PL0设计

编译器结构

PLO编译器的结构如下：

- 全局变量：控制整个编译过程，同时存储一些指令/符号集合
- 错误处理
- 读取符号
- 语句块处理器
- 解释器

接口设计

- `error`：输出错误
- `getsym`：读取下一个token
- `gen`：生成一条编译出的目标代码
- `test`：检查能否访问符号
- `block`：处理一个代码块
  - `enter`：将变量/常量写入符号表
  - `position`：根据标识符查找符号表表索引
  - `constdeclaration`：处理常量声明
  - `vardeclaration`：处理变量生命
  - `listcode`：输出目前已经编译出的目标代码
  - `statement`：处理语句
    - `expression`：处理表达式
    - `term`：处理项
      - `factor`：处理因子
    - `condition`：处理条件表达式
- `interpret`：解释执行编译出的程序

文件结构

pl0编译器仅有一个文件组成。
由main函数调用子程序控制编译过程进行。
编译过程为block函数。
解释过程为interpret函数。

编译器结构

- `Lexer`：词法分析器
- `Parser`：语法分析器
- `Compiler`：一次编译过程

文件组织

- `lexer`: 词法分析相关
- `CMakeLists.txt`: 编译文件，并通过`add_definitions`来控制debug输出与每次作业的输出
- `core`: 核心部分，例如编译过程的抽象
- `parser`: 词法分析相关
- `error`: 错误处理相关
- `symbol`: 符号相关
- `util`: 工具方法
- `main.cpp`: 程序入口

接口设计

Token

Token相关的设计有TokenType和Token,TokenType结合X-Macro实现了enum到string的快速转换；Token存储了TokenType,原内容和行数。

- `Token()`: 创建空的Token
- `Token(Token::TokenType, std::string const&, int)`:创建初始化的Token
- `tokenTypeToString(Token::TokenType)`:将TokenType转换为字符串值。
- `reserve(std::string const&)`:使用了哈希表实现关键字查询。
- `operator<<(std::ostream&, Token const&)`: 将Token输出

Lexer

词法分析器
explicit Lexer(std::istream& in, ErrorReporter& errorReporter) : input*(in),* *errorReporter*(errorReporter) {};

- `Lexer(std::istream&, ErrorReporter&)`:使用输入流创建词法分析器
- `next(Token&)`:获取字符流中的下一个token，这部分使用了有限状态机来实现。在词法分析中，如果出现了a类异常，则抛出CompilerException(ILLEGAL_SYMBOL)，如果是无法识别的字符，则抛出CompilerException(ILLEGAL_CHARACTER)
- `setLogger(std::shared_ptr<Logger>)`:设置输出
- `tokenize(TokenStream&)`:将输入流转化为Token流，将错误输出到ErrorReporter中

TokenStream

Token流

- `put(Token const&)`:将token加入TokenStream末尾
- `peekType(Token::TokenType)`:预读并检查Token类型
- `peekType(int, Token::TokenType)`:预读第offset个Token并检查Token类型
- `peekType(int, std::vector<Token::TokenType>)`:预读并检查Token类型（多个备选项）

- `peekForward(std::function<bool(Token::TokenType)>)`:向前预读直到返回true
- `empty()`:流内是否为空
- `peekForward(std::function<bool(Token const &)>)`:向前预读直到返回true
- `peek()`:返回首个Token
- `peek(int offset)`:返回第offset个Token
- `next()`:取出Token

Parser

- `Parser(TokenStream&, ErrorReporter&)`:创建语法分析器
- `currentToken()`:返回当前Token
- `nextToken()`:取出当前Token
- `tryMatch(Token::TokenType)`:如果类型匹配，取出Token
- `match(Token::TokenType)`:如果类型匹配，取出Token，否则输出错误
- `submit(T&)`:结束语法成分
- `parseXXX()`:分析语法成分

Logger

- `Logger(std::string const&)`:创建日志
- `~Logger()`:完成日志输出
- `stream()`:获取日志流

ErrorReporter

- `error(CompilerException const&)`:保存错误
- `hasErrors()`:是否有错误
- `printErrors(std::shared_ptr<Logger>)`:将错误输出到日志

Compiler

一次编译过程

- `Compiler(std::string const&)`:创建一个编译过程
- `lexer()`:进行词法分析，存储解析的token
- `parse()`:进行语法分析，构建抽象语法树
- `printErrors()`:输出错误

CompilerException

编译过程中的异常

- `errorType`: 异常类型
- `line`: 异常行数

AbstractSyntaxTree

定义了抽象语法树中的各种节点

Symbol

符号基类

- `id`: 符号id
- `scopeId`: 符号作用域
- `ident`: 符号标识符
- `typeString()`: 符号类型字符串
- `symbolType()`: 符号类型：变量/函数

VariableType

变量类型

- `type`: 变量类型，int/char
- `isConst`: 是否是常量
- `isArray`: 是否是数组

VariableSymbol

变量符号

- `type`: 变量类型
- `constVal`: 常数值
- `constVals`: 常量数组

FunctionSymbol

函数符号

- `type`: 返回类型
- `paramTypes`: 形参列表

SymbolTable

符号表

- `SymbolTable(int, std::shared_ptr<SymbolTable>)`:创建符号表
- `hasSymbolInScope(const std::string&)`:当前作用域内是否有该符号
- `findSymbol(const std::string&)`:查找符号
- `addSymbol(std::shared_ptr<Symbol>)`:添加符号
- `getScopeId()`:获取作用域id
- `print(std::ostream&)`:输出符号表

ASTVisitor

抽象语法树访问器，提供了访问每个节点的接口

SymbolTableBuilder

ASTVisitor的实现，用于遍历语法树构建符号表

词法分析

编码前设计：通过某个方法对字符流进行逐字符处理，使用有限状态机来解析token，最后存储到token表中。同时，单独存储错误的token, 方便后续输出

编码完成之后的修改：通过调用`Compiler::lexer`来完成一次编译过程中的词法分析，在`Lexer::tokenize`中不断调用词法分析器`Lexer::next`并记录解析出来的token，并加入`TokenStream`中，如果是错误的token(如`&`,`|`)，则保存错误到`ErrorReporter`。

语法分析

编码前设计：利用词法分析得到的TokenStream,使用递归下降法进行词法分析，通过CompUnit来开始构建抽象语法树。使用预读方式来处理语法成分的多种情况。

编码完成之后的修改：通过调用`Compiler::parse`来完成一次编译过程中的语法分析，在`Parser::parseCompUnit`中开始，并逐级调用`Parser::parseXXX`来分析语法成分，返回构建的抽象语法树节点，如果遇到错误，则保存到`ErrorReporter`中。

符号表

编码前设计： 利用语法分析得到的抽象语法树，遍历语法树分析作用域，创建符号表，并针对所有的声明节点创建符号来加入符号表。

编码完成之后完成的修改：通过`ASTVisitor`来代表一次遍历语法树的过程，并为构建符号表创建一次遍历过程`SymbolTableBuilder`，遇到Block则创建作用域和符号表，遇到Decl则准备插入符号表，如果遇到错误，则保存到`ErrorReporter`中。