# 编译技术实验指导书

## 一、绪论

### （一）编译器架构

编译课程实验是计算机科学与技术专业中一门重要的课程，旨在帮助学生深入理解编译原理和技术，并掌握实际编译器的设计和实现方法。本实验的主要任务是将一个类C语言的子集——sysY语言的源代码，通过编译器的各个阶段，转换成不同的目标代码形式，包括Pcode、LLVM和MIPS代码。

编译器是一种将高级程序语言转换成底层计算机硬件能够执行的机器码的工具。编译器通常包括词法分析、语法分析、语义分析、中间代码生成、优化以及目标代码生成等多个阶段。在本实验中，学生将通过逐步实现这些阶段，深入了解编译器的工作原理和实现细节。

sysY语言是一个简化版的C语言，包括了基本的语法和语义规则，例如控制结构、数据类型、函数定义和调用等。学生需要在本实验中理解sysY语言的语法和语义，实现相应的编译器功能，将sysY源代码转换成目标代码形式。

本实验涉及到三种目标代码形式，包括Pcode、LLVM和MIPS代码。Pcode是一种虚拟机代码，通常用于编译器前端和后端之间的中间代码表示。LLVM是一种低级虚拟指令集，用于编译器优化和代码生成。MIPS是一种常见的处理器架构，相比前两种，难度要更大一些。学生需要了解这些不同的目标代码形式，掌握其语法和语义，实现相应的目标代码生成功能。课程组为大家推荐的架构为前端-中端-后端。其中前端包含了词法分析，语法分析，中端主要为语义分析以及一些可能的优化，将源语言翻译为中间代码，后端将中间代码再次翻译为目标语言。

通过本实验，你们将能够深入理解编译器的工作原理和实现细节，掌握编译器的设计和实现方法，提高自己的编程能力和软件工程能力。

### （二）实验任务概述

待确定

## 二、词法分析

### （一）词法分析概述

#### 1. 编译器输入源程序

在介绍词法分析之前，我们先了解一下输入编译器的源程序实际上是什么形式。从人的角度看，源程序可以带有各种结构，如顺序结构、分支结构等等，而且我们还可以通过空格、换行等来把一个个单词清晰地分开。但是从编译器的角度看，源程序仅仅是一个**线性的字符串**。

例如对于下面这个简单的分支结构源程序：

```
int main() {
    int var = 8+2;
    if (var != 10) {
        printf(“error!”);
    }
    else {
        printf(“correct!”);
    }
    return 0;
}
```

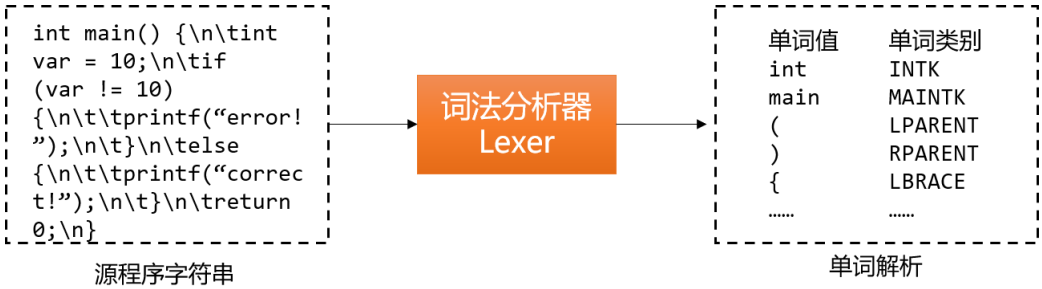
对于编译器而言，输入的是这样一个字符串：

```
int main() {\n\tint var = 8+2;\n\tif (var != 10)
{\n\t\tprintf(“error!”);\n\t}\n\telse {\n\t\tprintf(“correct!”);\n\t}\n\treturn
0;\n}
```

这个字符串是由一个个字符组成的连续序列，不具有其他复杂结构。这样一个非结构化的线性字符串对于编译器后续的分析工作是十分不利的，因此，编译器的第一步就是要把这样一个线性字符串分割成一个个单词，便于后续分析。

### 2. 词法分析器的作用

词法分析器作为编译器的第一部分，承担的任务就是**通过扫描输入的源程序字符串，将其分割成一个个单词，同时记录这些单词的类别信息。**而对于源程序中一些对编译没用的符号，如'\n'和注释等，词法分析器也会进行适当的处理（如忽视跳过，记录当前行号等）。




如图所示，经过词法分析器的解析，我们就可以从词法分析器依次获取每个单词的信息，包括单词值和单词类别，用于后续的编译。

需要注意的是，词法分析器只负责单词的划分解析，不涉及具体的语法和语义判断。

## （二）词法分析器实现思路

### 1. 词法分析器工作过程

在了解了词法分析器的作用后，我们来进一步学习词法分析器的工作过程。

一般而言，词法分析器包含以下部分：

- 源程序字符串`source`：输入编译器的源程序字符串
- 位置指针`p`：指向**下一个待解析的单词的起始字符**，初始化时指向source的第一个字符
- 单词值`token`：解析的单词值，为字符串
- 单词类别`type`：解析的单词类别，枚举类型
- 行号`line`：当前解析位置在源程序中的行号
- 数值`number`：解析的单词如果为数字（该单词本身是一个字符串），则设置number为该单词表示的数值

- `next`：词法分析器对外提供的接口，用于进行一次解析

此外，也可以根据实际需要额外添加其他部分。下面我们来看词法分析器大致的工作过程（黄色部分为本次解析的部分，灰色部分为已经解析完成的部分）。

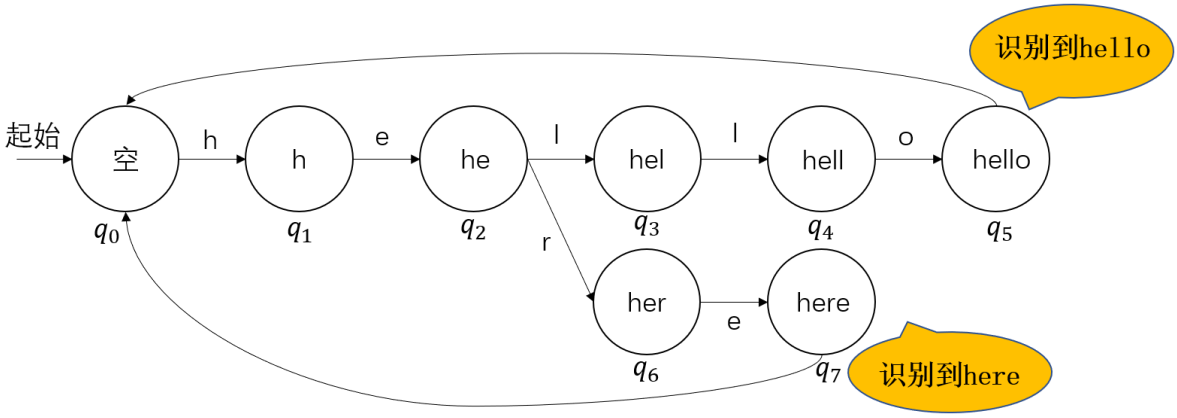

1. 初始化时，`p` **指向** `source` **的第一个字符**，`line=1`，其他部分为空。
2. 解析第一个单词"int"，并设置`token`和`type`为相应的单词值和单词类别，`p` **指向下一个待解析的单词的初始字符。**
3. **跳过空格字符**，解析第二个单词"main"，并设置`token`和`type`为相应的单词值和单词类别。
4. 经过若干步后，跳过"\n\t"并`line` **加1**，解析单词"int"，并设置`token`和`type`为相应的单词值和单词类别。
5. 经过若干步后，跳过空格字符，解析单词"10"，设置`token`和`type`为相应的单词值和单词类别，同时设置`number=10`。
6. 解析最后一个单词";"，设置`token`和`type`为相应的单词值和单词类别，解析结束。

编译器的其他部分通过每次调用`next`来解析一个单词，并通过`token, type, line, number`来获取本次解析得到的单词的相关信息。其中，最为重要的是每次调用`next`时，如何解析出一个单词（即图中的黄色部分）。下面介绍这一部分常用的方法——有限自动机。

## 2. 有限状态自动机

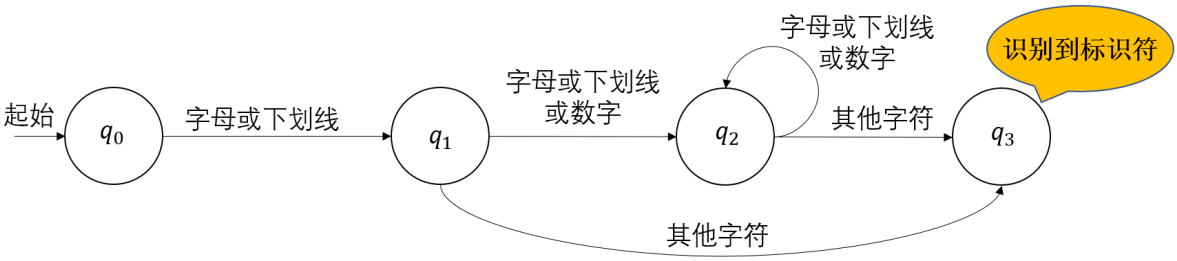
有限状态自动机FSA是处理形式化语言的重要工具，常常用于识别特定模式的字符串。有限状态自动机FSA内部包含一个当前状态$q$和转移函数$\delta$，对于一个输入的字符$a$，FSA将根据当前状态$q$和字符$a$转移到下一个状态$p$，即有$p = \delta(q, a)$。

这里用一个简单的例子来说明FSA的工作流程。




上图是一个用于识别"hello"和"here"两个单词的FSA，初始时FSA处于起始状态$q_0$，后续每读入一个字符$a$，都将根据当前状态和$a$进行一次状态转移（即沿着图中的对应边进入下一个状态），在到达$q_5$或$q_7$后，FSA就成功识别了一个"hello"或"here"单词。需要注意的是，在读入"he"时，我们不能判断当前这个单词是"hello"还是"here"，需要根据读入的下一个字符进行判断并转移到不同的状态分支（对应图中的$q_2$）。此外，这个例子也没有考虑异常情况，如读入字符'z'。

那么，FSA和我们的词法分析器有什么关系呢？不难发现，词法分析的单词中，很多都具有特定的结构：保留字和许多运算符号具有固定的字符串值，如main，break，+，&&，这些固定的串值可以通过类似上面例子的FSA识别；标识符和常数具有规定的结构，如标识符由一个字母或下划线开始，后面跟若干个字母或数字或下划线（如_a1_2b3），而常数则由数字开始，后面跟若干个数字（如123），虽然它们没有固定的串值，但是也可以通过FSA来进行识别。例如，下图是一个识别标识符的FSA。




因此，我们完全可以对每种单词都构造一个小的FSA，最后再将它们合成一个大的FSA，帮助我们的词法分析器识别所有的单词。

但是，在实际实现的过程中，为每种单词都构造一个FSA并最终将它们合并，将会产生数量巨大的中间状态和状态转移，这是十分繁琐且没有必要的。这里给出一种简单的基于贪心策略的实现方法，这种方法通过”贪心“地读取一串字符来简化许多繁琐的状态转移。

1. 每次调用next时，词法分析器先跳过当前位置的所有空白符，如果有'\n'则行号加1. 如果当前位置有注释，则跳过。
2. 如果当前字符是字母或下划线，则继续往后读取一串由字母或下划线或数字组成的字符串，直到结束或遇到不是这三者的字符，然后对于刚刚读取的这个字符串，到保留字表中进行查询，如果这个字符串属于保留字，如int，则按对应的保留字处理，否则按标识符处理。
3. 如果当前字符是数字，则继续往后读取一串由数字组成的字符串，直到结束或遇到非数字字符，然后按常数来处理刚刚读取的这个字符串。
4. 对于剩下的符号，如+,(等，只需要进行一次简单的判断即可。

事实上，可以发现前面词法分析器工作过程的例子正是按照这种方式来解析单词的。

## 3. 注意事项

注释对于后续的编译过程没有任何影响，因此应该在词法分析阶段就过滤掉。我们需要考虑的注释有两种，一种是单行注释，一种是多行注释。

```
// 这是一个单行注释
/* 这是一个
    多行注释 */
```

对于单行注释而言，一旦识别到"//"，则后面的内容全部跳过，直到遇到换行符；对于多行注释而言，一旦识别到"/*"，则到下一个"*/"中间的内容都要跳过。这里需要特别注意如下几种情况：

```
/* ***** 这是一个注释 ****123/ ****
********* 4/2 */

/* 这也是一个//注释  */

// /* 这还是一个注释 */

int a = 2048 / /* 这依然是一个注释 */ 1024; // 注意不要把除号当成注释
```

此外，我们会发现单词中有一些单词存在前缀重叠的情况，如'<'和'<='，对于这些前缀重叠的单词，我们单单读入一个符号是无法判断的，因此需要"偷看"下一个字符来判断是哪一种单词。

### 4. 词法错误

一个完整的编译器，除了能够编译出正确的目标代码，还需要能够对各种错误进行识别和检测。在词法分析阶段，我们可以完成部分词法错误的识别，当遇到下列错误时，你的编译器需要报错：

| 错误类型 | 错误类别码 | 解释 | 对应文法 |
| --- | --- | --- | --- |
| 非法符号 | a | 格式字符串中出现非法字符，报错行号为**<FormatString>**所在行数。 | <FormatString> → ""{<Char>}"" |

# 二、语法分析

## （一）语法分析概述

在上一节中，我们通过词法分析将输入的源程序解析成一个个单词。在语法分析部分，我们将对这些单词进行进一步地分析，将其建立成结构化的语法树，以便于后续的编译。

### 1. 语法树

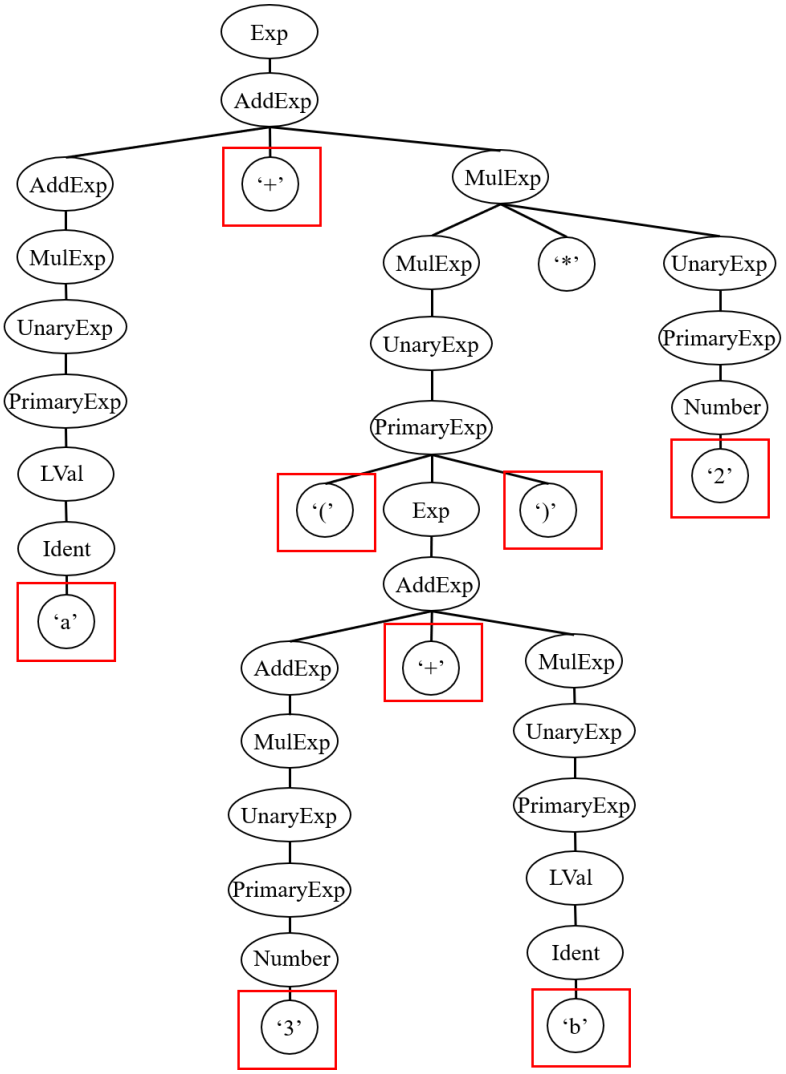
语法树是按照给定的文法，将一个句子转化为有结构层次的树结构。**树中的每个节点都代表一个语法成分，而其子节点则代表组成该语法成分的其他语法成分，根节点为开始符号。**以下面的几条文法为例，假设Exp为开始符号（可以理解成最高级的语法成分）：

```
左值表达式 LVal → Ident {'[' Exp ']'}
数值 Number → IntConst
基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number
一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
单目运算符 UnaryOp → '+' | '−' | '!'
乘除模表达式 MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp
表达式 Exp → AddExp
```

那么对于下面这个简单的句子（其中a和b为Ident）

```
a + (3 + b) * 2
```

我们则可以建立如下的语法树：




语法树展现了**从上到下的推理过程**，即我们是如何从一个语法成分推理出我们的句子（在上述例子中，从Exp推理出`a+(3+b)*2`），也就是语法树从根开始往下生长的过程；另一方面，语法树还展现了**从下到上的归约过程**，即我们是如何从一个句子不断地"合成"，最终合并成一个语法成分的（在上述例子中，从`a+(3+b)*2`最终合成Exp语法成分），也就是语法树从叶子结点向上不断合并的过程。

除此之外，不难发现，语法树中的叶子结点都是终结符（因为到终结符就无法继续拆分了），而且叶子结点从左到右刚好组成了我们输入的句子。

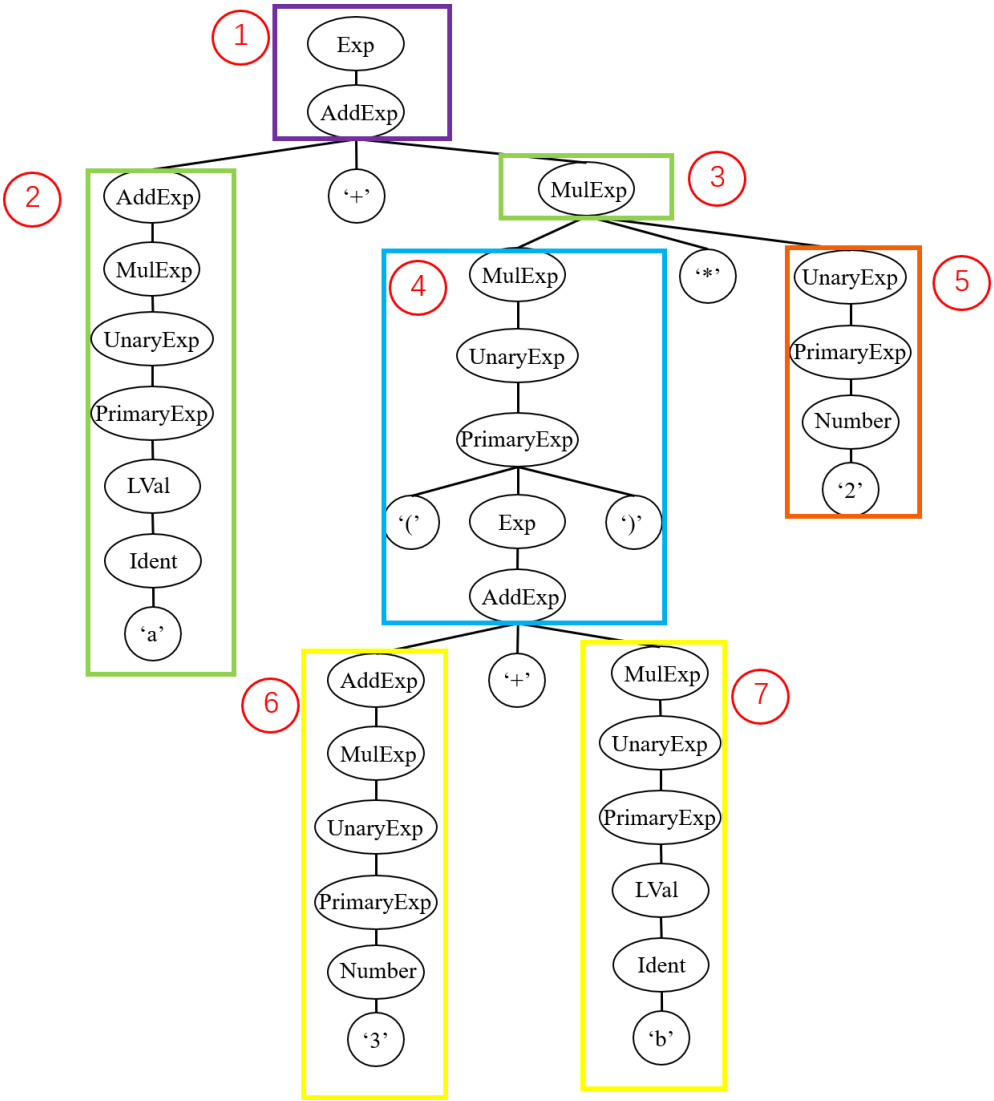
## 2. 语法分析的作用

前面我们介绍了给定文法时，一个句子的语法树是什么样的，以及语法树的一些特点。那么，**为什么我们要将一个简单的句子转化成如此复杂的语法树呢？**原因就在于，一个简单的句子是线性结构，不包含相应的语法结构信息，如果直接对线性的句子进行处理，是非常困难的。而语法树的层次化结构可以很好地表现语法结构，以及语法成分之间的关系；而树结构又可以很方便地利用**递归**等方法来进行处理，因此，将输入的源程序转化成包含语法结构的语法树，将为后续的处理带来极大的便利。

例如，对于上面例子中的表达式


```
a + (3 + b) * 2
```

对于人来说，可以很容易地确定表达式的运算顺序。但是，从计算机角度而言，要编写一个程序仅仅根据这个线性字符串来确定运算的顺序是比较困难的，尤其是当表达式更为复杂的时候。如果我们建立了语法树，则可以很轻松地利用递归等方法来进行计算。




例如，在上面的例子中，我们要求表达式Exp的值（图中序号1），那么我们只需要求出AddExp（序号2）和MulExp（序号3）的值并将它们相加即可。要求MulExp（序号3）的值，我们只需要求出MulExp（序号4）和UnaryExp（序号5）的值并将它们相乘即可。同样地，要求MulExp（序号4）的值，我们只需要求出AddExp（序号6）和MulExp（序号7）的值并将它们相乘即可。对于每种语法成分的处理，如AddExp和MulExp的计算求值，我们都可以采用递归的方法很方便地实现，而且**语法树的层次结构自然保证了计算顺序的正确性。**

事实上，就像上面介绍的表达式计算，后续的编译过程也是基于语法树，利用递归等方法来进行的。因此，**语法分析的作用就是根据词法分析器解析出来的一个个单词，生成这样一棵具有层次化结构的语法树**，方便后续的编译。

## （二）语法分析器实现思路

接下来介绍实现语法分析器（Parser）的常用方法——递归下降。

### 1. 递归下降

由于树结构可以很方便的用递归来处理（例如DFS），因此，我们同样可以采用递归的方法来生成语法树。

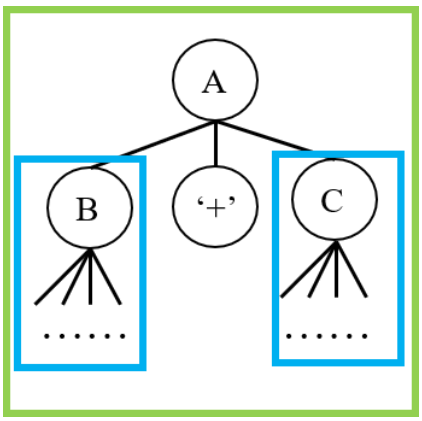
递归下降的主要思路，就是**为每个语法成分都编写一个子程序，该子程序会调用其他的子程序来解析组成该语法成分的其他语法成分，每个子程序返回的结果都是一棵对应语法成分的子树。**具体来说，例如有以下文法：

```
A -> B '+' C
```

那么，就可以为语法成分A编写一个这样的子程序（可以暂时忽略词法分析器lexer的调用，只关注其他子程序的调用顺序）：

```
parseA() {
    B = parseB(); // 调用B的子程序，解析语法成分B
    if(lexer.type != PLUS) error();
    lexer.next();
    C = parseC();   // 调用C的子程序，解析语法成分C
    return new A(B,C);  // 返回由B和C组成的语法成分A
}
```

如下图所示，解析A的子程序分别调用解析B和C的子程序，得到了两棵蓝色的子树，再将两棵子树和终结符'+'合并为绿色的子树（即语法成分A的语法树）。而**在解析A的子程序中，我们并不关心解析B和C的子程序究竟做了什么，只关注其返回的结果。**




当然，解析A的子程序也可能被其他子程序调用，它们也不关心解析A的子程序究竟做了什么。而且解析A的子程序在调用其他子程序时，可能会再次调用解析A的子程序，例如以下文法：

```
A -> B '+' C
C -> A | '1'
```

那么，解析C的子程序就有可能调用解析A的子程序，这就是递归下降中的递归思想。

**总而言之，我们只需要为文法中的每个语法成分都编写一个子程序并确保它们之间的正确调用，最后调用开始符号的解析子程序，就可以生成整棵语法树。**

## 2. 词法分析和语法分析的配合

前面我们强调了语法分析利用词法分析解析出的一个个单词来构建语法树，因此，在编写递归下降程序的过程中，需要特别注意语法分析和词法分析的配合。如果任何一个地方错误调用了词法分析，就可能对整个语法树的构建造成影响。

为了确保语法分析和词法分析的配合，我们做出如下规定供参考：

- 一个子程序在调用其他子程序前，需要调用词法分析器来预读一个单词
- 一个子程序在退出时，需要调用词法分析器来预读一个单词

有了上述规定，就可以确保：

- 刚进入一个子程序时，词法分析器已经预读好了一个单词
- 从一个子程序返回时，词法分析器已经预读好了一个单词

因此，对于前面例子中的解析A的子程序，词法分析器的调用顺序如下：

```
parseA() {
    // 进入子程序A时，词法分析器已经预读好一个单词，从而确保了进入子程序B时，词法分析器已经预读好一个单词
    B = parseB();
    // 从子程序B返回时，词法分析器已经预读好了一个单词（B后面的单词），这个单词应该是'+'，检查是否符合
    if(lexer.type != PLUS) error(); // 检查'+'
    // 进入子程序C前，预读一个单词
    lexer.next();
    C = parseC();
    // 从子程序C返回时，词法分析器已经预读好了一个单词（C后面的单词，也是A后面的单词）
    // 从而保证了子程序A退出时，词法分析器已经预读好了一个单词（A后面的单词）
    return new A(B,C);
}
```

这样我们就可以确保语法分析和词法分析的协调配合。

## 3. 多产生式

如果一个语法成分只有一条产生式，那么其解析方法就是唯一确定的。但是文法中可能存在某个语法成分有多条产生式，例如

```
语句 Stmt → LVal '=' Exp ';'
| [Exp] ';'
| Block
| 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
| 'while' '(' Cond ')' Stmt
| 'break' ';'
| 'continue' ';'
| 'return' [Exp] ';'
| LVal '=' 'getint''('')'';'
| 'printf''('FormatString{','Exp}')'';'
```

那么，**在解析Stmt的子程序中，就要考虑选择哪一条产生式进行后续的解析。**这里，我们可以考虑产生式右边的FIRST集。

假设A是某一条产生式的右部。那么A的FIRST集是一个终结符的集合，它由A能推理出的所有句子的第一个终结符组成。具体来说，对于产生式

```
Stmt -> 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
```

很明显，该产生式的右部所能推出的任何句子的第一个终结符一定是'if'，因此，其FIRST集中只包含一个终结符'if'。

对于产生式

```
Stmt -> Block
Block → '{' { BlockItem } '}'
```

则该Stmt产生式的右部所能推出的任何句子的第一个终结符一定是'{'，其FIRST集只包含一个终结符'{'。

对于产生式

```
Stmt -> Exp
Exp → AddExp // EXP的FIRST={(,Number,Ident,+,-,!}
AddExp → MulExp | AddExp ('+' | '−') MulExp // AddExp的FIRST=
{(,Number,Ident,+,-,!}
MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp // MulExp的FIRST=
{(,Number,Ident,+,-,!}
UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp //
UnaryExp的FIRST={(,Number,Ident,+,-,!} 其中，(,Number,Ident属于PrimaryExp的FIRST，因
此也属于UnaryExp的FIRST
UnaryOp → '+' | '−' | '!'
PrimaryExp → '(' Exp ')' | LVal | Number // PrimaryExp的FIRST={(,Number,Ident},
其中Ident属于LVal的FIRST，因此也属于PrimaryExp的FIRST
LVal → Ident {'[' Exp ']'}  // LVal的FIRST={Ident}
```

按照**自底向上**的顺序，根据注释，我们可以求出该Stmt产生式右部Exp的FIRST集为$\{+,-,!,(,Ident,Number\}$（这里将Ident和Number视为终结符，因为它们是通过词法分析直接得到的单词）。

求出Stmt每条产生式的右部的FIRST集后，我们就可以根据这些FIRST集来编写解析Stmt的子程序。

```
Stmt -> 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
```

例如，由于终结符'if'只在上面这条产生式的FIRST集中出现，因此，如果当前词法分析的单词为'if'，则只能利用这条产生式来进行解析。同样地，'while'和'break'等也是如此，据此，可以编写出下面部分的子程序：

```
parseStmt() {
    if(lexer.type == IFTK) {
        // 用 Stmt -> 'if' '(' Cond ')' Stmt [ 'else' Stmt ] 解析
    }
    else if (lexer.type == WHILETK) {
        // 用 Stmt -> 'while' '(' Cond ')' Stmt 解析
    }
    ......
}
```

需要特别注意Stmt的这几条产生式

```
 Stmt → LVal '=' Exp ';'    // FIRST={Ident}
| LVal '=' 'getint''('')'';'    // FIRST={Ident}
| [Exp] ';'     // FIRST={(,Number,Ident,+,-,!}
```

其产生式右部的FIRST集存在交集$\{Ident\}$，因此，**如果当前词法分析的单词为Ident，我们就不能直接判断选择哪一条产生式进行解析。**

这里提供一种参考思路。考虑到Exp可以推理出LVal（Exp -> AddExp -> MulExp -> UnaryExp -> PrimaryExp -> LVal），因此我们**可以用Exp的解析方法来解析LVal。**如果当前单词为Ident，则

1. 首先利用调用Exp的子程序来解析出语法成分Exp，判断下一个单词是';'还是'='，如果是';'，则按第三条产生式处理，完成Stmt解析，否则转第2步，从前两条产生式中选择一条解析，.

2. 从Exp提取出LVal（该Exp一定由唯一的LVal组成），继续判断下一个单词是不是'getint'，如果是则按第二条产生式处理，否则按第一条产生式处理，完成Stmt解析

注意，如果采用Exp的解析方法来解析LVal，你需要确保输出的结果正确（即输出应该是按照LVal的解析方法来解析LVal）。

对于除Stmt以外的其他语法成分的多产生式，也需要仔细考虑产生式右部的FIRST集，以编写正确的解析子程序。

## 4. 左递归文法

文法中存在左递归文法，如果直接据此编写递归下降子程序，将会导致无限递归，最终栈溢出程序崩溃。

```
AddExp → MulExp | AddExp ('+' | '−') MulExp
MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
```

我们需要通过改写文法来解决左递归问题。一般来说，有两种改写方法：

1. 将左递归改为右递归。我们可以将上述两条文法改写为：

   ```
   AddExp → MulExp | MulExp ('+' | '−') AddExp
   MulExp → UnaryExp | UnaryExp ('*' | '/' | '%') MulExp
   ```

   以解析AddExp为例，先调用MulExp的解析子程序，然后判断后面的单词是否是'+'或'-'，如果是，再递归调用解析AddExp的子程序。

2. 改写为BNF范式。通过分析结构，不难发现AddExp本质上是由若干个MulExp组成，MulExp本质上是由若干个UnaryExp组成，因此可以将上述两条文法改写为

   ```
   AddExp → MulExp {('+' | '−') MulExp}
   MulExp → UnaryExp {('*' | '/' | '%') UnaryExp}
   ```

   其中，{}表示其中的语法成分可以出现0到若干次。

   以解析AddExp为例，先调用MulExp的解析子程序，然后判断后面的单词是否是'+'或'-'，如果是，则再次调用MulExp的解析子程序，直至解析完MulExp后的单词不是'+'也不是'-'。

需要特别注意的是，尽管编写程序时按照改写后的文法编写，但是需要确保输出的解析次序和原来的文法一致。以第二种改写方法为例，可以在每次解析`('+' | '−') MulExp`之前，先将之前已经解析出的若干个MulExp合成一个AddExp，输出一次<AddExp>。

## 5. 语法错误

在语法分析阶段，需要在检测词法错误的基础上，进一步检测语法错误，包括以下几种类型。

| 错误类型 | 错误类别码 | 解释 | 对应文法 |
| --- | --- | --- | --- |
| 缺少分号 | i | 报错行号为分号**前一个非终结符**所在行号。 | <Stmt>,<ConstDecl>及<VarDecl>中的';' |
| 缺少右小括号 | j | 报错行号为右小括号**前一个非终结符**所在行号。 | 函数调用(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>中的')' |
| 缺少右中括号']' | k | 报错行号为右中括号**前一个非终结符**所在行号. | 数组定义(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)中的']' |

# 三、中间代码生成

## （一）四元式

在完成词法分析、语法分析和错误处理后，我们需要通过语义分析，将源程序翻译为中间代码，然后便是通过分析中间代码，将其翻译为最终的目标代码。一个好的中间代码，可以简化编译器的实现，便于调试和代码的优化。课程组给出了三种中间代码的实现的渠道，分别是LLVM，PECODE和自己设计的中间代码。下面，我主要就第三点，给出一些自己设计的中间代码（四元式）的参考格式，同学们完全可以基于自己的想法来自由发挥。只要设计的中间代码可以准确无误地表达源程序的逻辑，并且便于优化，就是好的中间代码。

### 四元式

此处我们介绍一种中间代码的形式：四元式。四元式是一种形如(op, arg1, arg2, result)的指令格式，其中op是运算符，arg1和arg2是操作数，result是运算结果。例如，赋值语句 x = y + z 可以表示为：

```
+, y,  z, t1
=, t1,  ,x
```

其中，+ 表示加法操作，= 表示赋值操作，y 和 z 是加法操作的两个操作数，t1 是临时变量，用于存储加法的结果，x 是赋值操作的结果。这段四元式的意思，是先把y和z的数值加和，赋值给t1，然后再把t1赋值给x。需要注意的是，在生成中间代码之前，需要为每个变量分配一个临时变量。临时变量是在中间代码生成过程中创建的，用于存储中间结果。临时变量可以采用 t1、t2、t3 等命名方式。

对于实验要求的文法来说，下面简单分析一下在生成中间代码阶段的各类语句（可自行设计）：

### 主函数

主函数和block块都是编译器中生成中间代码的基本单元。

主函数是整个程序的入口，通常包含全局变量的定义、函数的声明和调用以及其他一些初始化操作。在主函数中，编译器需要生成全局变量的初始化代码、函数的调用代码和程序结束时的清理代码等。Block块是一个语句块，通常由一组语句组成，可以是一个函数体、一个循环体或者一个条件语句块等。在block块中，编译器需要生成相应的控制流程代码，如条件分支和循环控制等。

对于主函数和block数据块，均可使用begin和end来描述其作用范围，大致四元式框架可描述如下：

```
main_begin,_,_,_

block_begin,_,_,_
xxxxxxx
block_end,_,_,_

block_begin,_,_,_
xxxxxxx
block_end,_,_,_

main_end,_,_,_
```

对于每个主函数与block块，均需要维护一个对应的符号表，有关符号表相应的知识我们会在接下来的指导书中讲到。

## 赋值语句

生成赋值语句的过程比较简单。假设要生成赋值语句 x = y + z，可以遵循以下步骤：

- 为加法操作分配一个临时变量 t1。
- 生成加法四元式 + y z t1。
- 生成赋值四元式 = t1 x。

对于数组元素，我们也有相应的表示方法，例如 a = num[0] 可以用如下步骤表示：

```
[], num, 0, t1
=, t1, , a
```

对于二维数组，可将二维数组展开，当作一维数组处理即可。

## 条件语句

条件语句需要将条件表达式翻译成中间代码。假设条件语句为 if (x > y) then z = x + y，可以遵循以下步骤：

- 为比较操作符 >的结果分配一个临时变量 t1。
- 生成比较四元式 > x y t1（x大于y给t1赋值为1，否则赋值为0）。
- 生成条件跳转四元式 jnz t1 L（t1不为0则跳转至L处，否则继续运行）。
- 生成无条件跳转四元式 jmp M（不运行下方语句块）。
- 生成标记 L。
- 生成赋值四元式 = x + y t2。
- 生成赋值四元式 = z t2。
- 生成标记 M。

如果由if语句中存在else和elif，处理方法与上述一致，具体请同学们自行思考。

## 循环语句

循环语句需要使用跳转指令和标记来实现循环控制。假设循环语句为 while (x < y) do x = x + 1，可以遵循以下步骤：

- 生成标记 L。
- 为比较操作符 < 分配一个临时变量 t1。
- 生成比较四元式 < x y t1。
- 生成条件跳转四元式 beq t1 0 M(t1为0则跳转至M处，否则继续运行)。
- 生成赋值四元式 + x  1 t2。

- 生成赋值四元式 = t2 x。
- 生成无条件跳转至L。
- 生成标记M。

## 函数定义

函数定义也是编译器中生成中间代码的基本单元之一。函数定义描述了一个函数的参数列表、返回类型和函数体。在函数定义中，编译器需要生成函数的入口代码、参数初始化代码、局部变量的声明和初始化、函数体中的控制流程代码和函数的返回值等。我们可以模仿block和主函数的处理方法进行函数定义，例如对于如下代码

```
int f(int a,int b){
    xxxx
    return xxx;
}
```

我们可以这么定义

```
func_begin,f,_,_
para_int,a,_,_
para_int,b,_,_
xxxxxxxx
ret xxxxxx
func_end,f,_,_
```

在上述示例中，`func_begin`是函数的入口标签，`ret`指令用于将计算结果作为返回值返回给调用者，`func_end`是函数的结束标签。函数体中的局部变量可以使用临时变量来实现。

此外，函数定义中还需要维护函数的符号表，包括参数和局部变量的声明。函数调用时需要将参数传递给被调用函数，并根据函数定义的参数和返回值类型生成相应的中间代码。

## 函数调用

- 在函数调用之前，需要将函数参数压入栈中。对于每个参数，可以生成一条对应的四元式，将参数值存储到一个临时变量中，并将该临时变量的地址压入栈中。例如，对于调用函数f(x, y)，可以生成以下四元式：

  ```
  =, x, _, t1    // 将x的值存储到t1中
  parm, t1, _, _ // 将t1的地址压入栈中
  =, y, _, t2    // 将y的值存储到t2中
  parm, t2, _, _ // 将t2的地址压入栈中
  ```

- 在所有参数都被压入栈之后，可以生成一个函数调用四元式，将控制权转移给被调用函数，例如，对于调用函数f(x, y)，可以生成以下四元式：

  ```
  call, f, _, _  // 调用函数f
  ```

- 在函数调用结束之后，需要将函数的返回值从栈中取出，并将其存储到一个临时变量中，并需要清空参数栈。

当然，这里给出的代码示例只包含了一些比较基础的语法，并没有覆盖文法中可能含有的所有情况。对于更复杂的情况，还需要同学自行思考中间代码的设计方案。

以上给出的四元式设计和中间代码规范格式都仅仅是中间代码参考架构，同学在设计过程中可以结合自己的想法对中间代码格式进行调整。中间代码的设计将直接影响到目标代码生成，所以同学在设计中间代码时一定要考虑到“如何将中间代码翻译成目标代码”，最好能具体到一些细节，以减少目标代码生成阶段中对中间代码的重构。若目标代码为MIPS指令集等汇编语言，中间代码需要尽可能向着汇编语言贴近，有些甚至可能就直接用汇编语言代替。

## 读入和输出

`getint()`非常简单，如遇到`b=getint()`的输入，直接使用以下四元式即可表示：

```
getint,_,_b
```

对于`printf`，可以将需要输出的字符串通过%d分割成多个字符串，对于每个%d，生成print_int中间代码，对于每一个字符串，生成print_str中间代码。

例如，对于如下代码：

```
printf("%d hello",x);
```

我可以用如下四元式表示：

```
print_int,a[1],_,_
print_string,hello,_,_
```

## 符号表设计

在编译器中，符号表是一个重要的数据结构，用于存储程序中的变量、常量、函数等信息，为编译器的各个模块提供了必要的信息。在中间代码生成过程中，符号表的作用是维护程序中变量、函数等信息，并为中间代码生成器提供必要的信息。

符号表通常包含以下信息：

- 符号名称：变量、函数等的名称；
- 符号类型：变量的数据类型、函数的返回值类型等；
- 存储地址：变量在内存中的地址、函数入口地址等；
- 作用域：变量、函数等的作用域；
- 其他属性：变量、函数等的其他属性，如是否是常量、是否被定义过等。

在中间代码生成过程中，维护符号表的过程可以分为以下几个步骤：

1. 符号表的建立和初始化：在编译器的初始化阶段，需要建立符号表并初始化，为后续的代码生成过程做好准备。符号表可以用一个哈希表或二叉查找树等数据结构来实现，以实现快速的查找和插入操作。
2. 插入符号信息：在编译过程中，当遇到变量或函数等符号时，需要将其信息插入符号表中。如果符号已经存在，则需要给出错误信息。插入符号信息时，需要包含符号名称、符号类型、存储地址、作用域等信息。
3. 查找符号信息：在代码生成过程中，当需要访问符号信息时，需要在符号表中查找相应的符号信息。查找符号信息时，需要提供符号名称和作用域等信息。如果符号不存在，则需要给出错误信息。
4. 更新符号信息：在程序的执行过程中，符号信息可能会发生变化。例如，当变量被赋值时，需要更新符号表中相应变量的值。更新符号信息时，需要提供符号名称和作用域等信息，并更新相应的属性信息。
5. 符号表的销毁：在编译器执行结束时，需要销毁符号表。销毁符号表时，需要释放所有动态分配的内存，并将符号表指针置为空。

维护符号表是编译器中的重要工作之一，它为编译器的各个模块提供必要的信息，并为代码生成过程提供了必要的支持。通过正确地维护符号表，可以保证编译器生成的中间代码的正确性和有效性。

# （二）llvm

从这一部分开始，将正式开始进入**代码生成**阶段。课程组给出了三种目标码，分别是生成到 `PCode` ，`LLVM IR` ，以及 `MIPS` 。写MIPS需要额外进行**代码优化**的操作。这里建议有时间的同学们可以去提前调研一下这几种代码再进行选择。由于理论课上，以及编译原理的教材上主要介绍了**四元式**，且在课本最后也介绍了PCode，所以采用PCode作为目标码的同学可以主要参考**编译原理教材**。

`LLVM` 可能看上去上手比较困难，毕竟相信大部分同学是第一次接触，而在往年的编译原理课程中，LLVM的代码生成是软件学院的课程要求，指导书也是针对往届软件学院的编译原理实验。在2022年与计算机学院合并之后，课程组虽然也添加了LLVM的代码生成通道，但是由于课程合并后，例如**文法，实验过程，实现要求**等的不同，课程组同学在去年实验中收到了许许多多同届同学关于LLVM的问题，包括**看不懂指导书，无从下手**等问题，所以在今年的指导书中将作出以下改进：

- 根据今年的实验顺序，**重新编排**每一个小实验部分的顺序，使得同学们在实验过程中更加顺畅。
- 对每一个部分进行**相关的说明**，帮助同学们更好地理解LLVM的代码生成过程。
- 会在指导书的每一个章节结束给出一些相对**较强的测试样例**，方便同学们做一个部分就测试一个部分。
- 由于LLVM本身就是一种很优秀的中间代码，所以对于想最终生成到MIPS的同学，今年的指导书中将新增LLVM的**中端优化部分**，帮助同学们更方便地从LLVM生成MIPS代码。

## 1. 简单介绍

### （1）LLVM是什么

`LLVM` 最早叫底层虚拟机 (Low Level Virtual Machine)，最初是伊利诺伊大学的一个研究项目，目的是提供一种现代的、基于SSA的编译策略，能够支持任意编程语言的静态和动态编译。从那时起，LLVM已经发展成为一个由多个子项目组成的伞式项目，其中许多子项目被各种各样的商业和开源项目用于生产，并被广泛用于学术研究。

现在，LLVM被用作实现各种静态和运行时编译语言的通用基础设施（例如，GCC、Java、.NET、Python、Ruby、Scheme、Haskell、D以及无数鲜为人知的语言所支持的语言族）。它还取代了各种特殊用途的编译器，如苹果OpenGL堆栈中的运行时专用化引擎和Adobe After Effects产品中的图像处理库。LLVM还被用于创建各种各样的新产品，其中最著名的可能是OpenCL GPU编程语言。

> 一些参考资料：
>
> - https://aosabook.org/en/v1/llvm.html#footnote-1
> - https://llvm.org/

### （2）三端设计

传统静态编译器，例如大多数C语言的编译器，最主流的设计是**三端设计**，其主要组件是前端、优化器和后端。前端解析源代码，检查其错误，并构建特定语言的**抽象语法树** `AST` (Abstract Syntax Tree)来表示输入代码。AST可以选择转换为新的目标码进行优化，优化器和后端在代码上运行。

`优化器` 的作用是**增加代码的运行效率**，例如消除冗余计算。 `后端` ，也即代码生成器，负责将代码**映射到目标指令集**，其常见部分包括指令选择、寄存器分配和指令调度。

当编译器需要支持**多种源语言或目标体系结构**时，使用这种设计最重要的优点就是，如果编译器在其优化器中使用公共代码表示，那么可以为任何可以编译到它的语言编写前端，也可以为任何能够从它编译的目标编写后端，如图 0-1 所示。

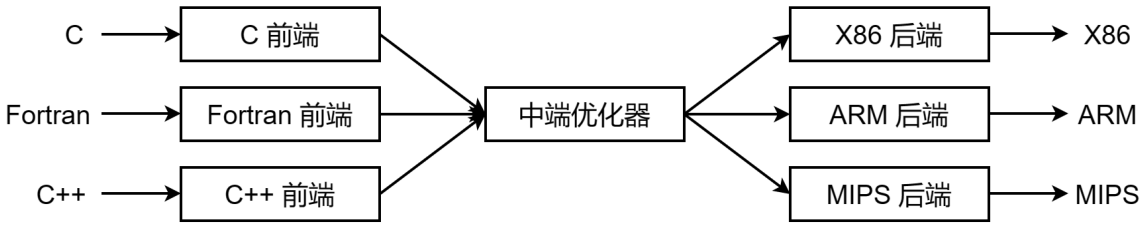



**图 0-1 三端设计示意图**

官网对这张设计图的描述只有一个单词 **Retargetablity**，译为**可重定向性**或**可移植性**。通俗理解，如果需要移植编译器以支持新的源语言，只需要实现一个新的前端，但现有的优化器和后端可以重用。如果不将这些部分分开，实现一种新的源语言将需要从头开始，因此，不难发现，支持 $N$ 个目标和 $M$ 种源语言需要 $N \times M$ 个编译器，而采用三端设计后，中端优化器可以复用，所以只需要 $N + M$ 个编译器。例如同学们熟悉的Java中的 `JIT`， `GCC` 都是采用这种设计。

`IR` (Intermediate Representation) 的翻译即为中间表示，在基于LLVM的编译器中，前端负责解析、验证和诊断输入代码中的错误，然后将解析的代码转换为LLVM IR，中端（此处即LLVM优化器）对LLVM IR进行优化，后端则负责将LLVM IR转换为目标语言。

### (3) 工具介绍

> 这一节应该配合下一节一起使用，但是下一节需要用到这些工具，故先放前面写了。

课程实验目标是将C语言的程序生成为LLVM IR的中间代码，尽管有指导书，但不可避免地，同学们还会遇到很多不会的情况。所以这里给出一个能够自己进行代码生成测试的工具介绍，帮助大家更方便地测试和完成实验。

这里着重介绍 `Ubuntu` (20.04或更新) 的下载与操作，一是方便，二是同学们或多或少有该系统的Vmware或者云服务器等等。如果真的没有，也可以在Windows或MacOS上直接装。MacOS和Windows安装Clang和LLVM的方法请自行搜索。

首先安装 `LLVM` 和 `Clang`

```
$ sudo apt-get install llvm
$ sudo apt-get install clang
```

安装完成后，输入指令查看版本。如果出现版本信息则说明安装成功。

```
$ clang -v
$ lli --version
```

> **注意：** 请务必保证llvm版本至少是**10.0.0 及以上**，否则**会影响正确性！**

如果使用apt无法安装，则将下列代码加入到 `/etc/apt/sources.list` 文件中

```
deb http://apt.llvm.org/focal/ llvm-toolchain-focal-10 main
deb-src http://apt.llvm.org/focal/ llvm-toolchain-focal-10 main
```

然后在终端执行

```
wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add -
apt-get install clang-10 lldb-10 lld-10
```

`MacOS` 上的安装也稍微提一嘴，需要安装XCode或XCode Command Line Tools, 其默认自带Clang

```
xcode-select --install
brew install llvm
```

安装完成后，需要添加LLVM到$PATH

```
echo 'export PATH="/usr/local/opt/llvm/bin:$PATH"' >> ~/.bash_profile
```

这时候可以仿照之前查看版本的方法，如果显示版本号则证明安装成功。

`Clang` 是 LLVM 项目中 C/C++ 语言的前端，其用法与 GCC 基本相同。
`lli` 会解释.bc 和 .ll 程序。

具体如何使用上述工具链，将在下一章介绍。

**(4) LLVM IR示例**

LLVM IR 具有三种表示形式，一种是在**内存中**的数据结构格式，一种是在磁盘二进制 **位码(bitcode)** 格式 `.bc` ，一种是**文本格式** `.ll` 。生成目标代码为LLVM IR的同学要求输出的是 `.ll` 形式的 LLVM IR。

作为一门全新的语言，与其讲过于理论的语法，不如直接看一个实例来得直观，也方便大家快速入门。

例如，给出源程序 `main.c` 如下

```
int a=1;
int add(int x,int y){
    return x+y;
}
int main(){
    int b=2;
    return add(a,b);
}
```

现在，倘若想知道其对应的LLVM IR长什么样，就可以用到Clang工具。下面是一些常用指令

```
$ clang main.c -o main # 生成可执行文件
$ clang -ccc-print-phases main.c # 查看编译的过程
$ clang -E -Xclang -dump-tokens main.c # 生成 tokens
$ clang -fsyntax-only -Xclang -ast-dump main.c # 生成语法树
$ clang -S -emit-llvm main.c -o main.ll -O0 # 生成 llvm ir (不开优化)
$ clang -S main.c -o main.s # 生成汇编
$ clang -c main.c -o main.o # 生成目标文件
```

输入 `clang -S -emit-llvm main.c -o main.ll` 后，会在同目录下生成一个 `main.ll` 的文件。在LLVM中，注释以';'打头。

```
; ModuleID = 'main.c'
source_filename = "main.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-
n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"
```

```
; 从下一行开始，是实验需要生成的部分，注释不要求生成。
@a = dso_local global i32 1, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add i32 %5, %6
  ret i32 %7
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 2, i32* %2, align 4
  %3 = load i32, i32* @a, align 4
  %4 = load i32, i32* %2, align 4
  %5 = call i32 @add(i32 %3, i32 %4)
  ret i32 %5
}

; 实验要求生成的代码到上一行即可

attributes #0 = { noinline nounwind optnone uwtable ...}
; ...是手动改的，因为后面一串太长了

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

用`lli main.ll`解释执行生成的 .ll 文件。如果一切正常，输入`echo $?`查看上一条指令的返回值。

在本次实验中，同学们会用到一些库函数。使用的时候请将`libsysy.c`和`libsysy.h`放在同一目录下，对使用到了库函数的源程序进行编译时，需要用到如下指令：

```
# 1. 分别导出 libsysy 和 main.c 对应的 .ll 文件
$ clang -emit-llvm -S libsysy.c -o lib.ll
$ clang -emit-llvm -S main.c -o main.ll

# 2. 使用 llvm-link 将两个文件链接，生成新的 IR 文件
$ llvm-link main.ll lib.ll -S -o out.ll

# 3. 用 lli 解释运行
$ lli out.ll
```

粗略一看，LLVM IR很长很麻烦，但仔细一看，在实验需要生成的代码部分，像是一种特殊的三元式。事实上，LLVM IR使用的是**三地址码**。下面对上述代码进行简要注释。

- Module ID：指明 `Module` 的标识
- source_filename：表明该Module是从什么文件编译得到的。如果是通过链接得到的，此处会显示 `llvm-link`
- target datalayout 和 target triple 是程序标签属性说明，和硬件/系统有关。其各个部分说明如下图所示。

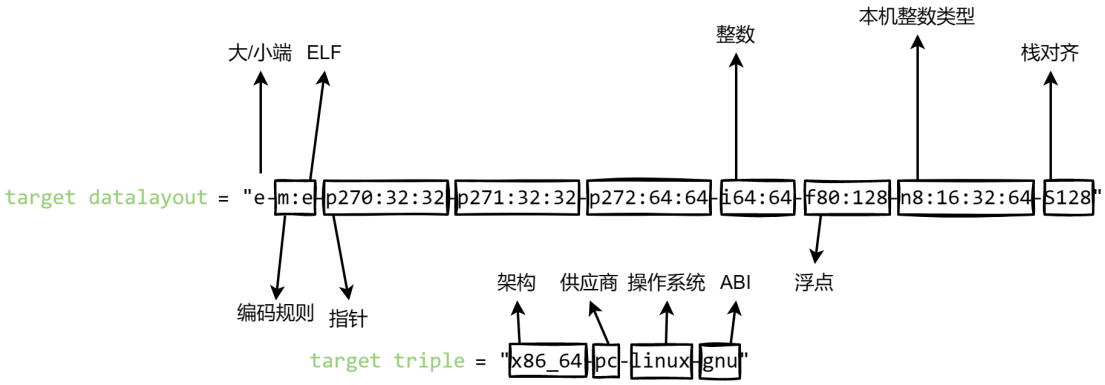



**图 0-2 target解释**

- `@a = dso_local global i32 1, align 4`：全局变量，名称是a，类型是i32，初始值是1，对齐方式是4字节。dso_local 表明该变量会在同一个链接单元内解析符号。
- `define dso_local i32 @add(i32 %0, i32 %1) #0`：函数定义。其中第一个i32是返回值类型，%add是函数名；第二个和第三个i32是形参类型，%0，%1是形参名。

> llvm中的标识符分为两种类型：全局的和局部的。全局的标识符包括函数名和全局变量，会加一个`@`前缀，局部的标识符会加一个`%`前缀。

- #0指出了函数的`attribute group`。在文件的最后，也能找到对应的attributes #0。因为attribute group可能很包含很多attribute且复用到多个函数，所以IR使用attribute group ID(即#0)的形式指明函数的attribute，这样既简洁又清晰。
- 而在大括号中间的函数体，是由一系列 `BasicBlock` 组成的。每个BasicBlock都有一个**label**，label使得该BasicBlock有一个符号表的入口点，其以terminator instruction(ret、br等)结尾的。每个BasicBlock由一系列 `Instruction` 组成。Instruction是LLVM IR的基本指令。
- %7 = add i32 %5, %6：随便拿上面一条指令来说，%7是一个临时寄存器，是Instruction的实例，它的操作数里面有两个值，一个是%5，一个是%6。%5和%6也是临时寄存器，即前两条Instruction的实例。

下面给出一个Module的主要架构，可以发现，LLVM中几乎所有的结构都可以认为是一个 `Value` ，结构与结构之间的Value传递可以简单理解为继承属性和综合属性。而 `User` 类和 `Use` 类则是LLVM中的重要概念，简单理解就是，User类存储使用Value的列表，而Use类存储Value和User的使用关系，这可以让User和Value快速找到对方。

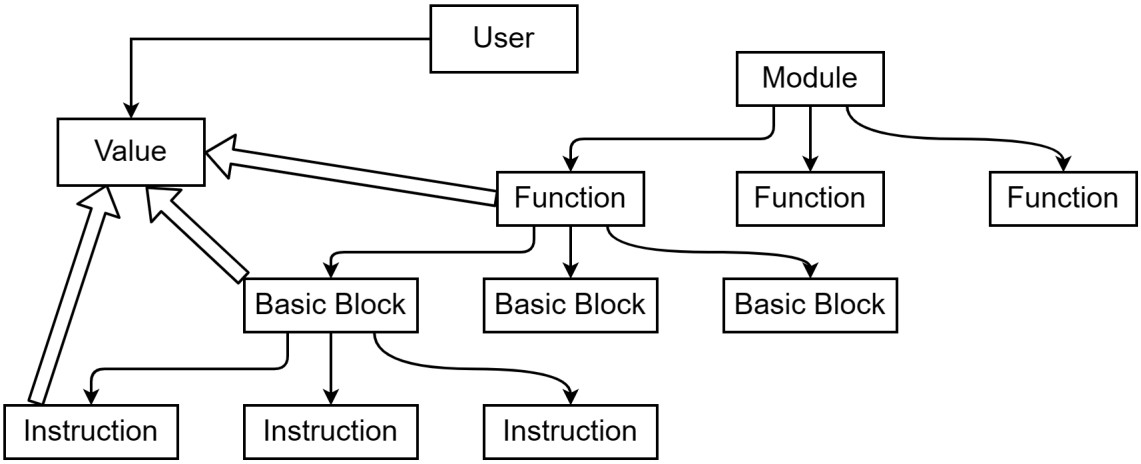



**图 0-3 LLVM架构简图**

这其中架构的设计，是为了方便LLVM的优化和分析，然后根据优化过后的Module生成后端代码。同学们可以根据自己的需要自行设计数据类型。但如果只是想生成到LLVM的同学，这部分内容其实没有那么重要，可以直接面向AST生成代码。

对于一些常用的Instructions，下面给出示例。对于一些没有给出的，可以参考LLVM IR指令集。

| llvm ir | 使用方法 | 简介 |
| --- | --- | --- |
| add | `<result> = add <ty> <op1>, <op2>` | / |
| sub | `<result> = sub <ty> <op1>, <op2>` | / |
| mul | `<result> = mul <ty> <op1>, <op2>` | / |
| sdiv | `<result> = sdiv <ty> <op1>, <op2>` | 有符号除法 |
| icmp | `<result> = icmp <cond> <ty> <op1>, <op2>` | 比较指令 |
| and | `<result> = and <ty> <op1>, <op2>` | 与 |
| or | `<result> = or <ty> <op1>, <op2>` | 或 |
| call | `<result> = call [ret attrs] <ty> <fnptrval>(<function args>)` | 函数调用 |
| alloca | `<result> = alloca <type>` | 分配内存 |
| load | `<result> = load <ty>, <ty>* <pointer>` | 读取内存 |
| store | `store <ty> <value>, <ty>* <pointer>` | 写内存 |
| getelementptr | `<result> = getelementptr <ty>, * {, [inrange] <ty> <idx>}*` `<result> = getelementptr inbounds <ty>, <ty>* <ptrval>{, [inrange] <ty> <idx>}*` | 计算目标元素的位置（这一章会单独详细说明） |
| phi | `<result> = phi [fast-math-flags] <ty> [ <val0>, <label0>], ...` | / |
| zext..to | `<result> = zext <ty> <value> to <ty2>` | 将 `ty` 的 `value` 的type扩充为 `ty2` |
| trunc..to | `<result> = trunc <ty> <value> to <ty2>` | 将 `ty` 的 `value` 的type缩减为 `ty2` |
| br | `br i1 <cond>, label <iftrue>, label <iffalse>` `br label <dest>` | 改变控制流 |
| ret | `ret <type> <value>`，`ret void` | 退出当前函数，并返回值 |

**一些说明**

- 如果目标生成到LLVM语言的同学请注意，clang 默认生成的虚拟寄存器是**按数字顺序**命名的，LLVM 限制了所有数字命名的虚拟寄存器必须严格地**从 0 开始递增**，且每个函数参数和基本块都会占用一个编号。如果你不能确定怎样用数字命名虚拟寄存器，请使用**字符串命名**虚拟寄存器。
- 由于本指导书的大量测试样例都是LLVM IR的代码，所以指导书主要讲述通过 `AST` 生成对应的代码。想通过LLVM IR的同学可以根据对应的 `Module` 结构自行存储数据，然后根据Module生成对应的LLVM IR代码自测。

## 2. 主函数与常量表达式

### （1）主函数

首先从最基本的开始，即只包含return语句的主函数（或没有参数的函数）。主要包含文法包括

```
CompUnit    → MainFuncDef
MainFuncDef → 'int' 'main' '(' ')' Block
Block       → '{' { BlockItem } '}'
BlockItem   → Stmt
Stmt        → 'return' Exp ';'
Exp         → AddExp
AddExp      → MulExp
MulExp      → UnaryExp
UnaryExp    → PrimaryExp
PrimaryExp  → Number
```

对于一个无参的函数，首先需要从AST获取函数的名称，返回值类型。然后分析函数体的Block。Block中的Stmt可以是return语句，也可以是其他语句，但是这里只考虑return语句。return语句中的Number在现在默认是**常数。**
所以对于一个代码生成器，同学们需要实现的功能有：

- 遍历**AST**，遍历到函数时，获取函数的**名称**、**返回值类型**
- 遍历到**BlockItem**内的**Stmt**时，如果是**return**语句，生成对应的**指令**

### （2）常量表达式

新增内容有

```
Stmt       → 'return' [Exp] ';'
Exp        → AddExp
AddExp     → MulExp | AddExp ('+' | '−') MulExp
MulExp     → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
UnaryExp   → PrimaryExp | UnaryOp UnaryExp
PrimaryExp → '(' Exp ')' | Number
UnaryOp    → '+' | '−'
```

对于常量表达式，这里只包含常数的四则运算，正负号操作。这时候就需要用到之前的Value思想。举个例子，对于 `1+2+3*4`，根据文法生成的AST样式如下

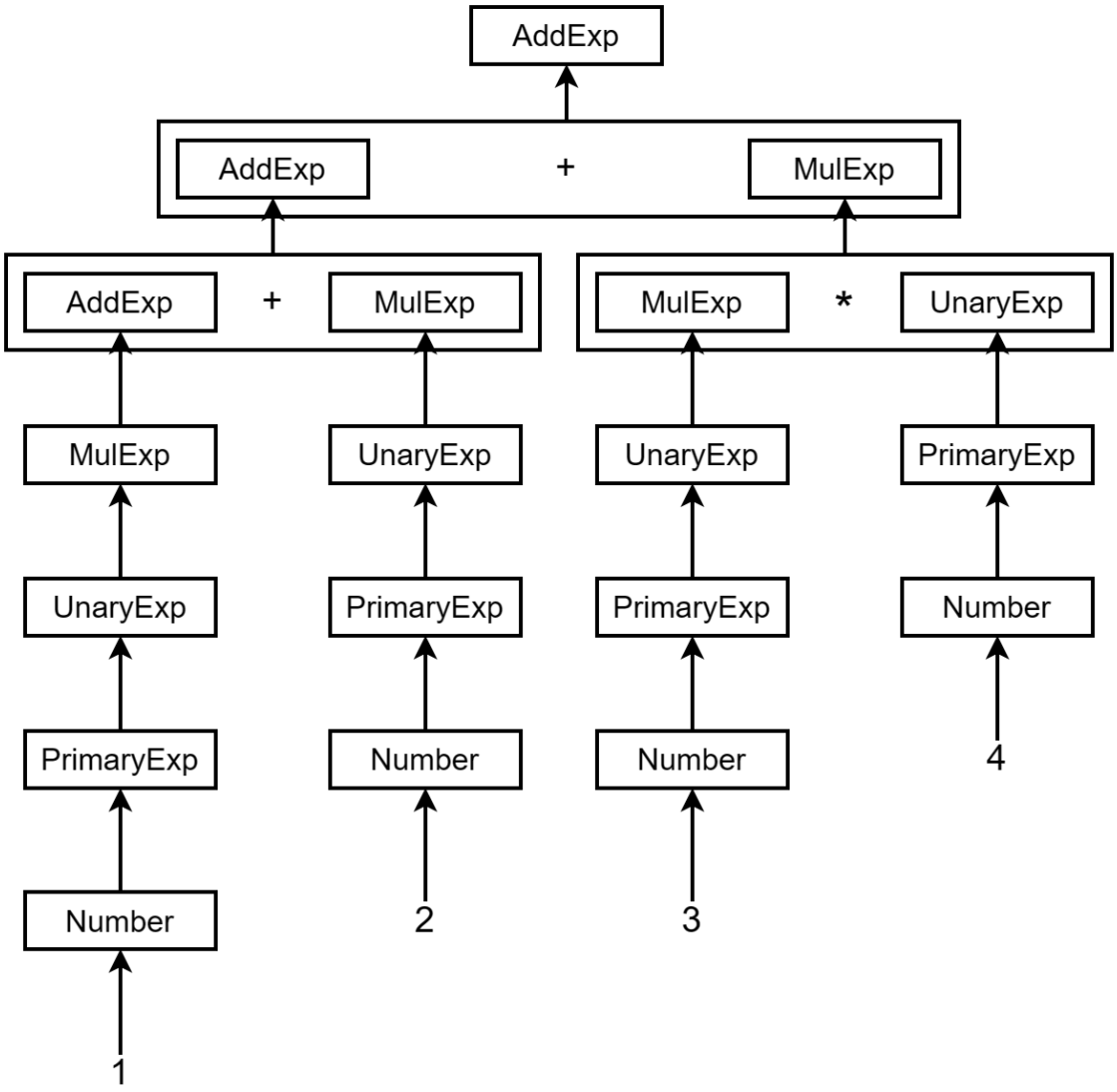



**图 1-1 简单四则运算AST参考图**

在生成的时候，顺序是从左到右，从上到下。所以先生成 `1`，然后生成 `2`，然后生成 `1+2`，然后生成 `3`，然后生成 `4`，然后生成 `3*4`，最后生成 `1+2+3*4`。那对于1+2的**AddExp**，在其生成的指令中，1和2的值就类似于综合属性，即从AddExp的实例的值（3）由产生式右边的值（1和2）推导出来。而对于3*4的**MulExp**，其生成的指令中3和4的值就类似于继承属性，即从MulExp的实例的值（12）由产生式左边的值（3和4）推导出来。最后，对于1+2+3*4的**AddExp**，生成指令的实例的值就由产生式右边的AddExp的值（3）和MulExp的值（12）推导出来。

同理，对于数字前的**正负**，可以看做是**0和其做一次AddExp**，即+1其实就是0+1 （其实正号甚至都不用去管他），-1其实就是0-1。所以在生成代码的时候，可以当作一个特殊的AddExp来处理。

> 特别注意：`MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp`运算中，`%`模运算的代码生成不是mod哦

**（3）测试样例**

源程序

```
int main() {
    return --+---+1 * +2 * ---++3 + 4 + --+++5 + 6 + ---+--7 * 8 + ----++--9 *
++++++10 * -----11;
}
```

生成代码参考（因为最后的代码是扔到评测机上重新编译去跑的，所以生成的代码不一定要一样，但是要确保输出结果一致）

```
define dso_local i32 @main() {
    %1 = sub i32 0, 1
    %2 = sub i32 0, %1
    %3 = sub i32 0, %2
    %4 = sub i32 0, %3
    %5 = sub i32 0, %4
    %6 = mul i32 %5, 2
    %7 = sub i32 0, 3
    %8 = sub i32 0, %7
    %9 = sub i32 0, %8
    %10 = mul i32 %6, %9
    %11 = add i32 %10, 4
    %12 = sub i32 0, 5
    %13 = sub i32 0, %12
    %14 = add i32 %11, %13
    %15 = add i32 %14, 6
    %16 = sub i32 0, 7
    %17 = sub i32 0, %16
    %18 = sub i32 0, %17
    %19 = sub i32 0, %18
    %20 = sub i32 0, %19
    %21 = mul i32 %20, 8
    %22 = add i32 %15, %21
    %23 = sub i32 0, 9
    %24 = sub i32 0, %23
    %25 = sub i32 0, %24
    %26 = sub i32 0, %25
    %27 = sub i32 0, %26
    %28 = sub i32 0, %27
    %29 = mul i32 %28, 10
    %30 = sub i32 0, 11
    %31 = sub i32 0, %30
    %32 = sub i32 0, %31
    %33 = sub i32 0, %32
    %34 = sub i32 0, %33
    %35 = mul i32 %29, %34
    %36 = add i32 %22, %35
    ret i32 %36
}
```

## 3. 全局变量与局部变量

### （1）全局变量

本章实验涉及的文法包括：

```
CompUnit     → {Decl} MainFuncDef
Decl         → ConstDecl | VarDecl
ConstDecl    → 'const' BType ConstDef { ',' ConstDef } ';'
BType        → 'int'
ConstDef     → Ident  '=' ConstInitVal
ConstInitVal → ConstExp
ConstExp     → AddExp
VarDecl      → BType VarDef { ',' VarDef } ';'
VarDef       → Ident | Ident '=' InitVal
InitVal      → Exp
```

在llvm中，全局变量使用的是和函数一样的全局标识符 `@`，所以全局变量的写法其实和函数的定义几乎一样。在本次的实验中，全局变/常量声明中指定的初值表达式必须是**常量表达式**。不妨举几个例子：

```
//以下都是全局变量
int a=5;
int b=2+3;
```

生成的llvm如下所示

```
@a = dso_local global i32 5
@b = dso_local global i32 5
```

可以看到，对于全局变量中的常量表达式，在生成的llvm中需要直接算出其**具体的值**。

### （2）局部变量与作用域

本章内容涉及文法包括：

```
BlockItem → Decl | Stmt
```

局部变量使用的标识符是 `%`。与全局变量不同，局部变量在赋值前需要申请一块内存。在对局部变量操作的时候也需要采用**load/store**来对内存进行操作。
同样的，举个例子来说明一下：

```
//以下都是局部变量
int a=1+2;
```

生成的llvm如下所示

```
%1 = alloca i32
%2 = add i32 1, 2
store i32 %2, i32* %1
```

### （3）符号表设计

这一章将主要考虑变量，包括**全局变量**和**局部变量**以及**作用域**的说明。不可避免地，同学们需要进行符号表的设计。

涉及到的文法如下：

```
Stmt       → LVal '=' Exp ';'
           | [Exp] ';'
           | 'return' Exp ';'
LVal       → Ident
PrimaryExp → '(' Exp ')' | LVal | Number
```

举个最简单的例子：

```
int a=1;
int b=2+a;
int main(){
  int c=b+4;
  return a+b+c;
}
```

如果需要将上述代码转换为llvm，应当怎么考虑呢？直观来看，a和b是**全局变量**，c是**局部变量**。直观上来说，过程是首先将全局变量a和b进行赋值，然后进入到main函数内部，对c进行赋值。那么在`return a+b+c;`的时候，根据上个实验，llvm最后几行应该是

```
%sumab = add i32 %a, %b
%sumabc = add i32 %sumab, %c
ret i32 %sumabc
```

问题就是，如何获取标识符`%a，%b，%c`，这时候符号表的作用就体现出来了。简单来说，符号表类似于一个索引。通过符号表可以很快速的找到变量对应的标识符。

对于上面的c语言程序，llvm生成如下：

```
@a = dso_local global i32 1
@b = dso_local global i32 3
define dso_local i32 @main() {
    %1 = alloca i32           ;分配c内存
    %2 = load i32, i32* @b    ;读取全局变量b
    %3 = add i32 %2, 4        ;计算b+4
    store i32 %3, i32* %1     ;把b+4的值存入c
    %4 = load i32, i32* @a    ;读取全局变量a
    %5 = load i32, i32* @b    ;读取全局变量b
    %6 = add i32 %4, %5       ;计算a+b;
    %7 = load i32, i32* %1    ;读取c
    %8 = add i32 %6, %7       ;计算(a+b)+c
    ret i32 %8                ;return
}
```

不难发现，对于全局变量的使用，可以直接使用全局变量的**全局标识符**（例如`@a`），而对于**局部变量**，则需要使用分配内存的标识符。由于标识符是**自增的数字**，所以快速找到对应变量的标识符就是符号表最重要的作用。同学们可以选择遍历一遍AST后造出一张统一的符号表，然后根据**完整的符号表**进行代码生成，也可以在遍历AST的同时造出一张**栈式符号表**，根据实时的栈式符号表生成相应代码。符号表存储的东西同学们可以自己设计，下面给出符号表的简略示例，同学们在实验中可以根据自己需要自行设计。

同时，同学们需要注意变量的作用域，即语句块内声明的变量的生命周期在该语句块内，且内层代码块覆盖外层代码块。

```
int a=1;
int b=2;
int c=3;
int main(){
  int d=4;
  int e=5;
  {//blockA
    int a=7;
    int e=8;
    int f=9;
  }
  int f=10;
}
```

在上面的程序中，在**blockA**中，a的值为7，覆盖了全局变量a=1，e覆盖了main中的e=5，而在main的最后一行，f并不存在覆盖，因为main外层不存在其他f的定义。

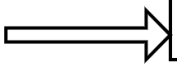
同样的，下面给出上述程序的llvm代码：

```
@a = dso_local global i32 1
@b = dso_local global i32 2
@c = dso_local global i32 3

define dso_local i32 @main() {
    %1 = alloca i32
    store i32 4, i32* %1
    %2 = alloca i32
    store i32 5, i32* %2
    %3 = alloca i32
    store i32 7, i32* %3
    %4 = alloca i32
    store i32 8, i32* %4
    %5 = alloca i32
    store i32 9, i32* %5
    %6 = alloca i32
    store i32 10, i32* %6
}
```

上述程序的符号表简略示意如下：

| | | | |
|---|---|---|---|
| a | int | 0 | global |
| b | int | 0 | global |
| c | int | 0 | global |
| d | int | 1 | main |
| e | int | 1 | main |
| a | int | 2 | block |
| e | int | 2 | block |
| f | int | 2 | block |
| f | int | 1 | main |



| | | | | |
|---|---|---|---|---|
| a | int | 0 | 1 | @a |
| b | int | 0 | 2 | @b |
| c | int | 0 | 3 | @c |
| d | int | 1 | 4 | %1 |
| e | int | 1 | 5 | %2 |
| f | int | 1 | 10 | %6 |

**图 2-1 完整符号表与栈式符号表示意图**

**(4) 测试样例**

源程序

```
int a=1;
int b=2+a;
int c=3*(b+------10);
int main(){
  int d=4+c;
  int e=5*d;
  {
    a=a+5;
    int b=a*2;
    a=b;
    int f=20;
    e=e+a*20;
  }
  int f=10;
  return e*f;
}
```

llvm参考如下：

```
@a = dso_local global i32 1
@b = dso_local global i32 3
@c = dso_local global i32 39
define dso_local i32 @main() {
    %1 = alloca i32
    %2 = load i32, i32* @c
    %3 = add i32 4, %2
    store i32 %3, i32* %1
    %4 = alloca i32
    %5 = load i32, i32* %1
    %6 = mul i32 5, %5
    store i32 %6, i32* %4
```

```
    %7 = load i32, i32* @a
    %8 = load i32, i32* @a
    %9 = add i32 %8, 5
    store i32 %9, i32* @a
    %10 = alloca i32
    %11 = load i32, i32* @a
    %12 = mul i32 %11, 2
    store i32 %12, i32* %10
    %13 = load i32, i32* @a
    %14 = load i32, i32* %10
    store i32 %14, i32* @a
    %15 = alloca i32
    store i32 20, i32* %15
    %16 = load i32, i32* %4
    %17 = load i32, i32* %4
    %18 = load i32, i32* @a
    %19 = mul i32 %18, 20
    %20 = add i32 %17, %19
    store i32 %20, i32* %4
    %21 = alloca i32
    store i32 10, i32* %21
    %22 = load i32, i32* %4
    %23 = load i32, i32* %21
    %24 = mul i32 %22, %23
    ret i32 %24
}
```

echo $?的结果为**198**

> 相信各位如果去手动计算的话，会算出来结果是4550。然而由于echo $?的返回值只截取最后一个字节，也就是8位，所以 `4550 mod 256 = 198`

## 4. 函数的定义及调用

> 本章主要涉及**不含数组**的函数的定义，调用等。

**库函数**

涉及文法有：

```
Stmt → LVal '=' 'getint''('')'';'
     | 'printf''('FormatString{','Exp}')'';'
```

首先添加**库函数**的调用。在实验的llvm代码中，库函数的声明如下：

```
declare i32 @getint()
declare void @putint(i32)
declare void @putch(i32)
declare void @putstr(i8*)
```

只要在llvm代码开头加上这些声明，就可以在后续代码中使用这些库函数。同时对于用到库函数的llvm代码，在编译时也需要使用llvm-link命令将库函数链接到生成的代码中。

对于库函数的使用，在文法中其实就包含两句，即`getint`和`printf`。其中，`printf`包含了有Exp和没有Exp的情况。同样的，这里给出一个简单的例子：

```
int main(){
  int a;
  a=getint();
  printf("hello:%d",a);
  return 0;
}
```

llvm代码如下：

```
declare i32 @getint()
declare void @putint(i32)
declare void @putch(i32)
declare void @putstr(i8*)

define dso_local i32 @main() {
    %1 = alloca i32
    %2 = load i32, i32* %1
    %3 = call i32 @getint()
    store i32 %3, i32* %1
    %4 = load i32, i32* %1
    call void @putch(i32 104)
    call void @putch(i32 101)
    call void @putch(i32 108)
    call void @putch(i32 108)
    call void @putch(i32 111)
    call void @putch(i32 58)
    call void @putint(i32 %4)
    ret i32 0
}
```

不难看出，`call i32 @getint()`即为调用getint的语句，对于其他的任何函数的调用也是像这样去写。而对于`printf`，则需要将其转化为多条`putch`和`putint`的调用，或者使用`putstr`以字符串输出。这里需要注意的是，`putch`和`putint`的参数都是`i32`类型，所以需要将字符串中的字符转化为对应的ascii码。

**函数定义与调用**

涉及文法如下：

```
CompUnit    → {Decl} {FuncDef} MainFuncDef
FuncDef     → FuncType Ident '(' [FuncFParams] ')' Block
FuncType    → 'void' | 'int'
FuncFParams → FuncFParam { ',' FuncFParam }
FuncFParam  → BType Ident
UnaryExp    → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
```

其实之前的main函数也是一个函数，即主函数。这里将其拓广到一般函数。对于一个函数，其特征包括**函数名**，**函数返回类型**和**参数**。在本实验中，函数返回类型只有`int`和`void`两种。由于目前只有零维整数作为参数，所以参数的类型统一都是`i32`。FuncFParams之后的Block则与之前主函数内处理方法一样。值得一提的是，由于每个**临时寄存器**和**基本块**占用一个编号，所以没有参数的函数的第一个临时寄存器的编号应该从**1**开始，因为函数体入口占用了一个编号0。而有参数的函数，参数编号从**0**开始，进入Block后需要跳过一个基本块入口的编号（可以参考测试样例）。

当然，如果全部采用字符串编号寄存器，上述问题都不会存在。

对于函数的调用，参考之前库函数的处理，不难发现，函数的调用其实和**全局变量**的调用基本是一样的，即用`@函数名`表示。所以函数部分和**符号表**有着密切关联。同学们需要在函数定义和函数调用的时候对符号表进行操作。对于有参数的函数调用，则在调用的函数内传入参数。对于没有返回值的函数，则直接`call`即可，不用为语句赋一个实例。

**（3）测试样例**

源代码：

```
int a=1000;
int aaa(int a,int b){
        return a+b;
}
void ab(){
        a=1200;
        return;
}
int main(){
        ab();
        int b=a,a;
        a=getint();
        printf("%d",aaa(a,b));
        return 0;
}
```

llvm输出参考：

```
declare i32 @getint()
declare void @putint(i32)
declare void @putch(i32)
declare void @putstr(i8*)
@a = dso_local global i32 1000
define dso_local i32 @aaa(i32 %0, i32 %1){
  %3 = alloca i32
  %4 = alloca i32
  store i32 %0, i32* %3
  store i32 %1, i32* %4
  %5 = load i32, i32* %3
  %6 = load i32, i32* %4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
define dso_local void @ab(){
  store i32 1200, i32* @a
  ret void
}

define dso_local i32 @main(){
  %1 = alloca i32
  %2 = alloca i32
  call void @ab()
  %3 = load i32, i32* @a
  store i32 %3, i32* %1
  %4 = call i32 @getint()
  store i32 %4, i32* %2
  %5 = load i32, i32* %2
  %6 = load i32, i32* %1
```

```
  %7 = call i32 @aaa(i32 %5, i32 %6)
  call void @putint(i32 %7)
  ret i32 0
}
```

- 输入：1000
- 输出：2200

## 5. 条件语句与短路求值

### （1）条件语句

涉及文法如下

```
Stmt    → 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
Cond    → LOrExp
RelExp  → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
EqExp   → RelExp | EqExp ('==' | '!=') RelExp
LAndExp → EqExp | LAndExp '&&' EqExp
LOrExp  → LAndExp | LOrExp '||' LAndExp
```

在条件语法中，需要同学们进行条件的判断与选择。这时候就涉及到基本块的标号。在llvm中，每个**临时寄存器**和**基本块**占用一个编号。所以对于纯数字编号的llvm，这里就需要进行**回填**操作。对于在代码生成前已经完整生成符号表的同学，这里就会显得十分容易。对于在代码生成同时生成符号表的同学，也可以采用**栈**的方式去回填编号，对于采用字符串编号的则没有任何要求。

要写出条件语句，首先要理清楚逻辑。在上述文法中，最重要的莫过于下面这一条语法

```
Stmt    → 'if' '(' Cond ')' Stmt1 [ 'else' Stmt2 ]  (BasicBlock3)
```

为了方便说明，对上述文法的两个Stmt编号为Stmt1和2。在这条语句之后基本块假设叫BasicBlock3。不难发现，条件判断的逻辑如左下图。

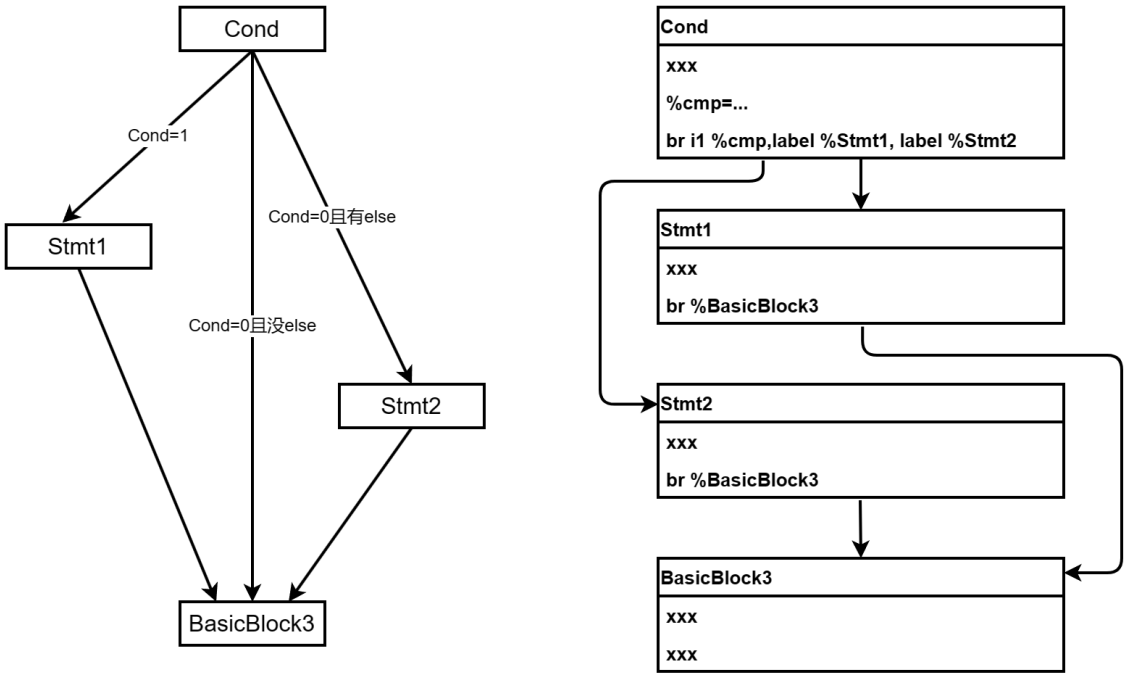


**图 4-1 条件判断流程示意图与基本块流图**

首先进行Cond结果的判断，如果结果为**1**则进入**Stmt1**，如果Cond结果为**0**，若文法有else则将进入**Stmt2**，否则进入下一条文法的基本块**BasicBlock3**。在Stmt1或Stmt2执行完成后都需要跳转到BasicBlock3。对于一个llvm程序来说，对一个含else的条件分支，其基本块构造可以如右上图所示。

如果能够理清楚基本块跳转的逻辑，那么在写代码的时候就会变得十分简单。

这时候再回过头去看Cond里面的代码，即LOr和Land，Eq和Rel。不难发现，其处理方式和加减乘除非常像，除了运算结果都是1位(i1)而非32位(i32)。同学们可能需要用到 `trunc` 或者 `zext` 指令进行类型转换。

### （2）短路求值

可能有的同学会认为，反正对于llvm来说，跳转与否只看Cond的值，所以只要把Cond算完结果就行，不会影响正确性。不妨看一下下面这个例子：

```
int a=5;
int change(){
  a=6;
  return a;
}
int main(){
  if(1||change()){
    printf("%d",a);
  }
  return 0;
}
```

如果要将上面这段代码翻译为llvm，同学们会怎么做？如果按照传统方法，即先**统一计算Cond**，则一定会执行一次 `change()` 函数，把全局变量的值变为**6**。但事实上，由于短路求值的存在，在读完1后，整个Cond的值就**已经被确定**了，即无论 `1||` 后面跟的是什么，都不影响Cond的结果，那么根据短路求值，后面的东西就不应该执行。所以上述代码的输出应当为**5**而不是6，也就是说，llvm不能够单纯的把Cond计算完后再进行跳转。这时候就需要对Cond的跳转逻辑进行改写。

改写之前同学们不妨思考一个问题，即什么时候跳转。根据短路求值，只要条件判断出现“短路”，即不需要考虑后续与或参数的情况下就已经能确定值的时候，就可以进行跳转。或者更简单的来说，当**LOrExp值为1**或者**LAndExp值为0**的时候，就已经没有必要再进行计算了。

```
Cond    → LOrExp
LAndExp → LAndExp '&&' EqExp
LOrExp  → LOrExp '||' LAndExp
```

- 对于连或来说，只要其中一个LOrExp或最后一个LAndExp为1，即可直接跳转Stmt1。
- 对于连与来说，只要其中一个LAndExp或最后一个EqExp为0，则直接进入下一个LOrExp。如果当前为连或的最后一项，则直接跳转Stmt2（有else）或BasicBlock3（没else）

上述两条规则即为短路求值的最核心算法，示意图如下。

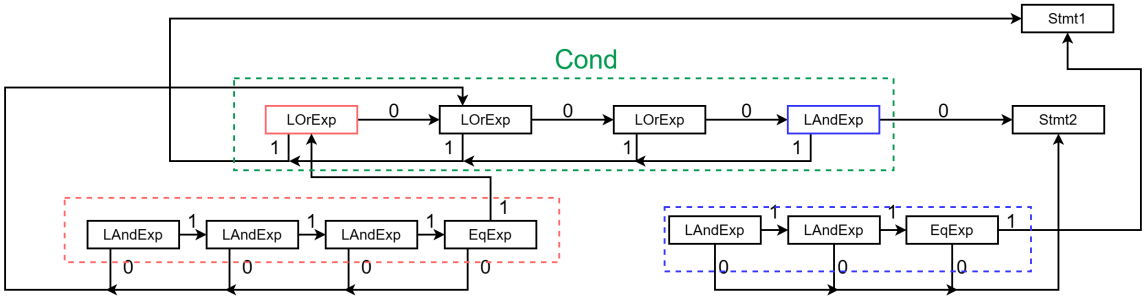



**图 4-2 短路求值算法示意图**

### (3) 测试样例

```
int a=1;
int func(){
    a=2;return 1;
}

int func2(){
    a=4;return 10;
}
int func3(){
    a=3;return 0;
}
int main(){
    if(0||func()&&func3()||func2()){printf("%d--1",a);}
    if(1||func3()){printf("%d--2",a);}
    if(0||func3()||func()<func2()){printf("%d--3",a);}
    return 0;
}
```

```
declare i32 @getint()
declare void @putint(i32)
declare void @putch(i32)
declare void @putstr(i8*)
@a = dso_local global i32 1
define dso_local i32 @func() {
    %1 = load i32, i32* @a
    store i32 2, i32* @a
    ret i32 1
}
define dso_local i32 @func2() {
    %1 = load i32, i32* @a
    store i32 4, i32* @a
    ret i32 10
}
define dso_local i32 @func3() {
    %1 = load i32, i32* @a
    store i32 3, i32* @a
    ret i32 0
}

define dso_local i32 @main() {
    br label %1
```

```
1:
    %2 = icmp ne i32 0, 0
    br i1 %2, label %12, label %3

3:
    %4 = call i32 @func()
    %5 = icmp ne i32 0, %4
    br i1 %5, label %6, label %9

6:
    %7 = call i32 @func3()
    %8 = icmp ne i32 0, %7
    br i1 %8, label %12, label %9

9:
    %10 = call i32 @func2()
    %11 = icmp ne i32 0, %10
    br i1 %11, label %12, label %14

12:
    %13 = load i32, i32* @a
    call void @putint(i32 %13)
    call void @putch(i32 45)
    call void @putch(i32 45)
    call void @putch(i32 49)
    br label %14

14:
    br label %15

15:
    %16 = icmp ne i32 0, 1
    br i1 %16, label %20, label %17

17:
    %18 = call i32 @func3()
    %19 = icmp ne i32 0, %18
    br i1 %19, label %20, label %22

20:
    %21 = load i32, i32* @a
    call void @putint(i32 %21)
    call void @putch(i32 45)
    call void @putch(i32 45)
    call void @putch(i32 50)
    br label %22

22:
    br label %23

23:
    %24 = icmp ne i32 0, 0
    br i1 %24, label %34, label %25

25:
    %26 = call i32 @func3()
    %27 = icmp ne i32 0, %26
```

```
    br i1 %27, label %34, label %28

28:
    %29 = call i32 @func()
    %30 = call i32 @func2()
    %31 = icmp slt i32 %29, %30
    %32 = zext i1 %31 to i32
    %33 = icmp ne i32 0, %32
    br i1 %33, label %34, label %36

34:
    %35 = load i32, i32* @a
    call void @putint(i32 %35)
    call void @putch(i32 45)
    call void @putch(i32 45)
    call void @putch(i32 51)
    br label %36

36:
    ret i32 0
}
```

> 注：由于历史遗留问题，跳转参考代码较乱，参考价值不大，同学们可以自行设计，仅需要保证短路求值正确性即可。

- 输出：4--14--24--3

## 6. 条件判断与循环

### 循环

涉及文法如下：

```
Stmt    → 'for' '(' [forStmt] ';' [Cond] ';' [forStmt] ')' Stmt
        | 'break' ';'
        | 'continue' ';'
forStmt → LVal '=' Exp
```

如果经过了上一章的学习，这一章其实难度就小了不少。对于这条文法，同样可以改写为

```
Stmt    → 'for' '(' [forStmt1] ';' [Cond] ';' [forStmt2] ')' Stmt (BasicBlock)
```

如果查询C语言的for循环，其中对for循环的描述为：

```
for(initialization;condition;incr/decr){
   //code to be executed
}
```

不难发现，实验文法中的forStmt1，Cond，forStmt2分别表示了上述for循环中的**初始化(initialization)**，**条件(condition)**和**增量/减量(increment/decrement)**。同学们去搜索C语言的for循环逻辑的话也会发现，for循环的逻辑可以表述为

- 1.执行初始化表达式forStmt1
- 2.执行条件表达式Cond，如果为1执行循环体Stmt，否则结束循环执行BasicBlock
- 3.执行完循环体Stmt后执行增量/减量表达式forStmt2

- 4.重复执行步骤2和步骤3

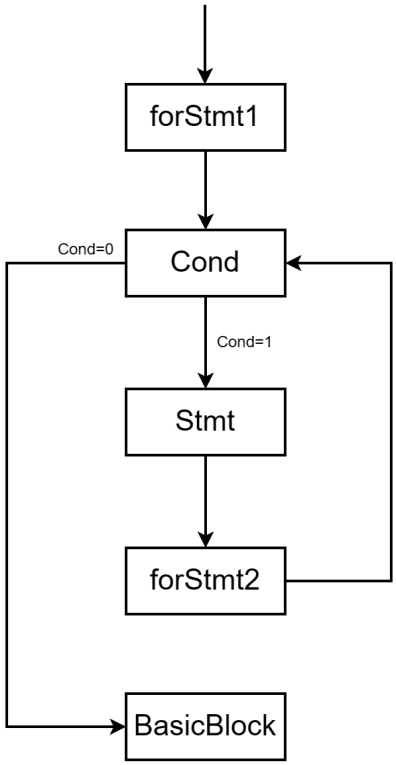



**图 5-1 for循环流程图**

### (2) break/continue

对于`break`和`continue`，直观理解为，break**跳出循环**，continue**跳过本次循环**。再通俗点说就是，break跳转到的是**BasicBlock**，而continue跳转到的是**Cond**。这样就能达到目的了。所以，对于循环而言，跳转的位置很重要。这也是同学们在编码的时候需要着重注意的点。

同样的，针对这两条指令，对上图作出一定的修改，就是整个循环的流程图了。

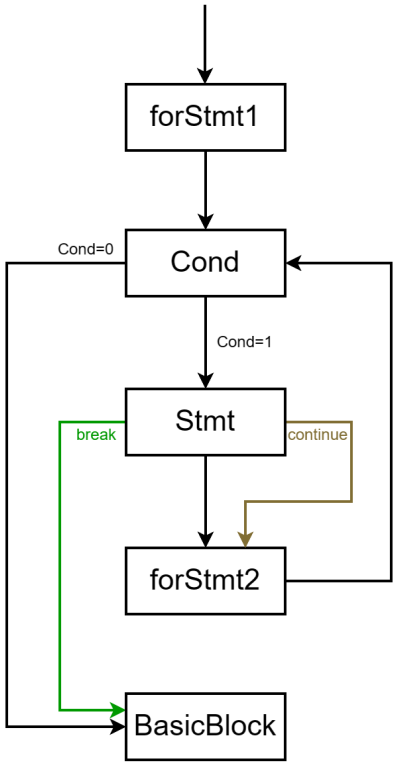


图 5-2 for循环完整流程图

**(3) 测试样例**

```c
int main(){
    int a1=1,a2;
    a2=a1;
    int temp;
    int n,i;
    n=getint();
    for(i=a1*a1;i<n+1;i=i+1){
        temp=a2;
        a2=a1+a2;
        a1=temp;
        if(i%2==1){
            continue;
        }
        printf("round %d: %d\n",i,a1);
        if(i>19){
            break;
        }
    }
    return 0;
}
```

```
declare i32 @getint()
declare void @putint(i32)
declare void @putch(i32)
declare void @putstr(i8*)
define dso_local i32 @main() {
    %1 = alloca i32
    store i32 1, i32* %1
    %2 = alloca i32
    %3 = load i32, i32* %1
    store i32 %3, i32* %2
    %4 = alloca i32
    %5 = alloca i32
    %6 = alloca i32
    %7 = call i32 @getint()
    store i32 %7, i32* %5
    %8 = load i32, i32* %1
    %9 = load i32, i32* %1
    %10 = mul i32 %8, %9
    store i32 %10, i32* %6
    br label %11

11:
    %12 = load i32, i32* %6
    %13 = load i32, i32* %5
    %14 = add i32 %13, 1
    %15 = icmp slt i32 %12, %14
    %16 = zext i1 %15 to i32
    %17 = icmp ne i32 0, %16
    br i1 %17, label %18, label %44

18:
```

```
    %19 = load i32, i32* %2
    store i32 %19, i32* %4
    %20 = load i32, i32* %2
    %21 = load i32, i32* %1
    %22 = load i32, i32* %2
    %23 = add i32 %21, %22
    store i32 %23, i32* %2
    %24 = load i32, i32* %4
    store i32 %24, i32* %1
    br label %25

25:
    %26 = load i32, i32* %6
    %27 = srem i32 %26, 2
    %28 = icmp eq i32 %27, 1
    %29 = zext i1 %28 to i32
    %30 = icmp ne i32 0, %29
    br i1 %30, label %31, label %32

31:
    br label %41

32:
    %33 = load i32, i32* %6
    %34 = load i32, i32* %1
    call void @putch(i32 114)
    call void @putch(i32 111)
    call void @putch(i32 117)
    call void @putch(i32 110)
    call void @putch(i32 100)
    call void @putch(i32 32)
    call void @putint(i32 %33)
    call void @putch(i32 58)
    call void @putch(i32 32)
    call void @putint(i32 %34)
    call void @putch(i32 10)
    br label %35

35:
    %36 = load i32, i32* %6
    %37 = icmp sgt i32 %36, 19
    %38 = zext i1 %37 to i32
    %39 = icmp ne i32 0, %38
    br i1 %39, label %40, label %41

40:
    br label %44

41:
    %42 = load i32, i32* %6
    %43 = add i32 %42, 1
    store i32 %43, i32* %6
    br label %11

44:
    ret i32 0
}
```

- 输入：10
- 输出：

```
round 2: 2
round 4: 5
round 6: 13
round 8: 34
round 10: 89
```

- 输入：40
- 输出：
- ```
  round 2: 2
  round 4: 5
  round 6: 13
  round 8: 34
  round 10: 89
  round 12: 233
  round 14: 610
  round 16: 1597
  round 18: 4181
  round 20: 10946
  ```

> 本质为一个只输出20以内偶数项的斐波那契数列

## 7. 数组与函数

### （1）数组

数组涉及的文法相当多，包括以下几条：

```
ConstDef     → Ident { '[' ConstExp ']' } '=' ConstInitVal
ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
VarDef       → Ident { '[' ConstExp ']' } | Ident { '[' ConstExp ']' } '='
InitVal
InitVal      → Exp | '{' [ InitVal { ',' InitVal } ] '}'
FuncFParam   → BType Ident ['[' ']' { '[' ConstExp ']' }]
LVal         → Ident {'[' Exp ']'}
```

在数组的编写中，同学们会频繁用到 `getElementPtr` 指令，故先系统介绍一下这个指令的用法。

getElementPtr指令的工作是计算地址。其本身不对数据做任何访问与修改。其语法如下：

```
<result> = getelementptr <ty>, <ty>* <ptrval>, {<ty> <index>}*
```

现在来理解一下上面这一条指令。第一个 `&lt;ty&gt;` 表示的是第一个索引所指向的类型，有时也是**返回值的类型**。第二个 `&lt;ty&gt;` 表示的是后面的指针基地址 `&lt;ptrval&gt;` 的类型， `&lt;ty&gt; &lt;index&gt;` 表示的是一组索引的类型和值，在本实验中索引的类型为i32。索引指向的基本类型确定的是增加索引值时指针的偏移量。

说完理论，不如结合一个实例来讲解。考虑数组 `a[5][7]`，需要获取 `a[3][4]` 的地址，有如下写法：
```

```
%1 = getelementptr [5 x [7 x i32]], [5 x [7 x i32]]* @a, i32 0, i32 3
%2 = getelementptr [7 x i32], [7 x i32]* %1, i32 0, i32 4

%3 = getelementptr [5 x [7 x i32]], [5 x [7 x i32]]* @a, i32 0, i32 3, i32 4

%4 = getelementptr [5 x [7 x i32]], [5 x [7 x i32]]* @a, i32 0, i32 0
%5 = getelementptr [7 x i32], [7 x i32]* @4, i32 3, i32 0,
%6 = getelementptr i32, i32* %5, i32 4
```

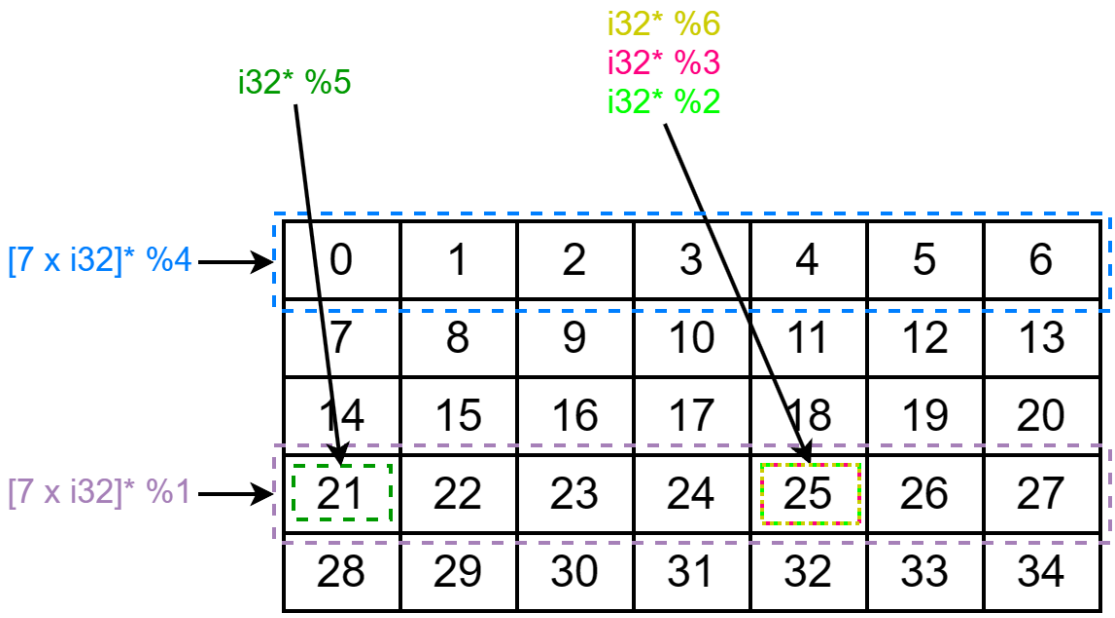



**图 6-1 getElementPtr示意图**

对于 `%6` ，其只有一组索引**i32 4**，所以索引使用基本类型为**i32**，基地址为 `%5` ，索引值为4，所以指针相对于%5（21号格子）前进了**4个i32**类型，即指向了25号格子，返回类型为i32*。

而当存在多组索引值的时候，每多一组索引值，索引使用基本类型就要去掉一层。再拿上面举个例子，对于 `%1` ，基地址为@a，类型为 **[5 x [7 x i32]]** 。第一组索引值为**0**，所以指针前进0个0x5x7个i32，第二组索引值为**3**，这时候就要**去掉一层**，即把[5 x [7 x i32]]中的5去掉，即索引使用的基本类型为**[7 x i32]** ，指针向前移动3x7个i32。返回类型为 **[7 x i32]***，而对于 `%2` ，第一组索引值为**0**，其首先前进了0x7个i32，第二组索引值为**4**，去掉一层，索引基本类型为i32，指针向前移动4个i32，指向25号格子

当然，可以一步到位，如 `%3` ，后面跟了三组索引值，第一组让指针前进0x5x7个i32，第二组让指针前进3x7个i32，第三组让指针前进4个i32，索引使用基本类型去掉两层，为**i32***。

对于 `%4` ，虽然其两组索引值都是0，但是其索引使用基本类型去掉了一层，变为了 **[7 x i32]** 。在 `%5` 的时候，第一组索引值为3，即指针前进3x7个i32，第二组索引值为0，即指针前进0个i32，索引使用基本类型变为i32，返回指针类型为 **i32***。

当然，同学们也可以直接将所有高维数组模拟为1维数组，例如对于**a[5][7]**中取**a[3][4]**，可以直接将a转换为一个**a[35]**，然后指针偏移7x3+4=25，直接取**a[25]**。

### （2）数组定义与调用

这一章将主要讲述数组定义和调用，包括全局数组，局部数组的定义，以及函数中的数组调用。对于全局数组定义，与全局变量一样，同学们需要将所有量**全部计算到特定的值。**同时，对于全局数组，对数组中空缺的值，需要**置0。**对于全是0的地方，可以采用 `zeroinitializer` 来统一置0。

```
int a[1+2+3+4]={1,1+1,1+3-1};
int b[10][20];
int c[5][5]={{1,2,3},{1,2,3,4,5}};
```

```
@a = dso_local global [10 x i32] [i32 1, i32 2, i32 3, i32 0, i32 0, i32 0, i32
0, i32 0, i32 0, i32 0]
@b = dso_local global [10 x [20 x i32]] zeroinitializer
@c = dso_local global [5 x [5 x i32]] [[5 x i32] [i32 1, i32 2, i32 3, i32 0, i32
0], [5 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5], [5 x i32] zeroinitializer, [5
x i32] zeroinitializer, [5 x i32] zeroinitializer]
```

当然，zeroinitializer不是必须的，同学们完全可以一个个**i32 0**写进去，但对于一些很阴间的样例点，不用zeroinitializer可能会导致 `TLE` ，例如全局数组 `int a[1000];` ，不使用该指令就需要输出**1000次i32 0**，必然导致TLE，所以还是推荐同学们使用zeroinitializer。

对于局部数组，在定义的时候同样需要使用 `alloca` 指令，其存取指令同样采用**load和store**，只是在此之前需要采用 `getelementptr` 获取数组内应位置的地址。

对于数组传参，其中涉及到维数的变化问题，例如，对于参数中**含维度的数组**，同学们可以参考上述 `getelementptr` 指令自行设计，因为该指令很灵活，所以下面的测试样例仅仅当一个参考。同学们可以将自己生成的llvm使用**lli**编译后自行查看输出比对。

### （3）测试样例

```
int a[3+3]={1,2,3,4,5,6};
int b[3][3]={{3,6+2,5},{1,2}};
void a1(int x){
    if(x>1){
        a1(x-1);
    }
    return;
}
int a2(int x,int y[]){
    return x+y[2];
}
int a3(int x,int y[],int z[][3]){
    return x*y[1]-z[2][1];
}
int main(){
    int c[2][3]={{1,2,3}};
    a1(c[0][2]);
    int x=a2(a[4],a);
    int y=a3(b[0][1],b[1],b);
    printf("%d",x+y);
    return 0;
}
```

```
declare i32 @getint()
declare void @putint(i32)
```

```
declare void @putch(i32)
declare void @putstr(i8*)
@a = dso_local global [6 x i32] [i32 1, i32 2, i32 3, i32 4, i32 5, i32 6]
@b = dso_local global [3 x [3 x i32]] [[3 x i32] [i32 3, i32 8, i32 5], [3 x i32]
[i32 1, i32 2, i32 0], [3 x i32] zeroinitializer]
define dso_local void @a1(i32 %0) {
    %2 = alloca i32
    store i32 %0, i32 * %2
    br label %3

3:
    %4 = load i32, i32* %2
    %5 = icmp sgt i32 %4, 1
    %6 = zext i1 %5 to i32
    %7 = icmp ne i32 0, %6
    br i1 %7, label %8, label %11

8:
    %9 = load i32, i32* %2
    %10 = sub i32 %9, 1
    call void @a1(i32 %10)
    br label %11

11:
    ret void
}
define dso_local i32 @a2(i32 %0, i32* %1) {
    %3 = alloca i32*
    store i32* %1, i32* * %3
    %4 = alloca i32
    store i32 %0, i32 * %4
    %5 = load i32, i32* %4
    %6 = load i32*, i32* * %3
    %7 = getelementptr i32, i32* %6, i32 2
    %8 = load i32, i32* %7
    %9 = add i32 %5, %8
    ret i32 %9
}
define dso_local i32 @a3(i32 %0, i32* %1, [3 x i32] *%2) {
    %4 = alloca [3 x i32]*
    store [3 x i32]* %2, [3 x i32]* * %4
    %5 = alloca i32*
    store i32* %1, i32* * %5
    %6 = alloca i32
    store i32 %0, i32 * %6
    %7 = load i32, i32* %6
    %8 = load i32*, i32* * %5
    %9 = getelementptr i32, i32* %8, i32 1
    %10 = load i32, i32* %9
    %11 = mul i32 %7, %10
    %12 = load [3 x i32] *, [3 x i32]* * %4
    %13 = getelementptr [3 x i32], [3 x i32]* %12, i32 2
    %14 = getelementptr [3 x i32], [3 x i32]* %13, i32 0, i32 1
    %15 = load i32, i32 *%14
    %16 = sub i32 %11, %15
    ret i32 %16
}
```

```
define dso_local i32 @main() {
    %1 = alloca [2 x [ 3 x i32]]
    %2 = getelementptr [2 x [3 x i32]], [2 x [3 x i32]]*%1, i32 0, i32 0, i32 0
    store i32 1, i32* %2
    %3 = getelementptr [2 x [3 x i32]], [2 x [3 x i32]]*%1, i32 0, i32 0, i32 1
    store i32 2, i32* %3
    %4 = getelementptr [2 x [3 x i32]], [2 x [3 x i32]]*%1, i32 0, i32 0, i32 2
    store i32 3, i32* %4
    %5 = getelementptr [2 x [3 x i32]], [2 x [3 x i32]]*%1, i32 0, i32 0, i32 2
    %6 = load i32, i32* %5
    call void @a1(i32 %6)
    %7 = alloca i32
    %8 = getelementptr [6 x i32], [6 x i32]* @a, i32 0, i32 4
    %9 = load i32, i32* %8
    %10 = getelementptr [6 x i32], [6 x i32]* @a, i32 0, i32 0
    %11 = call i32 @a2(i32 %9, i32* %10)
    store i32 %11, i32* %7
    %12 = alloca i32
    %13 = getelementptr [3 x [3 x i32]], [3 x [3 x i32]]* @b, i32 0, i32 0, i32
1
    %14 = load i32, i32* %13
    %15 = mul i32 1, 3
    %16 = getelementptr [3 x [3 x i32]], [3 x [3 x i32]]* @b, i32 0, i32 0
    %17 = getelementptr [3 x i32], [3 x i32]* %16, i32 0, i32 %15
    %18 = getelementptr [3 x [3 x i32]], [3 x [3 x i32]]* @b, i32 0, i32 0
    %19 = call i32 @a3(i32 %14, i32* %17, [3 x i32]* %18)
    store i32 %19, i32* %12
    %20 = load i32, i32* %7
    %21 = load i32, i32* %12
    %22 = add i32 %20, %21
    call void @putint(i32 %22)
    ret i32 0
}
```

- 输出：24

# 四、目标代码生成

## （一）动态内存管理

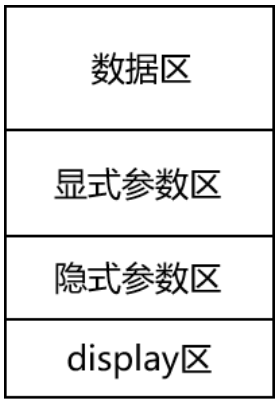
在介绍不同的目标代码之前，先介绍动态内存管理。在生成目标代码时，由于目标代码在执行时是动态的，而编译过程是静态的，因此我们必须采用动态内存管理的方法来在编译过程中合理地进行内存分配。例如下面这段代码，在编译时我们无法确定`fibo()`会调用几次，因此不可能直接完成静态的内存分配。

```
int fibo(int n) {
    if(n <= 1) return 1;
    int x = fibo(n-1);
    int y = fibo(n-2);
    return x+y;
}

int main() {
    int n;
    n = getint();
    n = fibo(n);
```

```
    printf("%d",n);
    return 0;
}
```

实现动态的内存分配，往往需要借助于**活动记录（Activity Record, AR）**这种结构。通常来说，活动记录是为一个过程的一次执行所分配的一块连续内存空间。这里的过程在我们的文法中可以理解为Block，也就是说我们为每一个Block分配一个AR。这块内存空间包含该过程执行所需的数据（如变量、临时变量等），同时还包含一些信息（如返回地址等）。通常来说，AR具有如下的结构。




其中，数据区用于存储定义的变量或计算过程中的临时变量。显式参数区用于函数传参。隐式参数区包含返回地址、返回值、调用者AR基地址等，主要用于该过程执行完毕后返回调用者。display区中包含着对外层活动记录的索引，可以理解为对外层block的AR的索引，便于访问在外层block定义的变量。

AR的分配主要采用**栈式结构**，即进入一个block后会为其分配一个AR，并设置好AR中相应的部分，而在离开一个block后会收回为其分配的AR。

在静态的编译过程中，我们可以借助AR来动态管理内存。比如要访问变量，我们只需要通过AR中的信息来进行相对寻址（通过display区找到外层block的活动记录基地址，再加上变量的偏移地址），而不需要计算出绝对地址（大多数情况下也算不出来）。

当然，上面提到的活动记录是一种通用的动态存储管理方法，具体实现上可能因不同目标代码而异，比如MIPS中可能通过$fp和$sp寄存器来寻址和管理，pcode中可能通过动态链和静态链来寻址，你可以根据自己的需要在原有基础上设计活动记录的结构和动态存储管理的方法，在指导书中我们也给出了相应的示例。

# （二）Pcode代码

## 1. Pcode概述

pcode源于Pascal编译器产生的基于栈式虚拟机的中间代码。尽管中间代码（如四元式等）通常与机器无关，但是由于pcode是基于栈式虚拟机的，因此在生成pcode时必须考虑栈式虚拟机的一些特性，以及运行时的内存管理等。我们首先简单介绍关于栈式虚拟机的一些基本知识。

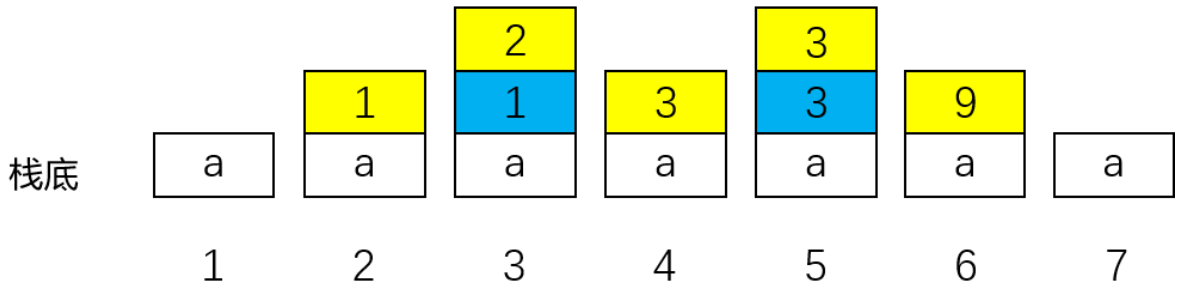
### （1）栈式虚拟机的工作过程

栈式虚拟机可以理解为一个解释执行程序（如python解释器），它能够执行输入的pcode代码并产生相应的结果。

和我们学过的MIPS架构不同，栈式虚拟机中没有众多寄存器用于保存运算结果或变量（如MIPS中的`$t`寄存器和`$s`寄存器等），只有5个基本的寄存器，而且这些寄存器有特殊用途，不能用来保存运算结果。这就意味着变量、中间结果等都必须保存在内存的数据区中。

此外，栈式虚拟机的最大特点，就在于其将内存的数据区视为一个栈，大量的运算都是针对栈顶的入栈、退栈、计算来完成的，内存的动态管理也是基于栈的。例如，对于下列代码

```
int a = 1;
printf("%d", (a+2)*3);
```




栈式虚拟机的栈变化如上图所示（黄色为栈顶，蓝色为次栈顶）：

1. 为变量a分配内存空间，并初始化
2. 栈顶压入变量a的值（为1）
3. 栈顶压入常数2
4. 栈顶与次栈顶相加，弹出两个元素，并压入运算结果（为3）
5. 栈顶压入常数3
6. 栈顶与次栈顶相乘，弹出两个元素，并压入运算结果（为9）
7. 弹出栈顶，并输出其值（输出9）

可以看到，在栈式虚拟机中，变量都保存在内存中，运算的中间结果也保存在内存中，且操作都是基于栈结构进行的。

**（2）栈式虚拟机的5个寄存器**

栈式虚拟机中只有5个寄存器：PC、SP、MP、NP、EP。这里简单介绍每个寄存器的作用：

- PC：程序计数器，和MIPS中的PC相同，指向下一条将要执行的指令
- SP：栈指针，指向栈顶，便于入栈退栈
- MP：标志指针，指向当前活动记录基地址（活动记录后续会介绍）
- NP：堆指针，指向空闲堆空间起始位置
- EP：顶指针，指向当前程序模块在栈中的最高位置

其中，由于我们的文法不涉及内存的运行时分配（如`new()`），因此实现的栈式虚拟机不需要堆区，也不需要NP寄存器；且可以假定栈式虚拟机有无限大的内存空间，因此也不需要EP寄存器来防止栈溢出。所以，我们真正需要实现的只有PC、SP和MP三个寄存器。

## 2. Pcode内存管理

**（1）Pcode的活动记录**

前面介绍了动态内存管理的相关知识。在实现Pcode的动态内存管理时，活动记录和前面提到的AR结构有些许不同，主要是动态链和静态链两个部分，同时Pcode代码的执行和这两个部分有很大的关系。Pcode动态内存管理一个典型的活动记录AR结构如下图所示。


在栈式虚拟机中，一个活动记录主要包含以下几种数据：

- 返回值，表示该过程执行结束的返回值，可以理解为函数返回值
- 返回地址，表示该过程执行结束后应该返回的指令地址，类似MIPS中`$ra`寄存器的作用
- 静态链Static Link（SL），指向外层活动记录的基地址，可以理解为当前block外层的block的AR基地址。其"静态"体现在可以根据程序源代码直接确定block之间的嵌套关系。
- 动态链Dynamic Link（DL），指向调用者的活动记录的基地址。其"动态"体现在编译时无法确定被谁调用，只有运行时才能确定。
- 参数区，调用过程（函数）时用于传递参数，类似MIPS中`$a`寄存器的作用
- 数据区，包括局部数据区和临时变量区。其中，局部数据区存储定义的局部变量，临时变量区存储运算的中间结果等。

接下来我们重点分析AR中的静态链和动态链的作用。

**（2）静态链SL**

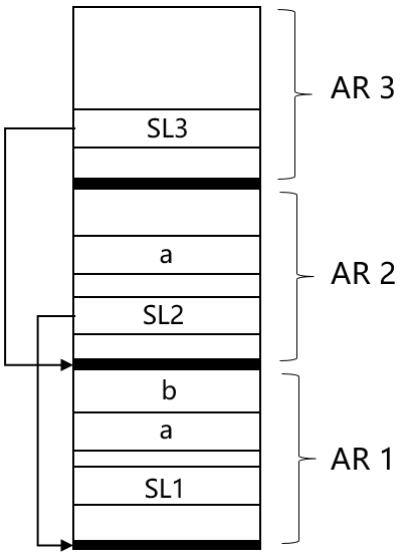
静态链的值是外层活动记录的基地址，作用主要是进行不同作用域下的变量访问。

对于如下代码：

```
int main() {                // block1
    int a = 10, b = 20;
    if(a < b) {             // block 2
        int a = 0;          // 覆盖block1中a的定义
        while (a < 10) {    // block 3
            b = b + 1;      // 引用的block1的b  -------------------------1
            a = a + 1;      // 引用的block2的a  -------------------------2
        }
    }
    printf("%d",b);
    return 0;
}
```

根据静态的程序，如注释所示，我们可以确定三个block之间的嵌套关系，以及每个变量定义的作用域。

如下图所示，在程序执行到block3时，内层中存在3个AR，对应三个block。图中只简单画出了AR的局部变量区和静态链SL，黑色加粗表示AR基地址。




在程序执行到block3中的1处时，此时需要引用变量b，但是变量b在当前AR（AR3）中并没有定义，因此通过SL3找到AR2，发现在AR2中也没有变量b的定义，再次通过SL2找到AR1，在AR1中发现了变量b的定义。事实上，从源程序中也可以发现，引用的确实是block1定义的变量b。在执行到2处时，引用变量a，类似上面的过程，在AR2中找到变量a的定义，说明引用的是此处的变量a。上述过程说明，我们只要沿着静态链找，找到的第一个定义点，就是对应的变量定义。

在这里，我们定义两个重要概念：**level和addr**。level表示活动记录的深度，例如处于AR3中时，变量a的level为1（通过一次静态链搜索即可在AR2中找到定义），变量b的level为2（通过2次静态链搜索在AR1中找到定义）。addr表示变量与其定义AR的基地址的相对偏移。有了这两个概念，我们就可以很方便地来跨活动记录访问变量。例如，当前处于AR3，要访问变量b，则变量b的level为2，通过2次静态链搜索，找到AR1的基地址，再加上变量b的addr（距离基地址的相对偏移），即可确定变量b的绝对地址，访问变量b。

因此，在编译生成代码时，我们对变量的访问只需要给出level和addr两个值即可。注意，**level、addr的值在编译时是可以通过符号表确定的**，其中level可以根据block的相互嵌套关系（或符号表的深度）确定，addr则可以根据该block内的变量大小和数量确定，这也是“静态”的一种体现。

在上面的例子中，我们似乎可以直接确定变量a和变量b的绝对地址，甚至不用通过level、addr和静态链来访问。但是，对于更复杂的情况，如递归调用（`fibo(n)`），编译时，我们无法确定当前是第几次调用，也就无法确定AR的基地址（因为每次调用都要分配一个AR），无法通过绝对地址来访问，但可以通过相对的level和addr来访问。这也是静态链的必要性所在。

**（3）动态链DL**

动态链的作用主要是在函数调用完成后，能够使标志指针MP回到调用前的状态（指向调用者的AR基地址）。因此，调用者在调用函数时，需要将当前MP的值填入函数AR的DL中，函数执行完毕后，将DL的值赋给MP。由于编译时无法预知调用者的AR基地址，因此DL的值在编译时无法确定，只能在程序运行过程中再填入。

最后，建议在实现的过程中，按前面提到的AR结构来实现，预留出返回值、返回地址、SL、DL等区域（即使有些时候不需要），确保AR结构统一。

## 3. Pcode代码介绍

在了解完栈式虚拟机和基于活动记录的动态内存管理后，我们正式介绍pcode代码。pcode代码围绕栈式虚拟机的特点，其操作主要针对栈来进行。下面给出一些供参考的pcode代码，当然你也可以根据自己的需要来自行设计代码。

| 指令名称 | 指令格式 | 说明 |
| --- | --- | --- |
| LIT | LIT imm | 加载立即数imm到栈顶 |
| OPR | OPR opcode | 进行栈顶与次栈顶（或者单独栈顶）的运算，并将结果压入栈顶；opcode决定运算类型 |
| LOD | LOD level, addr | 以栈顶元素为动态偏移，加载内存中`(level, addr)+动态偏移`处的变量值到栈顶 |
| STO | STO level, addr | 以栈顶元素为动态偏移，把次栈顶的元素写入内存中`(level, addr)+动态偏移`处的变量 |
| CAL | CAL label | 函数调用（分配AR、设置SL、DL、参数等） |
| BLKS | BLKS | 开启新的block（分配新的AR，设置SL、DL等） |
| BLKE | BLKE level | 结束当前block（回收当前AR） |
| JMP | JMP label | 无条件跳转到label |
| JPC | JPC label | 栈顶为0时跳转到label，否则顺序执行 |
| JPT | JPT label | 栈顶不为0时跳转到label，否则顺序执行 |
| INT | INT imm | 栈顶寄存器加imm，用于控制栈顶指针的移动 |
| RED | RED | 读入数字到栈顶 |
| WRT | WRT | 输出栈顶数字 |
| WRTS | WRTS str | 输出字符串 |
| LABLE | label: | 标识label，用于跳转 |
| RET | RET | 返回调用者并回收AR |
| LEA | LEA level, addr | 加载(level, addr)处变量的绝对地址到栈顶 |

下面对生成上述代码中的一些注意事项进行讲解。

**（1）内存访问：LOD与STO**

前面我们提到，通过level和addr，即可对内存中的变量进行存取。但是为什么在LOD和STO指令中，我们还需要以栈顶值为动态偏移呢？考虑下面的例子

```
int main() {
    int arr[4] = {1,2,3,4};
    int i;
    i = getint();
    i = i % 4;
    printf("%d",arr[i]);  // ----------------- 1
    return 0;
}
```

尽管我们知道变量arr的(level, addr)，但是由于变量i的值在编译时无法确定，因此需要在运行时将变量i的值加载到栈顶作为动态偏移，再和(level, addr)配合进行数组元素的访问。

其中，1处输出arr[i]的pcode代码示例如下

```
LIT 0
LOD 0, 8    // 加载i到栈顶 i在当前AR中定义，level=0，addr=8，动态偏移为0
LOD 0, 4    // 加载arr[i]到栈顶，arr的level=0， addr=4，动态偏移为栈顶的i
WRT     // 输出arr[i]
```

**(2) Block与AR：BLKS和BLKE**

在sysY文法中，进入一个block相当于进入一个新的过程，需要分配一个活动记录AR；而从一个block退出时，也需要回收其AR。

需要注意的是，对于AR的分配，编译的时候，只需要生成BLKS指令即可，不必填充AR中的SL、DL（编译时也无法确定）；栈式虚拟机在解释执行BLKS指令时，会分配AR并填充上述字段。

通常来说，对于每个block，block的开始通过BLKS（block start）指令分配AR，结束时通过BLKE（block end）指令回收AR。但是，还需要考虑下面几种特殊情况：

```
int func(int a, int b) {  // 函数AR
    int x = a + b;
    if(x < 10) {
        int y = x * a;
        if(y < 10) {
            return y;   // ----------------------1
        }
    }
    return 0;
}
```

在如上的函数中，执行到1处时，分配了3个AR，在执行return返回调用者时，需要一次性回收这三个AR。

```
int main() {
    int i = 10;
    while(i > 0) { // 循环AR
        int a;
        a = getint();
        if(a < 10) {
            int b = a + i;
            if(b > 10) {
                break; // -----------------1
            }
        }
        i = i - 1;
    }
    return 0;
}
```

在如上代码中，进入循环后，执行到1处break时，需要回收循环体的三个AR。（continue同理）

对于这两种情况，建议**在编译过程中记录函数定义block和循环block的深度**，在遇到return/break/continue时，通过block深度确定需要回收多少个AR，从而在return/break/continue之前生成多少条BLKE指令。

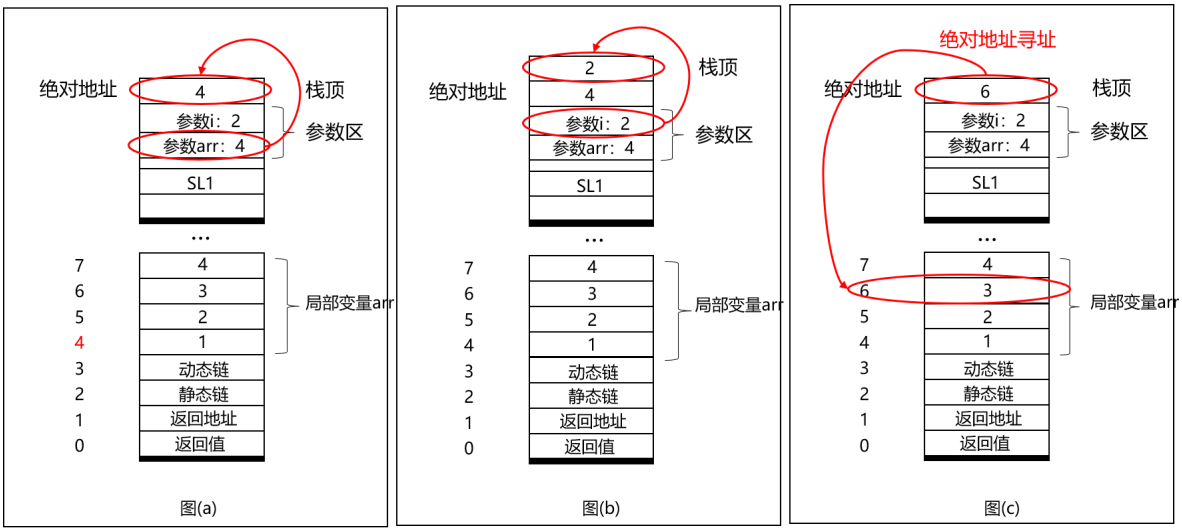
### (3) 传地址的数组存取

在函数中，可能涉及到传地址参数，并以此来进行数组的存取。例如下面例子：

```
void func(int arr[], int i) {
    printf("%d",arr[i]);
}

int main() {
    int arr[4] = {1,2,3,4};
    func(arr,2);
    return 0;
}
```

对于这种情况，建议先读取形参的值作为绝对基地址（图a），然后由绝对基地址+偏移值得到绝对地址存储在栈顶，来进行绝对寻址（图c）。注意，在绝对地址寻址时，可以将LOD/STO指令中的level和addr设置为特殊值以示区分。




下面给出上面例子中`func()`一种供参考的pcode代码。

```
INT 1 // 为形参arr分配空间（参数的值已经由调用者设置好）
INT 1 // 为形参i分配空间（参数的值已经由调用者设置好）
LIT 0
LOD 0, 4 // 加载参数arr的值（level为0，addr为4，动态偏移栈顶为0）
LIT 0
LOD 0, 5 // 加载参数i的值（level为0，addr为5，动态偏移栈顶为0）
OPR 1 // 栈顶与次栈顶相加 得到绝对地址压入栈顶
LOD -1, 0 // 按栈顶绝对地址寻址 level设为-1 addr设为0 表示按栈顶存储的绝对地址寻址
WRT // 输出栈顶元素
RET
```

## 4. Pcode执行程序

除了pcode代码生成，还需要完成一个栈式虚拟机来解释执行生成的pcode。该栈式虚拟机需要包括：代码区（用于存储pcode代码），数据区（栈），三个寄存器（PC、MP、SP）。大部分指令的解释执行只需要按其对栈的操作进行实现即可，这里主要介绍一下函数调用和返回的解释执行过程。

函数调用（CAL指令）的执行过程包括：

1. 准备好若干个参数，存储在栈顶。
2. 为函数分配AR，并设置返回地址，SL（为全局变量所在的AR的基地址），DL（为调用者AR基地址，存储在MP中），参数。其中，实参原本在调用者的栈顶，需要复制到函数AR的参数

区，之后函数只需要通过INT指令来为形参分配空间即可（参数值已经在参数区中存好了）。建议函数AR的起始可以直接覆盖栈顶实参，从而代替参数从调用者AR退栈的操作。
3. 分配好AR后，将MP设置为新的AR的基地址，栈顶指针SP指向参数区起始位置。
4. 设置PC为函数入口指令地址。

其中，第1、2步和BLKS的解释执行过程较为类似。

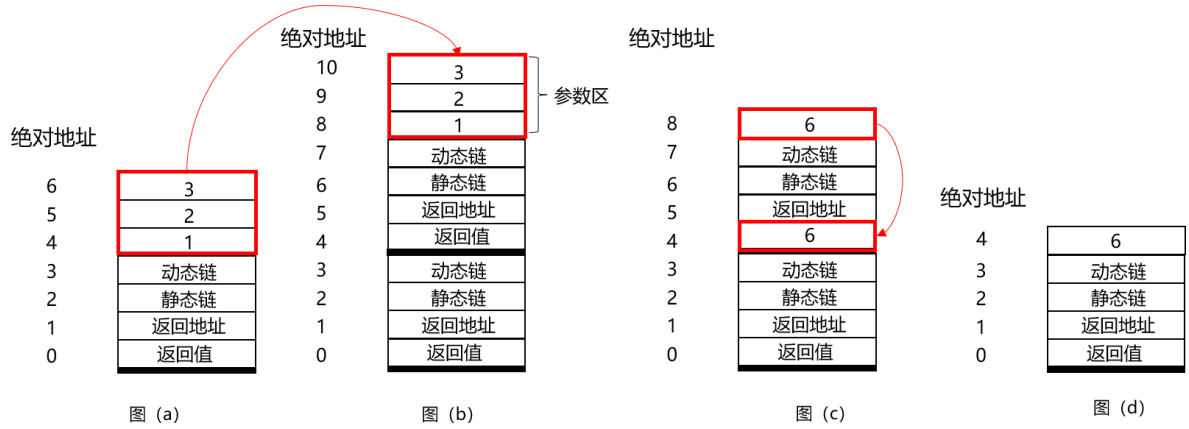
函数返回（主要是RET指令、BLKE指令）的执行过程包括：

1. 如果有返回值，将返回值填入函数第一个AR的返回值区域。注意，return语句可能出现在函数体中的任一个block，但是返回值建议填入函数的第一个AR（也就是CAL指令分配的AR）。
2. 回收所有函数AR（可以通过若干BLKE指令来回收）。
3. 恢复PC（返回地址）、SP（函数第一个AR基地址-1，刚好指向调用者栈顶）、MP（动态链）。如果有返回值，则执行INT 1指令，使SP+1，此时返回值正好处于调用者栈顶。

以下面这个简单的例子为例来进行具体说明：

```
int add(int a, int b, int c) {
    int r = a + b + c;
    return r;
}

int main() {
    int r = add(1,2,3);
    printf("%d",r);
    return 0;
}
```




- 图（a）：main中准备好三个参数，存储在栈顶，准备调用add()
- 图（b）：执行CAL指令，为函数分配AR，设置返回地址、SL、DL，同时把参数填入到参数区，更新PC、SP、MP寄存器。注意，分配的AR可以直接覆盖原本栈顶的参数。
- 图（c）：add()中计算完结果，把结果填入AR中的返回值，准备返回
- 图（d）：执行RET指令，回收CAL分配的AR，恢复PC、SP、MP寄存器。由于有返回值，sp+1，使得返回值6刚好在栈顶，便于后续操作。

## （三）MIPS

中间代码生成后，我们就来到了编译器设计的最后阶段了。目标代码生成通常以编译器此前生成的中间代码、符号表及其他相关信息作为输入，输出与源程序语义等价的目标程序代码。 代码生成模块需要面向某一个特定的目标体系结构生成目标代码，这种目标体系结构可以是 X86、MIPS、ARM等，因此我们可以把目标代码生成理解成是对中间代码的翻译。对于不同体系结构，中间代码翻译成目标代码的处理方式也不同。下面我们给出一个中间代码为四元式，目标代码为MIPS指令结构的指导方案。

## 生成全局数据段

在 MIPS 汇编语言中，全局数据段使用 .data 伪指令来定义。全局数据段的定义应该包括程序中所有全局变量的声明。在生成四元式时，对于每个全局变量，需要记录其标识符、类型、变量名和初始值（如果有的话）。

生成全局数据段的步骤如下：

- 在汇编文件中使用 .data 伪指令来定义全局数据段。
- 对于每个全局变量，生成对应的指令。指令的格式应该根据变量的类型和初始化情况来确定。例如，对于 int 类型的变量，可以使用 .word 伪指令来分配 4 字节的空间。对于printf中的字符串，可以使用.asciiz来分配空间。

例如对于以下代码：

```
int a=2,b[10]={1,2,3,4,5,6,7,8,9,10},c;
int main(){
    printf("ssss");
}
```

我们可以生成以下MIPS代码

```
a: .word 2
b: .word 10,9,8,7,6,5,4,3,2,1,
c: .word 0
str0: .asciiz "ssss"
```

这里需要注意的是，在MIPS代码生成过程中，我们直接使用.word来给变量赋值，对于数组元素，其值的顺序与下标顺序正好相反。对于未赋初值的变量，我们默认给其赋值为0。

通过生成全局数据段，我们可以在程序运行之前为所有的全局变量分配内存空间，并在必要时将它们初始化为指定的值。这样，在程序的执行过程中，全局变量的值就可以被保存在内存中，并随时被读取和修改。

## 生成全局代码段

全局代码段包括了所有的全局函数，以及全局变量的初始化代码。在 MIPS 汇编语言中，全局代码段使用 .text 伪指令来定义。在生成四元式时，需要将每个全局函数转化为 MIPS 汇编语言的形式，并生成相应的代码段。对于全局变量的初始化代码，可以将其转化为 MIPS 汇编语言中的常量池，以便在程序运行时进行初始化。

#### 寄存器分配

在不考虑优化的情况下，完全可以将所有变量都存储在内存中，寄存器仅暂存变量的值， 操作结束后将结果写入内存中的相应位置。因此，对于不考虑优化的代码生成部分来说，我们至多只需使用4个寄存器便可完成，而将最后的计算结果存入内存之中，我们可以通过循环利用的方式使用这些寄存器，例如，对于以下代码：

```
int a=1,b=2;
int c=a+b;
```

我们可以将mips代码写成这样：

```
li $t1, 1    //a=1
sw $t1, 0($sp)

li $t2, 2     //b=2
sw $t2, -4($sp)

lw $t3, 0($sp)   //c=a+b
lw $t0, -4($sp)
addu $t1, $t3, $t0
sw $t1, -8($sp)
```

我们需要在代码生成的过程中维护一个符号表，对于a=1这段代码，我们先用一个临时寄存器记录值，然后将其存入到内存的相应部分，并在符号表中记录它的位置，b=2同理，当运行到c=a+b时，我们会去符号表中找到存放a和b值的位置，并将其取出至临时寄存器中，进行计算，在计算结束后获得c的值，并继续存到相应内存中。这边需要注意的是，如果c并不是第一次被定义，那么应该从符号表中找到c最初被定义的地方，并将值存到上述地点。这些临时寄存器的值没有必要被长久保存，因此在使用后可以立即释放。

在代码生成时，我们不考虑复杂的寄存器分配方法，大家只要学会最基本的寄存器使用即可，后续在优化部分我们会着重讲述寄存器的分配方法。

**数组的处理**

不同类型数组在存储空间中的分配方案分别为：

- 如果数组是全局数组，那么它里面的各个值依次排列在全局数据区
- 如果数组是函数中定义的局部数组，则数组中的值依次排列在对应函数的活动记录中
- 如果数组是参数数组，那么活动记录中记录的是数组的基地址

需要注意的是，为数组分配存储空间时不特别区分一维数组与二维数组：它们的区别仅仅 在于访问数组元素时的偏移计算方法有差异。 我们将与数组相关的操作分为两种：数组存取，数组地址传递。

- 数组存取：当从某个数组中获取值或者向某个数组中存储值时，首先判断这个数组是全局数 组还是局部数组，然后获得其相对于静态数据区（或者当前活动记录基地址）的偏移，与 $gp$（或fp）求和，获得数组基地址。然后根据访问数组的位置，求出相对于数组基地址 的偏移。最后用lw或者sw访存。
- 数组地址传递：仅出现在函数调用时。 当调用某个函数时，如果这个函数需要一个数组作 为参数，则需要把某个全局数组或者局部数组或参数数组的地址传入。若需要传入一个非参 数数组的地址，那么可以在编译时确定数组相对于静态数据区或当前活动记录的偏移，从而 与 $gp$或fp求和，获得数组地址并传入。如果是参数数组，那么从活动记录中获取到的数 组值已经是数组地址，直接传入即可。如果需要传入二维数组中的某个一维数组的地址，那 么还需要加上偏移（动态计算）。

例如，对于以下代码

```
int a[2]={1,2};
a[1]=3;
```

我们可以如此操作

```
li $t1, 1  //第一行
li $t2, 0
sll $t2, $t2, 2
subu $t3, $fp, 0
subu $t3, $t3, $t2
sw $t1, ($t3)
```

```
li $t0, 2
li $t1, 1
sll $t1, $t1, 2
subu $t2, $fp, 0
subu $t2, $t2, $t1
sw $t0, ($t2)

li $t3, 3   //第二行
li $t0, 1
sll $t0, $t0, 2
subu $t1, $fp, 0
subu $t1, $t1, $t0
sw $t3, ($t1)
```

在定义数组时，我们首先找到该值对应地址的偏移量（相对于fp指针位置），在该位置赋以相应的值，并在符号表中记录。在需要使用该数组对应下标的值时，仅需要从符号表中找到其相对于fp指针的偏移量，并从内存中读出即可。当然，不要忘记在最后将值重新存到内存的相应部分。

**短路求值**

对于 if 条件分支语句的 && 短路， 可作如下变换：

```
// 变换前
if (a && b) {
    xxx
} else {
    xxx
}
// 变换后
if (a) {
    if (b) {
        xxx
    } else {
        xxx
    }
} else {
    xxx
}
```

对于 if 条件分支语句的 || 短路，可作如下变换：

```
// 变换前
if (a || b) {
    xxx
} else {
    xxx
}
// 变换后
if (a) {
    xxx
} else {
    if (b) {
        xxx
    } else {
        xxx
    }
```

```
}
```

若有 && 和 || 嵌套，可以按照符号的优先级将其拆成多个判断语句来处理。解析时遵循以下几点：

- 对 LAndExp 中每个 RelExp 进行判断：若该RelExp为真，则往后判断下一个 RelExp；若 为假，则直接跳转到当前 LAndExp 的尾部
- 对 LOrExp 中的每个 LAndExp 进行判断：若该LAndExp为真，则直接跳转进入到 if 语句 块内部；若为假，则往后判断下一个 LAndExp
- 若所有的 LOrExp 判断完毕后还没发生跳转，说明不满足判断条件，此时跳转到 if 语句块 尾部（或 else 语句块首部）

如下图所示代码

```
if (a || ( b && c ) || d) {
    xxxx
}
```

可以修改为：

```
if (!a) {
    goto IF_OR_1
}
goto IF_BEGIN

IF_OR_1:
if (!b) {
    goto IF_OR_2
}
if (!c) {
    goto IF_OR_2
}
goto IF_BEGIN

IF_OR_2:
if (!d) {
    goto IF_OR_3
}
goto IF_BEGIN

IF_OR_3:
goto IF_END

IF_BEGIN:
    xxxx
IF_END:
```

我们一般采用如此方法进行短路求值。

**函数调用**

在四元式生成Mips汇编代码时，函数调用部分需要考虑以下几个步骤：

1. 函数调用的参数传递 在调用函数之前，需要将函数的参数压入栈中，以便函数内部可以访问到这些参数。参数的压入顺序一般是从右到左，即先将最后一个参数压入栈底，最后将第一个参数压入栈顶。

2. 保存当前函数的现场 在调用函数之前，需要将当前函数的现场保存起来，以便在函数返回时可以恢复。保存现场的主要内容包括程序计数器PC、栈指针SP和寄存器。
3. 跳转到函数入口地址 调用函数时，需要将程序计数器设置为被调用函数的入口地址。
4. 函数返回时的处理 在被调用函数返回时，需要将返回值存储在合适的寄存器中，并将调用函数的现场恢复。

例如，对于最简单的函数调用，f(1)，我们使用如下代码mips进行执行：

```
sw $ra, 0($sp)
subu $sp, $sp, 4

# PUSH:push 1
li $t1, 1
sw $t1, 0($sp)

# CALL:call f
move $fp, $sp
jal f

addiu $sp, $sp, 4
move $fp, $sp
lw $ra, 0($sp)
move $t4, $v0
```

首先第一步，将存到ra寄存器中存的返回地址存入内存，防止其被覆盖，第二步，将参数存到合适的地方以便接下来使用，第三步，调用函数，第四步，结束函数调用，恢复现场（取出函数调用前保存下来的值）并取出相应的函数返回值。

一般来说，$ra$和fp 这两 个寄存器是函数调用时必须保存的寄存器，需要压入栈中。而其他寄存器是否需要保存，并没有标准答案。保存的寄存器越多，会带来更高的安全性，但同时造成的时间开销也会越大。具体需要保存和恢复哪种类型的寄存器，保存寄存器的数目是否要受到限制，这些需要同学们兼顾正确性和效率自行设计。

# 五、代码优化

## （一）初识SSA

首先欢迎各位同学勇于突破自我，来到编译器优化的部分！本部分优化教程适用于以SSA形式语言作为中端的编译器，例如课程组推荐的中端语言llvm。我们将从了解什么是SSA开始，到构建SSA，再到后面以SSA为基础的各种优化。编译器优化的道路可能充满各种bug与未知的困难，但是在完成一个又一个优化的过程中，你一定会充满成就感！经过mem2reg，你的代码将被去除掉大部分alloca和load指令；经过DCE(Dead Code Elimination)，你的代码会减少很多废话文学；经过func_inline，大部分程序经过你的编译器可能只剩下了一个函数；经过GVN&GCM，你将灵活的重排指令，创造更多可优化的空间…… 不过，万丈高楼平地起，我们先从初识SSA入手，进入中端优化的世界。

### 概述

本章看完你会了解：

- SSA的基本概念
- 非SSA转化为SSA的基本方法
- $\phi$运算与它的一些性质

## 什么是SSA

SSA的全写，即Static Single-Assignment（静态**单赋值**）。静态单赋值的含义为**每个变量在程序中只被赋值一次。**如下列程序便不是SSA程序：

```
x = 1;  //  语句1
y = 2;  //  语句2
x = y;  //  语句3
```

'='作为赋值操作，左边的变量会赋上右边的值。上述代码中的语句1为x赋值1，语句3为x赋值y，这段程序对x进行了两次赋值，即违反了SSA的定义。显然，我们删除语句1或语句3，就可以让这段程序成为一段SSA代码。

在一段正常的c代码中，可能会出现很多非SSA的现象，如：

```
x = 1;      //  语句1
y = x + 1;  //  语句2
x = 2;      //  语句3
z = x + 1;  //  语句4
```

上述的代码中x变量被赋值了两次（语句1和语句3），这会使得很多优化很难进行。如果我们只看语句2和语句4，我们很容易会认为y和z值是一样的，均为x + 1。但是由于x中途被重赋值了，导致这个想法是错误的。如果我们能把程序改写成SSA形式，如下：

```
x = 1;          //  语句1
y = x + 1;      //  语句2
x_2 = 2;        //  语句3
z = x_2 + 1;    //  语句4
```

这里我们用了一个新变量x_2来替代x的重赋值操作，同时将所有后续用到x的地方均换成了x_2。这样的话，我们就会明显的看出y和z的值大概率是不一样的，使得优化过程不会因此出错。

这样的一个小小的例子应该能让大家了解到SSA形式程序的好处，也展示了一个将非SSA程序转换成SSA程序的方法。但实际的情况可能比上面的程序复杂的多，如：

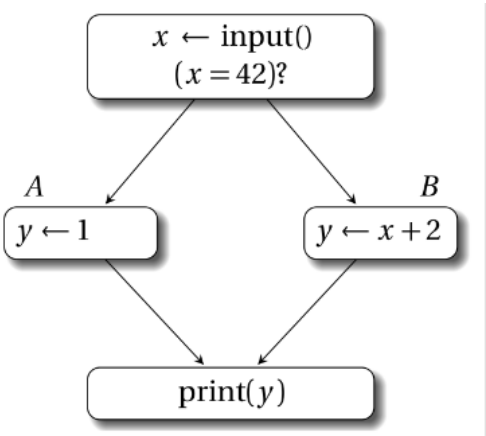
```
x = input()
if(x == 42)
    y = 1;
else
    y = x + 2;
print(y)
```

上面的伪代码描述了一个存在分支的情况。这样的程序能不能够被叫做SSA程序呢？答案是不能。因为显然，按照SSA的定义，y变量在程序中被赋值了两次（尽管这两条语句不可能都被执行）。

那么我们应该怎么将这种程序转化为SSA呢？如果我们仍采用上述的命名方法，产生y_2来替代第二次命名，那么最后的print(y)应该用y还是y_2呢？我们直观的一个想法就是，如果有一种运算能够智能的判断选择y还是y_2就好了。没错，因此我们需要引入新的运算方法，$\phi$运算。

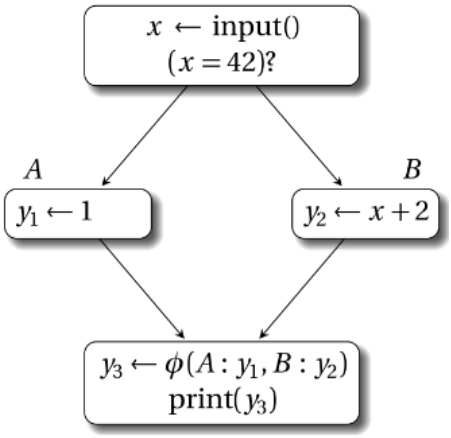
## $\phi$运算

还是用上面的例子，我们画出上述程序执行的流程图（后续我们会将其叫做CFG图）：




该程序目前无法转为SSA的关键点就在于，print(y)的时候，我们无法预先知道y会取左右两边哪个值。由此我们定义$\phi$运算，如果我们是从左边的基本块来的，就用左边的值；如果我们是从右边的基本块来的，就用右边的值。我们将这种**根据来源基本块来决定给这个变量赋什么值**的操作定义为$\phi$运算。

添加$\phi$运算后的流程图如下：




图中给左右基本块编号为A，B，$\phi$运算中的A : y1表示如果我们是A块来到这的，那么我们给y3赋值y1;同理B : y2则表示如果我们是B块来到这的，那么我们给y3赋值y2。在后面的大部分情况中，出于简洁，我会简化$\phi$函数的书写形式（去除掉基本块标号以及冒号，来源基本块和对应取值按从左到右的顺序）。还有一点需要注意，$\phi$运算要放在基本块的首部，也就是说SSA化后的基本块的语句应该是先有一批$\phi$语句，随后跟着一堆非$\phi$语句的格式。

由此我们便实现了含有分支数据流的SSA转化。事实上，我们针对SSA的大部分讨论将围绕$\phi$运算展开。

## 多条SSA

上个例子是最基本的$\phi$函数使用的例子，实际情况仍然要比上述复杂的多($\phi$在下面代码块中用phi替代了)：

```
y_3 = phi(y_1, y_2)     //  语句1
y_5 = phi(y_3, y_4)     //  语句2
```

在上述代码中，上述语句2中的y_3应该取什么值（语句1运算前的值还是语句1运算后的值）？

答案是语句1运算前的值。一个直观的理解是每个$\phi$语句是独立且平等的，这里的顺序只是一种方便记录的形式。每条$\phi$语句都对应了一组数据流信息，而在进入这个基本块之前，这些数据流是不互相影响的，因此在一个基本块的$\phi$语句也应该保持这种独立性，同时执行，或者说是**并行执行**。

因此一定要记住多条$\phi$语句逻辑上要并行执行。至于我们怎么用串行模拟实现这种并行性，是我们之后需要讨论的。

### 小结

至此，你已经了解了SSA的基本知识。在后续的指导中，我们将逐渐深入讲解构建SSA和有关SSA的相关优化。有兴趣的同学可以按照以上提出的概念，自己先尝试将程序转成SSA形式，当然，后续我们也将讲解构建SSA的算法——mem2reg。

## （二）Mem2reg

> 本部分教程参考*Static Single Assignment book*，将其关键内容整合成指导书，如有困惑推荐参考原书

### 概述

这一章我们将介绍如何将一个非SSA的形式转化到含有phi函数的SSA形式。首先看下面的一个例子（仅作参考）：

```
//  源语言
int main() {
    int x, cond = 1;
    if (cond > 0){
        x = 1;
    }
    else{
        x = -1;
    }
    return x;
}
```

```
//  未经mem2reg的llvm
define i32 @main() {
    %1 = alloca i32
    %2 = alloca i32
    store i32 1, i32* %1
    %3 = load i32, i32* %1
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %8

5:
    store i32 1, i32* %2
    br label %6

6:
    %7 = load i32, i32* %2
    ret i32 %7

8:
    %9 = sub i32 0, 1
    store i32 %9, i32* %2
    br label %6
```

```
}
```

```
//  经过mem2reg的llvm
define i32 @main() {
    %1 = icmp sgt i32 1, 0
    br i1 %1, label %2, label %5

2:
    br label %3

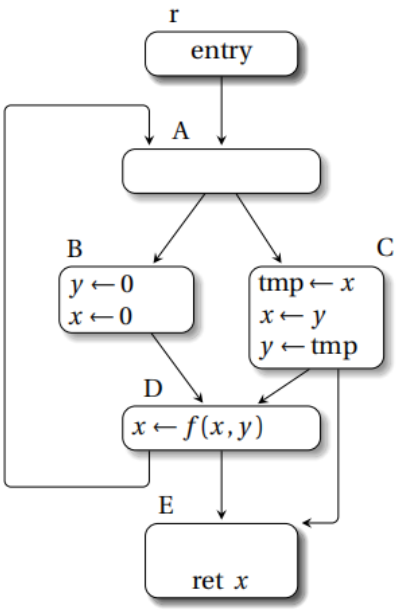
3:
    %4 = phi i32 [ 1, %2 ], [ %6, %5 ]
    ret i32 %4

5:
    %6 = sub i32 0, 1
    br label %3
}
```

回忆上次的内容，我们在介绍SSA时提及的两个主要的方法，一个是改名，一个是引入phi函数。我们这里也将主要按照这样的方法对SSA形式进行构建。SSA构建的标准算法包含两个过程：插入$\phi$ 指令和 变量重命名。

## 1. phi insertion

在这一个步骤里，我们会把构建SSA所需要插入的phi指令都一口气先插到里面。当我们遇到不同基本块汇集（converge）到同一个基本块的时候，便需要将phi指令插入到基本块中。看下面的例子：




我们将这种展示程序流程的图叫做控制流程图（**C**ontrol **F**low **G**raph），简记为**CFG**。上图中，B, C基本块汇聚到D，这导致我们无法确定y此时的取值来自于B还是C，只好插入phi函数实现SSA。一个直观的想法就是所有这种汇聚的基本块中都需要插入phi函数。用更加专业的语言来定义一下汇聚这个概念：

- 假设基本块n1和基本块n2是CFG中两个独立的节点，n3为另一个基本块节点
- CFG中存在两条路径n1->n3，n2->n3
- 这两条路径有且仅有n3这一个交点
- 那么我们定义n3是n1和n2的**join node**

定义完join node的概念后，我们便可以粗略的理解为：若n3 = join node(n1, n2)，且n1,n2定义了同一个变量v，那么我们要在n3中插入一个针对变量v的phi指令。但实际情况可能复杂的多，可能存在n3与其他基本块又汇聚回n1之类的情况，因此单一的join node概念肯定是不够的，我们拓广一下，提出join sets:

- 设S为一个基本块集合
- 当n为S中至少两个基本块n1, n2的join node时，我们称n为S的join node（注意此时S是一个集合）
- 我们将S的所有join node汇总起来，称为**join set**，记作J(S)

到这里，我们提出了一些新的概念来辅助我们找到将phi函数插到哪里。大家可能已经被绕晕了，我用上面图中的例子再具体化一下：

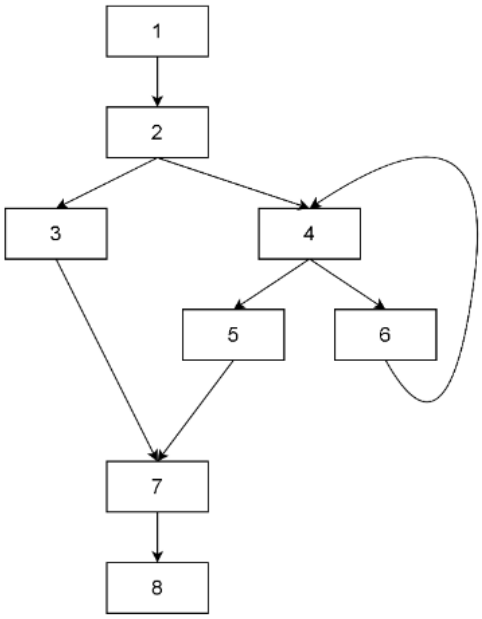
- $D = join\ node(B, C)$
- 令 $S_1 = \{B, C\}$, 则 $D \in J(S_1)$，且 $J(S_1) = \{D\}$
- 令 $S_2 = \{r, A, B, C, D, E\}$，则 $J(S_2) = \{A, D, E\}$

在上面的例子中，变量y在B,C中被定义(def)，然后我们发现 $J(\{B, C\}) = \{D\}$ ,因此在D中要构建phi指令。那么抽象出来就是对于一个变量v，我们先找到它的定义块集合 $Def_v$，简记为 $D_v$，然后再计算 $J(D_v)$，得到的即为我们应该插入phi指令的基本块集合。注意到，phi指令本身，也是对变量v的定义，但由于 $J(D_v \cup J(D_v)) = J(D_v)$，所以我们最终需要插入phi指令的基本块集合**还是** $J(D_v)$ 。

似乎问题到这里已经解决了，我们只需要遍历每个变量，计算出 $J(D_v)$，然后插入phi指令就好了，但是实际上求 $J(D_v)$ 的过程非常复杂。计算机科学家结合了图论中的一些概念，将问题转化的更好求解。下面介绍一下相关概念：

- **支配**(dominate)：如果CFG中从起始节点到基本块y的所有路径都经过了基本块x，我们说**x支配y**
- **严格支配**(strict dominate)：显然每个基本块都支配它自己。如果x支配y，且x不等于y，那么**x严格支配y**
- **直接支配者**（immediate dominator, idom）：严格支配n，且不严格支配任何严格支配 n 的节点的节点(直观理解就是所有严格支配n的节点中离n最近的那一个)，我们称其为n的直接支配者

下面是一个CFG：




我们可以举一些例子阐述一下上述的概念：

- 1严格支配3：因为程序如果要运行到3号基本块，必须要经过1号基本块（支配），且1!= 3（严格支配）。
- 3不严格支配7：因为程序如果要运行到7号基本块，可以不经过3，可以从4->5->7的线路走
- 7的直接支配者是2：依照上述概念，有1，2严格支配7，但显然2更近。严谨地说，1虽然严格支配7，但1还严格支配2（7的严格支配者），因此不符合上述概念，不是7的直接支配者。

需要注意的是，**每个节点的直接支配者有且只有一个**（除去entry节点）。

**支配边界**（**D**ominance **F**rontier）：节点 n 的支配边界是 CFG 中刚好**不**被 n 支配到的节点集合. 形式化一点的定义是：$DF(n) = \{x \mid n支配x的前驱节点，n不严格支配x\}$ .

我们还是举个例子辅助理解：CFG图中沿着节点4向下走，5是被4支配的（如果程序要进行到节点5一定经过4），而接着到7节点的时候，我们发现4不再支配7了。由此我们便说4的支配边界是7。

类似操作，我们对集合也可以定义DF操作，$DF(S) = \bigcup_{s \in S} DF(s)$ 。DF还可以进一步迭代，即

$$DF_1(S) = DF(S)$$
$$DF_{i+1}(S) = DF_i(S)$$
$$DF^+(S) = DF_{i->\infty}$$

还记得我们想要求的 $J(Def)$ 嘛？事实上:

$$DF^+(S) = J(S \cup entry)$$

这里的entry是程序的入口基本块。在SSA原文默认的程序中，我们需要在entry基本块中定义所有使用的变量，因此在没有entry基本块的基本块集合中，会有可能出现变量未被定义的情况。而在我们课程组推荐的llvm架构中，没有这样纯定义变量的entry块，因此可以大致理解为 $DF^+(S) = J(S)$ 。

那么任务就很明确了，对于每个变量v，求出 $DF^+(Def_v)$ 就知道了需要插入phi函数的位置。

首先，我们先计算出CFG图。

> 这个根据同学们自己的中端架构来实现，其实就是根据基本块跳转指令来记录每一个基本块的next和prev基本块们就可以

接下来，根据CFG图计算支配关系。

> 推荐的一个方法是迭代计算。按照"某基本块的dom <- 某基本块所有前驱的dom的交集加上自己本身" 的策略进行更新，直到该基本块的dom集合不发生变化

在得到基础的支配关系后，直接支配关系和严格支配关系可以直接按照定义计算出来。

> 按照定义即可

接下来，计算最重要的支配边界，这里给一个计算CFG中每个节点的支配边界的伪代码：

**Algorithm 3.2:** Algorithm for computing the dominance frontier of each CFG node.

```
for (a, b) ∈ CFG edges do
    x ← a
    while x does not strictly dominate b do
        DF(x) ← DF(x) ∪ b
        x ← immediate dominator(x)
```

上个步骤我们计算出了 $DF(S)$，下面我们提供一个参考插入phi函数的伪代码，我们的迭代支配边界也将在这个过程被计算出来：

**Algorithm 3.1:** Standard algorithm for inserting $\phi$-functions

```
for v: variable names in original program do
    F ← {}                          ▷ set of basic blocks where φ is added
    W ← {}                          ▷ set of basic blocks that contain definitions of v
    for d ∈ Defs(v) do
        let B be the basic block containing d
        W ← W ∪ {B}
    while W ≠ {} do
        remove a basic block X from W
        for Y: basic block ∈ DF(X) do
            if Y ∉ F then
                add v ← φ(...) at entry of Y
                F ← F ∪ {Y}
                if Y ∉ Defs(v) then
                    W ← W ∪ {Y}
```

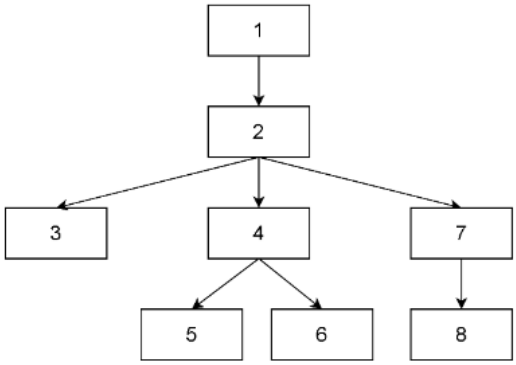
> 上述伪代码的思路很直接。对于某个变量v，首先W即 $Def_v$，对于其中的每个X，找到X的支配边界，由于它的支配边界不由X支配，即有可能来自于其他基本块，因此要插入phi指令（即如果从X过去，变量v就用X中定义的值）

在程序中插入 phi 函数后，程序中地每个变量依然会有多个定义，但程序中的每个定义都可以确定地到达对它的使用，在程序的任意一个位置，一定能够唯一地找到一个已定义变量的定义点，不会出现“这个变量可能在这里定义，也可能在那里定义”的情况（这种情况变成了 phi 函数定义了该变量，kill 了原有的多个定义）。

## 2. variable renaming

为了使程序变为 SSA 形式，我们还需要对变量进行重命名。在程序中插入 phi 函数使得每一个变量的存活区间（live-range）被切割成了几片，变量重命名要做的就是给每个单独的存活区间一个新的变量名。

在上文计算支配关系的时候，基本块之间的直接支配关系可以构成如下图的支配树（dom tree）：




在支配树上进行 DFS，DFS 的过程中，计算并更新每个变量 v 当前的定义 v.reachingDef，并创建新的变量。算法如下：

**Algorithm 3.3:** Renaming algorithm for second phase of SSA construction

▷ *rename variable definitions and uses to have one definition per variable name*

```
1  foreach v : Variable do
2      v.reachingDef ← ⊥
3  foreach BB: basic Block in depth-first search preorder traversal of the dom. tree do
4      foreach i : instruction in linear code sequence of BB do
5          foreach v : variable used by non-φ-function i do
6              updateReachingDef(v, i)
7              replace this use of v by v.reachingDef in i
8          foreach v : variable defined by i (may be a φ-function) do
9              updateReachingDef(v, i)
10             create fresh variable v′
11             replace this definition of v by v′ in i
12             v′.reachingDef ← v.reachingDef
13             v.reachingDef ← v′
14     foreach φ: φ-function in a successor of BB do
15         foreach v : variable used by φ do
16             updateReachingDef(v, φ)
17             replace this use of v by v.reachingDef in φ
```

**Procedure** updateReachingDef(v,i) Utility function for SSA renaming

**Data:** $v$ : variable from program
**Data:** $i$ : instruction from program

▷ *search through chain of definitions for v until we find the closest definition that dominates i, then update v.*reachingDef *in-place with this definition*

```
1  r ← v.reachingDef
2  while not (r == ⊥ or definition(r) dominates i) do
3      r ← r.reachingDef
4  v.reachingDef ← r
```

# （三）Function Inline(函数内联)

# （四）DCE(死代码删除)

## 概述

程序包含的一些代码可能并不会被运行或者不会对结果产生影响，那么我们称这种代码为死代码。我们将不会被运行到的称为**不可达代码**，将不会对结果产生影响的代码成为**无用代码**。删除无用或不可达代码可以缩减IR代码，可使程序更小、编译更快、执行也更快。

以如下的代码为例：

```
int main() {
    int a = 24;
    int b = 25; /* 无用代码 */
    int c;
    c = a << 2;
    return c;
    b = 24; /* 不可达代码 */
    return 0;
}
```

基于SSA的形式，我们很容易做死代码删除。整体思路为沿着def-use链将有用的代码全部找出来标记，这样没有被标记过的代码就是死代码。但是要注意的是我们删除代码的同时会改变控制流图等一系列结构，因此我们要在做删除的同时谨慎维护原有的数据结构。

## 实现

我们提供的一个可行的思路如下：

#### 删除无用函数

这一部分主要是遍历所有函数，删除没有被用到的函数。函数之间的调用关系可以通过向函数类添加一些属性来维护，例如每个函数维护自己调用过的函数与自己被哪些函数调用等。

#### 删除死代码

这一部分需要遍历有用函数中的每条指令，并沿着def-use链构建有用指令的**闭包**。最基本的一些有用指令包括br指令，ret指令，调用库函数的call指令等。

寻找闭包的过程其实是一个递归：先将基本的有用指令进行标记，随后标记每条有用指令用到的value，并接着递归寻找其用到的value，从而构建一个有用指令的闭包。该思路仅供参考，由于SSA中def-use提供了很丰富的指令之间的逻辑关系，因此可能对于不同的架构有不同的最优实现方案。

#### 维护数据结构

删除指令的时候，我们要注意维护之前构建的一些数据结构，如指令之间的def-use关系，basicblock一些相关的属性等等，根据不同的架构进行调整。

理论上，我们删除了除有用指令外的所有指令，易发现无用代码和不可达代码都已经被删除了。

### 拓展

我们这里介绍的只是较为基础版的死代码删除。一些可行的拓展方案如下：

- 进行**内存分析**，删除一些无用的store，load指令：例如有时我们会在load之前多次对同一个地址进行store，那么显然只有最后一个store是有效的
- 合并冗余的分支指令

另外在进行后续的常量传播等优化后，可能程序会暴露出新的可以做死代码删除的机会，因此可以合理安排优化的顺序与次数，运行多次死代码删除。

## （五）GVN&GCM(全局值标号&全局代码移动)

### 概述

GVN(Global Variable Numbering) 全局值编号：为全局的变量进行编号，实现全局的消除公共表达式。

GCM(Global Code Motion) 全局代码移动：根据Value之间的依赖关系，将代码的位置重新安排，从而使得一些不必要（不会影响结果）的代码尽可能少执行。

附官网论文链接：https://c9x.me/compile/bib/click-gvn.pdf

### GVN

GVN 寻找具有相同操作和相同输入的代数恒等式或指令，并为他们赋值为相同的编号。 GVN 的过程中也可以做折叠常量。我们通过查找哈希表来判断两条指令是否相同，因此GVN 在很大程度上依赖于SSA形式。 GVN 用一条指令替换多组等价指令，但是由于是全局性的编号，可能会导致代码结果不正确，所以GVN往往都**与GCM配套使用**，在GCM的过程中根据value之间的依赖关系重新调整指令顺序，保证GVN计算结果的正确性。

#### LVN

在正式介绍GVN前，我们先了解一下LVN(Local Variable Numbering)，也就是局部值编号，效果基本等同于教材上的公共子表达式消除。以下述的程序举例：

```
... define a, b
c <- a + b
d <- c - b
e <- a + b
f <- e - b
```

在上述的代码中，我们注意到c和e的操作数一致，操作符一致。我们可以把上述代码中的c和e取同一个值，不妨将所有的e换成c，即优化为：

```
... define a, b
c <- a + b
d <- c - b
f <- c - b
```

上述的优化又暴露了新的优化机会：d和f可以取相同的值，由此可以删除f的指令并将后续所有用到f的地方换成d。

```
... define a, b
c <- a + b
d <- c - b
```

按照上述的思想，我们构建了LVN算法。LVN算法的流程很简单：

- 对于每个basicblock初始化一个哈希表
- 对每条指令，获取它的操作数和操作符，根据操作数和操作符计算哈希值

> 这一步请同学自行设计放入的结构。只要能保证相同的指令在哈希表里也相同（注意有的运算符合交换律，要将不同顺序的情况也放入哈希表）

- 如果哈希表中已经存在该值，那么我们直接将basicblock里后续用到这个value的地方全部换成哈希表中存的value
- 否则将哈希值及对应的value存入哈希表

**常量折叠**

注意到LVN的过程中，我们可以顺便做常量折叠，这不仅节省了一个pass的时间，同时常量折叠也会暴露更多的LVN优化机会。

常数表达式计算（constant-expression evaluation），或称常数折叠（constant folding），指的是在编译时计算其操作数已知是常数的表达式。在多数情况下这是一种相对容易的转换。最基础的一个常数折叠操作是：在遍历指令的过程中，判别指令的所有操作数是否为常数值，并用计算结果替代该表达式。

回到 LVN 框架，如果一个运算的所有运算对象都具有已知的常数值，那么 LVN 可以在编译时执行该运算并将结果直接合并到生成的代码中。LVN 可以在哈希表中存储有关常数的信息，包括其值。如果 LVN 发现一个常量表达式，它可以将表达式替换为对相应结果的立即数加载操作。

更复杂的一些折叠操作涉及到代数恒等式：

- a + 0 = a
- a - 0 = a
- a * 0 = 0
- 2 * a = a + a
- a * 1 = a
- a / 1 = a
- a / a = 1

还可以利用二元运算的结合律进行更复杂的折叠：

- a + (b - a) = b
- a - (a - b) = b
- ...

这一部分可以尽情发挥大家的想象力，但注意一定要在保证准确率的前提下去做复杂的折叠优化。

**GVN**

跟 LVN 类似，GVN 的本质是基于哈希的操作，它尝试用一条指令处理一组重复，或者说冗余的指令（它们计算相同的值），根据具体情况不同，将其替换或者作为新值加入哈希表中。类似 LVN，GVN 的架构也是有利于常量折叠的，像其他表达式一样，GVN 会对常量表达式进行值编号。在这里，我们通过哈希表来查找具有相同操作和相同输入的两个指令。

但与LVN不同的关键点在于，GVN是一个针对全局的哈希表。但显然全局的值标号会产生bug，这里我们会在GCM时进行纠正。

## GCM

全局代码移动算法的功能是调度那些"浮动"的指令，把它们归位到一个个基本块中。这个过程必须保留已有的控制依赖和数据依赖。在这个大前提下，算法可以自由发挥，我们的目标就是尽可能把代码移动到循环外面，尽可能让代码的执行路径变少。它的执行步骤大致如下所示：

- 计算支配树与支配关系
- 利用循环分析计算循环深度
- 调度所有指令

前两个步骤都是为最后一个步骤服务的，调度指令分为几个流程：find_Pinned_Insts，schedule_Early，schedule_Late，select_block

### find_Pinned_Insts

有些指令是无法被灵活调度的，如phi，br/jump，ret等指令，这些指令我们叫做pinnedInst。pinnedInst受到控制依赖的牵制，无法被调度到其他基本块，也就是说这些指令和它们的基本块是绑定的。这一部分，我们需要先遍历指令将pinnedInst进行标记。

### schedule_Early

顾名思义，这一步骤我们会尽可能的把指令前移，**确定每个指令能被调度到的最早的基本块**，同时不影响指令间的依赖关系。当我们把指令向前提时，限制它前移的是它的输入，即每条指令最早要在它的所有输入定义后的位置。伪代码如下：

```
forall instructions i do
  if i is pinned then          // Pinned instructions remain
    i.visit := True;           // ... in their original block
    forall inputs x to i do    // Schedule inputs to pinned
      Schedule_Early( x );     // ... instructions

// Find earliest legal block for instruction i
Schedule_Early( instruction i ) {
  if i.visit = True then       // Instruction is already
    return;                    // ... scheduled early?
  i.visit := True;             // Instruction is being visited now
  i.block := root;             // Start with shallowest dominator
  forall inputs x to i do      //
    Schedule_Early( x );       // Schedule all inputs early
    if i.block.dom_depth < x.block.dom_depth then
      i.block := x.block;      // Choose deepest dominator input
}
```

### schedule_Late

同理，这部分我们会尽可能的把指令后移，**确定每个指令能被调度到的最晚的基本块。**每个指令也会被使用它们的指令限制，限制其不能无限向后移。伪代码如下：

```
forall instructions i do
  if i is pinned then          // Pinned instructions remain
    i.visit := True;           // ... in their original block
    forall uses y of i do      // Schedule pinned insts outputs
      Schedule_Late( y );

// Find latest legal block for instruction i
Schedule_Late( instruction i ) {
  if i.visit = True then       // Instruction is already
    return;                    // ... scheduled late?
  i.visit := True;             // Instruction is being visited now
  Block lca := NULL;           // Start the LCA empty
  forall uses y of i do {      // Schedule all uses first
    Schedule_Late( y );        // Schedule all inputs late
    Block use := y.block;      // Use occurs in y's block
    if y is a PHI then {       // ... except for PHI instructions
      // Reverse dependence edge from i to y
      Pick j so that the jth input of y is i
      // Use matching block from CFG
      use := y.block.CFG_pred[j];
    }
    // Find the least common ancestor
    lca := Find_LCA( lca, use );
  }
  ...use the latest and earliest blocks to pick final position
}
```

**select_block**

在确定每个指令可以被灵活调度的空间后，我们将进行最关键的一步，为指令选择它最终的基本块。这里选择的依据是循环深度尽可能浅且尽可能的靠前。经过这样的调度，循环中不发生变化的指令会被移出循环，更多优化的机会暴露出来。

```
... Found the LCA, the latest legal position for this inst.
... We already have the earliest legal block.
  Block best := lca;           // Best place for i starts at the lca
  while lca ≠ i.block do       // While not at earliest block do...
    if lca.loop_nest < best.loop_nest then
      // Save deepest block at shallowest nest
      best := lca;
    lca := lca.immediate_dominator; // Go up the dom tree
  }
  i.block := best;             // Set desired block
```

由此，我们便在不改变变量依赖的前提下，重新调度了指令。

前面提到的优化都是基于中端的优化。不同于中端优化，后端优化和体系结构的关联性很强。在优化的过程中，不仅要考虑指令的数据流，可能还要考虑CPU的流水线等等方面，而我们最终生成的mips程序是跑在Mars模拟器当中的，所以不需要考虑cpu的执行情况，但是对于mips架构的理解对减少目标代码的执行数一定有很大的帮助。所以在学习后端的优化技术之前，我们希望同学们能够通过阅读mips架构的英文指令集来熟悉体系结构，这样在实现后端优化的过程中一定能事半功倍。

# （六）后端消除PHI

**这部分内容需要你的中间代码是SSA形式的，如果没有使用SSA形式的中间代码可以自行忽略这部分的内容**

SSA形式的中间代码本身就携带了大量信息，可以方便的进行很多中端优化，但是在代码生成的时候PHI指令实际上是无法被常规的汇编指令表示的，所以在进行代码生成之前需要将SSA形式的代码转换为汇编指令集能够接受的代码。

首先介绍一下**关键边（critical edges）**的概念，关键边是从一个有多个后继的节点指向有多个前驱的节点的边。消PHI最简单的一个思路是在源基本块跳转之前将PHI指令拆成多条move指令，但是如果程序中存在关键边，以下面的llvm代码为例

```
...
BB1:
%1 = phi i32 [ %2, %BB2 ], [ %3, %BB3 ]
%2 = phi i32 [%1, %BB2], [%3, %BB3]
br label %BB4
BB2:
....
br  i1 %4, label %BB4 label %BB1
BB3:
...
br  i1 %5, label %BB4 label %BB1
BB4:
%6 = add i32 %1, %1
...
```

显然从BB2到BB1之间存在一条关键边，假设直接在BB3的尾部添加move指令，那么跳转到BB4时，move指令也会被执行，这个时候%1变量的值会被改变，而BB4对%1有引用，可能会造成错误，所以如果要在跳转前将PHI变成多条move指令需要消除关键边，一种显然的做法是在BB3和BB1之间再增加一个基本块BB5，BB3跳转到BB5之后再对BB1需要的PHI指令添加move指令，再从BB5跳转到BB1，需要注意的是，由于一个基本块的PHI指令有并行性，例如从BB2跳转到BB1中，%2正确的赋值应该是原来%1的值，而不是跳转之后赋值给%1的原来%2的值，所以这里的move指令用PC(parallel copy)指令代替。下面给出《the SSA book》当中一种参考的算法

ing line 13, replacing $a_i'$ by $a_0$ in the following lines, and adding "remove the $\phi$-function" after them.

**Algorithm 3.5:** Critical Edge Splitting Algorithm for making non-conventional SSA form conventional.

```
1  foreach B: basic block of the CFG do
2      let (E_1, ..., E_n) be the list of incoming edges of B
3      foreach E_i = (B_i, B) do
4          let PC_i be an empty parallel copy instruction
5          if B_i has several outgoing edges then
6              create fresh empty basic block B_i'
7              replace edge E_i by edges B_i → B_i' and B_i' → B
8              insert PC_i in B_i'
9          else
10             append PC_i at the end of B_i
11     foreach φ-function at the entry of B of the form a_0 = φ(B_1 : a_1, ..., B_n : a_n) do
12         foreach a_i (argument of the φ-function corresponding to B_i) do
13             let a_i' be a freshly created variable
14             add copy a_i' ← a_i to PC_i
15             replace a_i by a_i' in the φ-function
```

这个时候所有的PHI指令都被PC指令替代，但是mips体系结构里并没有类似PC的指令，所以我们需要生成一个有并行效果的move序列。这里我们介绍一种图算法，将变量的值视作图的节点，每一条PC指令视作图的边，算法循环的从PC集合当中遍历，如果对于一个$b \leftarrow a$指令，不存在$c \leftarrow b$指令，那说明b变量的赋值不会影响到并行性，直接将`move b, a`加入move序列当中，并将$b \leftarrow a$从PC集合中删除，如果存在$c \leftarrow b$指令，则添加一个临时变量$a^{'}$，将`move a', a`加入move序列当中，用$b \leftarrow a'$替换$b \leftarrow a$指令，这样加入一个中间变量之后$b$的赋值就不会影响$c$的赋值。具体的算法如下：

**Algorithm 3.6:** Replacement of parallel copies with sequences of sequential copy operations.

```
1  let pcopy denote the parallel copy to be sequentialized
2  let seq = () denote the sequence of copies
3  while ¬[∀(b ← a) ∈ pcopy, a = b] do
4      if ∃(b ← a) ∈ pcopy s.t. ∄(c ← b) ∈ pcopy then        ▷ b is not live-in of pcopy
5          append b ← a to seq
6          remove copy b ← a from pcopy
7      else                                                  ▷ pcopy is only made-up of cycles; Break one of them
8          let b ← a ∈ pcopy s.t. a ≠ b
9          let a' be a freshly created variable
10         append a' ← a to seq
11         replace in pcopy b ← a into b ← a'
```

至此，我们已经完成了PHI指令的消除，需要注意的是，这是最简单的一种消PHI的方法，会产生大量move指令和基本块，如果不进行后续的优化甚至可能取得不如非SSA中间代码的性能，幸运的是，我们可以通过在寄存器阶段合并move指令、窥孔优化以及基本块合并等多种方法优化指令序列。《the SSA book》当中介绍了更加有效的消PHI方法，如果同学们有兴趣可以自行选择阅读。

# （七）寄存器分配

## 1. 图着色寄存器分配

在之前的阶段当中我们都假定了有无限个寄存器作为临时变量，而实际上，寄存器是十分珍贵的计算资源，有着最为迅速的读写速度。编译课程的目标架构mips只有32个通用寄存器，显然无法满足"无限"寄存器的要求，所以如何进行寄存器分配是编译优化当中一个重要的问题。但是已有的数学证明已经指出寄存器分配是一个NP完全问题，所以一个好的寄存器分配策略应该能在较短的时间给出近似最佳的结果。课程组以虎书为蓝本，介绍一种近似线性时间的寄存器分配策略。

**构造(build)**

本阶段的主要任务是通过数据流分析方法，计算在每条指令执行时同时活跃的临时变量，该集合的每一对临时变量两两形成一条边加入冲突图中，

构造阶段的主要工作是进行数据流分析，数据流分析我们建议分为两个阶段，首先是进行基本块的数据流分析，分析的数据流方程如下（两个阶段的数据流分析均使用同一个方程）

$$in[m] = use[m] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

第二阶段是进行指令间的数据流分析，得到冲突图，指令冲突的条件是在变量定义处所有出口活跃的变量和定义的变量是互相冲突的，以及同一条指令的出口变量互相之间是冲突的，例如下面的代码

```
...
add $2, $2, $3              #1
sw $4, 0($5)                #2
..
```

显然`$4, $5`都是#1指令的出口寄存器，那么`$2、$4、$5`将会互相作为邻接点加入冲突图中

第一阶段进行基本块之间的数据流分析是为了在进行指令之间的数据流分析获得一个开始的数据流信息，例如以下的代码

```
block1:
add $2, $2, $3
j block2
...
...
...
block2:
add $4, $5, $6
```

显然`$5, $6`是出口活跃的，所以在进行第二阶段`block1`的数据流分析开始时，就需要把`$5, $6`加入活跃变量集合中

**简化（simplify）**

进行启发式图着色，删除度数小于K（通用寄存器个数，mips中是32个）的节点，简化冲突图，产生更多图着色机会。具体的做法是对于一个节点m，如果它的度数小于K，那么将它从冲突图当中删除，并且将其压入栈中，以便进行后续的着色。这样进行启发式删除之后，都会减少其他结点的度数，从而产生更多的简化的机会，并且也为之后的合并提供更多机会，这要求在这个阶段不会简化非冻结指令。

**合并（coalesce）**

进行保守的合并，减少最终代码的move指令。对于move指令（例如`move $2, $3`），源操作数和目的操作数实际上相同，这种情况下可以将两个节点合并，减少一条move指令的代价，但是，合并两个结点之后，两个节点的活跃范围会增加，其邻接结点的集合实际上是原来两个集合的并集，这种情况下可能会导致一些能够着色的节点变为被溢出的结点，这样就得不偿失了，因此我们给出的一种策略是在合并的时候进行判断，保证合并之后高度数（度数>=K）的结点不会增加。通过简化之后，冲突图很多节点的度数已经降低，所以这个时候合并的机会可能多于原有的冲突图。

合并的效果通常决定了整个图着色算法的质量，另外由于在进行指令选择之前需要消PHI，会在这个时候产生很多冗余的move指令，通过图着色的时候进行合并能够极大的减少冗余的move指令，根据经验来看，这对程序性能的提升非常大。

**冻结（freeze）**

在进行简化和合并之后，有一些变量无法进行合并，并且可以进行简化，这个时候需要将其标记为传送冻结的结点，重新开始简化阶段。

**溢出（spill）**

简化和合并完成之后，冲突图中只剩下高度数（度数>=K）的节点，这个时候我们需要在图中选择一个高度数的结点，将它存入内存当中，然后将它压入栈中。这个时候其他结点的度数降低，可以继续进行简化。

**选择（select）**

对虚拟寄存器指派颜色，即分配真实的寄存器。具体做法是以一个空图开始，从栈中弹出一个节点，加入到冲突图中，并且为他指派一种颜色，该节点必须是可着色的，可着色的条件是其邻接结点使用的颜色小于真实的寄存器数K，对于简化阶段入栈的变量显然一定能够着色，对于溢出阶段入栈的变量，通过合并之后，邻接点中有一些变量的颜色相同，最后邻接点的颜色小于K种，能够进行着色，可以将其加入冲突图中，否则将其加入无法进行着色的集合中。

**重新开始(restart)**

如果无法进行着色的集合不为空，那么则需要改写程序，为这些变量在内存当中分配空间，并且在每次使用需要将其从内存当中取出，每次修改需要存进内存当中，这种情况下，溢出的临时变量会转变为几个活跃范围很小的新的临时变量，这个时候需要重新进行活跃分析、寄存器分配，直到没有溢出和简化为止（通常只需要迭代一两次）。

至此我们已经完成了图着色寄存器分配的介绍，但是图着色寄存器分配具体的实现是一个非常复杂的过程，《现代编译原理：c语言描述》当中有非常详细的介绍，在理解每个阶段的原理之后同学们可以参考书中的算法进行具体实现。

# （八）乘除法优化

对于处理器来说，执行乘除法的代价要远大于执行其他指令，而执行除法的代价又远大于除法的代价，所以我们理想的情况是通过算术运算将除常数转化为乘法和移位指令，将乘常数转化为移位和其他指令。我们只针对mips架构提出一种可行的除法优化方案，更加详细的实现以及具体的数学原理可以参考**Division by Invariant Integers using Multiplication&**

## 乘法优化

乘法优化比较简单，一般只需要判端常数是否是2的整数次幂即可，例如`x * 1024`可以优化为`x << 10`，另外由于我们给出的rank计算标准是乘法的代价为加法的5倍，那么`x * 3`实际上可以考虑优化为`x + x + x`。一些特殊的常数也可以考虑优化例如`x * 10`可以考虑优化为`x << 3 + x + x`。

### 除法优化

除法优化的思路是将除法指令转化为乘法指令和移位指令，即如下公式：
$quotient = \frac{dividend}{divisor} = (dividend * multiplier) >> shift$ 该公式先乘一个较大的常数，然后用右移*shift*位得到最终的答案，使用该式子计算除法的核心是如何得到*multiplier*，另外由于mips架构是32位的，还必须要考虑溢出的问题。mips在执行乘法的时候会将结果保存到hi和lo寄存器当中，在被除数乘*multiplier*之后，很容易就会超过32位，这个就会出现一部分答案在hi寄存器中，一部分答案在lo寄存器当中，所以我们需要使*multiplier*尽量大（超过$2^{32}$），能够使答案的部分在hi寄存器当中。所以最终的公式可以写为如下形式 $quotient = \lfloor \frac{n}{d} \rfloor = \lfloor \frac{m*n}{2^{N+l}} \rfloor$ 其中N是机器码长度32，最终获得答案的公式为 $SRL(MULUH(m,n),shift)$，$MULUH$表示乘法之后取HI寄存器。

### 选择Multipier和获得Shift

论文当中提出了一种*mutiplier*和*shift*的获得方法，具体证明可以同学们可以查阅论文

affect the quotient when $n < 2^{\cdot\cdot}$.

**Theorem 4.2** *Suppose $m$, $d$, $\ell$ are nonnegative integers such that $d \neq 0$ and*

$$2^{N+\ell} \leq m * d \leq 2^{N+\ell} + 2^{\ell}. \qquad (4.3)$$

*Then $\lfloor n/d \rfloor = \lfloor m * n/2^{N+\ell} \rfloor$ for every integer $n$ with $0 \leq n < 2^N$.*

不等式当中的N是机器码的长度，在mips当中为32，该式子的意思是需要找到一个*m*使得*m*d*之后大于$2^{N+l}$并且小于$2^{N+l} + 2^l$这样在右移*l*位之后可以得到最终的答案。

## （九）基本块优化和窥孔优化

llvm中端代码的基本块要求必须有入口和出口，即在基本块尾部必须有br或者return语句，除了第一个基本块，其他基本块需要有br语句指向它。与llvm不同，mips汇编代码会默认执行继续执行下一个基本块的指令，并且在现实的CPU中，跳转指令由于需要访问内存、从内存当中取指令，所以执行的代价比普通的计算指令要高，这在我们竞速rank的计算方式当中也有体现。另外，消PHI部分为了去除关键边，增加了很多中间基本块以及跳转指令，所以，我们在生成汇编代码的过程中，对基本块进行优化能很有效的提高程序执行的性能。

### 基本块合并

首先在翻译llvm代码时，对于跳转到相邻的基本块的情况，可以直接进行合并，例如下面的代码

```
bb1:
...   #code1
br %bb2
bb2:
...
```

在翻译的时候可以直接省去跳转指令的翻译，两个基本块实际上合并成为了一个基本块。（需要注意的是如果bb2还有其他的前驱块，bb2的标签需要继续保持)。

另一方面，对于只有一个出口的基本块显然可以进行迭代合并，从一个没有后继的基本块开始迭代，判断它是否只有一个前驱基本块，一直迭代到无法合并为止。

例如如下的代码

```
bb1:
...     #code1
j bb2
...
bb2:
...    #code2
j bb3
...
bb3:
...    #code3
```

从bb3开始迭代，可以合并到他的前驱基本块bb2中显然可以合并为

```
bb1:
...   #code1 + code2 + code3
...
```

## 基本块排序

同样的一组基本块，如果序列的顺序不同完全可能带来不一样的执行效果，例如如下的c代码

```
int sum;
int num = getint();
for (int i = 0; i < num; i = i + 1) {
  sum = sum + i;
}
```

被翻译为如下的llvm代码

```
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %5 = call i32 @getint()
  store i32 %5, i32* %3, align 4
  store i32 0, i32* %4, align 4
  br label %6

6:                                                ; preds = %14, %0
  %7 = load i32, i32* %4, align 4
  %8 = load i32, i32* %3, align 4
  %9 = icmp slt i32 %7, %8
  br i1 %9, label %10, label %17

17:                                               ; preds = %6
  %18 = load i32, i32* %2, align 4
  ret i32 %18

10:                                               ; preds = %6
  %11 = load i32, i32* %2, align 4
  %12 = load i32, i32* %4, align 4
  %13 = add nsw i32 %11, %12
  store i32 %13, i32* %2, align 4
```

```
  br label %14

14:                                               ; preds = %10
  %15 = load i32, i32* %4, align 4
  %16 = add nsw i32 %15, 1
  store i32 %16, i32* %4, align 4
  br label %6, !llvm.loop !8
```

基本块6是循环判断基本块，下一个基本块是17，翻译成mips代码之后，6和17之间可以省略跳转语句，但是根据源代码可以得知，循环判断的基本块大概率会跳转到循环体当中，这个时候就可以改变基本块17和基本块10的位置，可以减少jump指令的数量。

另一方面，一个循环可以被变为do while形式的循环，即先判断是否执行循环，然后进入循环体，在循环结束的时候再次判断是否执行循环，例如下面的代码

```
bb1:
addi $1, $1, 1
beq $1, $2, bb3
j bb2
....
bb2:
...    #code1
j bb1
bb3:
....
```

可以改成如下的do while 形式，jump类指令的执行数量可以减少一半

```
bb1:
...     #code1
addi $1, $1, 1
bne $1, $2, bb1
bb3:
...
```

同学们在实现的时候可以根据自己的中间代码特性以及实际的基本块顺序灵活的选择基本块序列。

# （十）指令选择

指令选择是生成目标代码的一个重要的步骤，关乎后端代码的整体质量，指令选择的目标是尽量用小的指令代价覆盖更多的代码，这里的代价可能是执行时间、指令条数、CPI（执行一条指令所需要的时钟周期数）等等，在我们这里主要指的是指令执行的FinalCycle。

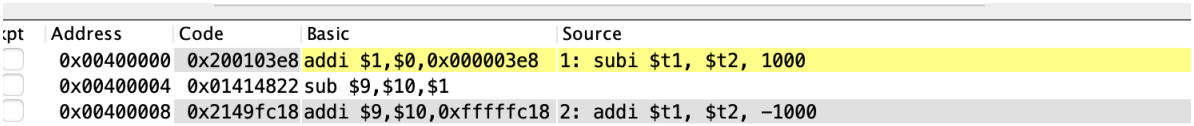
## Mars伪指令的限制

为了更方便的实现更多功能，以及更加方便翻译高级语言，Mars给我们提供了充分的伪指令使用，例如`subi $t1, $t2, 100`、`move $1, $2`、`ble $t1, $t2, label`等，能够缩短指令的条数，增加代码的可读性。但是由于Mars的局限性，以及体系结构的特性，很多时候伪指令虽然表面上降低了指令的条数，但是实际上反而会使FinalCycle增加，在非循环语句当中，这样的问题当然可以忽略，而在循环次数很多的循环体当中，哪怕每个语句多翻译了一条代码都会严重影响性能，而subi等语句其实是十分常见的计算语句，在循环当中也可能高频出现。另一方面，在进行图着色寄存器分配时，伪指令可能会隐蔽地更改寄存器的值，导致数据流分析错误，可能会产生一些很难发现的bug。所以不使用Mars低效的伪指令，转而自己封装一套翻译机制是提升性能的有效方法。

## 应用举例

我们给出一个Mars伪指令翻译不合理的例子，而更多伪指令不合理的地方，大家可以自行通过试验发现。

以llvm作中端代码为例，中端向后端发送一条`%3 = sub i32 %1, 1000`指令，后端可以将其翻译为伪代码指令`subi $t1, $t2, 1000`或基础指令`addi $9, $10, -1000`，在Mars当中汇编可以发现



| :pt | Address | Code | Basic | Source |
|---|---|---|---|---|
| | 0x00400000 | 0x200103e8 | addi $1,$0,0x000003e8 | 1: subi $t1, $t2, 1000 |
| | 0x00400004 | 0x01414822 | sub $9,$10,$1 | |
| | 0x00400008 | 0x2149fc18 | addi $9,$10,0xfffffc18 | 2: addi $t1, $t2, -1000 |

这条`subi $t1, $t2, 1000`指令，会被mars翻译成`addi $1, $0, 0x3e8`和`sub $9, $10, $1`两条指令，而通过`addi $9, $10, -1000`一条指令即可完成。