南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理大作业

从无到有实现编译器

年级：2020 级

专业：计算机科学与技术

指导教师：王刚 李忠伟

姓名：马琦

学号：2011600

2023 年 1 月 15 日

摘要

本学期，基于编译原理课程所学，我和沈冠翔同学完整实现了从源代码到目标代码的生成。我在本次报告所描述的工作主要如下：首先，借助 lex 词法分析器辅助完成了词法分析的过程，之后我们从产生式入手，设计了完整的产生式，利用 yacc 实现了对产生式的识别。之后为了实现更丰富的功能，我们建立了更多的语法树结点，层次更加丰富，实现了数组的操作。另外我在语法树的建立过程中即进行了类型的检查，实现了对非法数组的检查。之后根据建立的语法树，我们进一步的插入指令，生成对应的中间代码。最后在分配寄存器的时候，我们采用了快速且性能较好的 LinearScan 寄存器分配算法。

关键字：词法分析　寄存器分配　语法分析　中间代码生成　类型检查

目录

一、 引言

现代场景下的程序开发往往面临着巨量的代码和极为复杂的逻辑，如果单纯使用底层的汇编代码，那么我们开发程序的人工成本将高到我们无法接受，这就是编译器产生的原因，当然前提还是得有一些高级的语言，比如 C++、Java 等，这些语言可以让我们将注意力更多集中在程序本身，而非机器。编译器是为了让机器能够理解我们的高级语言，同时能够完成一些额外的工作，比如能够检查我们的代码是否存在语法错误，甚至可以去优化代码。

本学期基于王刚老师、李忠伟老师讲授的编译原理课程，我们开始从” 零” 搭建一个简单的基于 SysY 语言的编译器，主要借助了 Yacc、Lex、以及老师学长搭建的架构，能够成功将源代码 (.sy) 翻译生成目标代码 (.s)，在过程中包括对代码的类型检查，同时对中间代码进行了优化，对寄存器的分配进行了优化，使得目标代码的运行效率更高。在这过程中收获很多，具体的细节的一部分将在后文展开。

二、 编译器整体架构

对于一个编译器，简单来说可以划分为前端、中间代码、后端，其中，前端的目的主要是将源代码翻译生成中间代码，而后端则是将中间代码翻译生成机器码。详细来说，当我们传入源代码时，我们首先逐个字得进行识别，生成对应的代词流，将单词流传输给 yacc，进行语法分析，在此过程中我们建立起语法树，同时进行对应的类型检查。之后如果一切正常，我们将利用刚刚生成的语法树进一步地生成中间代码，之后再利用中间代码进一步地生成我们的目标代码。其架构图如下所示。

图 1: 朴素的编译器架构

其中，在中间代码到机器码这一个阶段，我们可以加入一些优化来提升代码的性能，首先我们将 IR 转化为 SSA IR(Static Single-Assignment), 之后利用生成的这种 IR 完成 Mem2Reg 和控制流优化算法，之后我们将优化后的 SSA IR 回复为普通的 IR，然后生成机器代码，同时使用 LinearScan 寄存器分配算法对机器代码进行进一步的优化，最后生成我们对应的.s 文件目标代码。

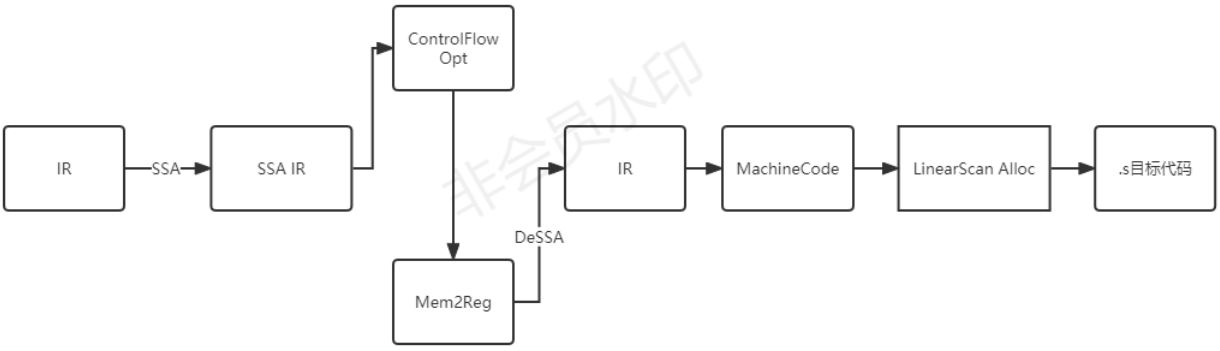

图 2: 带优化的编译器架构

三、 词法分析

这一块是编译器中最基础的部分，主要就是对源代码进行切分，将字符按照其类型进行整合，形成合适的单词流供后续分析。这一部分利用到的主要是课上所学的正则表达式、模式匹配，理论工作流程大致如下:

1. Lex 将源文件作为输入，读取源文件的每一个字符，并将它们存储在一个缓冲区中;
2. 然后，Lex 将缓冲区中的字符串与定义的模式（正则表达式）进行比较，如果有匹配的模式，则执行相应的动作;
3. 如果没有匹配的模式，则继续读取源文件，并将读取的字符添加到缓冲区中，重复上述步骤，直到找到匹配的模式。
4. 当找到匹配的模式时，Lex 会执行相应的动作，然后清空缓冲区，重复上述步骤，直到没有更多的字符可以读取。那么我们要做的首先就是设定合适的正则表达式识别不同类型的输入，之后我们还可以利用栈来完成括号匹配并且可以打印出作用域 (当然这一部分的内容在后续的 SymbolTable 中有更为完整和强大的功能，当时我们并不知情:(，并且这里确实可以打印出作用域)。

(一) 正则表达式

1. 正则表达式设计

设计正则表达式需要我们熟悉每一种元素其字符表示类型的特点，例如说对于 ID，第一个字符不可以是数字，对于浮点数来说 e 和 E 只可以出现在最后且并不一定出现，十六进制需要 0x 或者 0X 开头来声明，借助这些特性，我们设计的正则表达式如下:

正则表达式设计

```
DECIMAL [1-9][0-9]*|0|-2147483648
HEX 0(x|X)[1-9a-fA-F][0-9a-fA-F]*|0
OCT 0[1-7][0-7]*
DECIMAL_FLOAT ((([0-9]*[.][0-9]*([eE][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+))[
    fL|lL]?)
```

```
HEXADECIMAL_FLOAT (0[xX](([0-9A-Fa-f]*[.][0-9A-Fa-f]*([pP][+-]?[0-9]+)?)
    |([0-9A-Fa-f]+[pP][+-]?[0-9]+))[fL1L]?)
ID [[:alpha:]_][[:alpha:][:digit:]_]*
LINECOMMENT (\/\/[^\n]*)

commentblockbegin "/*"
commentelement .|\n
commentblockend "*/"
```

2. 举例说明

这其中就用到了类似状态机的原理，以一个简单的例子说明，比如要识别 22.33 这个十进制浮点数，首先我们得到 2，此时并不能区分他是浮点数还是十进制整数，但是可以判断他不可能是 ID、十进制和八进制及其他符号，之后识别第二个 2，此时还不能区分，当识别到小数点是，我们可以认为就是浮点数 (如果输入的源文件没有错误)，此时我们继续匹配，当最终匹配到最后的数字时，我们的识别结束，认定该数为十进制浮点数.

(二) 作用域的确定

这一部分和括号匹配有异曲同工之妙，其实就是如果左大括号入栈之后，右大括号入栈之前 (这里指的是一组匹配的大括号)，那么这之间的所有变量都属于处在作用域。我们只需要用一个全局栈来玩成即可，当然，我们在这里利用 set 加速了查找可用变量的过程 (原来可能需要遍历或者出栈)。由于这一部分在之后的 SymbolTable 中已经有所重复了，这里不做赘述。

作用域的一个例子

```
    {ID} {
    if(dump_tokens){
        if (!cur_str[part].count(std::string(yytext))) cur_str[part].insert(
            std::string(yytext));
        is_stack[std::string(yytext)].push_back(part);
        DEBUG_FOR_LEX("ID", std::string(yytext));
        offset += strlen(yytext);
    }
    yylval.strtype = (char*)malloc(strlen(yytext) + 1);
    strcpy(yylval.strtype, yytext);
    return ID;
}
```

(三) 函数符号表

在这里需要我们提前将系统库函数加入到符号表里 (函数也被视作 ID 的一种)，因为这些函数在后续不会进行声明定义，因此在类型检查中会被判定为 undefined。这里的操作依然很简单，并且由于我们结合了 multimap，可以在原来的基础上支持一些特性，比如我们可以满足函数和变量同名 (这个功能在 C 语言中有，详细原理在沈冠翔同学的报告中)，下面我们给出一段例子

函数加入符号表

```
"getint" {
    if(dump_tokens){
        DEBUG_FOR_LEX("ID", "getint");
        offset += strlen(yytext);
    }
    char *lexeme = new char[strlen(yytext) + 1];
    strcpy(lexeme, yytext);
    yylval.strtype = lexeme;
    Type* funcType = new FunctionType(TypeSystem::intType, {});//返回类型int, 无参数
    SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, 0);//作用域GLOBAL(0)
    globals->install(yytext, se);
    return ID;
}
```

四、 语法分析

(一) 产生式的设计

这一部分包含了我们编译过程中核心的过程，由之前的词法分析，我们可以得到具有语义的单词流，那么进一步的，我们要根据 SysY 语言的特性，建立对应的语法树，这里我们通过产生式，同样借用模式匹配的思路实现语法的分析，其理论原理为: 每次从输入的字符串的最左端开始读取一个字符，然后根据语法规则逐步构建一棵语法树，直到整个字符串都被读取完成，语法树也被构建完成。这里的语法规则其实就是我们之前设计的产生式，在之前的报告中已经详细地给出，这里不做过多赘述。

(二) 语法树的建立

语法树是我们这个部分需要着重考虑的，后续生成中间代码我们也是按照语法树去生成的。我们在原有的结点里加入一些新的结点，下面我们将按照语法树的生成顺序来介绍我们语法树的建立过程 (以目的为导向)。

1. 从整体到局部

这一部分本可以细分为很多步，但是为了方便理解避免拖沓我们选择集中说明，首先我们把整个源代码当做一个整体，然后不同的代码有不同的块 (也就是有很多作用域), 这其中包含很多不同的语句，包括空语句、赋值语句、条件判断语句、特殊语句 (break、continue、return 等)、循环语句、声明语句、函数定义语句等，我们的语法树也将主要就这些语句展开。

2. 声明语句

我们的程序一般都会操作很多数据，这些数据一般都是声明而来的。所以我们最先开始介绍声明语句 (DeclStmt), 这个语句主要是”TYPE VARDEFLIST;” 或者”TYPE CONSTVARDEFLIST;” 这样的形式，其中 VARDEFLIST 有单个定义语句构成，下面我们以非常量的声明语句为例进行说明。

在说明之前，我有必要介绍一下我们新生成的 InitNodeNode 结点，这是一个新的结点，继承自 StmtNode，主要用来存储声明式中的右值，那么相比于原来的 ExprNode，我们加入的这个结点可以满足数组的定义需求，其结构如下:

InitNode 结构

```
class InitNode : public StmtNode
{
private:
    bool isconst;
    ExprNode *leaf;
    std::vector<InitNode *> leaves;
    int cur_size = 0;

public:
    InitNode(bool isconst = false) : isconst(isconst), leaf(nullptr) {};
    void addleaf(InitNode *next) { leaves.push_back(next); };
    void setleaf(ExprNode *leaf1) { leaf = leaf1; leaves.clear(); };
    bool isLeaf() { return leaves.empty(); };
    void fill(int level, std::vector<int> d, Type* type);
    int getSize(int d_nxt);
    bool isFull();
    bool isConst() const { return isconst; }
    void output(int level);
    void genCode(int level);
    std::vector<InitNode *> getleaves() { return leaves; };
    ExprNode *getself() { return leaf; };
    ~InitNode()
    {
        if (leaf != nullptr)
            delete leaf;
        for (auto _leaf : leaves)
            delete _leaf;
    }
};
```

这样的一个结点方便我们进一步的递归迭代求解，其逻辑主要如下，当我们是单值或者数组中的单个元素时，leaf 值存储其表达式，而 leaves 为空，如果我们是一个数组时 (数组的子数组也算)，leaf 为空，leaves 存储其子节点。通过这样的一个简单的逻辑，我们就可以以一种统一的形式表示所有的 Initval。当然对于数组，我们还需要解决的是数组的填充问题，比如下面这一个相对复杂的例子

数组填充示例

```
int a[3][2][3] = {{{1}, {2}}, {{3,  4}, 5}};
```

那么对于这样的数组，我们首先需要自顶向下开始遍历，对于外层的大括号，或者就是其第一维度来说，应该有 3 个子节点而此时有两个，所以需要在末尾填充一个结点，如下

数组填充示例

```
int a[3][2][3] = {{{1}, {2}}, {{3, 4}, 5}, {}};
```

这其中最简单的是最后一个括号，我们只需要继续按照维度向下遍历，不断加入子节点，最终，他形成

数组填充示例

```
{{0, 0, 0}, {0, 0, 0}}
```

而对于第二种来说,我们直接判断叶节点的数量肯定是不行的,我们首先需要倒着遍历一遍 leaves,数一下尾部为单个元素的叶节点，之后将其补成可以被当前维度容量整除的数量，之后再按照下一维度的要求挨个递归插入，其过程如下

数组填充示例

```
{{3, 4}, 5, 0}
{{3, 4}, 5, 0, {}}
{{3, 4}, 5, 0, {0, 0}}
```

第一种也不是很麻烦，我们直接递归，当递归到数组的最后一个维度时，选择填充 0 元素，并且退出递归，其过程如下

数组填充示例

```
{{1}, {2}, {}}
{{1, 0}, {2, 0}, {0, 0}}
```

这一部分对应的代码如下

数组填充代码

```
void InitNode::fill(int level, std::vector<int> d, Type *type) // 传入维度信息和类型，并且记录递归深度
{
    if (level == d.size() || leaf != nullptr)
    {
        if (leaf == nullptr) // 递归终点
        {
            setleaf(new Constant(new ConstantSymbolEntry(Var2Const(type), 0)));
        }

        return;
    }
    int cap = 1, num = 0, cap2 = 1;
    for (int i = level + 1; i < d.size(); i++) // 计算数组容量
        cap *= d[i];
    for (int i = leaves.size() - 1; i >= 0; i--) // 计算尾部单个元素数量
        if (leaves[i]->isLeaf())
            num++;
        else
            break;
    while (num % cap) // 填充成为容量的整数倍
```

```
    {
        InitNode *new_const_node = new InitNode(true);
        new_const_node->setleaf(new Constant(new ConstantSymbolEntry(
            Var2Const(type), 0)));
        addleaf(new_const_node);
        num++;
    }
    int t = getSize(cap); // 补充数组结点
    while (t < d[level])
    {
        InitNode *new_node = new InitNode(true);
        addleaf(new_node);
        t++;
    }
    for (auto l : leaves) // 进一步递归
    {
        l->fill(level + 1, d, type);
    }
}
int InitNode::getSize(int d_nxt) // 计算当前结点下子节点的数量(非单个结点)
{
    int num = 0, cur_fit = 0;
    for (auto l : leaves)
    {
        if (l->leaf != nullptr)
        {
            num++;
        }
        else
        {
            cur_fit++;
        }
        if (num == d_nxt)
        {
            cur_fit++;
            num = 0;
        }
    }
    return cur_fit + num / d_nxt;
}
```


填充的时机在 DeclStmt 声明时即可，这样可以让我们尽早获得规整的数组结构。下面开始介绍声明式子，针对普通的整型或者浮点型变量来说，声明语句一般像下面这样

声明语句示例

```
int a = 1;
float b = 1.0;
```

我们首先在 DeclStmt 的级别 (即可以得到 TYPE 时) 先将类型存储起来，之后识别到 ID 时，

首先进行进行类型检查，判断之前是否重复定义过，之后我们将类型存入符号表中，并根据这个 SymbolEntry 生成一个新的 Id 结点，Id 结构主要如下

Id 结点的结构

```
class Id : public ExprNode
{
private:
    IndicesNode *indices; //用来存储Id的索引, 只对数组有用
    bool is_array = false, is_array_ele = false; // 用来区分数组还是数组元素
    bool isleft; // 判断是否为左值

public:
    Id(SymbolEntry *se, bool be_array = false, bool isleft = false) :
        ExprNode(se, be_array)
    {
        indices = nullptr;
        is_array_ele = se->getType()->isARRAY() && be_array;
        is_array = se->getType()->isARRAY();
    };
    void setIndices(IndicesNode *new_indices) { indices = new_indices; };
    IndicesNode *getIndices() { return indices; };
    void output(int level);
    bool is_Array() { return is_array; };
    bool is_Array_Ele() { return is_array_ele; };
    ArrayType* get_Array_Type() { return (ArrayType*)(getSymPtr()->getType())
        ; };
    // void typeCheck();
    void genCode();
    ~Id()
    {
        if (indices != nullptr)
            delete indices;
    };
};
```

之后将该 Id 存到符号表中，再将 Initval 所生成的 InitNode 和当前 Id 一起，通过如下语句生成一个新的声明语句结点。

Id 结点的结构

```
    $$ = new DeclStmt(new Id(se), (dynamic_cast<InitNode*>($3)));
```

对于数组的声明式来说，是一样的道理，主要区别在于数组有维度信息，所以我们又新定义了一个 IndicesNode 结点，结点的结构如下

IndicesNode 结构

```
  private:
    std::vector<ExprNode *> exprList; //存储维度信息

public:
```

```
    IndicesNode(){};
    void addNew(ExprNode *new_expr) { exprList.push_back(new_expr); };
    void addBefore(ExprNode *new_expr) { exprList.insert(exprList.begin(),
        new_expr); };
    void output(int level);
    std::vector<ExprNode *> getExprList() { return exprList; };
    void genCode();
    ~IndicesNode()
    {
        for (auto expr : exprList)
            delete expr;
    }
```

在数组维度的产生式中，我们每添加一个 Expr(或者 ConstExpr) 直接将其加入到数组容器的最后就可以了，最后我们依然是在 DeclStmt 层次，先将 IndicesNode 值赋给 Id，之后重复之前普通值的操作即可。

3. 赋值式子或者调用

这一部分就是如下这么几种情况

数组填充示例

```
int a = 1;
int b[2][2] = {a};
```

(三) 数组的类型检查

五、 中间代码生成