Files

- `def.h`: shared declarations
- `lexer.lex`: lexical analyzer
- `parser.y`: syntax analyzer
- `symtab.c`: symbol table
- `sem.c`: base functions for semantic analysis
- `gen.c`: base functions for code generation
- `stat.c`: translation of statements
- `expr.c`: translation of expressions
- `print.c`: functions for printing (abstract tree, T-code, ...)
- `main.c`: main file

File def.h: Data Structures

```c
typedef struct s_name {
    char *name;
    struct s_name *next;
} Name;

typedef struct snode {
    int type;
    Value value;
    int line;
    struct snode *child, *brother;
} Node;

typedef struct s_schema {
    char *name;
    int type;
    struct s_schema *next;
} Schema;

typedef struct s_symbol {
    int oid;
    int size;
    Schema schema;
    struct s_symbol *next;
} Symbol;

typedef struct s_context {
    int level;
    Pschema pschema;
    struct s_context *next;
} Context;
```

```c
typedef struct s_environment {
    int level;
    int numobj;
    Pname pname;
    struct s_environment *next;
} Environment;

typedef struct t_stat {
    int address;
    Operator op;
    Value args[MAXARGS];
    struct t_stat *next;
} Tstat;

typedef struct {
    Tstat *head;
    int size;
    Tstat *tail;
} Code;

typedef struct {
    Operator op;
    char *name;
    int indent;
    char *format;
} Opdescr;
```

File `def.h`: Function Prototypes

```
Boolean compatible(char*, char*),
        duplicated(char*, Pschema),
        homonyms(Pschema, Pschema),
        name_in_environment(char*),
        name_in_list(char*, Pname),
        repeated_names(Pname),
        type_equal(Schema, Schema);

char *clear_string(char *s),
     *get_format(Schema),
     *nameop(Operator),
     *operator(int),
     *update_lextab(char*),
     *valname(Pnode);

Opdescr *get_descr(Operator);

Operator codop(char*);

Pname id_list(Pnode, int*);

Pnode boolconstnode(int),
      idnode(char*),
      intconstnode(int),
      newnode(Typenode),
      qualnode(Typenode, int),
      strconstnode(char*);
```

```
Code appcode(Code, Code),
     assign_stat(Pnode),
     attr_code(Pnode),
     def_stat(Pnode),
     concode(Code, Code, ...),
     endcode(),
     expr(Pnode, Pschema),
     if_stat(Pnode),
     makecode(Operator),
     makecode1(Operator, int),
     makecode2(Operator, int, int),
     makecode3(Operator, int, int, int),
     make_get_fget(Operator, int, char*),
     make_ldint(int),
     make_ldstr(char *s),
     make_print_fprint(Operator, char*),
     make_sattr(char*),
     program(Pnode),
     read_stat(Pnode),
     specifier(Pnode),
     stat(Pnode),
     stat_list(Pnode),
     tuple_const(Pnode, Pschema),
     while_stat(Pnode),
     write_stat(Pnode);
```

File def.h: Function Prototypes (ii)

```
Pschema append_schemas(Pschema, Pschema),
        atomic_type(Pnode),
        clone_schema(Pschema),
        name_in_constack(char*, int*, int*),
        name_in_context(char*),
        name_in_schema(char*, Pschema),
        schemanode(char*, int),
        table_type(Pnode);

Psymbol insert(Schema),
        lookup(char*);

Schema type(Pnode);

Tstat *newstat(Operator);
```

```
void codeprint(Code, int),
     freemem(void*, int),
     idlprint(Pname),
     init_compiler(),
     init_lextab(),
     init_symtab(),
     insert_name_into_environment(char*),
     *newmem(int),
     noderror(Pnode),
     pop_context(),
     pop_environmet(),
     push_context(Pschema),
     push_environment(),
     eliminate(char*),
     relocate_address(Code, int),
     schprint(Schema),
     semerror(Pnode, char*),
     symprint(),
     syserror(char*),
     treeprint(Pnode, int);
```

File lexer.lex

```
%{
#include "parser.h"
#include "def.h"
int line = 1;
Value lexval;
%}

%option noyywrap

comment         --.*\n
spacing         ([ \t\r])+
letter          [A-Za-z]
digit           [0-9]
intconst        {digit}+
strconst        \"([^\"])*\"
boolconst       false|true
id              {letter}({letter}|{digit})*
sugar           [(){}:,;\+\-\*/\[\]=><]

%%

{comment}       ;
{spacing}       ;
\n              {line++;}
all             {return(ALL);}
and             {return(AND);}
boolean         {return(BOOLEAN);}
do              {return(DO);}
else            {return(ELSE);}
exists          {return(EXISTS);}
extend          {return(EXTEND);}
end             {return(END);}
"=="            {return(EQ);}
">="            {return(GE);}
if              {return(IF);}
integer         {return(INTEGER);}
join            {return(JOIN);}
"<="            {return(LE);}
"!="            {return(NE);}
not             {return(NOT);}
or              {return(OR);}
project         {return(PROJECT);}
program         {return(PROGRAM);}
read            {return(READ);}
rename          {return(RENAME);}
select          {return(SELECT);}
string          {return(STRING);}
table           {return(TABLE);}
then            {return(THEN);}
update          {return(UPDATE);}
while           {return(WHILE);}
write           {return(WRITE);}
{intconst}      {lexval.ival = atoi(yytext);
                 return(INTCONST);}
{strconst}      {lexval.sval = update_lextab(clear_string(yytext));
                 return(STRCONST);}
{boolconst}     {
                  lexval.ival = (yytext[0] == 'f' ? FALSE : TRUE);
                  return(BOOLCONST);
                }
{id}            {lexval.sval = update_lextab(yytext);
                 return(ID);}
{sugar}         {return(yytext[0]);}
.               {return(ERROR);}
<<EOF>>         {return(EOF);}

%%
```

File `symtab.c`

```c
#include "def.h"
#include "parser.h"

#define SHIFT   5
#define MAXFORMAT 1000

extern int oid_counter;

static Pname lextab[TOT_BUCKETS];

Psymbol symtab[TOT_BUCKETS];

void syserror(char *message)
{
  printf("System error: %s\n", message);
  exit(-1);
}

void noderror(Pnode p)
{
    printf("Inconsistent node (%d) in parse tree\n", p->type);
}

void *newmem(int size)
{
  char *p;
  static long size_allocated = 0;

  if((p = malloc(size)) == NULL)
    syserror("Failure in memory allocation");
  size_allocated += size;
  return(p);
}
```

File symtab.c (ii)

```c
void freemem(void *p, int size)
{
  static long size_deallocated = 0;

  free(p);
  size_deallocated += size;
}

int hash_function(char *s)
{
  int i, h=0;

  for(i=0; s[i] != '\0'; i++)
    h = ((h << SHIFT) + s[i]) % TOT_BUCKETS;
  return(h);
}

void init_lextab()
{
    int i;

    for(i = 0; i < TOT_BUCKETS; i++)
     lextab[i] = NULL;
}

void init_symtab()
{
  int i;

  for(i = 0; i < TOT_BUCKETS; i++)
    symtab[i] = NULL;
}
```

```c
char *update_lextab(char *s)
{
    int index;
    Pname p;
    char *ps;

    index = hash_function(s);
    for(p = lextab[index]; p != NULL; p = p->next)
     if(strcmp(p->name, s) == 0)
         return(p->name);
    ps = newmem(strlen(s)+1);
    strcpy(ps, s);
    p = lextab[index];
    lextab[index] = (Pname) newmem(sizeof(Name));
    lextab[index]->name = ps;
    lextab[index]->next = p;
    return(lextab[index]->name);
}

Psymbol lookup(char *name)
{
  int index;
  Psymbol psymbol;

  index = hash_function(name);
  for(psymbol = symtab[index]; psymbol != NULL;
                        psymbol = psymbol->next)
  if(psymbol->schema.name == name)
    return(psymbol);
  return(NULL);
}
```

File symtab.c (iii)

```
Psymbol insert(Schema schema)
{
  int index;
  Psymbol psymbol;

  index = hash_function(schema.name);
  psymbol = symtab[index];
  symtab[index] = (Psymbol) newmem(sizeof(Symbol));
  symtab[index]->oid = oid_counter++;
  symtab[index]->size = get_size(&schema);
  symtab[index]->schema = schema;
  symtab[index]->next = psymbol;
  return(symtab[index]);
}
```

```
int get_size(Pschema pschema)
{
  Pschema psch;
  int tupsize = 0;

  switch (pschema->type)
  {
    case INTEGER:
    case BOOLEAN:
      return sizeof(int);

    case STRING:
      return sizeof(char *);

    case TABLE:
      for(psch = pschema->next; psch;
                                psch = psch->next)
      {
        if(psch->type == STRING)
          tupsize += sizeof(char *);
        else
          upsize += sizeof(int);
      }
      return (tupsize);
  }
}
```

File symtab.c (iv)

```c
int get_attribute_offset(Pschema pschema, char *attrname)
{
  int attroffset;

  for(attroffset = 0; pschema->name != attrname && pschema != NULL; pschema = pschema->next)
    attroffset += get_size(pschema);
  if(pschema != NULL)
    return(attroffset);
  syserror("get_attribute_offset()");
}

char *get_format(Schema schema)
{
  char *format;
  Pschema pschema;
  char *attr_name, *atomic_type;
  Boolean first = TRUE;

  format = (char*) newmem(MAXFORMAT);
  switch(schema.type)
  {
    case INTEGER:
      sprintf(format, "i");
      break;

    case STRING:
      sprintf(format, "s");
      break;

    case BOOLEAN:
      sprintf(format, "b");
      break;

    ...
```

File symtab.c (v)

```
char *get_format(Schema schema)
{
  ...

   case TABLE:
      sprintf(format, "(");
      for(pschema = schema.next; pschema; pschema = pschema->next)
      {
        attr_name = (pschema->name ? pschema->name : "?");
        atomic_type = (pschema->type == INTEGER ? "i" : (pschema->type == STRING ? "s" : "b"));
        if(first == FALSE)
          strcat(format, ",");
        sprintf(&format[strlen(format)], "%s:%s", attr_name, atomic_type);
        first = FALSE;
      }
      strcat(format, ")");
      break;

    default: syserror("get_format()");
  }
  return(format);
}
```

File symtab.c (vi)

```
void eliminate(char *name)
{
  int index;
  Psymbol psymb, prec;

  index = hash_function(name);
  prec = psymb = symtab[index];
  while(psymb != NULL)
  {
    if(psymb->schema.name == name)
    {
      if(psymb == prec)
        symtab[index] = psymb->next;
      else
        prec->next = psymb->next;
      freemem(psymb, (int)sizeof(Symbol));
      return;
    }
    prec = psymb;
    psymb = psymb->next;
  }
  syserror("No name to be removed from symbol table");
}
```

File sem.c

```
#include "def.h"
#include "parser.h"

static Penvironment envstack = NULL;
static Pcontext constack = NULL;

int oid_counter = 0;

int numobj_in_current_env()
{
  return (envstack->numobj);
}

void push_environment()
{
    Penvironment temp = envstack;
    int lev = (temp == NULL ? 0 : temp->level + 1);

    envstack = (Penvironment) newmem(sizeof(Environment));
    envstack->level = lev;
    envstack->numobj = 0;
    envstack->pname = NULL;
    envstack->next = temp;
}

void insert_name_into_environment(char *name)
{
  Pname tempname = envstack->pname;

  envstack->pname = (Pname)newmem(sizeof(Name));
  envstack->pname->name = name;
  envstack->numobj++;
  envstack->pname->next = tempname;
}
```

File sem.c (ii)

```
Boolean name_in_environment(char *name)
{
  return(name_in_list(name, envstack->pname));
}

Boolean name_in_list(char *name, Pname pname)
{
  while(pname)
  {
    if(name == pname->name)
      return(TRUE);
    pname = pname->next;
  }
  return(FALSE);
}

void pop_environment()
{
    Penvironment penv = envstack;
    Pname pname, next;

    if(penv == NULL) syserror("pop_environment()");
    next= penv->pname;
    while(next)
    {
      pname = next;
      eliminate(pname->name);
      next = pname->next;
      freemem(pname, sizeof(Name));
    }
    oid_counter -= penv->numobj;
    envstack = penv->next;
    freemem((void*)penv, sizeof(Environment));
}
```

File sem.c (iii)

```
void push_context(Pschema pschema)
{
    Pcontext temp = constack;
    int lev = (temp == NULL ? 0 : temp->level + 1);

    constack = (Pcontext) newmem(sizeof(Context));
    constack->level = lev;
    constack->pschema = pschema;
    constack->next = temp;
}

void pop_context()
{
    Pcontext tempcontext;

    tempcontext = constack;
    if(tempcontext == NULL) syserror("pop_context()");
    constack = tempcontext->next;
    freemem((void*) tempcontext, sizeof(Context));
}
```

File sem.c (iv)

```
Pschema name_in_constack(char *name, int *pcontext_offset, int *pattribute_context)
{
  Pcontext pcontext = constack;
  Pschema pschema;

  for(*pcontext_offset = 0; pcontext != NULL; ++(*pcontext_offset), pcontext = pcontext->next)
    if((pschema = name_in_schema(name, pcontext->pschema)) != NULL)
    {
      *pattribute_context = get_attribute_offset(pcontext->pschema, name);
      return(pschema);
    }
    return(NULL);
}

Pschema name_in_schema(char *name, Pschema pschema)
{
  while(pschema != NULL)
  {
    if(pschema->name == name)
      return(pschema);
    pschema = pschema->next;
  };
  return(NULL);
}
```

File gen.c

```
Code appcode(Code code1, Code code2)
{
  Code rescode;

  if(code1.head == NULL)
    return (code2);
  else if(code2.head == NULL)
    return (code1);
  relocate_address(code2, code1.size);
  rescode.head = code1.head;
  rescode.tail = code2.tail;
  code1.tail->next = code2.head;
  rescode.size = code1.size + code2.size;
  return(rescode);
}

Code endcode()
{
  static Code code = {NULL, 0, NULL};

  return(code);
}

Code concode(Code code1, Code code2, ...)
{
  Code rescode = code1, *pcode = &code2;

  while(pcode->head != NULL)
  {
    rescode = appcode(rescode, *pcode);
    pcode++;
  }
  return(rescode);
}
```

```
Tstat *newstat(Operator op)
{
  Tstat *pstat;

  pstat = (Tstat*) newmem(sizeof(Tstat));
  pstat->address = 0;
  pstat->op = op;
  pstat->next = NULL;
  return(pstat);
}

Code makecode(Operator op)
{
  Code code;

  code.head = code.tail = newstat(op);
  code.size = 1;
  return(code);
}

Code makecode1(Operator op, int arg)
{
  Code code;

  code = makecode(op);
  code.head->args[0].ival = arg;
  return(code);
}
```

File gen.c (ii)

```
Code makecode2(Operator op, int arg1, int arg2)
{
  Code code;

  code = makecode1(op, arg1);
  code.head->args[1].ival = arg2;
  return(code);
}

Code makecode3(Operator op, int arg1, int arg2, int arg3)
{
  Code code;

  code = makecode2(op, arg1, arg2);
  code.head->args[2].ival = arg3;
  return(code);
}

Code make_ldint(int i)
{
  Code code;

  code = makecode(T_LDINT);
  code.head->args[0].ival = i;
  return(code);
}
```

File gen.c (iii)

```c
Code make_ldstr(char *s)
{
  Code code;

  code = makecode(T_LDSTR);
  code.head->args[0].sval = s;
  return(code);
}

Code make_sattr(char *s)
{
  Code code;

  code = makecode(T_SATTR);
  code.head->args[0].sval = s;
  return(code);
}

Code make_get_fget(Operator op, int oid, char *format)
{
  Code code;

  code = makecode1(op, oid);
  code.head->args[1].sval = format;
  return(code);
}

Code make_print_fprint(Operator op, char *format)
{
  Code code;

  code = makecode(op);
  code.head->args[0].sval = format;
  return(code);
}
```

File stat.c

```
#include "def.h"
#include "parser.h"

void semerror(Pnode p, char *message)
{
  printf("Line %d: %s\n", p->line, message);
  exit(-1);
}

int qualifier(Pnode p)
{
  return (p->value.ival);
}

char *valname(Pnode p)
{
  return (p->value.sval);
}

Code program(Pnode root)
{
  Code body = stat_list(root->child);

  return concode(makecode1(T_TCODE, body.size + 2),
                 body,
                 makecode(T_HALT),
                 endcode());
}

...
```

File `expr.c`

```c
#include "def.h"
#include "parser.h"

Boolean type_equal(Schema schema1, Schema schema2)
{
  Pschema p1, p2;

  if(schema1.type != schema2.type)
    return(FALSE);
  if(schema1.type == TABLE)
  {
    for(p1 = schema1.next, p2 = schema2.next; p1 != NULL && p2 != NULL; p1= p1->next, p2 = p2->next)
      if(p1->type != p2->type || !compatible(p1->name, p2->name))
        return(FALSE);
    return(p1 == NULL && p2 == NULL);
  }
  else
    return(TRUE);
}

Boolean compatible(char *name1, char *name2)
{
  return(name1 == NULL || name2 == NULL || name1 == name2);
}
```

File expr.c (ii)

```c
Pschema clone_schema(Pschema pschema)
{
  Pschema clone, psch;

  clone = psch = (Pschema) newmem(sizeof(Schema));
  *psch = *pschema;
  while(pschema->next)
  {
    psch->next = (Pschema) newmem(sizeof(Schema));
    *(psch->next) = *(pschema->next);
    psch = psch->next;
    pschema = pschema->next;
  }
  return (clone);
}

Pschema append_schemas(Pschema psch1, Pschema psch2)
{
  Pschema head = psch1;

  while(psch1->next)
    psch1 = psch1->next;
  psch1->next = psch2;
  return(head);
}
```

File expr.c (iii)

```
Code expr(Pnode root, Pschema pschema)
{
  ...

  pschema->name = NULL;
  pschema->next = NULL;
  switch(root->type)
  {
    case N_ID : ...

    ...

    case N_MATH_EXPR:
      code1 = expr(root->child, &schema1);
      code2 = expr(root->child->brother, &schema2);
      if(schema1.type != INTEGER || schema2.type != INTEGER)
        semerror(root, "Math operation requires integer types");
      pschema->type = INTEGER;
      switch(qualifier(root))
      {
        case '+' : op = T_PLUS; break;
        case '-' : op = T_MINUS; break;
        case '*' : op = T_TIMES; break;
        case '/' : op = T_DIV; break;
        default: noderror(root);
      }
      return concode(code1,
                     code2,
                     makecode(op),
                     endcode());

  ...
```

File main.c

```c
#include "def.h"

extern int yydebug;
extern Pnode root;

int main(int argc, char *argv[])
{
    Code code;

    if(argc > 1) yydebug = 1;
    init_compiler();
    yyparse();
    code = program(root);
    codeprint(code, 0);
}

void init_compiler()
{
    init_lextab();
    init_symtab();
}
```