**编译器设计与实现**

学校：NWPU

学院：计算机学院

目录

[1 词法分析器 4](#_Toc8391088)

[1.1 概述 4](#_Toc8391089)

[1.2 主要特性 4](#_Toc8391090)

[1.3 实现方式 4](#_Toc8391091)

[1.3.1 程序主体 4](#_Toc8391092)

[1.3.2 注释的实现方式 4](#_Toc8391093)

[1.3.3 记录行号实现 5](#_Toc8391094)

[1.3.4 输入方式实现 5](#_Toc8391095)

[1.3.5 其他复杂常量表示的实现 6](#_Toc8391096)

[2 语法分析器 6](#_Toc8391097)

[2.1 概述 6](#_Toc8391098)

[2.2 主要特性 6](#_Toc8391099)

[2.3 实现方式 7](#_Toc8391100)

[2.3.1 词法分析器主主体 7](#_Toc8391101)

[2.3.2 语法树的建立 7](#_Toc8391102)

[2.3.3 垂悬else问题解决 7](#_Toc8391103)

[2.3.4 运算符优先级 7](#_Toc8391104)

[3 抽象语法树的建立和打印 10](#_Toc8391105)

[3.1 概述 10](#_Toc8391106)

[3.2 主要特性 10](#_Toc8391107)

[3.3 实现方式 10](#_Toc8391108)

[3.3.1 语法树的建立 10](#_Toc8391109)

[3.3.2 语法树的化简 10](#_Toc8391110)

[3.3.3 打印语法树 10](#_Toc8391111)

[4 语法树的解析和中间代码生成 12](#_Toc8391112)

[4.1 概述 12](#_Toc8391113)

[4.2 主要特性 12](#_Toc8391114)

[4.3 实现方式 13](#_Toc8391115)

[4.3.1 程序主体 13](#_Toc8391116)

[4.3.2 复杂的变量初始化 13](#_Toc8391117)

[4.3.3 内置输入输出函数 15](#_Toc8391118)

[4.3.4 变量的定义域和生存期 15](#_Toc8391119)

[4.3.5 布尔表达式的短路求值 16](#_Toc8391120)

[4.3.6 ”,”表达式，返回左值 17](#_Toc8391121)

[4.3.7 if if-else else 语句 17](#_Toc8391122)

[4.3.8 while do-while for 三种循环语句 18](#_Toc8391123)

[4.3.9 continue，break 语句 18](#_Toc8391124)

[4.3.10 goto-label语句 18](#_Toc8391125)

[4.3.11 类型检查 19](#_Toc8391126)

[4.3.12 错误定位 19](#_Toc8391127)

[5 8086汇编代码生成 19](#_Toc8391128)

[5.1 概述 19](#_Toc8391129)

[5.2 主要特性 19](#_Toc8391130)

[5.3 实现方式 19](#_Toc8391131)

[5.3.1 主程序 19](#_Toc8391132)

[5.3.2 输入和打印功能 21](#_Toc8391133)

[5.3.3 函数的递归调用 21](#_Toc8391134)

[5.4 一个例子，递归方式求斐波那契数列的第n项（提供视频演示） 21](#_Toc8391135)

## 词法分析器

### 概述

* 基于flex工具实现的c语言词法分析器
* 对应文件：my\_compiler.l
* 输入：符合C语言语法的文本文件（参数为文件名） 或则是键盘输入流（不加参数）
* 输入：1. 用于bison处理的token流 2. 在内存中已经构造好的AST**叶节点**对象

### 主要特性

1. 记录每个token的行号，保存在AST结点中，用于将来的错误定位提示。
2. 支持// 格式的注释 和/\*\*/格式的注释
3. 可识别C语言全部32个关键字和各种运算符，包括"..."
4. 支持C语言复杂的常数表示，如8进制，16进制，表示浮点精度的数1.23f，1.1L,数指数表示，如1e3，整数的格式，如123u，123L，宽字符串L”hello”等。
5. 可以支持文件或直接在控制台输入

### 实现方式

#### 程序主体

使用flex可以简单的实现一个词法分析器，分析器没识别到一个token，做两件事，第一是调用 #include "AST.h" 中的new\_node()函数，它有三个参数：

参数1为字符串，表示改token的类型

参数2表示它的孩子数量，由于基本的token都为叶节点，所以孩子数量都为0

参数3为yylineno，表示改token所在的行数，为以后错误定位和提示做准备。

第二是返回给bison改token的类型，用于语法分析。

"auto"     { yylval.node = new\_node("AUTO",0,yylineno); return(AUTO); }

"break"    { yylval.node = new\_node("BREAK",0,yylineno);    return(BREAK); }

"case"     { yylval.node = new\_node("CASE",0,yylineno);     return(CASE); }

#### 注释的实现方式

我的flex貌似不支持最短匹配，所示实现方式比较复杂

1. // 的实现方式使用正则表达式 "//"[^\n]\* 没啥讲的
2. /\*\*/的实现方式使用正则表达式 "/\*"(([^\\*]\*(\\*[^\/])?)\*)"\*/"

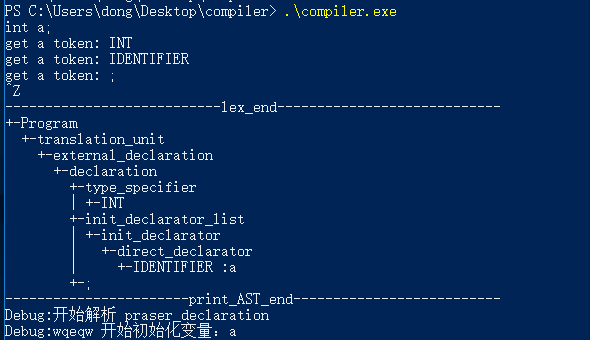
开头和结尾的"/\*"和"\*/"表示开始和结尾，中间的([^\\*]\*(\\*[^\/])?)\* 表示非\*号的任意个字符重复或者 \*号加上非/字符的重复

#### 记录行号实现

使用flex的%option yylineno 即开启记录行号功能，它自动的把行号记录到变量yylineno中。

#### 输入方式实现

提供两种输入方式，一种是程序运行时加文件参数，这是会自动第从文件中读取，二是在运行程序后直接输入C源程序，每按一次回车就会输出解析到的字符，输入^Z（Ctrl+Z）表示EOF，程序开始语法解析。



实现方式： （程序实际在my\_compiler.y的main函数中）

    yyin=stdin;

    if(argc>1) {

        if((yyin=fopen(argv[1],"r"))==NULL) {

            printf("Can't open file %s\n",argv[1]);

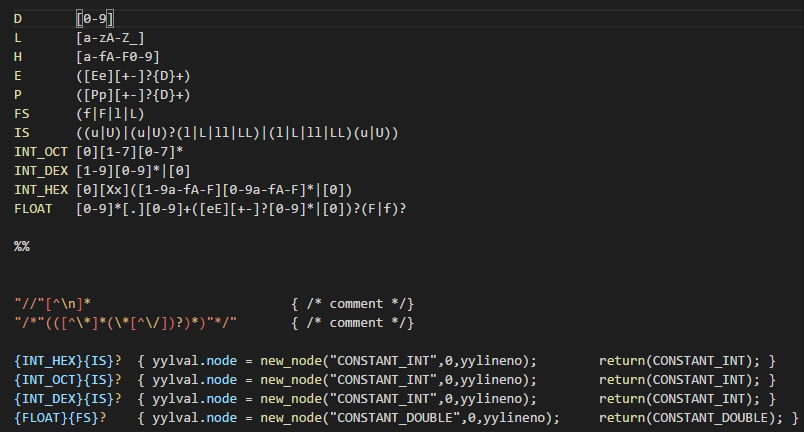
            return -1;

        }

    }

    yyparse();

#### 其他复杂常量表示的实现



## 语法分析器

### 概述

* 基于bison工具实现的C语言语法分析器
* 对应文件：my\_compiler.y
* 输入：flex生成的token流
* 唯一输出：完整的抽象语法树
* 参考：<http://www.quut.com/c/ANSI-C-grammar-l-1998.html>
* 详细语法解析结构见Visio文件**compiler.vsdx**

### 主要特性

1. 可以解析C语言几乎全部的语法，包括结构体，结构体的空间分配，三元表达式，强制类型转换，·号， 逗号表达式，指针，复杂的初始化等（指建立语法树，只有一部分语法实现了中间代码翻译）
2. 基于文法的运算符优先级
3. 通过主动标识优先级解决垂悬else的问题（这也是唯一一处用到主动标注优先级的地方）
4. 程序的唯一输出即为在内存中的抽象语法树，树的解析和错误提示等都在后续的程序中，到此，源文件不再提供任何信息。

### 实现方式

#### 词法分析器主主体

使用bison语法实现，具体含义见Visio文件**compiler.vsdx**

#### 语法树的建立

使用#include "AST.h" 中的new\_node()函数，参数详见上一节的介绍，考虑到各个结点的孩子数和兄弟数不相同，所以采用左孩子右兄弟的表示方法。

#### 垂悬else问题解决

使用%nonassoc表示不关心结合性，但写在后面的短语优先级较高，使用%prec主动表示优先级，后接一个变量，将THE\_FIRST\_IF\_IN\_FRONT 写在ELSE前表示ELSE的优先级更高，这样解析到ELSE时总是尝试和前一个没有ELSE的IF去匹配。

%nonassoc THE\_FIRST\_IF\_IN\_FRONT

%nonassoc ELSE

selection\_statement:

    IF '(' expression ')' statement %prec THE\_FIRST\_IF\_IN\_FRONT {

        $$ = new\_node("selection\_statement",5,$1,$2,$3,$4,$5);

    }

| IF '(' expression ')' statement ELSE statement {

        $$ = new\_node("selection\_statement",7,$1,$2,$3,$4,$5,$6,$7);

    }

| SWITCH '(' expression ')' statement {

        $$ = new\_node("selection\_statement",5,$1,$2,$3,$4,$5);

    }

;

#### 运算符优先级

严格按照C语言的优先级来设计，下表的优先级在程序中均有体现（在compiler.vsdx左下角，更加直观）

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| **1** | ++ -- | Suffix/postfix increment and decrement | Left-to-right |
| () | Function call |
| [] | Array subscripting |
| . | Structure and union member access |
| −> | Structure and union member access through pointer |
| (*type*){*list*} | Compound literal(C99) |
| **2** | ++ -- | Prefix increment and decrement | Right-to-left |
| + − | Unary plus and minus |
| ! ~ | Logical NOT and bitwise NOT |
| (*type*) | Type cast |
| \* | Indirection (dereference) |
| & | Address-of |
| sizeof | Size-of |
| \_Alignof | Alignment requirement(C11) |
| **3** | \* / % | Multiplication, division, and remainder | Left-to-right |
| **4** | + − | Addition and subtraction |
| **5** | << >> | Bitwise left shift and right shift |
| **6** | < <= | For relational operators < and ≤ respectively |
| > >= | For relational operators > and ≥ respectively |
| **7** | == != | For relational = and ≠ respectively |
| **8** | & | Bitwise AND |
| **9** | ^ | Bitwise XOR (exclusive or) |
| **10** | | | Bitwise OR (inclusive or) |
| **11** | && | Logical AND |
| **12** | || | Logical OR |
| **13** | ?: | Ternary conditional | Right-to-Left |
| **14** | = | Simple assignment |
| += −= | Assignment by sum and difference |
| \*= /= %= | Assignment by product, quotient, and remainder |
| <<= >>= | Assignment by bitwise left shift and right shift |
| &= ^= |= | Assignment by bitwise AND, XOR, and OR |
| **15** | , | Comma | Left-to-right |

## 抽象语法树的建立和打印

### 概述

* 为了节省空间，采用左孩子右兄弟的方式表示抽象语法树
* 由bison程序生成
* 后续的所有语法解析都是对抽象语法树的操作
* 对应文件AST.h ，AST.cpp

### 主要特性

1. 可以比较清晰，可视化好的方式**打印出语法树**
2. 语法树中有一些结点只有一个分支（如基本表达式推出乘法表达式，进而推出加法表达式，如果没有乘法，则基本表达式可以直接推出加法表达式），对于这些结点，可以选择保留或为了**简化语法树**的结构而选择不保留（甚至可以删除代码中的{}，（）等辅助信息）

### 实现方式

#### 语法树的建立

核心函数为struct AST\* new\_node(string type, int num,...)

它可以接收**数量可变的参数**，string type 表示结点类型，int num表示孩子个数，… 表示数量不定的孩子结点，每次调用时，将第一个孩子结点放在它的左分支，将剩余的孩子结点一次放在第一个孩子的右部，这样就实现了以左孩子右兄弟的方式建立语法树。

#### 语法树的化简

有些结点只有一个孩子，理论上讲，这些结点并不起关键作用，可以将他们删掉，（如基本表达式推出乘法表达式，进而推出加法表达式，如果没有乘法，则基本表达式可以直接推出加法表达式），还有一些信息在生成语法树之后就显得冗余了，比如(1+2)\*3中的括号，完全可以将他们删除（但是本程序中为了解析方便且统一，仍然保留一些）。

在实现的过程中，先定义一个关键类型的数组（在程序中为string essential\_type[]），表示这些类型在解析中起较大作用，即使是冗余结点，也保存在语法树中，对于非关键的类型，如果检测到它只有一个孩子，那么直接将它的孩子指向它的父亲。

#### 打印语法树

对于左孩子右兄弟的树，采用前序遍历可以较为直观的打印语法树，打印时将树的层数也传进参数中，一个结点处在树的第几层就打印几个空格，这样结构层级关系更加清晰。

为了使树的父子关系更为直观，只是打印空格远远不够，如果我们在打印一个结点的孩子结点时，都加上“|”，在孩子结点结束后换为空格，这样可以更加友好的向人们展示语法树。

如下图，表示if(n==2){ans=1;}else … 的语法树



实现代码;

void print\_AST(AST \*head,int leavel) {

if(head!=NULL) {

string type = head->type;

for(int i=0;i<leavel;++i) {

if(printtreestack[i]==0){

cout<<" ";

}else{

cout<<"| ";

}

}

cout<<"+-" << head->type;

//有真实值的类型

if(head->type == "IDENTIFIER"||head->type == "CONSTANT\_INT" || head->type == "CONSTANT\_DOUBLE" || head->type == "STRING\_LITERAL") {

cout << " :" << head->text;

}

cout << endl;

if(head->right!=NULL){

printtreestack[leavel]=1;

}

print\_AST(head->left,leavel+1);

printtreestack[leavel]=0; //将高位的1归零,很关键

print\_AST(head->right,leavel);

}

}

## 语法树的解析和中间代码生成

### 概述

* 输入：抽象语法树
* 输出：中间代码（IF-GOTO的形式），DEBUG信息
* 从根节点开始，不断地向下解析

### 主要特性

1. 支持多个函数定义，全局变量定义。
2. 支持**复杂的变量初始化方式**，如int a=1,b=1\*4,c,d=a+c+1;
3. 内置两个函数，输入sacn 和输出print
4. 实现了变量的**定义域和生存期**
5. 布尔表达式的**短路求值**
6. 支持**”,”表达式**，返回值为逗号左边的值，如a=1,2,3; //a=1
7. **if else-if else** 语句
8. **while do-while for** 三种循环语句
9. **continue，break** 语句
10. **goto-label语句**
11. 提供简单的**类型检查**
12. 提供简单的**错误定位**和提示
13. 使用**多个变量映射表**，如一个函数名为a，不影响一个变量名为a，也不影响标签名为a，多个函数之间的变量互不影响。

### 实现方式

#### 程序主体

从

void Praser::praserAST(struct AST\* node) {

开始，一般来说，每个函数负责解析一类结点，如praser\_function\_definition(node);就是解析函数定义的结点，他所做的工作是分析传入结点的子节点，根据子节点的不同做相应的动作，可以是生成代码，创建变量，创建代码块，将子节点交给更具体的函数解析等等，这样不断调用，就完成了树的解析，并且生成了中间代码。

其中用到的**重要数据结构**为：

    int varNum = 0; //临时变量个数

    int tempNum = 0; ////变量个数

    int labelNum = 0; //标签号

    vector<string> codeList; //中间代码

    AsmCode asmcode; //x86代码生成对象

    map<string, funcNode> funcPool; //函数池（函数名--> 函数结点）

    map<string, string> labelPool; //标签映射关系（统一把标签映射为lable\_i的形式）

    vector<struct Block> blockStack;//维护的代码块栈

#### 复杂的变量初始化

有如下程序：

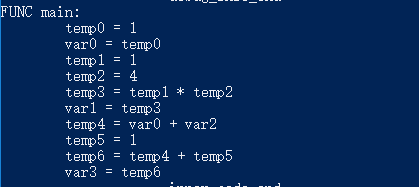
int main()

{

    int a=1,b=1\*4,c,d=a+c+1;

}

生成的中间代码为：（中间变量用temp表示，变量用var表示）



生成的语法树为：

+-declaration //声明

+-type\_specifier //类型

| +-INT

+-init\_declarator\_list //通过init\_declarator\_list的递归来声明多个变量

| +-init\_declarator\_list//init\_declarator\_list->init\_declarator\_list

| | +-init\_declarator\_list // + init\_declarator

| | | +-init\_declarator\_list

| | | | +-init\_declarator

| | | | +-direct\_declarator

| | | | | +-IDENTIFIER :a //表示变量

| | | | +-=

| | | | +-initializer //初始化语句

| | | | +-assignment\_expression//赋值语句

| | | | +-primary\_expression//基本表达式

| | | | +-CONSTANT\_INT :1//常量1

| | | +-,

| | | +-init\_declarator

| | | +-direct\_declarator

| | | | +-IDENTIFIER :b

| | | +-=

| | | +-initializer

| | | +-assignment\_expression

| | | +-multiplicative\_expression

| | | +-primary\_expression

| | | | +-CONSTANT\_INT :1

| | | +-\*

| | | +-primary\_expression

| | | +-CONSTANT\_INT :4

| | +-,

| | +-init\_declarator

| | +-direct\_declarator

| | +-IDENTIFIER :c

| +-,

| +-init\_declarator

| +-direct\_declarator

| | +-IDENTIFIER :d

| +-=

| +-initializer

| +-assignment\_expression

| +-additive\_expression

| +-additive\_expression

| | +-primary\_expression

| | | +-IDENTIFIER :a

| | +-+

| | +-primary\_expression

| | +-IDENTIFIER :c

| +-+

| +-primary\_expression

| +-CONSTANT\_INT :1

+-;

#### 内置输入输出函数

本程序为了使用方便，编译器里内置了两个函数，一个为scan，表示从标准输入中读入一个数，另一个为print，表示向标准输出中打印一个数。

在写程序时，可以不定义直接使用，不会发生找不到函数的错误，**后续还可以生成输入输出的汇编代码**。

实现方式为在解析语法树前执行如下代码，人为地将函数加入函数池中：

    //加入内置函数 scan：读入一个int 并返回 print：打印一个int值

    funcNode print;

    print.name = "print";

    print.rtype = "void";//返回值

    varNode pnode;//传入参数

    pnode.type = "int"; //传入参数的类型

    print.paralist.push\_back(pnode);

    funcPool.insert({ "print", print });

    funcNode scan;

    scan.name = "scan";

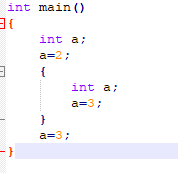
    scan.rtype = "int";

    funcPool.insert({"scan",scan});

#### 变量的定义域和生存期

C语言中，一个大括号{}内或者if else等表达式后面的一条语句被认为是一个新的代码段，在代码段内部可以定义新的变量，如果在代码段内定义的变量和代码段外的重复，并不会造成冲突，并且统一视为代码段内部的变量，跳出代码段时，变量被清除。

如有如下程序：



翻译成中间代码为，可以看到{}内外的a是两个不同的变量。



实现方式：

在解析 函数定义，{}，if，for，while等可以生成新的代码块语句时，生成一个block并将其push到blockStack中，代码块结束时再将刚才生成的block弹出。

block中包含函数结点（如果这个block是函数），变量map等数据结构。

//添加一个新的block

  Block newblock;

  blockStack.push\_back(newblock);

//中间的代码////

//弹出添加的block

  blockStack.pop\_back();

查找变量时，先从外层的block，如果没找到，一层一层深入，直到遍历整个blockStack，如果还未找到，提示出现错误。

#### 布尔表达式的短路求值

解析 （表达式1）||（表达式2） 的伪代码与实现为

|  |  |
| --- | --- |
| 解析 表达式1  如果 表达式1 为假，解析表达式2  为真，解析完毕，直接跳出，结果为真  解析表达式2  如果 表达式2 为假，结果为假  为真，解析完毕，结果为真 | temp1=exp1  if temp1==FALSE goto label\_1  value=TRUE  goto exit  label\_1：  if temp2==FALSE goto label\_2  value=TRUE  goto exit  label\_2:  value=FALSE  exit: |

实现：

varNode Praser::praser\_logical\_or\_expression(struct AST\* logical\_or\_exp) {

    //logical\_or\_expression -> logical\_or\_expression OR\_OP logical\_and\_expression

    varNode node1 = praser\_assignment\_expression(logical\_or\_exp->left);

    string label1 = getLabelName();

    string label2 = getLabelName();

    string tempname = "temp" + inttostr(tempNum);

    ++tempNum;

    varNode newnode = createTempVar(tempname, node1.type);

    addCode("IF " + getNodeName(node1) + " == FALSE" + " GOTO " + label1);

    addCode(newnode.name + " = " + node1.name);

    addCode("GOTO " + label2);//跳出整个判断体

    addCode(label1 + ":");

    varNode node2 = praser\_assignment\_expression(logical\_or\_exp->left->right->right);

    addCode(newnode.name + " = " + node2.name);

    addCode(label2 + ":");

    return newnode;

}

&& 表达式实现方式类似，不再赘述，详细可参考函数praser\_logical\_and\_expression

#### ”,”表达式，返回左值

对于表达式：a=exp1,exp2;

解析时先解析exp2，但不做动作（可能语句本身的副作用会修改值，如i++），后解析exp1，并返回exp1的值，这样就实现的解析”,”表达式。

    else if(node->left->type == "expression") {

        praser\_assignment\_expression(node->left->right->right);

        return praser\_expression(node->left);

    }

该实现方式对a=exp1,exp2,exp3,exp4; 等嵌套的”,”表达式也适用。

#### if if-else else 语句

比较简单，注意要记得添加一个新的block

Block newblock;

blockStack.push\_back(newblock);

AST\* expression = node->left->right->right;

varNode exp\_rnode = praser\_expression(expression);

AST\* statement = node->left->right->right->right->right;

string label1 = getLabelName();

string label2 = getLabelName();

addCode("IF " + getNodeName(exp\_rnode) + " != 0" + " GOTO " + label1);

addCode("GOTO " + label2);

addCode(label1 + ":");

praser\_statement(statement);

addCode(label2 + ":");

//弹出添加的block

blockStack.pop\_back();

#### while do-while for 三种循环语句

比较简单，这里就不贴代码了。

注意for要分别处理(exp;exp;exp)和(exp;exp;)两种情况。

要记得创建新的block 以支持for(int i=0;i<n;i=i+1)的写法

#### continue，break 语句

在进入循环语句的代码块段，block结构体中的breakLabelname和continueLabelname发挥作用，分别记录循环的跳出位置和开始位置，解析时如果遇到continue，break，将其转为对应的goto语句即可。（如果breakLabelname和continueLabelname为空时遇到continue，break 语句，则报错）。

struct Block { //独立的代码段，有独立的命名空间

    funcNode func;

    bool isfunc = false; //记录是否是函数

    map<string, struct varNode> varMap; //变量的map

    string breakLabelname=""; //块的break位置

    string continueLabelname="";

};

#### goto-label语句

C程序中遇到label：标签时，利用map<string, string> labelPool;将标签统一映射为label\_n的形式，并加入labelPool，当遇到goto label 时，从labelPool找出label标签名即可。

#### 类型检查

赋值时检查等号两边是否是同一种类型，如果类型不同，则提示错误。

另外，函数的返回值也检查是否和声明的类型相符。

#### 错误定位

当解析器认为出现错误时，调用以下函数，第一个参数为出错的行号，第二个参数为错误类型。结尾使用exit(1);结束进程。

每个结点的行号是结点的一个属性。

void Praser::error(int line, string error) {

    cout << "line:" << line << "[Error] "<<error << endl;

    exit(1);

}

## 8086汇编代码生成

### 概述

* 输入：中间代码
* 输出：x86汇编代码，（emu8086格式）
* 可以在**emu8086**上运行，并生成结果

### 主要特性

1. 将中间代码生成对应的汇编语言
2. 数据的输入和打印汇编实现（功能有限，目前只支持整形数的输入和打印）
3. 支持多函数，每次运行程序从**main**函数开始
4. 支持函数的**递归调用**
5. JG，JNG，JE，JNE，JL，JNL多种跳转方式

### 实现方式

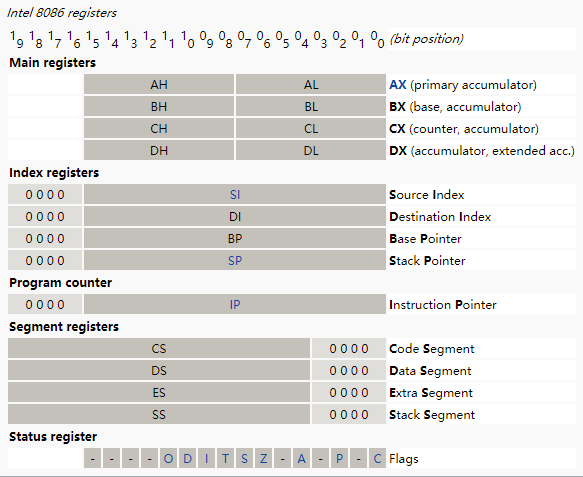
#### 主程序

实现从中间代码到汇编代码的转换

|  |  |  |
| --- | --- | --- |
| **中间代码** | **汇编代码** | **描述** |
| label\_n : | labe\_n: | 设置标签 |
| FUNC f : | f PROC | 定义函数 f |
| x= y | mov x,y | 变量赋值操作 |
| x = 1 | mov x,1 | 立即数赋值 |
| y[+-\*/]z | op y,z | 四则运算 |
| IF x [relop] y GOTO label\_n | cmp x,y  je|jl|jg label\_n | 条件跳转 |
| RETURN n | RET  f ENDP | 退出函数,返回n |
| PARAM\_IN var0 | mov ax,var0 | 函数传入实参 var0 |
| CALL f | CALL f | 调用函数 f |
| PARAM\_PREP var |  | 函数形参 |

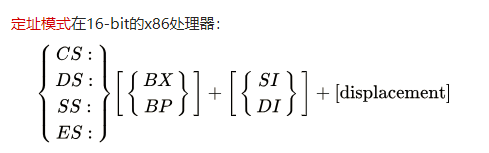
一些约定：

寄存器：



规定函数的返回值总是存在寄存器CX中，传入参数依次为AX，BX，CX，DX，暂时还没有实现传入参数大于4的情况，可以考虑使用栈或内存来传递参数。

栈是朝着低地址的方向伸长的,人为的入栈需要用减法。



初始时将DS的值设为0500H

#### 输入和打印功能

使用宏DEFINE\_SCAN\_NUM，来实现输入一个10进制数,并不复杂，读者可以自行实现

#### 函数的递归调用

函数调用时，总是**保护现场寄存器信息和内存信息**，那么不管递归调用几次，函数都不会混淆，总能找到正确的返回位置。

### 一个例子，递归方式求斐波那契数列的第n项（提供视频演示）

int fib(int n)

{

    int ans;

    if(n==1){

        ans=1;

    }else if(n==2){

        ans=1;

    }

    else{

        ans=fib(n-1)+fib(n-2);

    }

    return ans;

}

int main()

{

    int a,v;

    a=scan(); //从键盘中读入一个数

    v=fib(a);

    print(v);

    return 0;

}

中间代码：

FUNC fib:

PARAM\_IN var0

temp0 = 1

temp1 = var0 == temp0

IF temp1 != 0 GOTO label\_0

GOTO label\_1

label\_0:

temp2 = 1

var1 = temp2

GOTO label\_2

label\_1:

temp3 = 2

temp4 = var0 == temp3

IF temp4 != 0 GOTO label\_3

GOTO label\_4

label\_3:

temp5 = 1

var1 = temp5

GOTO label\_5

label\_4:

temp6 = 1

temp7 = var0 - temp6

PARAM\_PREP temp7

temp8 = CALL fib

temp9 = 2

temp10 = var0 - temp9

PARAM\_PREP temp10

temp11 = CALL fib

temp12 = temp8 + temp11

var1 = temp12

label\_5:

label\_2:

RETURN var1

FUNC main:

temp13 = CALL scan

var2 = temp13

PARAM\_PREP var2

temp14 = CALL fib

var3 = temp14

PARAM\_PREP var3

CALL print

temp15 = 0

RETURN temp15

汇编代码：

include 'emu8086.inc'

ORG 100h

MOV AX,0500H

MOV DS,AX

CALL MAIN

.EXIT

fib PROC

MOV DX,DS

ADD DX,0010h

MOV DS,DX

MOV [00H],AX

MOV AX,1

MOV [02H],AX

MOV AX,[00H]

MOV BX,[02H]

SUB AX,BX

MOV [04H],AX

JE label\_0

JMP label\_1

label\_0:

MOV AX,1

MOV [06H],AX

MOV AX,[06H]

MOV [08H],AX

JMP label\_2

label\_1:

MOV AX,2

MOV [0AH],AX

MOV AX,[00H]

MOV BX,[0AH]

SUB AX,BX

MOV [0CH],AX

JE label\_3

JMP label\_4

label\_3:

MOV AX,1

MOV [0EH],AX

MOV AX,[0EH]

MOV [08H],AX

JMP label\_5

label\_4:

MOV AX,1

MOV [10H],AX

MOV AX,[00H]

MOV BX,[10H]

SUB AX,BX

MOV [12H],AX

MOV AX,[12H]

CALL fib

MOV [14H],CX

MOV AX,2

MOV [16H],AX

MOV AX,[00H]

MOV BX,[16H]

SUB AX,BX

MOV [18H],AX

MOV AX,[18H]

CALL fib

MOV [1AH],CX

MOV AX,[14H]

MOV BX,[1AH]

ADD AX,BX

MOV [1CH],AX

MOV AX,[1CH]

MOV [08H],AX

label\_5:

label\_2:

MOV CX,[08H]

MOV DX,DS

SUB DX,0010h

MOV DS,DX

RET

fib ENDP

main PROC

CALL scan\_num

MOV [1EH],CX

MOV AX,[1EH]

MOV [20H],AX

MOV AX,[20H]

CALL fib

MOV [22H],CX

MOV AX,[22H]

MOV [24H],AX

MOV AX,[24H]

CALL print\_num

MOV AX,0

MOV [26H],AX

MOV CX,[26H]

RET

main ENDP

DEFINE\_SCAN\_NUM

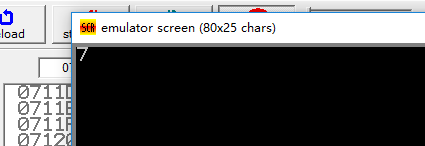
DEFINE\_PRINT\_NUM

DEFINE\_PRINT\_NUM\_UNS

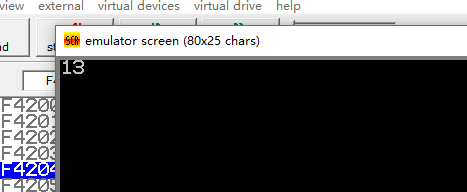
END

运行结果：

我们输入7，表示求斐波那契数列的第7项



得到结果：



程序运行正确