

浙江大学

本科实验报告

课程名称：编译原理

姓名：陈则衔；范天行；叶培峰

学院：计算机科学与技术

系：计算机科学与技术

专业：计算机科学与技术

学号：3140102231；3140102232；3140104023

指导教师：陈纯 冯雁

1.序言

此次实验作业因为设计内容实在太广，我们选择在虎书课本给予的框架基础上，构建我们的整个编译器。我们按照课本中的顺序，依次完成了前 1-7 章的各项内容，以课本提及的方法完成了词法分析，语法分析，语义分析，中间树的生成；并使用自己设计的方法，完成了汇编代码的生成，与抽象树、中间树的打印。

我们最终提交的文件说明：

Code 文件

该文件中存放着我们编译器所用的所有源代码，具体如下：

- Lex.yy.c,词法分析器。
- y.tab.c、y.tab.h,语法分析器
- absyn.h,Tiger 的抽象语法声明
- absyn.c, 构造函数的实现代码
- prabsyn.[ch],抽象语法树输出程序
- errmsg.[ch], 存放出错信息的数据结构，有助于产生带有文件名和行号的报错信息。
- parse.[ch], 一个驱动程序，它运行你的分析器来分析一个输入文件。
- symbol.[ch],将字符串转换为符号的模块。
- table.c、table.h, hash
- type.h、type.c, 描述了 Tiger 语言的数据类型
- temp.h,temp.c,支持临时变量和标号的模块
- frame.h\frame.c,uframe.c, 抽象接口
- escape.h、escape.c, 计算逃逸变量
- sement.c、sement.h, 局部变量分配存储单元，并且调用 translate 将抽象语法树转成中间树
- translate.c、ttranslate.h,构建中间语法树

exe 文件

该文件中存放着可执行所需要的文件

- displaydemo.tig, 代表运行的 tiger 程序（只能将程序写在 displaydemo.tig 中）
- bin 文件夹为 c 语言编译器，我们需要 c 语言编译器来编译
- tiger.exe 为可执行文件，它会编译 tiger 程序，生成 out.c, IR.txt, IR.json,ST.json 文件；其中，out.c 文件即生成的目标代码（x86 汇编），IR.txt 为中间语法树,IR.json 为中间语法树，ST.json 为抽象语法树，用于 js 可视化显示。
- vcvars32.bat, 当用 tiger.exe 生成 out.c 之后，该程序将编译 out.c 并且产生最后的可执行文件 out.exe.执行 out.exe 即为 tiger 程序执行的结果。

display 文件

该文件中存放着中间语法树以及抽象语法树可视化的源文件及 html 文件。

- displayIR.html, displayIR.js 中间语法树可视化
- Displayst.html, displayst.js 抽象语法树可视化

Tips

值得注意的事，我们需要将 exe 文件中生成的 IR.json 以及 ST.json 拷贝到 display 文件

夹中才可以正常显示。并且，由于 IR.json 中可能不仅仅只含有一颗树，所以在中间树可视化时，需要人工选取其中一颗语法树显示，ST.json 不存在该问题，直接显示即可。

运行环境

请在装有 vs2015 的环境下运行

由于 js 语言的特殊性，请确保 js 文件可以正确读取文件

组员分工

叶培峰：生成目标汇编代码，中间树生成

陈则衔：词法分析、语法分析，部分中间树生成，测试用例与报告撰写

范天行：部分报告，语义分析，与词法分析的抽象树串联，树形结构可视化

2.词法分析

在词法分析这一过程中，我们通过使用 lex 文件，达到分词、记录位置、初步识错的功能。

我们对该三个功能进行一一说明。

分词是词法分析中最重要的目的。主要用正则表达式来对应程序中允许和不允许出现的代码形式。整个分词部分在书本 chapter2 和 chapter4 中的 lex 相关文件上进行修改和完善。

我们可以简单的把代码片段分成三类，分别为注释部分，字符串部分以及剩余的代码片段。显然，在注释部分和字符串部分，可以任意形式的代码形式，也无需去判断其中含有的代码是否有着特殊含义，例如 for,while 等这些关键词。而在除这两者之外的代码片段才是我们需要进行分词的大头部分。因此，我们需要用 %x 来切换分词的状态，我们使用一下几个状态。

%x comment 进行注释部分的分词

%x string 进行字符串部分的分词

%s nocomment 回到非注释部分状态，且在这个部分需要识别大量的关键词以及变量和变量类型。

2.1 非注释部分

在非注释部分的代码段中，我们需要把每一个允许出现的代码形式进行分词，并且识别程序允许出现的关键词。

关键词包括：array break do end else function for if in let of nil then to type var while

变量名应为： [A-Za-Z][_A-Za-z0-9]* 必须以字母开头，由字母、数字以及_组成

数字： [0-9]+ 即 INT 类型

小数： [0-9]+\.[0-9]+ 即 DOUBLE 类型

ws: [\t]+ 多余的空格以及 Tab 符

特殊符号包括 , := ; () { } [] . + - * / = < > <= >= & |

其中，特殊符号最先判断句，:=要优于:先判断；>=要优于>先判断；<=要优于<先判断；其次关键词要优于变量先判断。

2.2 注释部分

每当遇见 “/*”，则标志着注释的开始，遇见 “*/” 则意味着注释的结束，两者一一对应。

“*” {adjust();comment_nest++; BEGIN comment} 标志着注释的开始，进入注释状态的分词

<comment>“*/” {adjust(); comment_nest--; if (!comment_nest) BEGIN nocomment;} 标志着一个注释的结束，所有注释都结束后则结束注释状态，回到非注释状态。这是由于 tiger 语言支持注释嵌套，我们需要一个 comment_nest 来计数，保证嵌套结束才能结束注释状态。

显然注释部分的总体优先级要高于字符串的优先级，接下来我们讨论字符串部分的实现。

2.3 字符串部分

与注释部分一样，当遇见 “” 时，进入 string 状态。同样，遇见再次遇到 “” 代表字符串结束。值得注意的是，在我们的 tiger 语言中，字符串常量的长度不允许超过 512。

但是由于，字符串部分内容有比较特殊的表达方式以及转义符 \ 的存在，我们需要对其进行特殊识别。

```
\“ {adjust(); init_string(); BEGIN string;} 代表字符串开始，进入 string 状态。
<string>{
    \“ {...}代表转义符\本身
    “\\” {...} 即遇见\“，即将\“转义成字面意义本身
    \n {...} 字符串中的换行
    \t {...} 字符串中的 Tab
    \[0-7]{3} {...} 字符串中八进制的用法，如\111\112 代表 IJ
    \“ {adjust(); end_string(); yylval.sval = strdup(str); BEGIN (0); return STRING;}
    即再次碰到\“，结束字符串状态，将字符串返回
    ...
}
```

2.4 记录位置与识错

我们发现，在前面分词是我们频繁的使用了 adjust() 函数，此函数就是记录下当前这个 token 的位置，位置信息是一维的，其代码非常简单，如下所示。

```
void adjust(void)
{
    EM_tokPos=charPos;
    charPos+=yyleng;
}
```

并且，每当遇到 \n 时，即代表行数+1，我们调用了 erromsg 中的 EM_newline() 函数，进行行数的递增，并且将该行的 EM_tokPos（代表该行开始时的位置）以及当前行数加入到一个维护位置的链表当中。即，我们拥有一个变量 lineNum，代表当前的行数，一个链表 linePos 代表从第一行到第 lineNum，每一行的 EM_tokPos 位置，则，第 k 行的长度即为 linePos[k+1]-linePos[k]。当前列数即为当前 EM_tokPos - linePos.当前行。

这样，纠错时，即能反馈当前位置。

由于前面分词已经遍历了所有的可能性，因此剩下的即为错误代码部分，如下

```
. {adjust(); EM_error(EM_tokPos,"illegal token");}
```

我们将该分词放在 lex 最后，技能达到识别错误的目的。

3.语法分析

在该部分中，我们主要通过 yacc 来构建抽象语法。该部分主要由 tiger.y 以及 absyn.h/absyn.c 来完成。

tiger.y 负责将 lex 分词的结果，进行匹配，将匹配的结果通过调用 absyn 中的函数来构成一个抽象语法。抽象语法树传递源程序的短语结构，其中已解决了所有语法分析问题，但不带有任何语义解释，抽象语法要求将抽象语法树表示成数据结构，并且对它进行操作。这便是这一章节所做的工作。

由于编译只有一边，词法分析、语法分析是同时进行的，当我们检查出一个错误，且需要向用户进行汇报时，词法分析器的当前位置就理所应当是最接近错误源的位置，这个就是我们上一部分所讲的 EM_tokPos，一个表示“当前位置”的全局变量，同样，这个全局变量在语法分析中也有着相当重要的作用。因为，我们需要记住抽象语法树的每个节点在源文件中对应的位置，以防万一该结点发生语义错误，因此，为了记住准确的位置，抽象语法数据结构上导出都必须带有 pos 域。pos 域指明了导出抽象语法树的字符在源程序中的位置。这样，类型检查器就能产生有用的报错信息。这样，当我们在语法分析时，可以通过 EM_tokPos 的方式，传入该结点在程序中的位置，以下程序是 lvalue ASSIGN exp 的一个例子：

```
lvalue ASSIGN exp {$$ = A_AssignExp(EM_tokPos, $1, $3);}
```

3.1 抽象语法类型

用 pos 域记录每一个抽象语法树节点只是其中一小部分，在进行抽象语法树构建之前，我们需要对抽象语法树的类型做一个定义。

与书本中一样，我们需要的类型主要包括 A_var, A_exp, A_dec 和 A_ty。

其中，A_var 为变量类型，其相关函数为 A_SimpleVar, A_FieldVar 以及 A_SubscriptVar。在出现左值时，我们需要调用其相关函数。

A_exp 是一个通用的抽象表达式类型，我们整棵树的最后结果可以看做是一整个 A_exp 类型，其类型包括：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。是一个很复杂的数据类型。

A_dec 为声明类型，包括数据类型，变量以及函数的声明，其相关函数主要为 A_FunctionDec, A_TypeDec, A_VarDec。

A_ty 主要为名字、记录以及数组的命名。

与书本一致，我们给出 A_var 的定义写法，并做一个简单的解释。

```
/*absyn.h*/
```

```
typedef struct A_var_ * A_var;
```

```
struct A_var_ {
```

```
    enum {A_simpleVar, A_fieldVar, A_subscriptVar} kind;
```

```
    A_pos pos;
```

```
    union {
```

```
        S_symbol simple;
```

```
        struct {A_var var; S_symbol sym;} field;
```

```
        struct {A_var var; A_exp exp;} subscript;
```

```
    } u;
```

```
};
```

其中, kind 代表这个变量的类型。

simpleVar: id,变量,通过 id 引用一个根据作用于规则可访问的变量或参数。

fieldVar: lvalue.id 记录域,点号表示法允许选择一个记录值相应的命名域。

subscriptVar: lvalue[exp] 数组下标,方括号表示法允许选择与编号对应的数组元素。
数组以从 0 开始的连续整数作为索引。

其在 yacc 中的写法如下:

```
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);}
```

分别对应:

```
lvalue    → id
          → lvalue.id
          → lvalue[exp]
```

再以此为展开,一个简单的赋值语句,例如 a:=5;其包含了左值 a,其整个式子的抽象语法如下:

```
A_AssignExp(4,A_SimpleVar(1,S_Symbol("a")),A_IntExp(3,5))
```

由于其本身是一个赋值语句,因此调用了 A_AssignExp 函数,其在 yacc 中会如下所示:

```
lvalue ASSIGN exp {$$ = A_AssignExp(EM_tokPos, $1, $3);}
```

其中参数,4 代表着 pos,由于 a 为左值,会调用 A_SimpleVar,同样,1 代表了 pos, S_Symbol("a"),代表了 a 这个符号。5 是一个赋值常量,则调用 A_IntExp(3,5),其中 3 同样代表了 pos。

3.2 Tiger 的抽象语法

前面介绍了主要的集中抽象语法类型,并且给出了一个很简单的例子作为抽象语法的说明。现在,我们对 Tiger 的抽象语法做出详细的说明。

在 yacc 中,我们最终得到的结果是一个 program,而 program 可以看做一个个 exp 构成,而 exp 的构成是很丰富的,其包含了 tiger 的语法。关于 exp 的 Yacc 写法如下:

```
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);}
    | LET decs IN explist END {$$ = A_LetExp(EM_tokPos, $2, A_SeqExp(EM_tokPos, $4));}
    | IF exp THEN exp ELSE exp {$$ = A_IfExp(EM_tokPos, $2, $4, $6);}
    | IF exp THEN exp {$$ = A_IfExp(EM_tokPos, $2, $4, NULL);}
    | exp PLUS exp {$$ = 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);}
| MINUS exp %prec UMINUS {$$ = A_OpExp(EM_tokPos, A_minusOp,
A_IntExp(EM_tokPos, 0), $2);}
| exp NEQ exp {$$ = 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 {$$ = A_OpExp(EM_tokPos, A_andOp, $1, $3);}
| exp OR exp {$$ = A_OpExp(EM_tokPos, A_orOp, $1, $3);}
| record {$$ = $1;}
| array {$$ = $1;}
| WHILE exp DO exp {$$ = A_WhileExp(EM_tokPos, $2, $4);}
| FOR lvalue ASSIGN exp TO exp DO exp {$$ = A_ForExp(EM_tokPos, $2, $4, $6, $8);}
| BREAK {$$ = A_BreakExp(EM_tokPos);}

```

我们可以发现，我们调用了很多的抽象语法函数，这些函数的构造函数在书本图 4-2 中有着详细描述，本编译器与书本一致。

我们可以发现，在对于表达式的处理中，并没有“&”和“|”的抽象语法；代替的，我们将 `exp1 AND exp2` 转成了 `if exp1 then exp2 else 0`，而 `exp1 | exp2` 则将转换陈好像它被写为 `if exp1 then 1 else exp2` 一样。这是书中的做法，然而在实际的操作中，这样做并不对。我们用操作 `andOp` 和 `orOp` 来代替 `AND` 和 `OR` 可以得到正确的结果。

该两者的函数调用如下：

```

exp AND exp {$$ = A_OpExp(EM_tokPos, A_andOp, $1, $3);}
exp OR exp {$$ = A_OpExp(EM_tokPos, A_orOp, $1, $3);}

```

类似地，在抽象语法中，一元负（-i）应当表示为减（0-i）。此外，当一个 `LetExp` 的函数题有多个语句是，我们必须使用 `seqExp`。空语句用 `A——seqExp（NULL）` 来表示。而一元负和计算符号的优先级我们用 `yacc` 自行实现，如下

```

/*%nonassoc LOW*/
%right ASSIGN
%left AND OR
%nonassoc EQ NEQ LT LE GT GE
%left PLUS MINUS
%left TIMES DIVIDE
/*%right DO OF ELSE optional*/
%left UMINUS

```

词法分析器返回的 ID 单词携带 `string` 类型的值，而抽象语法要求标识符具有 `symbol` 值。函数 `S_symbol`（见 `symbol.h`）可将字符串转为符号，函数 `S_name` 则可将符号转换为字符串。

编译器的语义分析阶段需要知道哪些局部变量会被嵌套的函数内使用。`varDec` 或 `field`

类型的 `escape` 成员用于记录这种信息。在书本的构造函数的参数中没有提及这个 `escape`（可逃逸）域，但它总被初始化为 `TRUE`，这是一个保守的近似值。`field` 类型及用于形式参数，也用于记录域；`escape` 对形式参数有意义，但对于记录成员则可忽略它。

前文所说，`A_VarDec` 和 `A_Field` 会将 `escape` 成员初始化。而调用 `A_VarDec` 和 `A_Field` 是在 `tiger` 声明时被调用。其相关 `yacc` 语句，根据以下的声明顺序来匹配。

```

decs → {dec}

dec → tydec
    → vardec
    → fundec

tydec → tyde type-id = ty
      ty → type-id
          → {typefield}
          → array of type-id
tyfields → 空字符串
          → id:type-id{,id:type-id}

vardec → var id := exp
        → var id:type-id:= exp

fundec → function id (tyfields) = exp
        → function id (tyfields):type-id=exp

```

最后，我们举一个类型声明的例子。

```
type tree = {key: int, children:treelist}
```

```
Type treelist = {head:tree,tail:treelist}
```

```
var h:tree=nil
```

其抽象语法树转换如下(为了清除期间省略了关于位置的表示):

```

A_DecList(
  A_TypeDec(
    A_NametyList(A_Namety(S_Symbol("tree"),
      A_RecordTy(
        A_FieldList(A_Field(S_Symbol("key"),S_Symbol("int")),
          A_FieldList(A_Field(S_Symbol("children"),S_Symbol("treelist")),
            ,NULL))),
    A_NametyList(A_Namety(S_Symbol("treelist")
      A_RecordTy(
        A_FieldList(A_Field(S_Symbol("head"),S_Symbol("tree")),
          A_FieldList(A_Field(S_Symbol("tail"),S_Symbol("treelist")),
            ,NULL))),
    NULL))),
  A_VarDec(
    S_Symbol("h"),S_Symbol("tree"),A_NilExp()));

```


4.语义分析

在经过词法分析与语法分析模块后，我们获得了一颗抽象语法树。但抽象语法树直接翻译为汇编代码是做不到的。生成抽象语法树的过程中，只能检查出词法、语法方面的错误。但是诸如类型、变量未定义等错误，是无法被检查出来的。通过语义分析模块，我们将各种类型、变量与他们的使用相联系，检查每一个表达式是否有正确的类型，并尝试着将抽象语法转化为更简单，适用于生成汇编的表示，为目标代码的生成铺路。

语义分析模块主要新涉及到：

Semant、translate、frame、temp、table、env、symbol 等文件，

其中 temp、table 模块，使用的是虎书提供的代码，而 semant、translate、frame、env、symbol 模块为我们拓展书本上的示例代码后写得。

因为这个模块与中间代码的生成关系很大，translate 模块与 tree 模块息息相关，因此可能会涉及一些应写在代码生成模块的内容。

接下来，我们会在以下几个方面，介绍我们的语义分析模块：

1. 命令式符号表的搭建与环境初始化
2. 活动记录，栈帧与静态链的实现
3. 语义分析模块入口介绍，与内部逻辑分析

4.1 命令式符号表的搭建与环境初始化

命令式符号表的搭建较为简单易懂，因此我们选择这种方式建立程序内部的变量、类型环境。

命令式符号表将所有环境中的符号，散列到同一张散列表中。因此在符号表结构层面，是无法区分不同层环境的。因此我们使用链表的方式维护一个给出符号压入符号表顺序的辅助栈，在创建新的环境时，压入一个标记符，在退出此环境时，按照辅助栈链表记录的压栈顺序，抛出符号表中的内容，直到抛出之前压入的标记符为止。这是课本上的内容，就不加赘述了。

因为符号表是抽象的结构，不会反映在中间代码中，在符号表中需要同时保存其对应的栈中位置，这部分会在之后加以说明。

在 Tiger 语法中，存在一些预定义的类型与内置函数。这些类型、函数无需用户定义，可以直接被使用。因此在符号表中初始环境生成时，就需要将这些符号预定义在符号表中，已让之后用户的所有操作，都可以使用这些类型与函数。

以下类型需要被定义在类型表中：

int 类型、string 类型、double 类型。Tiger 语法支持 int 与 string，虽然我们加入了 double 类型，但在汇编层面，我们没有将其实现。

而 array、record 类型的内容，可以由基本类型组合、迭代获得。其他的类型也是这样。

我们将以下函数定义在变量表中：

Print、flush、getchar、ord、chr、size、substring、concat、not、exit。这十个函数均被定义在 chap12 章节，课本给予的代码中，使用 C 语言实现。用户可以在 Tiger 语言中直接使用这十个函数。不过我们没有将 initArray、allocRecord、string 预定义，因为我们不希望这几个函数被用户直接使用。

4.2 活动记录，栈帧与静态链的实现。

静态链是一种清晰的结构，其指向上一层的环境，并维护一个自己层的栈。使用静态链可以轻松找到属于父层的栈中变量。静态链被保存在符号表中，对每一个变量、函数，都保存其静态链信息。这部分信息可以用来在生成中间树时，输出变量在栈中的位置。

我们最终决定汇编代码使用 x86 实现。并将所有变量视作逃逸，全部保存在栈中，以简化最终的情形。虽然在抽象的活动记录中，我们使用了静态链这种结构，但是在最终汇编的代码中并没有体现出这种耗费极大的寻找变量的方式，而是通过使用定长的栈，并记录在静态链的跳跃次数，直接计算出变量在栈中的位置，根据这个位置生成汇编代码，并祈祷用户不会栈溢出。这样可以节省多次寻址的时间。

不过在抽象层，使用静态链是有好处的。与 temp 合用，可以在汇编落地前，生成一个清楚明白的、易查询的链状结构，且具有可拓展性。

4.3 语义分析模块入口介绍，与内部逻辑分析

语义分析模块使用 SEM_transProg 函数作为入口，输入抽象语法树，并最终生成中间树与 fraglist，完成到第七章的所有内容。

语义分析一开始，因为汇编代码的关系，需要初始化 F_FP(), 作为当前栈的栈帧标号，接着初始化变量与类型符号表，然后采用递归的过程，分析抽象语法树的每一层结构。对于每一类操作，若其不能达成时，将报错。其行为如下：

Declaration	Var	在变量表中定义 var 并赋初值，生成静态链，压栈
	Type	在类型表中定义 type
	Function	在变量表中定义 func 与参数、静态链信息，并输出
Expression	Var	根据类型，在变量栈中取出变量信息
	Nil	返回一个空类型，若代表空类型的临时变量尚未存在，新建他
	Call	从变量表中或外部，使用一个函数
	Record	调用 allocRecord，将一个新的 record 的地址存放在临时变量位置，并初始化 record 的各项 field
	Array	调用 iniArray，在栈中存放数组指针
	Seq	顺序执行其下的抽象树中指令
	While	分析 while 的三部分：退出条件表达式、while 程序体、退出位置，生成 while 程序整体
	For	与 while 类似，分析 for 的各部分并生成 for 程序整体
	Assign	取变量表中的 var，使用表达式为其赋予新的值
	Break	退出当前循环体
	Let	分析 let 的定义部分、let 程序体，并生成 let 整体
	Op	进行表达式的算数运算与比较
	If	分析 if 的各个模块，如条件判断表达式、then, else 后的函数体，并生成 if 程序整体
Var	SimpleVar	被 exp 中的 var 调用，取出 simple 变量
	FieldVar	被 exp 中的 var 调用，取出 field 变量
	subscriptVar	被 exp 中的 var 调用，取出 subscript 变量

5.代码生成 （所有语句的代码生成的处理）

中间树代码部分

在语义分析的最后部分，就是调用课本给予的 `tree.c` 模块，生成中间树的过程。当然，`tree.c` 中只提供了基本结构的构造接口，其如何连接是需要我们自己设计的。接下来，我们会给出所有语句的构造方式。

我们在 `translate` 模块中构造了一种联合类型 `Tr_exp`，用来代表表达式 `T_exp`、`T_stm` 与条件判断语句 `Cx`。`Translate` 模块中所有与生成中间树有关的函数，其返回值都会使用 `Tr_exp` 的构造函数(`Tr_Ex, Tr_Nx, Tr_cx`)，转换为 `Tr_exp` 类型。并且，我们定义了 `unEx`、`unNx`、`unCx` 函数，用来将一个 `Tr_exp` 类型还原为 `T_exp, T_stm, Cx`。这样的做法，有两个好处。一为保持统一性，写程序更加易读，可以使用更简单的模式来生成函数；二为，这样的函数提供了三种原类型，相互转化的接口，从而可以满足 `tiger` 语言的所有需要。三种原类型相互转化的接口是必要的。

对于一个类似于 `a=b` 的表达式，我们的语法树会将其转化为条件判断语句 `Cx`，但是有时候，我们需要将一个 `Cx` 语句转化为有值的 `T_exp`。如下面这个例子：

`A := b = c`

此时，转化的接口就相当重要了。

在课本上，提供了 `unEx` 的实现方式，但未给出 `unCx` 与 `unNx` 的实现。为了实现编译器，这两个函数自然是必须的，我们是这样实现这两个函数的：

1.unNx

```
static T_stm unNx(Tr_exp e) {
    switch (e->kind) {
    case Tr_ex:
        return T_Exp(e->u.ex);
    case Tr_nx:
        return e->u.nx;
    case Tr_cx: {
        Temp_temp r = Temp_newtemp();
        Temp_label t = Temp_newlabel(), f = Temp_newlabel();
        doPatch(e->u.cx.trues, t);
        doPatch(e->u.cx.falses, f);
        return T_Exp(T_Eseq(T_Move(T_Temp(r), T_Const(1)),
                             T_Eseq(e->u.cx.stm,
                                     T_Eseq(T_Label(f),
                                             T_Eseq(T_Move(T_Temp(r), T_Const(0)),
                                                    T_Eseq(T_Label(t),
                                                            T_Temp(r))))))));
    }
    }
    assert(0);
}
```

`Nx`, `Ex` 到 `T_stm` 的转化非常的显然，而 `Cx` 到 `T_stm` 的转化，也只需要在 `Cx` 到 `T_exp`

的转化上套一个 `T_Exp` 即可。`UnNx` 的实现是非常简单的。

2.unCx

`UnCx` 的实现，建立在特殊对待 `CONST 0` 和 `CONST 1` 的基础上，其实现如下：

```
static struct Cx unCx(Tr_exp e) {
    switch (e->kind) {
        case Tr_cx:
            return e->u.cx;
        case Tr_ex: {
            struct Cx cx;
            cx.stm = T_Cjump(T_ne, e->u.ex, T_Const(0), NULL, NULL);
            cx.trues = PatchList(&(cx.stm->u.CJUMP.true), NULL);
            cx.falses = PatchList(&(cx.stm->u.CJUMP.false), NULL);
            return cx;
        }
        case Tr_nx:
            assert(0);
    }
    assert(0);
}
```

对于 `Tr_cx`,只需要剥离原本的构造函数即可,对于 `Tr_nx`,一个没有返回值的语句,是无法被表达为一个条件判断语句的。而对于 `Tr_ex`,情况就不一样了。我们采用和 C 语言一样的实现。当 `ex` 的值为 0 时,表达式的值为假,不然为真。

完成这两个转换函数后,所有的抽象语法到中间树的转化,就都可以进行了。下面,我们将列出所有语法的转化逻辑。

1.Var

中间代码与汇编的关系很大。对于出现在中间树中的变量,我们并不需要其名字信息,只需要知道其在什么位置,进行什么操作。因为我们将所有变量都放在栈中进行操作,因此只需要其通过静态链的寻址,以及在当前层栈中的偏移这两个信息即可。

在经过了语义分析后,中间树生成模块可以获得这两个关键信息。为了减少贴的代码,在这里只分析中间树生成模块的代码,并只给出最关键的代码部分。

根据这样的思路,我们对 `simpleVar`、`fieldVar`、`subscriptVar` 的转化如下:

SimpleVar

```
Tr_Exp(T_Mem(T_Binop(T_plus, addr, T_Const(ac->access->u.off)),
```

根据静态链寻址信息 `addr`,与栈内偏移,即可完成对一个 `simpleVar` 的寻址,生成中间树代码。

FieldVar

```
Tr_Exp(T_Mem(T_Binop(T_plus, unEx(base), T_Const(off * F_WORD_SIZE))))
```

根据静态链寻址获得的基地址信息 `base`,与 `field` 内的偏移,即可完成对 `fieldVar` 的寻址。

SubscriptVar

```
Tr_Exp(T_Mem(T_Binop(T_plus, unEx(base), T_Binop(T_mul, unEx(index),
```

```
T_Const(F_WORD_SIZE)))));
```

根据静态链寻址获得的基址信息 **base**，与数组下标的偏移，即可完成对 **subscriptVar** 的寻址。

2.Exp

因为汇编的特殊性，在语义分析模块中，变量、函数通过声明确定了其在栈中的位置，生成中间树时，变量的声明可以当做赋值一样处理，而函数的声明，其函数体可以保存在 **fraglist** 中输出。函数体本身也可以分解为 **exp** 语句的组合。因此现在只需要讨论 **exp** 表达式部分如何实现就可以了。

接下来，我们将根据从 **yacc** 开始便定义的所有 **exp** 的种类，分析各种 **exp** 表达式的中间树该如何生成。

1) intExp

```
Tr_Ex(T_Const(i))
```

2) doubleExp

```
Tr_Ex(T_Double(f))
```

3) nilExp

```
Tr_Ex(T_Eseq(alloc, T_Temp(nilTemp)))
```

Nil 作为一个空类的代表，指向编译器指定的特定位置。若 **nilTemp** 未被初始化，则需要先被初始化。

4) noExp

```
Tr_Ex(T_Const(0))
```

使用 **const 0** 表示无需写入中间树的结构，如函数定义、类型定义等。这些在语义分析阶段就能被处理完，或写入 **fraglist** 中。

5) recordExp

```
Tr_exp Tr_recordExp(int n, Tr_expList l) {
    Temp_temp r = Temp_newtemp();
    T_stm alloc = T_Move(T_Temp(r),
                        F_externalCall(String("allocRecord"), T_ExpList(T_Const(n *
F_WORD_SIZE), NULL))));

    int i = n - 1;
    T_stm seq = T_Move(T_Mem(T_Binop(T_plus, T_Temp(r), T_Const(i * F_WORD_SIZE))),
                    unEx(l->head));

    for (l = l->tail; l; l = l->tail, i--) {
        seq = T_Seq(T_Move(T_Mem(T_Binop(T_plus, T_Temp(r), T_Const(i * F_WORD_SIZE))),
                    unEx(l->head)),
                    seq);
    }
    return Tr_Ex(T_Eseq(T_Seq(alloc, seq), T_Temp(r)));
}
```

Record 的中间树处理需要经过三个部分，首先是初始化 **record**，并使用临时变量代表其存放地址，然后初始化 **record** 内的各个位置，最后将三个步骤合成中间树输出。

6) arrayExp

```
Tr_Ex(F_externalCall(String("initArray"), T_ExpList(unEx(size), T_ExpList(unEx(init),
NULL))))
```

外部调用 `initArray`，初始化 `array` 后，形成中间树输出。

7) seqExp

```
Tr_exp Tr_seqExp(Tr_expList l) {
    T_exp reslist = unEx(l->head);
    for (l = l->tail; l; l = l->tail) {
        reslist = T_Eseq(T_Exp(unEx(l->head)), reslist);
    }
    return Tr_Ex(reslist);
}
```

这个结构在 `let` 的 `body` 部分，以及括号中被使用，中间树生成模块将其使用 `Eseq` 串联，形成与原本类似的链状结构，作为中间树部分输出。

8) doneExp

```
Tr_Ex(T_Name(Temp_newlabel()));
```

`DoneExp` 作为 `while` 等循环的跳出位置使用，新建一个 `templelabel`，将其作为中间树的部分输出。

9) assignExp

```
Tr_Nx(T_Move(unEx(lval), unEx(exp)));
```

`AssignExp`，将一个 `exp` 的值赋给一个 `lval`。

10) arithExp

```
Tr_Ex(T_Binop(opp, unEx(left), unEx(right)))
```

`ArithExp`，输出表达式运算中间树，运算符部分将在汇编阶段被处理。

11) compExp

```
T_stm cond = T_Cjump(opp, unEx(left), unEx(right), NULL, NULL);
patchList trues = PatchList(&cond->u.CJUMP.true, NULL);
patchList falses = PatchList(&cond->u.CJUMP.false, NULL);
return Tr_Cx(trues, falses, cond);
```

使用 `patch`，生成 `Cx` 结构。`CJump` 的信息将在执行完 `ifExp` 后被填充

12) ifExp

`IfExp` 内容太长，就不全贴在这里了。在 `if` 模块，第一步生成填在 `compExp` 中需要填入的 `label t` 与 `f`，填充在 `Cjump` 中。再判断 `else` 模块是否存在，根据其存在与否，决定了在跳转位置 `T_label(f)` 中是否存在执行的内容。最后输出的格式如下：

```
Tr_Nx(T_Seq(cond.stm,
    T_Seq(T_Label(t),
        T_Seq(thenStm,
            T_Seq(joinJump,
                T_Seq(T_Label(f),
                    T_Seq(elseeStm,
                        T_Seq(joinJump, T_Label(join))))))))))
```

13) whileExp

```
Tr_Ex(T_Eseq(T_Jump(T_Name(test1), Temp_LabelList(test1, NULL)),
```

```

T_Eseq(T_Label(body1),
      T_Eseq(unNx(body),
            T_Eseq(T_Label(test1),
                  T_Eseq(T_Cjump(T_eq, unEx(test), T_Const(0), unEx(done)->u.NAME,
body1),
                        T_Eseq(T_Label(unEx(done)->u.NAME), T_Const(0))))))))

```

WhileExp 的格式和书上的形式相同，其逻辑如下：

```

Test:
  If not(condition) goto done
  Body
  Goto test
done

```

14)forExp

```

Tr_Ex(T_Eseq(T_Move(unEx(lval), unEx(lo)),
            T_Eseq(T_Jump(T_Name(test1), Temp_LabelList(test1, NULL)),
                  T_Eseq(T_Label(body1),
                        T_Eseq(unNx(body),
                              T_Eseq(T_Move(unEx(lval), unEx(Tr_arithExp(A_plusOp, lval,
Tr_Ex(T_Const(1))))), // i:= i + 1
                                      T_Eseq(T_Label(test1),
                                            T_Eseq(T_Cjump(T_le, unEx(lval), unEx(hi), body1,
unEx(done)->u.NAME), // i<=hi
                                                  T_Eseq(T_Label(unEx(done)->u.NAME),
T_Const(0))))))))))));

```

WhileExp 的格式和书上的形式相同，其逻辑如下：

```

for i:=lo to hi
do body
==>
let var i:=lo
  var limit:=hi
in while i<=limit
  do (body;i:=i+1)
end

```

6. 汇编代码生成

6.1 概述：

本项目中我们生成的具体目标代码是适用于 Intel x86 处理器的 Windows 平台下的汇编代码，即编译器本身运行在 Windows 平台下，生成的汇编代码经过我们的编译器编译后生

成的可执行 exe 文件也是需要 Windows 环境。具体本次项目中我们所使用的环境是 VS2015 + Win8.1 SDK。

在本章节中，我们将介绍具体实现汇编代码生成的思路，汇编代码生成的细节处理，相应运行环境的搭建，遇到的问题以及对应的妥协或者解决办法等内容，这也是我们所做的项目最重要最有特色的也是最精彩的部分。

6.2 汇编代码生成的思路

6.2.1: 适用于 x86 架构的中间代码

1. 函数栈帧和 level 栈帧:

在中间代码层，我们构建了携带有最简化的操作信息的中间代码树，但是 Tiger 语言在中间代码层的变量访问与函数访问规则却是在实际 x86 环境下的函数调用和变量访问规范下需要有相应的调整，我们举如下的例子来说明这个问题：

```
let
    var a = 0
    function g () =
        a := 1
in
    g()
end
```

函数 g 访问了位于函数 g 上一个 level 的一个变量 a，那么考虑一下对于 a 变量的寻址的做法，编译器在看到函数 g 中看到了变量啊，然后从该层 level 开始向上遍历 level 直到找到 a 所在的 level，实现伪代码如下：

```
addr = fp;
While(curLevel!= a->level)
{
    curLevel=curLevel->parent;
    addr = T_Mem(addr);
}
```



```
}
```

```
...
```

也就是我们对当前栈帧取其中的值来作为下一个 level 的寻址地址，但是在 x86 函数调用有着如下的规范：

```
proc_call:
    push ebp
    mov ebp, esp
    ...
```

其中 `ebp` 为当前函数栈帧，这两个动作的目的就是保存上一个函数也就是这个函数调用者的栈帧，然后把 `ebp` 赋值为当前函数栈帧，这样只要通过取 `ebp` 地址的值就可以得到上一个函数的栈帧。

如果中间树中的 `frame pointer` 是对应于规范中的 `ebp`，则函数调用层次和 level 层次不符合时会产生严重的冲突，也就是说如果同属于 level 1 层次的 `fun1` 如果调用了 `fun2` 则 `fun2` 在寻址 level 0 时会出现问题。

所以我们不得不增加 Level 栈帧来解决这个问题，当出现对当前函数栈帧的引用如 `T_Mem(fp)` 时，不把这句话解析为对当前函数栈帧的引用，而转换为对静态链的引用，具体实现在汇编中有如下伪代码：

```
如果对栈帧的引用是 T_Mem(fp)
    则翻译的汇编代码是 mov reg, LevelFrame[curLevel]
否则
    翻译的结果 Mov reg, ebp
```

在中间代码中对栈帧 `fp` 的嵌套的 `T_Mem` 将决定是对哪一层 Level 栈帧的寻址，记录这个的数据结构叫 `LevelFrame`。

2. 变量下标引用处理

在使用 C 语言的时候，声明一以另外变量为长度的变量本身就是一件不合法的事情，即如下代码不合法：

```
int N = 8;
int m[N];
```

原因也很简单，x86 的内存结构中不可能在堆栈中开辟一块你自己都不知道有多大的内存，同时也会对之后的局部变量寻址造成困扰。但是 `tiger` 声明的时候必须要兼容这个功能，但是程序在编译阶段是不可能知道变量的值的，这时候为 `array` 类型的变量分配栈空间就是一件极为困难的事情。

所以注定 `array` 类型的变量的实体存放不可能在堆栈中，所以这里我们的处理方法是所有的 `array` 类型的变量在堆栈中均是指针，也就是下面的 `tiger` 声明语句对应的汇编与如下的 C 语言声明阶段汇编代码相同，但是初始化方式不同。

```
Tiger:
var N := 8
type intArray = array of int
```

```
var m := intArray [ N ] of 0
```

```
C;  
int N = 8;  
int *m
```

也就是说 Tiger 类型的 array 必须是 C 语言中的指针类型才能保证其在堆栈中所占的大小也是 4 个字节，这样才不会对后面的变量寻址造成干扰。

同时这也就意味着对于有下标引用的变量必须比简单变量多一次地址访问，也就是寻址到 array 变量在堆栈中的地址后，还需要再对这个地址进行一次访问才能拿到 array 的真实地址，这也就是 C 语言中指针的功能。而对于一般的变量在堆栈中的地址就是他的真实地址。

6.2.2: 寄存器分配规则

实验做到这个问题的时候非常的尴尬，因为在课上讲过寄存器分配之后 4 天就是本实验的 ddl 了，所以本实验中没有应用复杂的寄存器分配规则，尽管 x86 架构中通用寄存器只有 4 个 `eax, ebx, ecx, edx`，还有两个用于保存地址的 `esi` 和 `edi`，四色问题的证明也能够说明尽管寄存器数量这么少，但是对于数据流图上色问题也是足够的了。

我们遍历中间代码树来生成相应的汇编，我们制定遍历规则来使得寄存器分配有序不至于混乱。树的遍历采取后续遍历我们举例如下来说明我们是怎样应用寄存器分配的：

遍历树结构所用的函数调用如下：

```
static void asm_tree_exp(T_exp exp, int d, int reg)  
  
static void asm_stm(T_stm stm, int d)
```

在 `exp` 类型的树遍历中，我们传入一个 `reg` 参数，这个参数代表我们希望这个调用结束后的 `exp` 的值保存在 `reg` 中，`reg` 具体则是指寄存器编号，如 0 号代表 `eax`，1 号代表 `ebx`。

所以对于树顶层节点就有如下的实现：

```
case T_ESEQ:  
  
    asm_stm(exp->u.ESEQ.stm, d + 1);  
  
    asm_tree_exp(exp->u.ESEQ.exp, d + 1, reg);  
  
    break;
```

ESEQ 这个语句并不产生任何汇编，它所做的就是把他携带的 `reg` 参数向下继续传递，保证儿子 `exp` 的返回值在所需要的 `reg` 中

而对于叶子节点，则就是对希望的寄存器进行赋值例子如下：

```
case T_CONST:

{

    char buf[33];

    itoa(exp->u.CONST, buf, 10);

    assignRegTo(reg);

    strcat(asmBuf, buf);

    strcat(asmBuf, "\n");

    break;

}
```

asmBuf 是保存有当前汇编指令的 buffer，可见上述的动作就是把 const 的值赋值给传来的 reg。

但是这种架构在遇到有两个以上 exp 子节点的时候就会遇到问题，如果 exp 节点 2 在运算中不小心改了 exp1 节点所希望结果保存在的寄存器，就会出现寄存器使用冲突的问题。所以需要用 push pop 动作保存相应的寄存器数据不被破坏。

```
asm_tree_exp(exp->u.BINOP.left, d + 1, 0);

strcat(asmBuf, "push eax\n"); // 保护exp1的结果 eax

asm_tree_exp(exp->u.BINOP.right, d + 1, 2);

strcat(asmBuf, "pop eax\n");
```

值得注意的是 STM 的节点也就是本身不具备表达式值的节点不能改动任何寄存器，也就是在进出 STM 节点的前后寄存器的上下文应该保持一致，这样在返回上层的 EXP 调用的过程中不至于出现错误，因此 asm_stm 函数中任何临时使用到的寄存器都必须保护如：

```
case T_EXP:

    strcat(asmBuf, "push eax\n");

    asm_tree_exp(stm->u.EXP, d + 1, 0);

    strcat(asmBuf, "pop eax\n");

    break;
```

6.3 汇编代码生成的细节与技巧

6.3.1: Call 指令

函数调用的难点在于函数参数的个数难以确定,这是使用调用 `asm_tree_exp` 对儿子节点进行计算的时候无法平衡寄存器冲突,并且中间树的 Call 指令的参数顺序也与 `_cdecl` 相悖,所以需要额外的开销来完成这个动作。

本项目在这里的具体实现是先把参数的计算结果保存在全局临时变量 `ARG[N]` 中,然后待所有参数的 `exp` 都计算完成后,再把 `ARG[N]` 的内容依次读取逆序压入堆栈。调用 `call` 后由于是 `_cdecl` 规范,所以需要手动清理堆栈,所以需要根据参数个数来计算抬升栈顶的数目:

```
add esp, n*4
```

6.3.2: Move 指令

Move 指令的特殊性在于当 Move 的左侧是 Mem 的时候,这里的 Mem 不再是取内存内容的意思,而是取该内存的地址。

在我们具体实现中,上述书中提过的情况不仅仅发生在 Mem 中,当引用 Temp 时也会出现这个问题,这是因为在中间代码层中产生的 temp 会影响到寄存器的分配,因此 temp 变量决不能存储在寄存器中,所以 temp 变量也和 ARG 变量一样以全局的临时变量的形式存储在出现的全局变量区。

这里的 trick 是在我们进行 Mem 和 Temp 时,把 Mem 和 Temp 的值传入给定的 reg,但是把他们的地址传入 esi, edi。Mem 的地址传入 esi, Temp 的地址传入 edi,这两个寄存器当且仅当这里使用,别的地方没有用到这两个寄存器,所以不存在冲突的问题。

然后到 Move 层则会视左子节点是什么类型来决定用哪一个寄存器来执行什么指令,如果是 Mem 或 Temp 则会执行 memory store 而非寄存器转移:

```
if (stm->u.MOVE.dst->kind == T_MEM)

    strcat(asmBuf, "mov [esi], eax\n");
```

```

else if(stm->u.MOVE.dst->kind == T_TEMP)

    strcat(asmBuf, "mov [edi], eax\n");

else

    strcat(asmBuf, "mov ebx, eax\n");

```

6.3.3: Jump 指令

Jump 是个 STM 类型的操作，但是 Jump 的参数是一个 exp，所以理所当然的需要寄存器的保护，但是对应的汇编指令 jmp 后的下一条指令应该是恢复寄存器，也就是正确的逻辑应该是

```

push eax
asm_tree_exp(eax)
jmp eax
...pop eax

```

但是程序没有运行不肯能知道 jmp 的地址是多少，也就不可能知道去哪插这一条 pop eax 来恢复被修改的 eax，这时候就只能用脏栈的手段来达到这个目的：

```

case T_JUMP:

{

    strcat(asmBuf, "push eax\n");

    asm_tree_exp(stm->u.JUMP.exp, d + 1, 0);

    strcat(asmBuf, "push eax\n");

    strcat(asmBuf, "pop eax\n");

    strcat(asmBuf, "pop eax\n");

    strcat(asmBuf, "jmp [esp-8]\n");

    break;

}

```

可以看到 eax 已经恢复，但是 jmp 还是可以跳转的正确的地址，按照汇编规范来说，位于当前栈指针 esp 下的内存是不可使用的，就是脏内存没有被初始化申请的，但是实际上还是可以使用的，这里只能做出妥协。

6.3.4: Level 层次初始化

Tiger 语言本身就是一个 EXP，也就是最外层的 let 和 in 也可以看做是写在函数中的也就是任何 tiger 语言等价于

```
Function main() = (  
    Let  
        ...  
    In  
        ...  
    End  
)
```

但是在生成中间树时最外层的 main 不会显示，所以最外层 level 的初始化工作需要专门定制初始化的代码来吧这层 main 的 shell 加上。

中间树的组合顺序在这里至关重要，进入一个 function 需要有以下层次，这应该与中间树的层次相对应：

- 将当前栈帧写入 LevelFrame
- 开辟新的栈区用于保存该域下的局部变量
- 根据 let 的 var 声明进入变量初始化过程
- in 中的汇编代码
- 回收堆栈退出函数
- let 中的 function dec 汇编下一层 function 递归上面的过程

具体实现如下，对于最外层 main 需要特殊处理：

```
char buf[100], sbuf[100];  
  
itoa((curLevel + 1) * 4, sbuf, 10);  
  
strcpy(buf, S_name(f->u.proc.name));  
  
strcat(asmBuf, buf);  
  
strcat(asmBuf, ":\n");  
  
strcat(asmBuf, "push ebp\nmov ebp, esp\n");  
  
strcat(asmBuf, "sub esp, 0xFF\n");  
  
if (strcmp(buf, S_name(main_label))!=0)
```

```

{

    strcat(asmBuf, "mov LevelFrame[");

    strcat(asmBuf, sbuf);

    strcat(asmBuf, "], ebp\n");

}

toASM(Tr_Nx(f->u.proc.body));

strcat(asmBuf, "add esp, 0xFF\n");

if (strcmp(buf, S_name(main_label)) == 0)

    strcat(asmBuf, "pop ebp\njmp end\n");

else

    strcat(asmBuf, "pop ebp\nret\n");

```

6.4 汇编运行环境

很明显，我们所生成出来的汇编举例 exe 还有这相当的距离，因为一个 exe 的编译中即使是最简单的 hello world 程序对应的汇编代码也是非常庞杂的，其中很多是对于 C 运行库的配置，也有很多操作系统层面的注册工作。此外即使能得到具有完备功能的汇编把它解析成一个 PE 文件也不是一件容易的事情。

因此我们在这个问题上采取折中的手段，我们不可能自己去把操作系统相关的和 C 语言程序运行相关的库的汇编自己编写，我们所生成的就是对应于 hello world 程序中如下函数体的汇编代码，这一部分代码是我们可以控制的。

```

int main(){
    printf("hello world");
}

```

具体我们采取的折中手段就是用 C 语言内嵌汇编的形式来运行我们的汇编，这里的内嵌汇编提供一个相当于虚拟机的环境可以供我们调试执行我们编译器生成的汇编代码，这个虚拟机是一个.c 文件，通过编译调试这个文件来帮助我们验证我们的工作。

这个.c 文件有以下几个部分组成，举例如下：

```
/******函数头部******/  
  
#include <stdio.h>  
  
#include <stdlib.h>  
  
int *initArray(int len, int init);  
  
int ARG[2048];  
  
int TEMP[2048];  
  
int LevelFrame[2048];  
  
int execute();  
  
/*******/  
  
/******Tiger程序中的静态字符串常量******/  
  
char L19[] = "\x20\x4f";  
  
char L20[] = "\x20\x4f";  
  
char L21[] = "\x20\x2e";  
  
char L22[] = "\x20\x2e";  
  
char L28[] = "\xa";  
  
char L29[] = "\xa";  
  
/*******/  
  
int main()  
  
{
```



```

    execute();

    system("pause");
}

int execute()
{
    __asm {...} /*生成的汇编代码*/

    __asm {

        end:

        add esp, 0xFF

        pop ebp

    }
}

int *initArray(int len, int init)
{
    int *p = (int *)malloc(4 * len);

    for (int i = 0; i < len; i++)

        p[i] = init;

    return p;
}

```

最后用 VS 编译运行这个程序即可

7 测试案例

我们对 tiger 语言的语法进行一一测试。

7.1 声明

Tiger 程序如下：

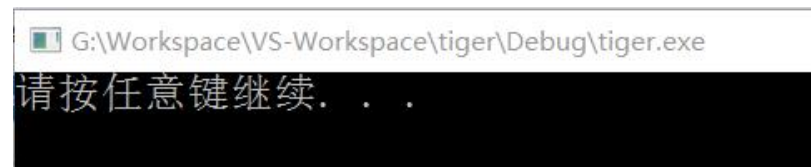
```
let
  type intlist = int
  var a := 0
  var b:intlist := 1

  function test(c:int) =
    print("test\n")

in test(0)
end
```

该程序包括了数据类型的声明，变量的声明以及函数的声明，以及一个带参函数的调用。结果将会输出一个 test。

编译之后的结果如下所示：



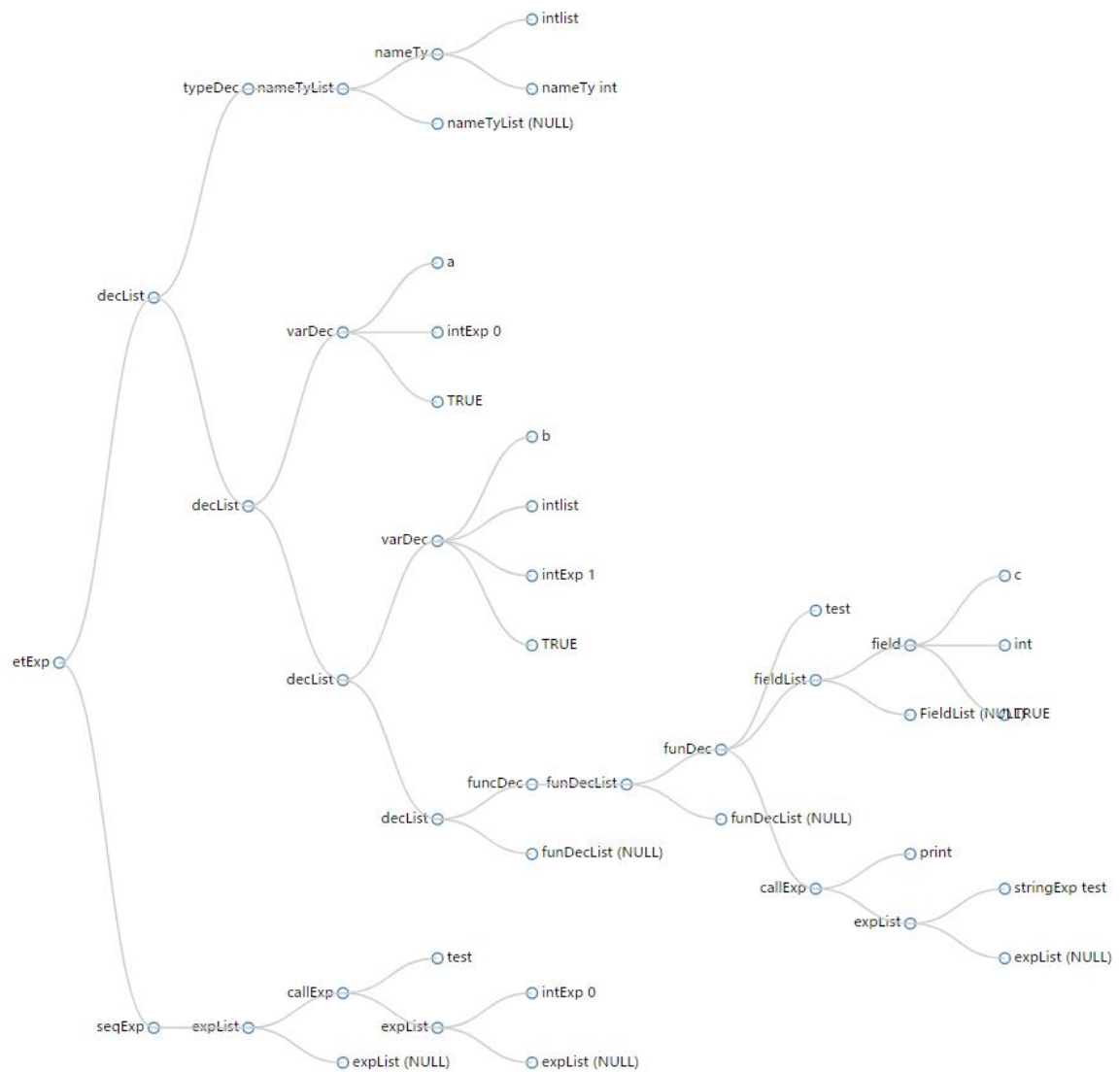
没有任何 warning 和 error.

此时，已经生成了目标汇编代码，我们运行目标汇编代码，得到如下结果：



符合预期，正确。

我们再看整个程序的抽象语法树，如下所示（放大可获取细节）



7.2 嵌套声明与记录

Tiger 程序如下：

```
let
  type intlist = {hd:int, tl:intlist}
  var a := 0
  var b:= intlist{hd=10,tl=nil}

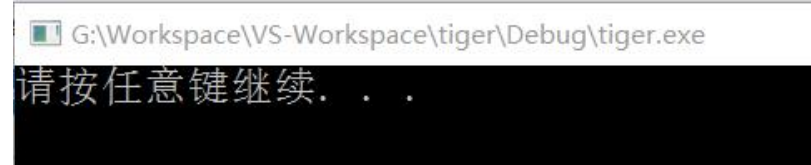
  function test(c:int) =
    let function test1() =
      if b.hd = 10 then print("test1\n")
    in (test1();print("test\n"));
end
```

```
in test(0)
end
```

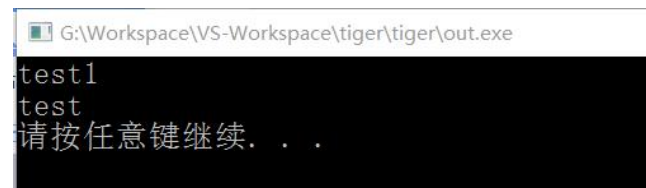
该程序包括了记录的声明和变量类型为记录的声明。

同时，在函数 `test` 中，我们嵌套定义了无参函数 `test1()`，判断 `b.hd` 是否为 10 来测试记录是否可用。调用函数 `test(0)`，将会先输出 `test1`，在输出 `test`。

编译之后的结果如下所示：



此时，已经生成了目标汇编代码，我们运行目标汇编代码，得到如下结果：



符合预期，正确。

整个程序的抽象语法树如下所示：

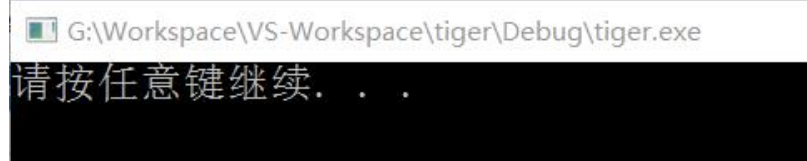

```

        row[r] := row[r-1] + 1;
        print("in while\n")
    );
    if (row[7] = 7)
        then print("yes row[7] = 7\n")
    )
end
in test(0)
end

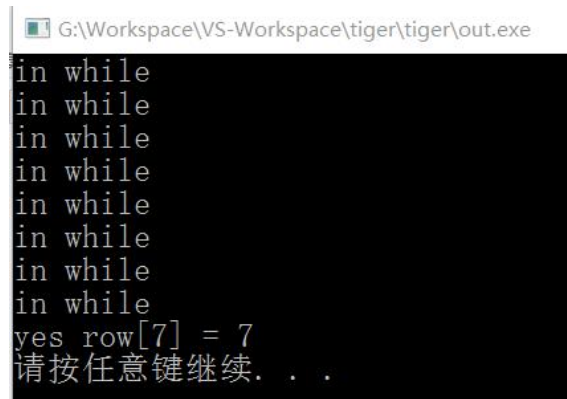
```

该程序包含了数组的声明和 while 循环的使用。同时在 while 循环内，包含了序列，算术操作，条件语句的比较，赋值。

编译之后的结果如下所示：



此时，已经生成了目标汇编代码，我们运行目标汇编代码，得到如下结果：



7.4 字符串以及条件判断语句

Tiger 程序如下：

```

let
    var str1 := "string"
    var str2 := "string"
    function test(c:int) =
        let var a:=0
            var b:=1
            var d:=-1
        in (
            if str1 = str2 then
                print("str1 = str2!\n");
            if a>b then
                print("a>b\n")
            else

```

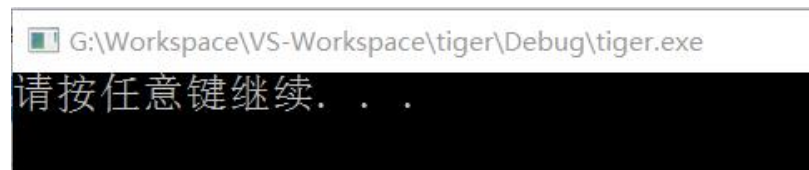
```

        print("a<=b\n");
    if b>=c then
        print("b>=c\n")
    else
        print("b<c\n");
    if d<0 then
        print("d<0\n");
    if a&b then
        print("a&b = 1\n")
    else
        print("a&b = 0\n");
    if (a|b) then
        print("a|b = 1\n");
    )
end
in test(1)
end

```

该程序包含了一个 `string` 变量的声明，以及各个条件语句的比较符以及布尔操作 `&` 和 `|`。

编译之后的结果如下所示：

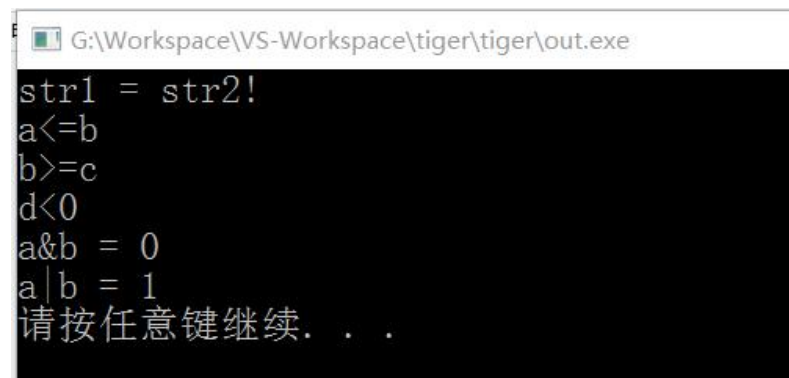


```

G:\Workspace\VS-Workspace\tiger\Debug\tiger.exe
请按任意键继续. . .

```

此时，已经生成了目标汇编代码，我们运行目标汇编代码，得到如下结果：



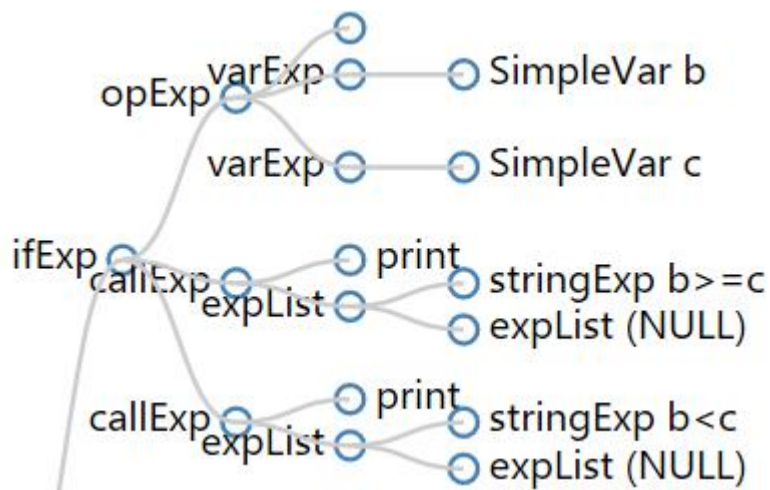
```

G:\Workspace\VS-Workspace\tiger\tiger\out.exe
str1 = str2!
a<=b
b>=c
d<0
a&b = 0
a|b = 1
请按任意键继续. . .

```

符合预期结果。

我们挑选其中一个 `if` 语句显示抽象语法树，如下：



其对应着 if b>=c then
 print("b>=c\n")
 else
 print("b<c\n");

接下去的部分由于抽象语法树过大不予以显示。

7.5 for 循环+*/操作符以及 break 运算

Tiger 程序如下：

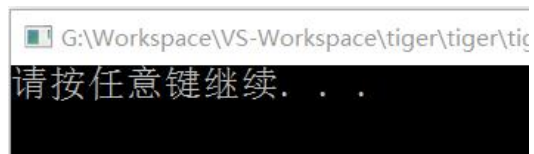
```
let
var N:=8

function test() =
let var i:=0
    var j:=3
in (
    for i:=0 to N/2+(3-1)*1
    do(
        if i=(j+1)/2 then
            (print("break!\n");break)

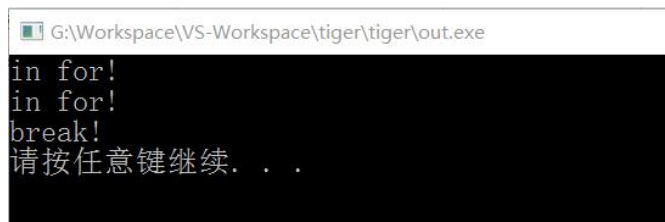
        print("in for!\n");
    )
)
end
in test()
end
```

该程序包含了一个 for 循环，for 循环的约束条件中包含了一个算式 $N/2+(3-1)*1$ 结果为 6。在循环体中，当 i 自增到 2 时，即退出循环体。

编译之后的结果如下所示：



此时，已经生成了目标汇编代码，我们运行目标汇编代码，得到如下结果：



符合预期结果。

7.6 八皇后程序

其 tiger 程序如下

```
/* A program to solve the 8-queens problem */

let
  var N := 8

  type intArray = array of int

  var row := intArray [ N ] of 0
  var col := intArray [ N ] of 0
  var diag1 := intArray [N+N-1] of 0
  var diag2 := intArray [N+N-1] of 0

  function printboard() =
    let var i:=0
    var j:=0
    in
      for i := 0 to N-1
      do (for j := 0 to N-1
          do if col[i]=j then print(" O") else print(" .");
            print("\n"));
          print("\n")
      end

  function try(c:int) =
    let var r:=0
    in
      if c=N
```

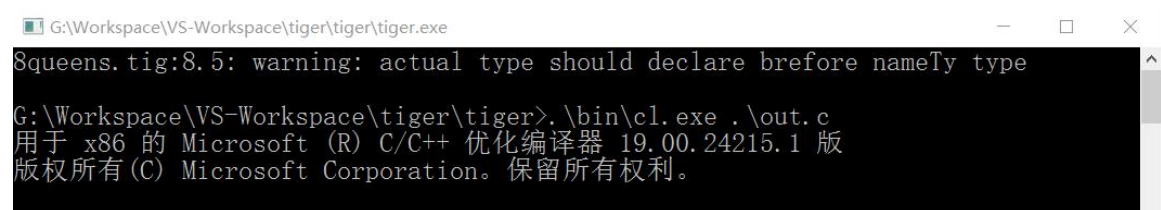
```

    then printboard()
  else for r := 0 to N-1
    do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
      then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
        col[c]:=r;
        try(c+1);
        row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)
    end

  in try(0)
end

```

编译之后的结果如下所示：



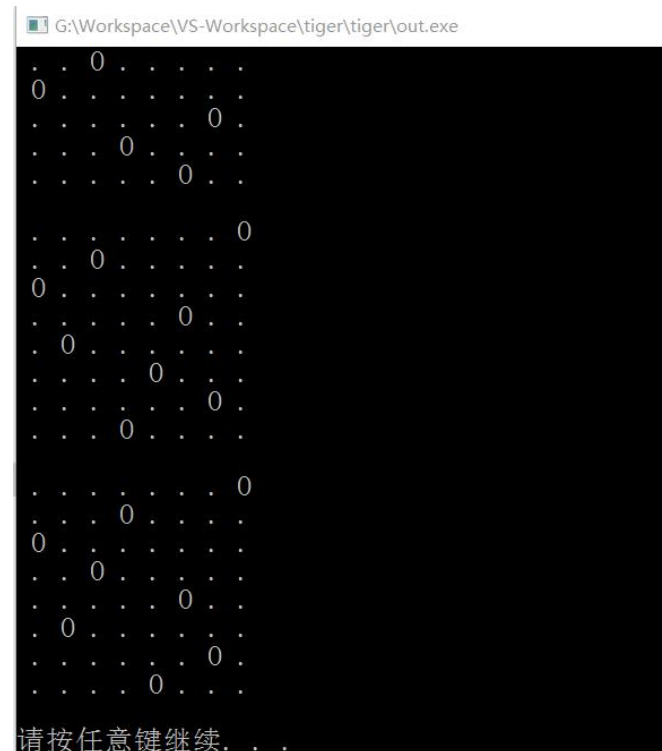
```

G:\Workspace\VS-Workspace\tiger\tiger.exe
8queens.tig:8.5: warning: actual type should declare before nameTy type

G:\Workspace\VS-Workspace\tiger\tiger>. \bin\cl.exe .\out.c
用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.00.24215.1 版
版权所有 (C) Microsoft Corporation。保留所有权利。

```

此时，已经生成了目标汇编代码，我们运行目标汇编代码，得到如下结果：



```

G:\Workspace\VS-Workspace\tiger\tiger\out.exe
. . 0 . . . .
0 . . . . .
. . . . 0 .
. . . 0 . .
. . . . 0 .

. . . . . 0
. . 0 . . . .
0 . . . . .
. . . . 0 .
. 0 . . . .
. . . 0 . .
. . . . 0 .
. . . 0 . .

. . . . . 0
. . . 0 . . .
0 . . . . .
. . 0 . . . .
. 0 . . . 0 .
. 0 . . . .
. . . . 0 .
. . . . 0 .

请按任意键继续. . .

```

程序输出了八皇后问题的所有解，符合预期结果。