# 中间代码生成

学号 SA19225231 姓名 李泽伟 实验三

## 实验目的

在词法分析、语法分析的基础上生成中间代码

## 实验要求

下面是实验三可以生成的语法结构：

1. 变量：包括各种类型，如基础数据类型（int，float, char）、指针、数组、结构体和对象；
2. 函数，包括参数和返回值
3. 表达式：算术和布尔表达式，支持各种常见操作符
4. 控制结构：包括if、while等
5. 语法糖：自增自减

## 实验步骤

1. 确定项目结构
    1. `e3.l` 是lex文件，包含对源代码的词法分析
    2. `e3.y` 是 `bison` 文件，包含对 `yylex` 的分析结果的语法分析
    3. `makefile` 是makefile文件，告诉make命令怎么编译和链接程序
    4. `tasm` 是编译生成的可执行文件
    5. `xxx.asm` 是生成的中间语言
2. 执行过程
    1. 首先用 `make` 命令生成可执行文件 `tasm`
    2. 然后 `make test` 输出 `functions.asm` 文件

    ```
    make
    make test
    ```

3. 执行结果

    输入命令如图所示

```
# lzw @ MacBook-Pro in ~/Documents/USTC/编译原理/ 实验/e3 中间代码生成 on
git:master x [19:55:01]
$ make test
bison -vdty e3.y
conflicts: 170 shift/reduce, 66 reduce/reduce
flex e3.l
gcc-8    -c -o lex.yy.o lex.yy.c
gcc-8    -c -o y.tab.o y.tab.c
gcc-8 -o tasm lex.yy.o y.tab.o
./tasm < functions.txt > functions.asm
```

Functions.txt中代码如下

```
int foo(int a, double b, bool c) {
  if(a == 0){
    a = a + 2;
  }else{
    a--;
  }
  return a + 2;
}

void main() {
  int b;
  int a;
  double d;

  d = 2 + 3 * 4 - (6 / 2);
  b = ReadInteger();
  a = b + 2;
  Print(a, b, d);
  foo(a, d, !true);
  foo(a + 2, d/2, a == b && d >= 1.0);
}
```

编译生成中间语言并输出结果在functions.asm中，functions.asm中间代码如下

```
FUNC @foo:
  arg a, b, c
_begIf_1:
  push #0
  cmpeq
  jz _elseIf_1
  push #2
  add
```

```
  pop a
  jmp _endIf_1
_elseIf_1:
  push #1
  sub
_endIf_1:
  push #2
  add
  ret ~
ENDFUNC

FUNC @main:
  var b
  var a
  var d
  push #2
  push #3
  push #4
  mul
  add
  push #6
  push #2
  div
  sub
  pop d
  pop b
  push #2
  add
  pop a
  push #1
  not
  push #2
  add
  push #2
  div
  push #1.000000
  cmpge
  and
  cmpeq
ENDFUNC
```

# 代码介绍

1. makefile文件

```
OUT      = tasm
```

```
TESTFILE = functions.txt
SCANNER  = e3.l
PARSER   = e3.y

CC       = gcc
OBJ      = lex.yy.o y.tab.o
TESTOUT  = $(basename $(TESTFILE)).asm
OUTFILES = lex.yy.c y.tab.c y.tab.h y.output $(OUT)

.PHONY: build test clean
# 执行make会生成tasm可执行文件
build: $(OUT)
# 声明目标文件
test: $(TESTOUT)

clean:
  rm -f *.o $(OUTFILES)
# 如果编译生成的文件版本比functions.asm新
# 执行./tasm < test.c > fucntions.asm
$(TESTOUT): $(TESTFILE) $(OUT)
  ./$(OUT) < $< > $@
# 哪个可执行文件版本旧就生成哪个
$(OUT): $(OBJ)
  $(CC) -o $(OUT) $(OBJ)
# 如果lex文件和头文件比lex.yy.c版本更新的话
# 执行flex
lex.yy.c: $(SCANNER) y.tab.c
  flex $<
# 如果yacc文件比y.tab.c版本更新的话
# 执行bison
y.tab.c: $(PARSER)
  bison -vdty $<
```

2. if、while结构分析

```
/* 原代码 */
if(a == 0){
  a = a + 2;
} else{
  a--;
}
/* 中间代码 */
  arg a
_begIf_1:        // 每个If有编号
  push #0
  cmpeq          // 判断相等
  jz _elseIf_1   // PSW为0跳转到else
  push #2
  add
```

```
  pop a
  jmp _endIf_1  // 结束跳转endif
_elseIf_1:
  push #1
  sub
_endIf_1:
```

bison代码如下

```
%{
/* IF栈 满递增栈 */
int ii = 0, itop = -1, istack[100];
#define _BEG_IF     {istack[++itop] = ++ii;}  // 每个if的编号都入栈
#define _END_IF     {itop--;}
#define _i          (istack[itop])            // 当前if编号

/* While栈 满递增栈 */
int ww = 0, wtop = -1, wstack[100];
#define _BEG_WHILE  {wstack[++wtop] = ++ww;}  // 每个while的编号都入栈
#define _END_WHILE  {wtop--;}                 // 栈顶指针-1
#define _w          (wstack[wtop])            // 当前while编号
%}
/* If */
IfStmt      :   If '(' BoolExpr ')' Then StmtBlock EndThen EndIf    { /*
empty */}
            |   If '(' BoolExpr ')' Then Stmt EndThen EndIf    { /* empty
*/}
            |   If '(' BoolExpr ')' Then StmtBlock EndThen T_Else
StmtBlock EndIf
                                    { /* empty */}
            |   If '(' BoolExpr ')' Then Stmt EndThen T_Else Stmt EndIf

                                    { /* empty */}
            ;

If          :   T_If                { _BEG_IF; printf("_begIf_%d:\n", _i);
}

Then        :   /* empty */         { printf("\tjz _elseIf_%d\n", _i); }
            ;

EndThen     :   /* empty */         { printf("\tjmp
_endIf_%d\n_elseIf_%d:\n", _i, _i); }
            ;

EndIf       :   /* empty */         { printf("_endIf_%d:\n\n", _i);
_END_IF; }
```

3. 表达式分析

```
/**
  * add 会将操作数栈中的两个操作数出栈
  * 相加以后结果入栈
  * 这里用的是pcode的语法
  */
Expr        :   Constant            { /* empty */          }
            |   LValue              { /* empty */                    }
            |   Call                { /* empty */               }
            |   '('Expr')'          { /* empty */           }
            |   Expr '+' Expr       { printf("\tadd\n");       }
            |   Expr '-' Expr       { printf("\tsub\n");      }
            |   Expr '*' Expr       { printf("\tmul\n");       }
            |   Expr '/' Expr       { printf("\tdiv\n");       }
            |   Expr '%' Expr       { printf("\tmod\n");      }
            |   '-' Expr            { printf("\tneg\n");            }
            |   Expr '<' Expr       { printf("\tcmplt\n");       }
            |   Expr T_Le Expr      { printf("\tcmple\n");    }
            |   Expr '>' Expr       { printf("\tcmpgt\n");       }
            |   Expr T_Ge Expr      { printf("\tcmpge\n");       }
            |   Expr T_Eq Expr      { printf("\tcmpeq\n");        }
            |   Expr T_Ne Expr      { printf("\tcmpne\n");     }
            |   Expr T_And Expr     { printf("\tand\n");          }
            |   Expr T_Or Expr      { printf("\tor\n");           }
            |   '!' Expr            { printf("\tnot\n");            }
            |   T_ReadInteger '(' LValue ')'    { printf("\treadint %s\n",
$3); }
            |   T_ReadLine '(' LValue')'  { printf("\treadline %s\n", $3);
       }
            ;
/* 常量入栈 */
Constant    :   T_IntConstant       { printf("\tpush #%d\n", $1);  }
            |   T_DoubleConstant    { printf("\tpush #%f\n", $1); }
            |   T_BooleanConstant   { printf("\tpush #%d\n", $1);   }
            |   T_StringConstant    { printf("\tpush #%s\n", $1); }
            |   T_Null              {                  }
            ;
```

4. 自增自减

i ++ 和 ++ i 的区别我没写出来...思路是#1最后入栈

```
/* 自增语句 */
IncrementStmt:   LValue T_Ic         { printf("\tpush #1\n\tadd\n");   }
             ;
/* 自减语句 */
DecrementStmt:   LValue T_Dc         { printf("\tpush #1\n\tsub\n");   }
             ;
```

# 代码清单

1. `e3.l`代码

```
%{
#include "y.tab.h"
int comment_flag;
int string_flag;
int cur_lineno = 1;
void lex_error(char* msg, int line);
%}

/*常量*/
INT ([+|-]?[0-9]+)
INT_HEX (0[xX][a-fA-F0-9]+)
INT_OCT (0[1-7][0-7]*)
INT_BIN (0[bB][01]+)
INTEGER {INT}|{INT_HEX}|{INT_OCT}|{INT_BIN}
FLOAT {INT}\.[0-9]+
SCIENCE (([0-9]+\.[0-9]*)|([0-9]*\.[0-9]+)|INT)[Ee][-+]?[0-9]+
DOUBLE {SCIENCE}|{FLOAT}
STRING \"[^\"\n]*\"
STRING_BEGIN \"

/*标识符*/
IDENTIFIER [a-z_A-Z][a-z_A-Z0-9]*
/*关键字*/
KW_LE <=
KW_GE >=
KW_EQ ==
KW_NE !=
KW_IC "++"
KW_DC "--"
KW_AND &&
KW_OR "||"
KW_VOID void
KW_INT int
KW_DOUBLE double
KW_BOOLEAN bool
```

```
KW_STRING string
KW_NULL null
KW_FOR for
KW_WHILE while
KW_IF if
KW_ELSE else
KW_RETURN return
KW_BREAK break
KW_PRINT print
KW_READINT readInt
KW_READLINE readLine
KW_TRUE true
KW_FALSE false
WHITESPACE   [ \t\r\a]+
OPERATOR [+*-/%=,;!<>(){}\[\]]

/*注释*/
COMMENT ("//".*)|("/*"([*]*(([^*/])+([/])*)*)*"*/")
COMMENT_BEGIN "/*"

/*错误*/
AERROR .
STRING_END_ERROR \"[^\"\n]*$

%%
[\n]             { cur_lineno++;           }
{OPERATOR}       { return yytext[0];         }
{KW_LE}          { return T_Le;              }
{KW_GE}          { return T_Ge;              }
{KW_EQ}          { return T_Eq;              }
{KW_NE}          { return T_Ne;              }
{KW_IC}          { return T_Ic;              }
{KW_DC}          { return T_Dc;              }
{KW_AND}         { return T_And;             }
{KW_OR}          { return T_Or;              }
{KW_VOID}        { return T_Void;            }
{KW_INT}         { return T_Int;             }
{KW_DOUBLE}      { return T_Double;          }
{KW_BOOLEAN}     { return T_Boolean;         }
{KW_STRING}      { return T_String;          }
{KW_NULL}        { return T_Null;            }
{KW_FOR}         { return T_For;             }
{KW_WHILE}       { return T_While;           }
{KW_IF}          { return T_If;              }
{KW_ELSE}        { return T_Else;            }
{KW_RETURN}      { return T_Return;          }
{KW_BREAK}       { return T_Break;           }
{KW_PRINT}       { return T_Print;           }
{KW_READINT}     { return T_ReadInteger;     }
```

```
{KW_READLINE}    { return T_ReadLine;         }

{KW_TRUE}        { yylval.bval = 1;             return T_BooleanConstant; }
{KW_FALSE}       { yylval.bval = 0;             return T_BooleanConstant; }

{INTEGER}        { yylval.ival = atoi(yytext); return T_IntConstant;     }
{DOUBLE}         { yylval.dval = atof(yytext); return T_DoubleConstant;  }
{STRING}         { yylval.sval = strdup(yytext);       return T_StringConstant;
}
{IDENTIFIER}     { yylval.sval = strdup(yytext);       return T_Identifier;
}
{WHITESPACE}     { /* skip */                }
{COMMENT}        { /* skip */                }

{AERROR}         { lex_error("Unrecognized character", cur_lineno);    }
{STRING_END_ERROR}  {lex_error("String whithout end", cur_lineno);}
<<EOF>>         { return 0;                  }


%%

void lex_error(char* msg, int line) {
    printf("\nError at line %-3d: %s\n\n", line, msg);
}

int yywrap(void) {
    return 1;
}
```

2. `e3.y`代码

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <sys/malloc.h>

extern int cur_lineno;
int yylex();
void yyerror(const char* msg);
/* IF栈 */
int ii = 0, itop = -1, istack[100];
#define _BEG_IF     {istack[++itop] = ++ii;}
#define _END_IF     {itop--;}
#define _i          (istack[itop])

/* While栈 */
int ww = 0, wtop = -1, wstack[100];
#define _BEG_WHILE  {wstack[++wtop] = ++ww;}
#define _END_WHILE  {wtop--;}
```

```
#define _w          (wstack[wtop])
%}

%start Program

/*bison可以从这个定义中产生yylval的定义*/
%union {
    int       ival;
    char      *sval;
    double    dval;
    int       bval;
}

%token    T_Le
%token    T_Ge
%token    T_Eq
%token    T_Ne
%token    T_Ic
%token    T_Dc
%token    T_And
%token    T_Or
%token    T_Void
%token    T_Int
%token    T_Double
%token    T_Boolean
%token    T_String
%token    T_Null
%token    T_For
%token    T_While
%token    T_If
%token    T_Else
%token    T_Return
%token    T_Break
%token    T_Print
%token    T_ReadInteger
%token    T_ReadLine
%token    <ival>   T_IntConstant
%token    <dval>   T_DoubleConstant
%token    <sval>   T_StringConstant
%token    <sval>   T_Identifier
%token    <bval>   T_BooleanConstant
%token    T_Lineend

/* 左值是string类型 */
%type <sval> LValue

%left ')' ']'
%left ','
%left '='
```

```
%left T_OR
%left T_AND
%nonassoc T_Eq T_Ne
%nonassoc '<' '>' T_Le T_Ge
%left '+' '-'
%left '*' '/' '%'
%right '!'
%left T_Ic T_Dc
%left '.' '(' '['
%nonassoc T_ELSE
%nonassoc T_IFX

%%
/* 解析起点 */
Program     :   Fieldlist           { /* empty */      }
            |   Fieldlist Function  { /* empty */    }
            |   error               { yyerror("Program");
              }
            ;
/* 函数 */
Function    :   Type FuncName  '(' FormalParams ')' StmtBlock
                                    { printf( "ENDFUNC\n\n"); }
            ;

FuncName    :   T_Identifier        { printf("FUNC @%s:\n", $1); }
            ;
/* 字段列表 */
Fieldlist   :   Fieldlist Field     { /* empty */      }
            |                       { /* empty */                    }
            |   error               { yyerror("Fieldlist");
              }
            ;
/* 字段 */
Field       :   Variable            { /* empty */                 }
            |   Function            { /* empty */          }
            ;
/* 变量声明 */
VariableDecl:   Type T_Identifier   { printf("\tvar %s",$2); }
            |   error               { yyerror("VariableDecl");
             }
            ;
/* 变量定义 */
Variable    :   /* empty */        { /* empty */ }
            |   Variable VariableDecl ';'   { printf("\n\n");   }
            |   error               { yyerror("Variable");
             }
            ;
/* 变量类型 */
Type        :   T_Int               { /* empty */               }
```

```
            |   T_Double            { /* empty */             }
            |   T_Boolean           { /* empty */               }
            |   T_String            { /* empty */             }
            |   T_Void              { /* empty */               }
            |   Type '[' ']'        { /* empty */             }
            |   error               { yyerror("Type");
             }
            ;
/* 形式参数 */
FormalParams:                       { /* empty */          }
            |    _Params            { printf("\n\n");          }
            ;
_Params     :   Type T_Identifier   { printf("\targ %s", $2);     }
            |   _Params ',' Type T_Identifier { printf(", %s", $4);  }
            |                       { /* empty */          }
            ;

/* 语句块 */
StmtBlock   :   '{' Stmtlist '}'    { /* empty */ }
            |   error               { yyerror("StmtBlock");
              }
            ;
/* 语句列表 */
Stmtlist    :   Stmtlist Stmt       { /* empty */  }
            |                       { /* empty */                    }
            ;
/* 语句 */
Stmt        :   Variable            { /* empty */           }
            |   SimpleStmt ';'      { /* empty */       }
            |   IfStmt              { /* empty */             }
            |   WhileStmt           { /* empty */         }
            |   ForStmt             { /* empty */           }
            |   BreakStmt ';'       { /* empty */        }
            |   ReturnStmt ';'      { /* empty */       }
            |   PrintStmt ';'       { /* empty */        }
            |   StmtBlock           { /* empty */               }
            ;

/* 简单语句 如赋值自增 */
SimpleStmt  :   LValue '=' Expr     { printf("\tpop %s\n\n", $1);      }
            |   IncrementStmt       { /* empty */      }
            |   DecrementStmt       { /* empty */      }
            |   Call                { /* empty */        }
            |                       { /* empty */            }
            |   error               { yyerror("SimpleStmt");
             }
            ;
/* 自增语句 */
IncrementStmt:   LValue T_Ic         { printf("\tpush #1\n\tadd\n");   }
```

```
            ;
/* 自减语句 */
DecrementStmt:   LValue T_Dc          { printf("\tpush #1\n\tsub\n");   }
            ;

/* 左值 */
LValue      :   T_Identifier          { $$ = $1;            }
            |   Expr '[' Expr ']'     { /* empty */   }
            |   error                 { yyerror("LValue");
             }
            ;

/* 调用 */
Call        :   T_Identifier '(' ActualParams ')'
                                      { /* empty */}
            ;
/* 实参 */
ActualParams:   ExprMore              { /* empty */   }
            |                         { /* empty */                }
            ;
/* 多个表达式 */
ExprMore    :   Expr                  { /* empty */           }
            |   ExprMore ',' Expr     { /* empty */           }
            ;
/* For语句 */
ForStmt     :   T_For '(' SimpleStmt ';' BoolExpr ';' SimpleStmt ')' Stmt
                                      { /* empty */}
            ;
/* while */
WhileStmt   :   While '(' BoolExpr ')' Do StmtBlock EndWhile
                                      { /* empty */ }
            ;

While       :   T_While         { _BEG_WHILE; printf("_begWhile_%d:\n", _w); }
            ;

Do          :   /* empty */     { printf("\tjz _endWhile_%d\n", _w); }
            ;

EndWhile    :   /* empty */     { printf("\tjmp
_begWhile_%d\n_endWhile_%d:\n\n", _w, _w); _END_WHILE; }
            ;
/* If */
IfStmt      :   If '(' BoolExpr ')' Then StmtBlock EndThen EndIf    { /* empty
*/}
            |   If '(' BoolExpr ')' Then Stmt EndThen EndIf    { /* empty */}
            |   If '(' BoolExpr ')' Then StmtBlock EndThen T_Else StmtBlock
EndIf    { /* empty */}
```

```
            |   If '(' BoolExpr ')' Then Stmt EndThen T_Else Stmt EndIf    {
/* empty */}
            ;

If          :   T_If                { _BEG_IF; printf("_begIf_%d:\n", _i); }

Then        :   /* empty */         { printf("\tjz _elseIf_%d\n", _i); }
            ;

EndThen     :   /* empty */         { printf("\tjmp _endIf_%d\n_elseIf_%d:\n",
_i, _i); }
            ;

EndIf       :   /* empty */         { printf("_endIf_%d:\n\n", _i); _END_IF; }
            ;
/* Return */
ReturnStmt  :   T_Return            { printf("\tret\n\n");       }
            |   T_Return Expr       { printf("\tret ~\n\n"); }
            ;
/* Break */
BreakStmt   :   T_Break             { printf("\tjmp _endWhile_%d\n", _w); }
            ;

PrintStmt   :   T_Print '(' ExprMore ')'{ /* empty */}
            ;

BoolExpr    :   Expr                { /* empty */           }
            ;
/* 表达式 */
Expr        :   Constant            { /* empty */          }
            |   LValue              { /* empty */                    }
            |   Call                { /* empty */               }
            |   '('Expr')'          { /* empty */           }
            |   Expr '+' Expr       { printf("\tadd\n");       }
            |   Expr '-' Expr       { printf("\tsub\n");      }
            |   Expr '*' Expr       { printf("\tmul\n");       }
            |   Expr '/' Expr       { printf("\tdiv\n");       }
            |   Expr '%' Expr       { printf("\tmod\n");      }
            |   '-' Expr            { printf("\tneg\n");             }
            |   Expr T_Dc           { /* empty */     }
            |   T_Dc Expr           { /* empty */     }
            |   Expr T_Ic           { /* empty */     }
            |   T_Ic Expr           { /* empty */     }
            |   Expr '<' Expr       { printf("\tcmplt\n");       }
            |   Expr T_Le Expr      { printf("\tcmple\n");    }
            |   Expr '>' Expr       { printf("\tcmpgt\n");       }
            |   Expr T_Ge Expr      { printf("\tcmpge\n");       }
            |   Expr T_Eq Expr      { printf("\tcmpeq\n");        }
            |   Expr T_Ne Expr      { printf("\tcmpne\n");     }
```

```
            |   Expr T_And Expr     { printf("\tand\n");          }
            |   Expr T_Or Expr      { printf("\tor\n");           }
            |   '!' Expr            { printf("\tnot\n");            }
            |   T_ReadInteger '(' LValue ')'   { printf("\treadint %s\n", $3);
}
            |   T_ReadLine '(' LValue')'  { printf("\treadline %s\n", $3);
   }
            ;

/* 常量 */
Constant    :   T_IntConstant       { printf("\tpush #%d\n", $1);  }
            |   T_DoubleConstant    { printf("\tpush #%f\n", $1); }
            |   T_BooleanConstant   { printf("\tpush #%d\n", $1);   }
            |   T_StringConstant    { printf("\tpush #%s\n", $1); }
            |   T_Null              {                  }
            ;
%%

int main() {
    return yyparse();
}

void yyerror(const char* msg) {
    printf("ERROR: %s at line %d \n", msg, cur_lineno);
}
```

3. `makefile`代码如下

```
OUT      = tasm
TESTFILE = functions.txt
SCANNER  = e3.l
PARSER   = e3.y

CC       = gcc-8
OBJ      = lex.yy.o y.tab.o
TESTOUT  = $(basename $(TESTFILE)).asm
OUTFILES = lex.yy.c y.tab.c y.tab.h y.output $(OUT)

.PHONY: build test simulate clean

build: $(OUT)

test: $(TESTOUT)

clean:
  rm -f *.o $(OUTFILES)

$(TESTOUT): $(TESTFILE) $(OUT)
  ./$(OUT) < $< > $@
```

```
$(OUT): $(OBJ)
  $(CC) -o $(OUT) $(OBJ)

lex.yy.c: $(SCANNER) y.tab.c
  flex $<

y.tab.c: $(PARSER)
  bison -vdty $<
```