浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	赵涵斌（组长）、叶陈韬、卢涛
学　　院：	计算机学院
系：	计算机科学与技术
专　　业：	计算机科学与技术
学　　号：	3140102417、3130102437、3140102441
指导教师：	陈纯、冯雁

2017 年　　06 月　　01 日

目录

O、序言

0.1.综述

我们小组此次实验实现的是一个 tiger 编译器，该编译器实现了 tiger 的全部语法，我们组按照书上的内容，一章一章完善编译器的功能，最终实现的功能有：输出抽象语法树、输出中间代码 IR 树，以及对代码的分析报错功能。工程一共包含 36 个文件，每个文件对应的功能如下。

0.2.文件说明

我们的编译器基于 Linux 的 Flex+Bison 环境编写调试，以下为具体文件说明。

文件名	文件说明
absyn.h、absyn.c	抽象语法树节点与构造函数定义
env.h、env.c	值环境与类型环境的符号表构造及初始化函数、符号表的值结构定义。
errormsg.h、errormsg.c	输出错误信息，包含文件名、错误位置以及具体的错误原因。
escape.h、escape.c	计算逃逸变量
frame.h、uframe.c	函数域和函数段结构定义
makefile	编译文件
parse.h、parse.c	主函数，用于连接各个模块
prabsyn.h、prabsyn.c	输出抽象语法树
printtree.h、printtree.c	输出中间代码 IR 树
semant.h、semant.c	语义分析、抽象语法树节点转化
symbol.h、symbol.c	环境与内层符号表之间的接口 ,名称表的定义与相关维护操作
table.h、table.c	实现内层符号表结构及功能 ,定义了记录符号插入的

	的堆栈。
temp.h、temp.c	对临时变量的记录和分配
test_translate.c	调用 prittree 里面的函数，输出 IR 树
tiger.lex	词法分析，包括注释处理和 token 位置记录
tiger.y	语法分析，生成抽象语法树
translate.h、translate.c	抽象语法树节点转化为 IR 树节点
tree.h、tree.c	中间代码 IR 树节点结构定义以及生成
types.h、types.c	Tiger 语言变量、操作等类型定义
util.h、util.c	基本函数定义

0.3.组员分工

赵涵斌	语法分析、中间代码生成
叶陈韬	抽象语法树、语义分析
卢涛	词法分析、中间代码生成

注：本实验完成过程中组员通力合作，上述分工仅为每个人负责的模块，各个模块在具体开发过程中都有全员参与，每个人总体工作量大致相当（1:1:1）。

一、词法分析

1.1.概念与原理

a) 词法分析概念

词法分析将输入分解成一个个独立的词法符号，即“单词符号”，简称单词。

b) 正则表达式的作用

语言是字符串组成的集合，字符串是符号的有限序列，为了用有限的描述来指明这类（很可能是无限的）语言，可以采用正则表达式表示法，每个正则表达式表示一个字符串集合。

1.2.Tiger 语言词法单词及词法分析实现的正规表达式描述

结合 Tiger 语言参考手册，将词法单词进行罗列。

a) 标识符：

标识符是以字母开头，由字母、数字和下划线组成的序列。区分大小写字母，用正则表达式[A-Za-z][_A-Za-z0-9]*表示。

b) 整型文字常数：

整型文字常数 ,由十进制数字组成的一个序列是一个代表对应整数值的整型常数，用正则表达式[0-9]+表示。

c) 浮点型文字常数

用正则表达式[0-9]+\.[0-9]+表示。

d) 字符串文字常数：

字符串是一个序列 ,由括在双引号之间的零至多个可打印字符、空白符、或转义序列组成。每一个转义序列由转义字符“\”引入，代表一个字符序列。Tiger 允许的转义序列有

\n：系统中表示换行的字符；

\t：制表符 Tab；

\^c：控制字符 c，适用于任何适当的字符；

\ddd：具有 ASCII 码 ddd（3 个十进制数字）的单个字符；

\” ：双引号字符（” ）

\\：反斜线自负（\）

\f_ _ _f\：此序列将被忽略

对于字符串常数 ,返回的字符串值应当包含所有已转换到其含义的转义字符。

e) 保留字：

```
array     {record(); return ARRAY;}
break     {record(); return BREAK;}
do        {record(); return DO;}
end       {record(); return END;}
function  {record(); return FUNCTION;}
for       {record(); return FOR;}
if        {record(); return IF;}
in        {record(); return IN;}
let       {record(); return LET;}
of        {record(); return OF;}
nil       {record(); return NIL;}
then      {record(); return THEN;}
to        {record(); return TO;}
type      {record(); return TYPE;}
var       {record(); return VAR;}
while     {record(); return WHILE;}
```

f) 使用的标点符号：

```
"+"       {record(); return PLUS;}
"-"       {record(); return MINUS;}
"*"       {record(); return TIMES;}
"/"       {record(); return DIVIDE;}
"&"       {record(); return AND;}
"|"       {record(); return OR;}
"="       {record(); return EQ;}
"<>"      {record(); return NEQ;}
"<="      {record(); return LE;}
"<"       {record(); return LT;}
">="      {record(); return GE;}
">"       {record(); return GT;}
"("       {record(); return LPAREN;}
")"       {record(); return RPAREN;}
"{"       {record(); return LBRACE;}
"}"       {record(); return RBRACE;}
"["       {record(); return LBRACK;}
"]"       {record(); return RBRACK;}
"."       {record(); return DOT;}
","       {record(); return COMMA;}
":="      {record(); return ASSIGN;}
":"       {record(); return COLON;}
";"       {record(); return SEMICOLON;}
{blank}   {record(); continue;}
\n        {record(); EM_newline(); continue;}
```

1.3.数据结构

由于词法分析通过 lex 或 flex 工具来完成，不需要手工编写有限自动机，因此需要用到的数据结构比较简单，只有一些用于保存当前 token 及 token 位置的简单变量或结构体

1.4.具体实现

利用 flex 工具实现词法分析器，先定义了 tiger 语言的所有 tokens，然后写出正则表达式（tiger.lex），最后用 flex 生成 C 源代码。

1.5.解决问题

a) 处理注释

/*....*/视为注释，程序会忽略注释内容，并可以处理循环注释。

```
"/*"                    {record(); comment_cnt++; BEGIN comment;}
<comment>"/*"           {record(); comment_cnt++; BEGIN comment;}
<comment>"*/"           {record(); comment_cnt--; if (!comment_cnt) BEGIN nocomment;}
```

b) 处理字符串

程序设置了一个 buffer，用于接受字符串。

string_start 表示开始接受字符串，string_end 表示结束接受字符串

```
\"      {record(); string_start(); BEGIN string;}
<string>{
\\n             {record(); char_to_string(0x0A);}
\\t             {record(); char_to_string(0x09);}
\\              {record(); char_to_string(0x5c);}
"\\\""          {record(); char_to_string(0x22);}

\\[0-9]{3}      {record(); char_to_string(atoi(yytext));}
\"              {record(); string_end();  yylval.sval = strdup(str); BEGIN (0); return STRING;}
```

c) 错误处理

根据词法分析记录的每个 token 的位置信息 ,以及语法分析和语义分析得到的错误原因，输出错误的文件、错误的位置以及错误的原因。

```
if (fileName)
        fprintf(stderr,"error : file : %s:",fileName);
if (line)
        fprintf(stderr, "%d.%d: ", linenum, position - line->i);
```

二、语法分析

2.1.概念与原理

a) 语法分析概念

语法分析用于分析程序的短语结构。

b) 上下文无关文法

用上下文无关文法描述一种语言，其产生式的右部有 0 至多个符号。每一个符号或者是终结符，来自该语言字符串字母表中的单词，或者是非终结符，出现在某个产生式的左部。

2.2.Tiger 语法分析文法实现描述

结合 Tiger 语言参考手册，实现.y 文件的编写。

a) 标识符

```
id: ID {$$ = S_Symbol($1); }
```

b) 声明

声明序列：

```
decs: dec decs {$$ = A_DecList($1, $2);}
| {$$ = NULL;}

dec: tydeclist {$$ = $1;}
| vardec {$$ = $1;}
| fundeclist {$$ = $1;}
```

Tiger 中类型和类型声明的语法：

```
tydeclist: tydec tydeclist {$$ = A_TypeDec(EM_tokPos,
    A_NametyList($1, $2->u.type));}
| tydec {$$ = A_TypeDec(EM_tokPos, A_NametyList($1, NULL));}

tydec: TYPE id EQ ty {$$ = A_Namety($2, $4);}

ty: id {$$ = A_NameTy(EM_tokPos, $1);}
| LBRACE typefields RBRACE {$$ = A_RecordTy(EM_tokPos, $2);}
| ARRAY OF id {$$ = A_ArrayTy(EM_tokPos, $3);}

typefields: id COLON id COMMA typefields {$$ =
    A_FieldList(A_Field(EM_tokPos, $1, $3), $5);}
| id COLON id {$$ = A_FieldList(A_Field(EM_tokPos, $1, $3), NULL);}
| {$$ = NULL;}
```

变量声明：

```
vardec: VAR id ASSIGN exp {$$ = A_VarDec(EM_tokPos, $2, NULL, $4);}
| VAR id COLON id ASSIGN exp {$$ = A_VarDec(EM_tokPos, $2, $4,
    $6);}
```

函数声明：

```
fundeclist: fundec fundeclist {$$ = A_FunctionDec(EM_tokPos,
    A_FundecList($1, $2->u.function));}
| fundec {$$ = A_FunctionDec(EM_tokPos, A_FundecList($1, NULL));}

fundec: FUNCTION id LPAREN typefields RPAREN EQ exp {$$ =
    A_Fundec(EM_tokPos, $2, $4, NULL, $7);}
| FUNCTION id LPAREN typefields RPAREN COLON id EQ exp {$$ =
    A_Fundec(EM_tokPos, $2, $4, $7, $9);}
```

使用好作用域规则。

c) 变量和表达式

左值：

```
lvalue: id {$$ = A_SimpleVar(EM_tokPos, $1);}
| lvalue DOT id {$$ = A_FieldVar(EM_tokPos, $1, $3);}
| id LBRACK exp RBRACK {$$ = A_SubscriptVar(EM_tokPos,
    A_SimpleVar(EM_tokPos, $1), $3);}
| lvalue LBRACK exp RBRACK {$$ = A_SubscriptVar(EM_tokPos, $1,
    $3);}
```

表达式：分为左值、无值表达式、序列、无值、整型文字常数、字符串文字常数、负值、函数调用、算数操作、比较、字符串比较、布尔操作、操作符的优先级、操作符的结合性、记录创建、数组创建、数组和记录赋值、生存期、赋值、if-then-else、if-then、while、for、break、let、圆括号。

```
exp: lvalue {$$ = A_VarExp(EM_tokPos, $1);}
| lvalue ASSIGN exp {$$ = A_AssignExp(EM_tokPos, $1, $3);}
| IF exp THEN exp ELSE exp {$$ = A_IfExp(EM_tokPos, $2, $4, $6);}
| IF exp THEN exp {$$ = A_IfExp(EM_tokPos, $2, $4, NULL);}
| WHILE exp DO exp {$$ = A_WhileExp(EM_tokPos, $2, $4);}
| BREAK {$$ = A_BreakExp(EM_tokPos);}
| NIL {$$ = A_NilExp(EM_tokPos);}
| seq {$$ = $1;}
| INT {$$ = A_IntExp(EM_tokPos, $1);}
| DOUBLE {$$ = A_DoubleExp(EM_tokPos, $1);}
| STRING {$$ = A_StringExp(EM_tokPos, $1);}
| funcall {$$ = $1;}
| MINUS exp %prec UMINUS {$$ = A_OpExp(EM_tokPos, A_minusOp,
    A_IntExp(EM_tokPos, 0), $2);}
```

```
| exp PLUS exp {$$ = A_OpExp(EM_tokPos, A_plusOp, $1, $3);}
| exp MINUS exp {$$ = A_OpExp(EM_tokPos, A_minusOp, $1, $3);}
| exp TIMES exp {$$ = A_OpExp(EM_tokPos, A_timesOp, $1, $3);}
| exp DIVIDE exp {$$ = A_OpExp(EM_tokPos, A_divideOp, $1, $3);}
| exp EQ exp {$$ = A_OpExp(EM_tokPos, A_eqOp, $1, $3);}
| exp NEQ exp {$$ = A_OpExp(EM_tokPos, A_neqOp, $1, $3);}
| exp GT exp {$$ = A_OpExp(EM_tokPos, A_gtOp, $1, $3);}
| exp LT exp {$$ = A_OpExp(EM_tokPos, A_ltOp, $1, $3);}
| exp GE exp {$$ = A_OpExp(EM_tokPos, A_geOp, $1, $3);}
| exp LE exp {$$ = A_OpExp(EM_tokPos, A_leOp, $1, $3);}
| exp AND exp {$$ = A_IfExp(EM_tokPos, $1, $3, A_IntExp(EM_tokPos,
    0));}
| exp OR exp {$$ = A_IfExp(EM_tokPos, $1, A_IntExp(EM_tokPos,1),
    $3);}

| LET decs IN explist END {$$ = A_LetExp(EM_tokPos, $2,
    A_SeqExp(EM_tokPos, $4));}
| record {$$ = $1;}
| array {$$ = $1;}
| FOR id ASSIGN exp TO exp DO exp {$$ = A_ForExp(EM_tokPos, $2, $4,
    $6, $8);}
```

程序：Tiger 程序没有参数，程序就是一个表达式 exp

```
program:  exp {absyn_root=$1;}
```

记录：

```
record: id LBRACE refields RBRACE {$$ = A_RecordExp(EM_tokPos, $1,
    $3);}

refields: id EQ exp COMMA refields {$$ = A_EfieldList(A_Efield($1,
    $3), $5);}
        | id EQ exp {$$ = A_EfieldList(A_Efield($1, $3), NULL);}
        | {$$ = NULL;}
```

序列：

```
seq: LPAREN explist RPAREN {$$ = A_SeqExp(EM_tokPos, $2);}
explist: exp SEMICOLON explist {$$ = A_ExpList($1, $3);}
| exp {$$ = A_ExpList($1, NULL);}
| {$$ = NULL;}
```

数组：

```
array: id LBRACK exp RBRACK OF exp {$$ = A_ArrayExp(EM_tokPos, $1,
    $3, $6);}
```

函数调用：

```
funcall: id LPAREN args RPAREN {$$ = A_CallExp(EM_tokPos, $1, $3);}

args: exp COMMA args {$$ = A_ExpList($1, $3);}
    | exp {$$ = A_ExpList($1, NULL);}
    | {$$ = NULL;}
```

2.3.数据结构

由于语法分析是通过 yacc 或 bison 等工具实现，并不需要手工实现下推自动机，因此在语法分析阶段需要用到的数据结构较少。在语法分析的过程中同时构造了抽象语法树。

2.4.具体实现

语法分析的实现通过 bison 工具完成。另外，为了在本阶段生成抽象语法树，在每条文法规则后加上动作，构造对应的树节点（见下表），经过递归调用得到完整的抽象语法树

三、抽象语法树

3.1.概念与原理

早期的一些编译器选择直接在语法制导翻译中生成代码，但是出于可阅读性和高维护性的考量，现代编译器往往采用了一种叫做抽象语法树的中间表达作为语法分析与代码生成之间的接口。抽象语法树是对具体分析树的抽象和浓缩。抽象语法树消除了具体分析树种诸多冗余的部分，但同时又保留了具体分析树中具有实际意义的部分，从而提取语法分析中产生的结果，并将其传递给后续分析环节。

我们的 tiger 编译器也使用了抽象语法树这一结构来作为语义分析及代码翻译的基础。下面介绍抽象语法树相关的数据结构及具体实现思路。

3.2.数据结构

抽象语法树的节点类型是抽象语法树的一个重要的信息载体。节点的类型往往决定了后续分析和代码翻译的具体动作。因此，针对语法中的每一个类别，我们都为其声明相应的抽象语法树节点类型，下面展示的是各个节点类型的名称。

```
typedef struct A_var_ * A_var;
typedef struct A_exp_ * A_exp;
typedef struct A_dec_ * A_dec;
typedef struct A_ty_   * A_ty;

typedef struct A_decList_ * A_decList;
typedef struct A_expList_ * A_expList;
typedef struct A_field_ * A_field;
typedef struct A_fieldList_ * A_fieldList;
typedef struct A_fundec_ * A_fundec;
typedef struct A_fundecList_ * A_fundecList;
typedef struct A_namety_ * A_namety;
typedef struct A_nametyList_ * A_nametyList;
typedef struct A_efield_ * A_efield;
typedef struct A_efieldList_ * A_efieldList;
```

针对每种类型的节点，我们要对其进行具体的结构声明。下面我们以其中较为复杂的 struct A_exp_为例，来介绍抽象语法树的节点结构。

下述代码片段展示的是 struct A_exp_结构的一部分，后续部分先用省略号来代替。可以看到其中有两个关键变量，一个是枚举类型变量 kind，另一个是 int（A_pos）型的变量 pos。其中 kind 变量指明了这个节点所属的具体类型，即这个节点对应到语法规则中具体的含义。通过分析 tiger 的语法我们可以了解，tiger 的文法中 exp 这一非终结符对应的文法很多，其中一大部分需要构造出 A_exp 节点，kind 记录了这一信息，在后续分析中只要查询 kind 变量的值就可以清楚当前节点对应的意义，并采取相应动作。pos 变量则存储了位置信息，便于精准报错。

```
struct A_exp_{
    enum {
        A_varExp,  A_nilExp,  A_intExp,A_doubleExp,  A_stringExp,
A_callExp,
        A_opExp, A_recordExp, A_seqExp, A_assignExp, A_ifExp,
        A_whileExp, A_forExp, A_breakExp, A_letExp, A_arrayExp
```

```
    } kind;
    A_pos pos;
    …
};
```

下述代码片段展示的是 struct A_exp_结构的余下部分，前述部分以省略号代替。该部分声明了一个 union 变量 u。u 用于存储上述所声明的各种具体 kind 中的个性化信息。如 var 表达式需要 A_var var 来存储变量信息；int 表达式需要 int intt 来存储 int 型常量；call 表达式需要结构变量 call 来存储函数调用时所调用的具体函数及传递的参数；nil 表达式则不需要 u 的任何变量。即该变量用以存储每种 kind 节点丰富的特有信息。

```
struct A_exp_{
    …
     union {
          A_var var;
           int intt;
          double doublee;
           string stringg;
           struct {S_symbol func; A_expList args;} call;
           struct {A_oper oper; A_exp left; A_exp right;} op;
           struct {S_symbol typ; A_efieldList fields;} record;
           A_expList seq;
           struct {A_var var; A_exp exp;} assign;
           struct {A_exp test, then, elsee;} iff;
           struct {A_exp test, body;} whilee;
           struct {S_symbol var; A_exp lo,hi,body; bool escape;} forr;
           struct {A_decList decs; A_exp body;} let;
           struct {S_symbol typ; A_exp size, init;} array;
    } u;
};
```

3.3.具体实现

有了上述定义的数据结构以后，即可以开始准备抽象语法树的实现。抽象语法树的实现过程主要由构造函数的声明和 yacc 源文件中的语义动作的定义构成。

构造函数用以提供一个方法构造出如上所示的各种节点。由于上述节点内部有定义具体的 kind，并且各种 kind 在 union u 中所存储的信息有所区别，因此我们在实现构造函数时，对同一种节点类型也需要建立不同的构造函数。

不过构造函数的逻辑相对比较清晰，即获取及存储所需的信息，并为节点的 kind 赋值。下面我们来看其中的一个例子：

```
A_exp A_VarExp(A_pos pos, A_var var)
{
    A_exp p = checked_malloc(sizeof(*p));
    p->kind=A_varExp;
    p->pos=pos;
    p->u.var=var;
    return p;
}
```

上述代码即定义了 var 表达式节点的构造函数，可以看到构造的逻辑比较清楚。新建一个 A_exp 的指针节点，为其分配内存空间，然后赋予对应的 kind 值，并将参数表中传递进来的参数一一存储，即完成了构造函数的定义。其他构造函数的定义过程也类似。

构筑完成构造函数后，我们需要用构造函数来完成抽象语法树的构造。这一环节可以借助语法分析过程中 Yacc 的语义动作来实现。在编写语义动作之前，我们需要先对 yacc 的终结符和非终结符类型进行相应的声明，因为我们需要通过规定语法分析中符号的类型来达到生成和存放节点指针的作用。

```
%union {
    int pos;
    int ival;
    double dval;
    string sval;
    S_symbol sym;
```

```
    A_var var;
    A_exp exp;
     A_dec dec;
    A_decList declist;
    A_expList explist;
    A_efieldList efieldlist;
    A_ty ty;
    A_fieldList fieldlist;
    A_fundec fundec;
    A_namety namety; }
```

通过在 union 中一一指明 ,我们便可以将自己定义的类型使用到 yacc 中去。接下来我们就可以通过形如%token <sval> ID STRING 的形式，对各个终结符非终结符进行类型绑定。

在绑定好类型后，我们在每条文法规则后加上{语义动作}，并在语义动作中生成该文法所需的抽象语法树节点。如下述代码所示：

```
exp: lvalue {$$ = A_VarExp(EM_tokPos, $1);}
    | funcall {$$ = $1;}
    | lvalue ASSIGN exp {$$ = A_AssignExp(EM_tokPos, $1, $3);}
    | NIL {$$ = A_NilExp(EM_tokPos);}
    | seq {$$ = $1;}
    | INT {$$ = A_IntExp(EM_tokPos, $1);}
    | DOUBLE {$$ = A_DoubleExp(EM_tokPos, $1);}
    | STRING {$$ = A_StringExp(EM_tokPos, $1);}
...
```

如第一条规则（exp: lvalue {$$ = A_VarExp(EM_tokPos, $1);}）所示，在归约时，调用相应的抽象语法树节点构造函数，并将所需的信息传给构造函数，最后得到节点指针保存在$$中，继续往根节点传递。持续这一过程，我们便可以在程序的 start 非终结符中得到整颗抽象语法树，并获得它的根节点。

四、语义分析

4.1.概念与原理

语义分析在编译器中的作用也十分重要，在这个阶段，编译器需要将变量的定义与他们的各个使用联系起来，检查每一个表达式是否有正确的类型，并将抽象语法树转换成更简单的、适合于生成机器代码的表示。

在我们的 tiger 编译器中，我们基于上个阶段中产生的抽象语法树，进行了类型检查，变量的上下文相关等语义分析内容。

4.2.数据结构

在语义分析过程中，由于要记录上下文相关的一些信息，创立和维护符号表这样的数据结构十分重要。我们在类型检查，变量与函数的声明调用环节中常常需要进行符号表操作。我们使用了哈希表作为符号表的主要存储数据结构。对可能频繁使用的字符串比较进行了优化，维护一个名称的哈希表，并以名称的指针作为 key 传递给 hash 表进行存储及相应的比较。同时我们也对类型和值的不同环境进行了分割。下面我们将结合代码介绍一下我们所定义的符号表这一数据结构。

我们由外而内地讨论这个问题，即从环境的考虑出发，到最内部的具体表结构。如下是环境中声明的数据结构和函数接口：

```
typedef struct E_enventry_ * E_enventry;

/* declaration of enventry */
struct E_enventry_ {
    enum {E_varEntry, E_funEntry} kind;
    union {
        struct {Tr_access access; Ty_ty ty;} var;
        struct {Tr_level level; Temp_label label; Ty_tyList formals; Ty_ty result;} fun;
```

```
    } u;
};

/* constructor of two types of entry */
E_enventry E_VarEntry(Tr_access access, Ty_ty ty);
E_enventry E_FunEntry(Tr_level level, Temp_label label, Ty_tyList, Ty_ty
result);

/* constructor of two types of table */
S_table E_base_tenv(void);
S_table E_base_venv(void);
```

声明 struct E_enventry_，用来分别作为值/类型与函数的 value。也是使用 kind 来进行类型区分，用 union 来存储对应信息。同时，我们需要提供两种 entry 的构造函数，这两个函数会分别返回两种类型的 struct E_enventry_指针。最后，为了区分值与类型两种环境，我们将符号表进行分离，分别用两个构造函数来构造值和类型两种符号表。在实现环节，我们需要调用符号表的构造函数，构造过程中还需对两种符号表进行不同的初始化处理（初始类型、初始函数）。

在符号名称表的阶段，我们需要向内调用实际存储使用的哈希表，并向外提供符号表的构造、插入、查找、删除等各种接口，同时还要在这个阶段实现符号名称表的维护以及一些过渡操作。下述代码即是符号表中定义的数据结构和接口：

```
typedef struct S_symbol_ *S_symbol;
struct S_symbol_ { string name; S_symbol next; };
#define SIZE 109
static S_symbol hashtable[SIZE];

S_symbol S_Symbol(string);
string S_name(S_symbol);

typedef struct TAB_table_ * S_table;
S_table S_empty(void);
void S_enter(S_table t, S_symbol sym, void *value);
```

```
void *S_look(S_table t, S_symbol sym);
void S_beginScope(S_table t);
void S_endScope(S_table t);
```

如上所示，通过 struct S_symbol_这一数据结构，我们可以构造存储 static S_symbol hashtable[SIZE]哈希表来存储相应的值和类型名称，而通过传递 struct S_symbol_结构的指针作为 key 来进行内层哈希 ,可以大大减少由字符串比较带来的额外时间开销。

内层的具体哈希表声明如下：

```
typedef struct TAB_table_ *TAB_table;
#define TABSIZE 109
typedef struct bucket_ *bucket;
struct bucket_ { void *key; void *value; bucket next; };
struct TAB_table_ {
    bucket table[TABSIZE];
    bucket stack;
};
/* Make a new table */
TAB_table TAB_empty(void);

/* Enter key->value into table t, shadowing any previous binding for key. */
void TAB_enter(TAB_table t, void *key, void *value);

/* Look up key in table t */
void *TAB_look(TAB_table t, void *key);

/* Pop the most recent binding and return its key. This may expose another binding for the same key. */
void *TAB_pop(TAB_table t);
```

struct bucket_结构为哈希表的基本元素，struct TAB_table_是整个 table 结构，其中 table 为实际存储使用的哈希表，而 stack 则是 FILO 地存放了元素

的添加顺序，主要用于退出 scope 时的清空操作。声明的各个函数接口即进行新建，增加、删除、查询等操作，具体实现方法在下一小节中展开。

4.3.具体实现

在有了符号表这一数据结构之后，即可以开始准备语义分析的实现。该部分的实现主要由符号表的具体功能实现、类型检查函数及外层的语义分析函数构成。

首先来看符号表实现，我们从内而外地实现符号表。底层 TAB_table 的各项函数实现如下：

```
TAB_table TAB_empty(void)
{
    TAB_table t = checked_malloc(sizeof(*t));
    int i;
    t->stack = NULL;
    for (i = 0; i < TABSIZE; i++)
        t->table[i] = NULL;
    return t;
}

void TAB_enter(TAB_table t, void *key, void *value)
{
    if (t == NULL || key == NULL)
    {
        printf("null pointer error.");
        return;
    }
    int index;
    index = ((unsigned)key) % TABSIZE;

    bucket b = checked_malloc(sizeof(*b));
    b->key = key; b->value = value; b->next = t->table[index];
    t->table[index] = b;
```

```
    bucket s = checked_malloc(sizeof(*s));
    s->key = key; s->next = t->stack;
    t->stack = s;
}

void *TAB_look(TAB_table t, void *key)
{
    if (t == NULL || key == NULL)
    {
        printf("null pointer error.");
        return;
    }
    int index;
    bucket b;
    index = ((unsigned)key) % TABSIZE;
    for (b = t->table[index]; b; b = b->next)
        if (b->key == key) return b->value;
    return NULL;
}

void *TAB_pop(TAB_table t) {
    if (t == NULL)
    {
        printf("null pointer error.");
        return NULL;
    }
    bucket s, b; int index;
    s = t->stack;
    if (s == NULL)
    {
        printf("the table is empty.");
```

```
        return NULL;
    }
    index = ((unsigned)s->key) % TABSIZE;
    b = t->table[index];
    if (b == NULL)
    {
        printf("unexpected pop error.");
        return NULL;
    }
    t->table[index] = b->next;
    t->stack = s->next;
    return b->key;
}
```

其中值得关注的是 key 的处理和 stack 变量的相关维护操作。在 enter 函数中 ,我们用 index = ((unsigned)key) % TABSIZE ,将指针 key 转化为 unsigned 类型再取余来获取 index 值，代替了原本字符串的哈希函数。同时，在 enter 时维护 stack 栈。在 pop 函数中，我们通过 stack 栈顶指针得到对应的元素，并通过 hash 找到其在 hash 表中的位置，由于 stack 的 FILO 特性，该元素一定在 table[index]指针所指的位置，即可以通过相应指针操作删除该元素。

名称符号表中，我们需要维护 static S_symbol hashtable[SIZE]这一结构，其函数实现与内层哈希操作比较接近。而对外的各个符号表接口则是通过调用内层哈希表的各种函数实现。其中值得注意的是为了保持 scope 的准确性，我们需要在进入 scope 和退出 scope 时对应做出处理，具体实现如下：

```
static struct S_symbol_ marksym = { "<mark>", 0 };

void S_beginScope(S_table t)
{
    S_enter(t, &marksym, NULL);
}

void S_endScope(S_table t)
```

```
{
    S_symbol s;
    do s = TAB_pop(t);
    while (s != &marksym);
}
```

即在进入 scope 时插入一个 marksym 元素，而在退出 scope 时需要一直做 pop 操作，直到 pop 出第一个 marksym 元素为止。

在具体语义分析函数中，我们的核心思想是通过对不同表达式类型（即不同抽象语法树节点的 kind）分别处理，创建并维护了值与类型两张符号表，并结合语法进行相应的语义分析动作。在函数的实现过程中，抽象的概念比较重要，我们将所需做的语义分析动作进行了相应的归类，并抽象出来形成独立的函数，这可以提高编译器的可读性，也为编译器的后续维护带来了一些遍历。例如在语义分析过程中，除去表达式处理的函数，我们还需要定义变量处理、类型检查、函数参数检查等函数，同时，由于后续的代码生成过程与之类似，所以我们可以预留一些接口以供后续使用。

五、翻译成中间代码

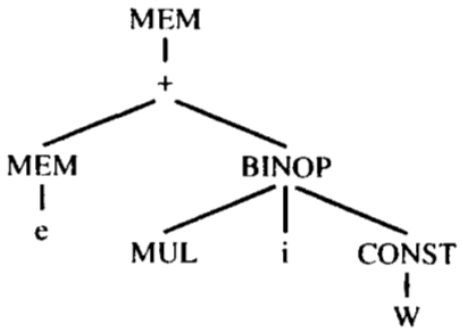
5.1.概念与原理

中间表示概念

中间表示是一种抽象机器语言，它可以表示目标机的操作，而不需太多地涉及机器相关的细节，它独立于源语言的细节。

1. 中间代码格式

中间代码采用 IR tree 来表示，直接将抽象语法树翻译成 IR tree。



MEM(+(MEM(*e*), BINOP(MUL, *i*, CONST *W*)))

2. 中间表示树及数据结构描述

IR 节点结构如下：

```
T_stm T_Seq(T_stm left, T_stm right);
T_stm T_Label(Temp_label);
T_stm T_Jump(T_exp exp, Temp_labelList labels);
T_stm T_Cjump(T_relOp op, T_exp left, T_exp right,
        Temp_label true, Temp_label false);
T_stm T_Move(T_exp, T_exp);
T_stm T_Exp(T_exp);

T_exp T_Binop(T_binOp, T_exp, T_exp);
T_exp T_Mem(T_exp);
T_exp T_Temp(Temp_temp);
T_exp T_Eseq(T_stm, T_exp);
T_exp T_Name(Temp_label);
T_exp T_Const(int);
T_exp T_Double(float);
T_exp T_Call(T_exp, T_expList);
```

记录了节点类型以及子节点的类型、值信息。

5.2.具体实现

设计 translate.h ,实现 translate.c ,并在语义分析(semant.c 和 semant.h)过程中调用 translate 模块的树节点构造函数接口，由于语义分析是通过函数的递归调用进行的，因此在语义分析完成时就同时生成了完整的 IR tree,最后的结果应当是一个 F_fragList。

六、测试案例

首先用一个正确的代码检验一下编译器的输出结果。这里我们选用的是书上附录中的 merge.tig：

正确用例 MERGE.TIG

```
let

 type any = {any : int}
 var buffer := getchar()

function readint(any: any) : int =
 let var i := 0
       function isdigit(s : string) : int =
            ord(buffer)>=ord("0") & ord(buffer)<=ord("9")
       function skipto() =
          while buffer=" " | buffer="\n"
             do buffer := getchar()
  in skipto();
       any.any := isdigit(buffer);
       while isdigit(buffer)
          do (i := i*10+ord(buffer)-ord("0"); buffer := getchar());
       i
 end

 type list = {first: int, rest: list}

 function readlist() : list =
     let var any := any{any=0}
          var i := readint(any)
```

```
        in if any.any
              then list{first=i,rest=readlist()}
              else nil
      end

 function merge(a: list, b: list) : list =
    if a=nil then b
    else if b=nil then a
    else if a.first < b.first
         then list{first=a.first,rest=merge(a.rest,b)}
         else list{first=b.first,rest=merge(a,b.rest)}

 function printint(i: int) =
  let function f(i:int) = if i>0
            then (f(i/10); print(chr(i-i/10*10+ord("0"))))
    in if i<0 then (print("-"); f(-i))
         else if i>0 then f(i)
         else print("0")
  end

 function printlist(l: list) =
    if l=nil then print("\n")
    else (printint(l.first); print(" "); printlist(l.rest))

    var list1 := readlist()
    var list2 := (buffer:=getchar(); readlist())


  /* BODY OF MAIN PROGRAM */
 in printlist(merge(list1,list2))
end
```

生成的抽象语法树如下所示：(由于树结构过于庞大，完整结构见 Test results 里的 absyn.tree)

absyn.tree ×

```
letExp(
   decList(
      typeDec(
         nametyList(
            namety(any,
               recordTy(
                  fieldList(
                     field(any,
                        int,
                        TRUE),
                     fieldList())))),
            nametyList())),
         decList(
            varDec(buffer,
               callExp(getchar,
                  expList()),
               TRUE),
            decList(
               functionDec(
                  fundecList(
                     fundec(readint,
                        fieldList(
                           field(any,
                              any,
                              TRUE),
                           fieldList()),
                        int,
                        letExp(
```

生成的 IR 树如下所示：(由于树结构过于庞大，完整结构见 Test results 里的 IR.tree)

IR.tree ×

```
NAME L36
NAME L22
NAME L21
NAME L17
NAME L16
NAME L15
NAME L14
SEQ(
    CJUMP(EQ,
        MEM(
            BINOP(PLUS,
                MEM(
                    BINOP(PLUS,
                        TEMP t100,
                        CONST 0)),
                CONST 4)),
        TEMP t105,
        L76,L77),
    SEQ(
        LABEL L76,
        SEQ(
            EXP(
                CALL(
                    NAME L0,
                    MEM(
                        BINOP(PLUS,
                            MEM(
                                BINOP(PLUS,
```

接下来展示几个错误代码实例。

词法检查

代码：

```
/* error: unexpected token */
let

/* calculate n! */
function nfactor(n: int) =
        if   n = 0
            then 1
            else n * nfactor(n-1)

at
    nfactor(10)
end
```

报错结果：

```
tao@ubuntu:~/codes/final_compiler$ ./a.out Test_cases/test1.tig
error : file : Test_cases/test1.tig:10.1: syntax error
!!!Syntax error
tao@ubuntu:~/codes/final_compiler$
```

语法检查

代码：

```
syntax error: no such grammer 'else-then' */
let

/* calculate n! */
function nfactor(n: int) =
```

```
        if   n = 0
             then 1
             else
             then n * nfactor(n-1)

in
    nfactor(10)
end
```

报错结果：

```
tao@ubuntu:~/codes/final_compiler$ ./a.out Test_cases/test2.tig
error : file : Test_cases/test2.tig:9.4: syntax error
!!!Syntax error
tao@ubuntu:~/codes/final_compiler$
```

表达式类型检查

代码：

```
/* error: comparison of incompatible types */

3 > "df"
```

报错结果：

```
tao@ubuntu:~/codes/final_compiler$ ./a.out Test_cases/test3.tig
error : file : Test_cases/test3.tig:3.8: !!!Unexpected type in comparsion
tao@ubuntu:~/codes/final_compiler$
```

函数返回值检查

代码：

```
/* error : procedure returns value    and procedure is used in arexpr */
let

/* calculate n! */
```

```
function nfactor(n: int) =
        if   n = 0
             then 1
             else n * nfactor(n-1)

in
    nfactor(10)
end
```

报错结果：

```
tao@ubuntu:~/codes/final_compiler$ ./a.out Test_cases/test4.tig
error : file : Test_cases/test4.tig:8.24: int or double required(op)
error : file : Test_cases/test4.tig:10.1: !!!Incorrect return type in function '
nfactor'
tao@ubuntu:~/codes/final_compiler$
```

数组类型检查

代码：

```
/* error : initializing exp and array type differ */

let
    type arrayty = array of int

    var a := arrayty [10] of " "
in
    0
end
```

报错结果：

```
tao@ubuntu:~/codes/final_compiler$ ./a.out Test_cases/test5.tig
error : file : Test_cases/test5.tig:7.1: !!!Unmatched array type in arrayty
tao@ubuntu:~/codes/final_compiler$
```

变量作用域

代码：

```
/* error : second function uses variables local to the first one, undeclared
variable */
let

function do_nothing1(a: int, b: string):int=
        (do_nothing2(a+1);0)

function do_nothing2(d: int):string =
        (do_nothing1(a, "str");" ")

in
    do_nothing1(0, "str2")
end
```

报错结果：

```
tao@ubuntu:~/codes/final_compiler$ ./a.out Test_cases/test6.tig
error : file : Test_cases/test6.tig:8.17: !!!Undefined var a
error : file : Test_cases/test6.tig:8.17: !!!Undefined var a
tao@ubuntu:~/codes/final_compiler$
```