成绩	

编译原理实验报告

学院： 教育实验学院

班级： HC002001

姓名： 孟祥宇

学号： 2020300092

日期： 2023-08-25

目 录

1 实验概述

1.1 要求与目标

完成 SysY 到 llvm-ir 的静态编译器前端。

1.2 总体完成情况

基本完成 SysY 源代码到 llvm-ir 的静态编译工作，。

1.3 团队分工情况

单人完成。

2 主要功能

SysY 的前端静态编译器。

使用方法

```
./parser.out \
    <input_file> \
    [-o <output_file>] \
    [--hide-AST]
```

其中，-o 指定 llvm-IR 的输出文件， --hide-AST 隐藏抽象语法树 AST 的输出。

2.1 功能展示

```
./parser.out function_test2021/001_var_defn.sy -o main.ll
```

将 function_test2021/001_var_defn.sy 编译为 llvm-ir 形式的中间表示。

```
./parser.out function_test2021/001_var_defn.sy -o main.ll
-hide-AST
```

将 function_test2021/001_var_defn.sy 编译为 llvm-ir 形式的中间表示，同时不输出抽象语法树。

```
python TestMachine.py function_test2021/001_var_defn.sy
```

使用测试机测试编译器生成的 llvm-IR 是否正确，测试机使用 parser.out 编译 SysY 源代码生成 llvm-IR，再使用 clang 编译得到可执行程序，运行可执行程序得到结果，然后将其和测试样例的正确输出比较，从而测试编译器是否正常工作，以及编译结果是否正确。

```
python TestBatch.py
```

使用测试机对所有测试样例进行批量测试，每个测试样例的输出精简至一行，并在整体测试完后统计出通过和未通过的样例编号，以及通过的样例个数。

3 软件总体结构

3.1 软件开发环境

flex ~= 2.6.4

bison ~= 3.8.2

make ~= 4.4.1

gcc ~= 13.2.0，使用默认版本（-std=gnu++17）

在 Linux 平台上开发和使用。

3.2 软件运行环境

非二进制，源码除了开发环境以外的内容无依赖，自行编译即可。

3.3 源代码组织与构建

front-end/，该目录为编译器源代码、头文件和 flex&bison 生成文件存放的位置，包含有 makefile，用于生成编译器本身，即 parser.out。

源代码分为了：scanner, parser, node (AST), IR_node, symtable, gen_IR 六个模块，顶层为 main.cpp，包含了 main()的程序入口。

function-test/，该目录为测试文件和样例存放的位置，包含有 makefile，用于测试。

4 详细设计

4.1 Scanner & Parser

4.1.1 两者之间数据的传递

```
// parser.ypp
%union {
    std::string* string;
    int token;
    int value;
    // -----------------
    AST_Node* ast_node;
}
```

通过在 parser.ypp 中使用%union 关键字定义共享变量，scanner 和 parser 都可以访问共享变量。例如变量名的传递，

```
// scanner.l
#define STRING_TOKEN        \
yylval.string = new std::string(yytext, yyleng)

[_a-zA-Z][0-9a-zA-Z]*       {
                              // 标识符
                              STRING_TOKEN;
                              return T_IDENT;
                            }

// parser.ypp
%token<string> T_IDENT

Ident:          T_IDENT {
                    $$ = new AST_Node();
                    $$->set_name(*$1);
                    delete $1;
                    $$->set_type(NodeType::IDENT);
                }
                ;
```

其中 T_IDENT 是 std::string*类型的变量，由 scanner.l 赋值，并在 parser.ypp 中赋值给 Ident 节点，实现了从 scanner 到 parser 值的传递。

4.1.2 抽象语法树节点的组织

所有节点由同一个数据结构 AST_NODE 组成，**其核心作用是在归约过程中记录和维护父节点和子节点的关系**，为此其 API 如下所示，构造函数通过可变参数实现任意数量子节点的指定，也可以通过 add_child 进行子节点的添加；set_{parent, name, type, value}() 用于对节点的属性进行设置。

```
    // default constructor
    AST_Node();

    // constructor with childs, it should end with nullptr
    AST_Node(AST_Node* child1, ...);

    // default deconstructor
    ~AST_Node();

    // add child
    void add_child(AST_Node* child);

    // set parent
    void set_parent(AST_Node* parent);

    // set name
    void set_name(const std::string &name);

    // set value
    void set_value(unsigned int value);

    // set type label
    void set_type(NodeType type);
```

在构造时首先将语法树完成搭建，在可以化简的地方会减少冗余的节点：

```
ConstInitVal:   ConstExp {
                    $$ = $1;
                    /* $$ = new AST_Node($1, nullptr);
                    $$->set_name("ConstInitVal");
                    $$->set_type(NodeType::JUST_PASS); */
                }
```

然后指定每个节点的 name 属性，便于理解抽象语法树和扫描时使用；对于字符串或数值类型的节点，其 name/value 是真实值，不仅用于理解语义；最后指定每个节点的类型，根据不同类型会调用不同的 gen-IR-handler 函数。

4.2 符号表模块

符号表用于遍历语法树的过程中记录变量/函数的作用域及相关属性，在函数调用、通过变量名获取地址等场合得到应用。这里**符号表的构造使用入栈/出栈的方式**。

```
// 进入新的作用域
scopes.push(new Scope(nullptr));

// 离开当前作用域
scopes.pop();
```

为了便于符号的查找，**每一个作用域保存了父节点的指针**，通过向上回溯到的方式，如果当前作用域中包含查找符号，则返回 1；如果一直到根节点都没有查找到查找符号，则返回 0。

```
Symbol* Scope::lookup(const string& symbol_name)
{
    Scope* cur = this;
    while (cur != nullptr) {
        for (Symbol* x : cur->symbols) {
            if (x->name == symbol_name) {
                return x;
            }
        }
        cur = cur->parent;
    }
    return nullptr;
}
```

符号的结构体则用于记录相应的属性，分为变量和函数两类，公用属性是名称，函数的特有属性是参数列表，变量的特有属性包括：是否初始化，变量类型，地址，是否常量，是否全局变量。

```
/*
| name | symbol_type |
       - func        | param_type_list |
       - var         | initialized | value_type | value | is_const | is_global |
*/
```

4.3 IR_node 模块

IR_node 模块区分于 AST_node 模块，每一个类型都有对应的 gen-IR-handler 函数生成对应的 IR。这里使用继承的方式，使用纯虚函数 print()定义生成 IR 的接口，根据不同生成格式分为 BinaryExpIR, ReturnStmtIR 等多个类。

以 BinaryExpIR 类为例，包含了所有的二元运算表达式的生成，因为这些表达式具有同样的生成格式。其定义如下：

```
class BinaryExpIR: public IR
{
public:
    string inst_name;
    string var_type;
    string operand_1, operand_2;
    string return_reg;

    void print(ofstream& output);
};

void BinaryExpIR::print(ofstream& output)
{
    make_table(scopes.size(), output);
    output << return_reg << " = " << inst_name << " " << var_type
    << " " << operand_1 << ", " << operand_2 << endl;
}
```

这里传入的 operand_1 和 operand_2 在生成 LVal 类型的节点时已经将其值加载到了局部变量中：

```
// 调用 LVal 节点的 gen-IR-handler
AST_Node* operand_1 = ast->childs[0];
AST_Node* operand_2 = ast->childs[1];
generate_IR(operand_1, output);

// 生成 LVal 节点对应的 IR
ir.is_initialized = true;
ir.is_reg = true;
ir.init_reg = t_symbol->reg_value;
ir.is_const = t_symbol->is_const;
ir.is_global = t_symbol->is_global;
ir.print(output);

void LValIR::print(ofstream& output)
{
    make_table(scopes.size(), output);
    output << left_reg_name << " = " << "load " << var_type << ",
"
    << var_type << "* " << right_reg_name << endl;
}
```

```
// 通过全局变量获得 LVal 的寄存器/值
if (is_reg)
    ir.operand_1 = last_reg;
else
    ir.operand_1 = to_string(last_const);
```

4.4 gen-IR 模块

gen-IR 模块位于 AST_Node 和 IR_Node 中间，根据 AST_Node 的类型执行相应的操作，初始化 IR_Node，并调用对应的 IR_Node 的生成函数。

仍以 AddExp 的 IR 生成为例，首先根据 AST_Node 的名称确定指令为”add”还是”sub”：

```
BinaryExpIR ir;

if (ast->name == "+")
    ir.inst_name = "add";
else if (ast->name == "-")
    ir.inst_name = "sub";
else
    perror("wrong add op");
```

使用全局编号确定临时变量的寄存器名称：

```
global_var_index++;
string reg_name = "%v" + to_string(global_var_index);
ir.return_reg = reg_name;
```

先将两个操作数转化为可以直接使用的值（局部寄存器或字面值）：

```
AST_Node* operand_1 = ast->childs[0];
AST_Node* operand_2 = ast->childs[1];
generate_IR(operand_1, output);
if (is_reg)
    ir.operand_1 = last_reg;
else
    ir.operand_1 = to_string(last_const);
generate_IR(operand_2, output);
if (is_reg)
    ir.operand_2 = last_reg;
else
    ir.operand_2 = to_string(last_const);
```

最后根据操作数的类型指定变量类型，调用 IR_Node 的 print()函数生成 IR：

```
ir.var_type = last_reg_type;

ir.print(output);
```

5 测试与结果

5.1 功能测试与结果

```
Pass Cases: total  44
[0, 1, 2, 3, 5, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32, 34, 35,
36, 37, 38, 39, 41, 43, 44, 45, 49, 52, 53, 54, 55, 91]
```

整体来看，编译器实现了大部分的功能，包括：变量定义、函数定义和调用、所有表达式、if 语句、while 语句、运算符优先级、注释、十六进制和八进制数字等，通过了 103 个测试样例中的 44 个。

5.2 抽象语法树

生成的抽象语法树见 parser-output.txt，这里给出一个示例：

```
<CompRoot, , >
    <VarDecl, , VAR_DECL>
        <int, , STRING>
        <VarDefList, , JUST_CONCAT>
            <VarDef, , JUST_PASS>
                <a, , IDENT>
                <initial =, , STRING>
                <InitVal, , JUST_PASS>
                    <Number, 3, NUMBER>
    <VarDecl, , VAR_DECL>
        <int, , STRING>
        <VarDefList, , JUST_CONCAT>
            <VarDef, , JUST_PASS>
                <b, , IDENT>
                <initial =, , STRING>
                <InitVal, , JUST_PASS>
```

```
                <Number, 5, NUMBER>
    <FuncDef, , FUNC_DEF>
        <int, , FUNC_TYPE>
        <main, , FUNC_NAME>
        <(), , FUNC_PARAMS>
        <Block, , BLOCK>
            <BlockItemList, , JUST_CONCAT>
                <VarDecl, , VAR_DECL>
                    <int, , STRING>
                    <VarDefList, , JUST_CONCAT>
                        <VarDef, , JUST_PASS>
                            <a, , IDENT>
                            <initial =, , STRING>
                            <InitVal, , JUST_PASS>
                                <Number, 5, NUMBER>
                <Stmt, , STMT>
                    <return, , STRING>
                    <+, , ADD_EXP>
                        <LVal, , LVAL>
                            <a, , IDENT>
                        <LVal, , LVAL>
                            <b, , IDENT>
```

5.3 生成的 llvm-IR

以上述的抽象语法树为例，生成的 llvm-IR 如下：

```
target triple = "x86_64-pc-windows-msys"


@a = global i32 3 align 4
@b = global i32 5 align 4
```

```
; func def
define i32 @main () {
  %a = alloca i32
  store i32 5, i32* %a
  %v2 = load i32, i32* %a
  %v3 = load i32, i32* @b
  %v1 = add i32 %v2, %v3
  ret i32 %v1
  ret i32 0
}
```

6 实验总结

6.1 调试和问题修改总结

1，编译器没有支持数组和常量数组两个 SysY 的特性，并且对于类型系统的处理还不是很妥当。

2，直接将 IR 输出到了文件中，在调试时不是很方便。

6.2 实验小结

一开始在如何构建抽象语法树的阶段卡了比较长的时间，没有理解好语法树的核心在于维护父节点与子节点的关系，为每个类型的节点定义了一个单独的类，工作量过大导致进度缓慢。之后使用统一节点，并采取了先将语法树构造出来、后期不断完善的策略，使开发过程加快了很多。

之后在生成 IR 时发现二元表达式具有一致的生成格式，于是新增了 IR_Node 这一抽象层，按照 ir_node 的属性进行填空，然后调用 print()函数生成结果，使 IR 生成过程简洁了很多。而 LVal 的处理也优化了很多次，一开始在变量定义的地方想将局部变量和全局变量进行统一，但过于混乱，后来修改到生成 LVal 的 IR 时再进行处理，将局部变量/全局变量变成 llvm-IR 中可以直接使用的局部寄存器和字面值，这样处理过程就很流畅。

完成编译器的过程有一点挑战，这个过程中不断克服了想去查找现成代码的心理，独自实现了符号表、生成 IR 这些部分，除了 flex&bison 部分参考了开源的代码，其他部分都是自己不断优化完成的，学到了不少的知识，项目开发能力也得到了一些提升。

7 实验建议

无特殊建议。