail, function->name)) {
                    EM_error(function->pos, "two functions has same name"); // two functions have the same name -_-'
                }

                // check whether the function has a result or not
                if (function->result != S_Symbol("")) {
                    // it has a result
                    Ty_ty resultTy = actual_ty(S_look(tenv, function->result));
                    // result type must exist
                    if (!resultTy) {
                        EM_error(function->pos, "#37");
                        return;
                    }
                    // trans all the parameters into type list
                    Ty_tyList formalTyList = makeFormalTyList(tenv, function->params);
                    // enter the function into environment
                    S_enter(venv, function->name, E_FunEntry(formalTyList, resultTy));
                } else {
                    // no return value
                    Ty_tyList formalTyList = makeFormalTyList(tenv, function->params);
                    S_enter(venv, function->name, E_FunEntry(formalTyList, NULL));
                }
            }
            // second pass transfer the body of the function
            for (functions = d->u.function; functions; functions = functions->tail) {
                A_fundec function = functions->head;
                E_enventry funEntry = S_look(venv, function->name);

                S_beginScope(venv);
                A_fieldList l; Ty_tyList t;
                for (l = function->params, t = funEntry->u.fun.formals; l; l = l->tail, t = t->tail) {
                    S_enter(venv, l->head->name, E_VarEntry(t->head));
                }
                struct expty body_expty = transExp(venv, tenv, function->body, FALSE, NULL);
                S_endScope(venv);

                // check whether the function has a result or not
```

```
            if (funEntry->u.fun.result) {
                // it has a result
                Ty_ty resultTy = funEntry->u.fun.result;
                // check its return type
                if (!ty_match(body_expty.ty, resultTy)) {
                    EM_error(function->body->pos, "return type mismatch");
                    return;
                }
            } else {
                // no return value
                // return type must be void
                if (body_expty.ty->kind != Ty_void) {
                    EM_error(function->body->pos, "procedure returns value");
                    return;
                }
            }
        }
    } break;
    case A_varDec: {
        if (!d->u.var.init) {
            S_enter(venv, d->u.var.var, E_VarEntry(Ty_Void()));
            return;
        }
        struct expty init_expty = transExp(venv, tenv, d->u.var.init, FALSE, NULL);
        if (d->u.var.typ == S_Symbol("")) {
            // has no explicit type declaration
            if (init_expty.ty->kind == Ty_nil) {
                EM_error(d->pos, "type required");
            }
            S_enter(venv, d->u.var.var, E_VarEntry(init_expty.ty));
        } else {
            Ty_ty ty = S_look(tenv, d->u.var.typ);
            if (!ty) {
                EM_error(d->pos, "undefined type");
                return;
            }
            // init exp type must be the same as typ
            if (ty_match(ty, init_expty.ty)) {
                S_enter(venv, d->u.var.var, E_VarEntry(init_expty.ty));
            } else {
                EM_error(d->u.var.init->pos, "type mismatch");
                S_enter(venv, d->u.var.var, E_VarEntry(NULL));
                return;
            }
        }

    } break;
    case A_typeDec: {
        A_nametyList types;
        // first pass, add into environment to handle mutually recursive
        for (types = d->u.type; types; types = types->tail) {
            if (in_type_dec_list(types->tail, types->head->name)) {
                EM_error(types->head->ty->pos, "two types has same name"); // two types have the same name -_-'
            } else {
                Ty_ty ty = Ty_Name(types->head->name, NULL);
                S_enter(tenv, types->head->name, ty);
            }
        }
        // second pass, handle body
        for (types = d->u.type; types; types = types->tail) {
            Ty_ty ty1 = S_look(tenv, types->head->name);

            switch (types->head->ty->kind) {
                case A_nameTy: {
                    Ty_ty ty = actual_ty(S_look(tenv, types->head->ty->u.name));
                    ty1->u.name.ty = Ty_Name(types->head->ty->u.name, ty);
                } break;
                case A_recordTy: {

                    A_fieldList f;
                    Ty_fieldList tys = NULL, head;

                    // to make field list
                    for (f = types->head->ty->u.record; f; f = f->tail) {
                        Ty_ty ty = S_look(tenv, f->head->typ);
                        if (!ty) {
                            EM_error(f->head->pos, "type %s is illegal", S_name(types->head->name));
                        } else {
                            Ty_field tmp = Ty_Field(f->head->name, ty);
                            if (tys) {
                                tys->tail = Ty_FieldList(tmp, NULL);
                                tys = tys->tail;
                            } else {
                                tys = Ty_FieldList(tmp, NULL);
                                head = tys;
                            }
                        }
```

```
                    }

                    ty1->u.name.ty = Ty_Record(head);

                } break;
                case A_arrayTy: {
                    Ty_ty ty = actual_ty(S_look(tenv, types->head->ty->u.array));
                    ty1->u.name.ty = Ty_Array(ty);
                } break;
                default: assert(0);
            }

        }
        // third pass, check illegal cycle
        for (types = d->u.type; types; types = types->tail) {
            ty_list ty_list = Ty_List(types->head->name, NULL);
            A_ty next = types->head->ty;
            while (TRUE) {
                if (next->kind == A_nameTy) {
                    if (in_list(ty_list, next->u.name)) {
                        EM_error(types->head->ty->pos, "illegal type cycle");
                        return;
                    } else {
                        A_namety temp = find_ty(d->u.type, next->u.name);
                        if (temp) {
                            add_to_list(ty_list, next->u.name);
                            next = temp->ty;
                            continue;
                        } else {
                            break;
                        }
                    }
                } else {
                    break;
                }
            }
        }
    } break;
    default: assert(0);
  }
}
```