SysY 编译器实验

https://github.com/Trip1eS/SysYCompiler

这是个一简单的，实验性质的 SysY 语言的编译器，主要在 2021 版本的 SysY 定义基础上完成了大部分词法分析（除了注释）、大部分语法分析和部分中间代码生成，其他很多地方，包括语义分析、错误处理和数组的支持等还没有完全实现，并且可能有一些未发现的 bug，将在后续的时间里补充、修复完整。

本文档将主要介绍这三个模块的实现思路，以及其他有特色的地方。

1 词法分析

除了标识符和数字之外，词法分析其实就是一个匹配字符串的过程，为了实现匹配的功能，同时让词法分析器的代码有一定的可拓展性，并且依旧有较高的效率，我为每个 Token 使用了一个类似状态机的模型。在进行词法分析时，词法分析器从输入流中读入字符，并传给每一个状态机，然后词法分析器再根据每个状态机的输出来决定现在是否匹配一个 Token，或者现在是否应该报错。

1.1 Matcher

我将状态机的状态枚举命名为 `MatchStatus`，共有四个。`Accpet` 代表状态机可以接受当前的字符，但不能接受下一个字符，表示要么现在匹配，要么等下次再输入时就匹配失败；`Reject` 表示匹配失败，不再参加之后的输入；`Error` 表示遇到了错误，只有在读取数字时会出现这个状态，表示数字不合法；`Reading` 表示既没有接受，也没有拒绝，可以继续读入。

```
enum class MatchStatus {
    Accept,  // accept curChar but not accept nextChar
    Reject,  // already failed
    Error,   // encountered error
    Reading  // accept curChar and nextChar
};
```

我将这个状态机的基类命名为 `Matcher`，最主要的就是 `read` 方法。

```
class Matcher {
   public:
    virtual void read(char curChar, char nextChar) = 0;
    // ...
};
```

这个函数接受两个参数，分别是当前输入流中的字符和输入流中的下一个字符，派生类应该根据自己的需求，处理这两个字符，并且在处理过后改变自己的状态。

派生类一共有三个，`StringMatcher`，`IntConstMatcher` 和 `IdMatcher`，分别用来匹配字符串（包括关键词、操作符、分隔符）、整型字面量和标识符。

下面以 `IdMatcher` 为例来介绍一下 `read` 方法的实现方法。

```
void IdMatcher::read(char curChar, char nextChar) {
    if (getCurrentStatus() == MatchStatus::Reject) return;
    if (_val.empty()) {
        // The first character can be '_' or [a-zA-Z]
        if (!(curChar == '_' || std::isalpha(curChar))) {
            setStatus(MatchStatus::Reject);
        } else if (nextChar == '_' || std::isalpha(nextChar) ||
        ↪  std::isdigit(nextChar)) {
            // Can accept next char
            setStatus(MatchStatus::Reading);
        } else {
            // Cannot accept next char
            setStatus(MatchStatus::Accept);
        }
    } else {
        if (curChar == '_' || std::isalpha(curChar) || std::isdigit(curChar))
        ↪  {
            if (nextChar == '_' || std::isalpha(nextChar) ||
            ↪  std::isdigit(nextChar)) {
                setStatus(MatchStatus::Reading);
            } else {
                setStatus(MatchStatus::Accept);
            }
        } else {
            setStatus(MatchStatus::Reject);
        }
    }
    _val += curChar;
}
```

首先，如果状态为 `Reject`，直接不处理。然后，如果现在读取的是第一个字符，则看 `curChar` 是否为字母或下划线，如果不是，直接设置状态为 `Reject`，否则，再看 `nextChar`，如果还可以接受，那就将状态设置为 `Reading`，表示还可以读取，不能接受的话，就设置为 `Accept`，告诉词法分析器，可以生成一个 Token。不是第一个字符的情况也是类似的。

1.2 Lexer

在前面的基础上，词法分析器其实就只是一个用来管理所有 `Matcher` 的类。

有一个最主要的方法 `getNextToken`，用于获取下一个 Token。在这个函数中，一直从输出流中把当前字符和下个字符交给所有 `Matcher`，直到下面几种情况的任意一种发生：

1. 有 `Matcher` 报错，则词法分析器也报错。
2. 所有 `Matcher` 都拒绝，说明无法识别，报错。
3. 有一个 `Matcher` 在 `Accept` 状态，而且没有其他的 `Matcher` 还在 `Reading` 状态（惰性匹配），说明识别到了一个 Token，返回这个 Token。

其中，词法分析器的错误处理直接通过 `C++` 的 `throw` 语句丢出，交给上层的函数来处理。

和用很多 `if-else` 语句的方法比起来，这种方法可能确实在效率上有些不佳，但是其实也只有 $O(n)$ 的时间复杂度，毕竟在 SysY 语言中并没有多少种 Token（2021 版有 34 个），多个 `Matcher` 带来的常数还是比较小的。

2 语法分析

语法分析使用的是自顶向下的，不带回溯的分析器，并对原文法进行了一些修改。

2.1 AstNode

每个（修改过后的文法的）非终结符都对应一个语法分析树的结点，每个结点类都继承于 `AstNodeBase` 这个类，每个类中都存放着自己的信息，包括自己的子节点等。为了后续设计的便利，对 `AstNode` 这些类使用了访问者模式，每个类都实现一个 `accpet` 方法，用于接受访问者的访问。

```
virtual void accept(AstNodesVisitor &visitor) const override {
    visitor.visit(*this);
}
```

然后在 `AstNodesVisitor` 抽象类中为每个派生类定义一个 `visit` 方法，这样就可以让算法和对象分离，实现一种双分派（double dispatch）的效果，可以让后面的模块更方便地访问语法解析树，而不是为每个结点的派生类添加一个方法。

2.2 Parser

语法分析器首先获取 Lexer 分析出来的 Token 序列，再以 `CompUnit` 为单位，循环解析，直到所有 Token 都用完。对一般的非终结符，都编写了一个函数来解析，每个函数的返回值就是这个非终结符对应的 `AstNode`。

其中，大多数的情况下只需要根据当前的 Token 就可以判断接下来如何推导，只有两处地方用到了向前看：

1. 在 `CompUnit -> Decl | FuncDef` 这里需要往前看 2 个 Token, 如果是左括号, 就推导为 `FuncDef`, 否则是 `Decl`。
2. 在 `UnaryExp -> PrimaryExp | FuncCall | UnaryOp UnaryExp` 中，需要往前看 1 个 Token, 如果是左括号就推导为 `FuncCall`。

所以严谨地来说，我将原来的文法修改为了 $LL(2)$ 文法，但是其实语法分析的效率还是接近线性的。

2.3 BinaryExp

其中，最不同的就是二元表达式的解析部分。原文法的二元表达式（包括乘除、加减、关系、相等性、逻辑与、逻辑或）是左递归的，不能用 LL 解析器来解析这种文法，所以需要进行一些修改。一种方法是直接消除左递归，添加几个非终结符；另一种方法就是把整个表达式都读取出来，用栈和运算符对应的优先级来生成语法解析树。我这里使用的是后者，后者只需要一个函数就可以构造任意的二元表达式，而且代码量少，可维护性好。

算法的基本思想是：首先，定义两个栈，一个栈储存语法解析树的结点，一个栈存运算符。然后，开始从 Token 序列中不断读取一元表达式和运算符，如果当前的运算符优先级小于等于运算符栈顶的运算符优先级，就从结点栈中取出两个结点，从运算符栈中取出一个运算符，构造一个二元表达式，再将这个二元表达式的结点入栈。

2.4 AstDumper

这个类用于输出 AST 树，作为结点的访问者，继承于 `AstNodesVisitor`。通过 `visit` 函数遍历 AST 树，然后输出对应的非终结符和 Token。

3 中间代码生成

这部分是将 AST 树转化为 LLVM IR,参考了 LLVM 的官方教程 https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html，使用 `llvm::IRBuilder` 可以方便地生成 SSA 格式（static single assignment form，静态单赋值形式）的中间代码。

3.1 IrGenerator

`IrGenerator` 也是 AST 结点的访问者，继承于 `AstNodesVisitor`，通过 `visit` 函数遍历语法解析树，然后对每个结点做对应的中间代码生成。有几个比较重要的私有成员：

- **`_context`** 是 LLVM 存储各种数据的对象，包括类型和常量表等。
- **`_builder`** 有许多方法用来生成 LLVM IR，并且管理当前插入指令的位置。
- **`_module`** 用来储存和管理生成的 IR，包括各种函数和全局变量。

- **`_namedValues`** 用来储存当前作用域中的局部变量。
- **`_retBB`** 指向当前函数的返回 BasicBlock，用于处理 `return` 语句。

3.2 IfStmt

对于 `if-else` 语句，可以用伪代码表示如下：

```
    ifcond = (cond == 1)
    如果 ifcond == 1 跳转到 if.then 否则跳转到 if.else
if.then:
    执行 if 中的内容
    跳转到 if.end
if.else:
    执行 else 中的内容
    跳转到 if.end
if.end:
    执行接下来的内容
```

如果没有 `else`，在 `if.else` 中直接跳转到 `if.end`。