

目录

第一章	课程设计题目、目标和要求	3
1.1	题目	3
1.2	目标	3
1.3	要求	3
第二章	需求分析	4
2.1	词法分析	4
2.2	语法分析	5
2.3	语义分析	6
2.3.1	符号表	6
2.3.2	作用域管理	7
2.3.3	类型转换和检查规则	7
2.3.4	错误处理	8
2.4	代码生成	8
2.5	用户接口	9
第三章	开发环境	11
第四章	总体设计	12
4.1	词法分析	12
4.1.1	数据结构设计	12
4.1.2	算法描述	13
4.2	语法分析	13
4.2.1	数据结构说明	13
4.2.2	详细设计说明	14
4.3	语义分析	18
4.3.1	数据结构设计	18
4.3.2	详细设计说明	18
4.4	语义分析表	20
4.4.1	数据结构说明	20
4.5	代码生成	21
4.5.1	数据结构说明	21
4.5.2	详细设计说明	21
4.6	用户接口	23

目录	2
4.6.1 浏览器交互界面	23
4.6.2 命令行界面	24
第五章 程序清单	25
5.1 compiler 文件夹	25
5.1.1 文件夹结构	25
5.1.2 编译器核心代码	26
5.1.3 代码质量	26
5.2 compiler_browser 文件夹	26
5.2.1 文件夹结构	26
5.2.2 浏览器交互界面	27
5.2.3 代码质量	28
5.3 compiler_server 文件夹	28
5.3.1 文件夹结构	28
5.3.2 命令行交互界面	28
5.3.3 代码质量	28
第六章 测试报告	29
6.1 平台测试	29
6.2 单元测试	29
6.2.1 词法分析器单元测试	29
6.2.2 语法分析器单元测试	31
6.2.3 补充测试	32
6.2.4 语义分析器测试	32
6.2.5 图形化界面实际运行测试	34
第七章 实验总结	37
7.1 实验亮点	37
7.2 各模块总结	37
7.2.1 词法分析模块	38
7.2.2 语法分析模块	38
7.2.3 语义分析模块	38
7.2.4 代码生成模块	38
7.2.5 用户接口模块	38
7.3 问题及解决方案	39
7.4 实验成果	39
7.5 实验收获	39
7.6 改进方向	40
附录 A 代码文档	41
附录 B Pascal-S 文法	42

第一章 课程设计题目、目标和要求

1.1 题目

本次课程设计的题目是“Pascal-S 语言编译程序的设计与实现”。

1.2 目标

本次课程设计的目标是：按照所给 Pascal-S 语言的**语法**，参考 Pascal-S 语言的**语义**，**设计并实现** Pascal-S 语言的编译程序。

1.3 要求

本次课程设计需要严格遵循软件工程方法论，采用**分阶段递进式**的开发流程：从需求分析和总体设计入手，依据功能模块合理分配开发任务，在确保各模块独立验证通过的基础上实现系统整体集成，整个开发过程应当采用“滚雪球”式的渐进策略，即以核心功能为起点，逐步向外围功能延伸，通过持续迭代和测试来不断完善系统质量。

第二章 需求分析

我们将分模块进行需求分析。图 2.1给出编译器的数据流图。

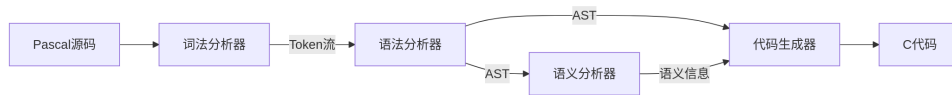


图 2.1: 数据流图

2.1 词法分析

词法分析器需要能够读取源代码，并将其转换为一系列的记号 (Token)。

具体而言，词法分析器要能识别以下类型的记号：

- **关键字 (Keyword)**：Pascal-S 语言的关键字，包括：program、const、var、procedure、function、begin、end、array、of、integer、real、boolean、char、if、then、else、for、to、do、while、read、write、true、false。
- **分隔符 (Delimiter)**：Pascal-S 语言的分隔符，包括：(、)、,、;、.、:、[、]。
- **算术和逻辑运算符 (Algebraic and Logic Operator)**：Pascal-S 语言的算术和逻辑运算符，包括：+、-、*、/、:=、div、mod、and、or、not。
- **关系运算符 (Relational Operator)**：Pascal-S 语言的关系运算符，包括：=、>、>=、<、<=、<>。
- **标识符 (Identifier)**：Pascal-S 语言的标识符，要求：(1) 由大小写字母和数字组成；(2) 由大小写字母开头。例如：abc、a0bc12。

- **数字 (Number)**: Pascal-S 语言的数字字面量, 包括整数和浮点数。整数要求: 由阿拉伯数字组成, 允许前导 0。示例: 1234 ; 浮点数要求: 形式为整数. 整数。示例: 123.45 。
- **字符串 (String)**: Pascal-S 语言的字符串字面量, 形式一对单引号括住的任意长度字符, 不包含换行符 (即字符串不能跨行)。示例: 'abcd' 。

词法分析器输出的记号序列中的每一个记号要包含 (1) 记号类型; (2) 记号所在行; (3) 对于标识符、字面量, 包含具体值。

词法分析器所接受的输入形式为字符串, 即 Pascal-S 语言的源代码, 输出形式为记号的序列。

错误情况包含两类: (1) 遇到非法字符, 例如 @ ; (2) 字符串未闭合, 例如 `const s='ab`。对以这两种错误, 词法分析器应该记录下错误后继续扫描, 以便发现尽可能多的词法错误, 直到扫描结束, 报告错误并终止编译程序。报告内容包括: (1) 出错行号; (2) 错误类型; (3) 错误原文; (4) 修改建议。

Pascal-S 中原无对于科学计数法的需求, 我们额外添加了对于科学计数法的支持, 能够将诸如 `1.2e+3` 的科学计数法表示的数字识别为一个 Number 记号。

2.2 语法分析

语法分析应该能接受词法分析提供的记号流, 根据语法生成式分析得出 AST。

语法分析应当可以分析如下成分:

- **program 的头部**。包含外部的输入参数以及 program 名称
- **program 的主体部分**。包含常量、变量、过程或函数、若干语句。
- **不同类型的定义**。包含 integer, real, boolean, char 这四个基本类型以及延伸出的数组类型。
- **过程和函数的头部**。头部包含其 名称、形参表、返回值类型 (过程没有返回值)。
- **过程和函数的主体**。包含常量、变量、若干语句。
- **语句的识别**。包含 if、for、read、write、赋值、函数调用这六类。
- **表达式的识别**。表达式的结构依次为 expression_list - expression - simple_expression - term - factor 。

在每个成分的识别过程中, 如果出现了错误, 应当进行相应的报错, 包含下面几类错误:

1. 关键字结构错误。如 begin 需要对应 end, 如果没有检测到, 需要报错 expect end after begin ; 或者是 if 后面需要识别到 then 。类似的任何关键字出现缺失时, 都需要提供相应的提示。
2. 成分结构错误。如 + 运算符左右应该是 expression, 而不应该是关键字或者别的, 此时也需要进行相应报错, 说明 + 的运算对象有误, 并说明应该是什么。类似的在什么地方应该出现什么成分, 但是却没有出现相应成分时, 应该进行相应报错。

语法分析的输出应该是一棵 AST, 每个节点对应一个成分。

2.3 语义分析

语义分析阶段的任务是接受通过语法分析阶段生成的抽象语法树（AST）输入，经过语义分析逻辑，确保程序在语法正确的基础上符合语言的语义规则，输出语义信息表。

2.3.1 符号表

语义分析阶段应当构造一个符号表。符号表是编译器用来记录各种标识符（变量、常量、类型、过程/函数）信息的数据结构。

符号表属性存储

记录每个标识符的**完整属性**，包括：

- 名称 (Name)
- 类型 (Type)
 - 基本类型 (`integer`、`real`、`boolean`、`char`)。
 - 复合类型 (`array`，包括维度信息)。
 - 函数/过程类型（包括返回类型和参数列表）。
- 作用域层级 (Scope Level)
- 常量标记 (`isConst`)
- 常量值 (`constValue`)
- 引用参数标记 (`isVar`)
- 参数列表 (`paraList`，每个参数的名称、类型、传递方式 `var` 或值传递)
- 返回值类型 (`returnType`)
- 维度信息（数组/结构体等复合类型）

符号表具备的操作

符号表需要具备如下功能：

- 存储标识符的属性（类型、作用域、存储位置）。
- 支持按作用域递归查找操作。
- 支持检查是否重复声明和类型是否正确的插入操作。
- 支持管理嵌套的作用域。
- 支持类型检查。
- 支持在离开作用域时，对符号表符号的删除操作。

2.3.2 作用域管理

作用域类型

- 全局作用域：
 - 存储 program 中定义的常量、变量、函数/过程。
 - 生命周期贯穿整个程序。
- 过程/函数作用域：
 - 每个 procedure 或 function 都有自己的作用域。
 - 可以访问外层作用域（如全局变量），但 不能访问同层级其他过程的作用域。
- 嵌套块作用域（如 begin ... end）：
 - compound_statement (begin ... end) 可能隐含局部作用域。
 - 如果支持，需处理变量遮蔽（如内层 x 覆盖外层 x）。

2.3.3 类型转换和检查规则

基础类型系统建模

Pascal-S 类型涵盖：

- 简单类型：integer、real、boolean、char。
- 数组：array[1..N] of T。

类型属性记录

- 在符号表中为每个符号（变量、函数等）附加类型信息。

类型检查

- 验证变量、表达式、函数等的类型是否匹配。

表达式类型检查

Pascal-S 是严格类型转换，不允许进行隐式转换。

- 运算符类型兼容性：
 - 检查操作数类型是否满足运算符要求
 - * integer + integer 合法，但 boolean + integer 不合法，其他四则运算同理。
 - * 关系运算符(relop)仅允许同类型比较(如 integer < integer, boolean = boolean), 其他关系运算符同理
 - * 逻辑运算符，如 not，只能对 boolean 运算符
- 赋值兼容性：

- 左值类型必须和右值类型相同（如 `var a : char; a := "hello";` 不合法）
- 数组类型需要检查范围（如 `array[1..10] of integer` 不能赋值为 `array[1..11] of integer`）。
- 控制流类型检查：
 - `if` 或 `while` 的条件必须是 `boolean` 类型

函数/过程调用检查

- 参数类型匹配检查：
 - 参数的实参形参数量和类型保持一致。
 - 处理 `var` 参数（引用传递）的约束：实参必须是左值（如变量，非常量）。
- 返回值检查：
 - 函数必须有返回值，且类型与声明一致。
 - 过程（`procedure`）禁止包含返回值语句。

2.3.4 错误处理

需要检测出以下错误并报告

1. **重复定义**：同一作用域内变量/函数名重复（如 `var x: integer; x: real;`）。
2. **未定义符号**：使用未声明的变量或函数（如 `x := y + z`，但 `y` 未定义）。
3. **作用域违规**：在函数外部访问局部变量（如 `procedure P(); var x: integer; begin end; begin x := 1; end`）。
4. **var 参数错误**：实参不是左值（如 `procedure P(var a: integer); begin end; P(1);` 错误，因为 `1` 不是变量）。
5. **数组越界错误**：检查数组的索引是否在合法范围内。例如，`arr[5]` 是否超出了数组的定义范围。
6. **常量赋值错误**：检查是否对常量进行了赋值操作。例如，`const x = 5; x := 10;`
7. **各种类型不匹配错误**：详见前文类型检查规则实现

2.4 代码生成

代码生成器需要能够接收语义分析提供的语义信息表和语法分析提供的 AST，根据翻译方案得到 C 语言代码。

由于分工规划，翻译方案设计归于代码翻译模块，翻译方案需要符合以下要求：

- 能够包含每个非终结符的 `str` 和 `type` 这两类综合属性，其中 `str` 为目标代码文本，`type` 为该节点对应变量的类型。

- 能够根据一些判定条件选择不同的翻译模式，以正确计算综合属性。如运算符翻译过程不同的变量类型对应不同操作，以及包含一个或多个变量的 idlist 如何读取
- 能够保证在根据设计的运算方法计算后，得到的 str 属性就是最后的目标代码组成部分，即翻译方案需要考虑语法规范。

另一方面，代码生成器需要有以下各类处理逻辑：

- 能够根据语法树中的节点信息以及语义信息表，设置变量名、变量类型、变量值（可能是对于非终结符的属性）这些信息。
- 能够实现对于每个生成式设计好的翻译方案，并且在递归下降地遍历语法树的过程中，求出每个非终结符的属性 str（对应代码文本）和 type（变量或常量节点的类型）。
- 能够根据求出的属性值与翻译方案，从叶子节点逐步生成代码，在向上每层中完成合并，汇总至根节点输出。

代码生成器需要输出 C 语言代码，且与原 Pascal-S 代码功能/逻辑完全相同。

在前面三个步骤执行正确的情况下，代码生成器给出的代码需要能够完成编译与运行，且确保执行结果与原 Pascal-S 程序一致。

2.5 用户接口

用户接口作为编译程序与外界交互的核心模块，其设计质量直接影响用户体验，在系统架构中具有关键作用。

为兼顾系统可访问性、交互多样性及使用便捷性，编译程序应当提供两种用户接口模式：第一种是浏览器交互界面（Web UI），该模式作为主要交互方式，需提供符合现代用户习惯的图形化操作体验；第二种是命令行接口（CLI），该模式作为备用方案，需确保在无浏览器环境等情况下仍能提供基础功能访问能力。下面分别对这两种接口进行需求分析。

在浏览器交互界面的设计中，应当包含以下功能组件：（1）源代码输入文本框，该区域应当支持用户直接输入或粘贴待编译的源代码内容；（2）显著的“编译”功能按钮，当用户点击该按钮时，系统自动将输入的源代码提交至后台服务进行处理；（3）输出文本框，将后台返回的编译结果（C 代码或错误信息）呈现给用户。

除了上述基础功能外，界面还需提供实用辅助功能如下：（1）应当设置“复制”操作按钮，用户可通过点击该按钮一键复制全部编译结果至剪贴板；（2）界面应当支持亮色与暗色两种显示模式的切换功能，以满足不同用户的使用偏好和使用环境需求；（3）应提供“显示历史记录”按钮，用户可以通过此按钮查看编译历史；（4）应提供“选择结果类型”的下拉框，用户可以选择结果类型为词法分析得到的记号流、语法分析得到的语法树，或最终 C 代码。

在交互方面，所有交互元素都应当提供明确的操作反馈，以提升用户体验。包括：按下编译按钮、等待后台结果时，弹出通知“正在编译”，同时这段时间内复制按钮禁用，变为灰色；接收到编译结果后且源代码无错误，弹出“编译成功”；接收到编译结果为错误信息，弹出“编译失败”；用户点击“复制”按钮且复制完成后，弹出“复制成功”。

在视觉设计方面，整体界面应当遵循简洁美观的设计原则。各功能组件的布局应当符合用户操作习惯，保持适当的间距和层次感。配色方案应当专业且舒适，确保文字内容在各种模式下都具有良好的可读性。所有交互元素都应当提供明确的操作反馈，以提升用户体验。

命令行接口应包含以下功能：(1) 通过命令行输入 Pascal-S 源代码，输出编译后的 C 代码或错误信息；(2) 通过指定文件输入 Pascal-S 源代码，输出编译后的 C 代码或错误信息；(3) 输出的 C 代码或错误信息可以输出到命令行，或指定文件。

命令行接口要有充分的提示，例如“请输入源代码”；要提供命令行选项来实现以上的不同功能；要提供 `--help` 输出帮助消息；应当提供选项来输出编译的中间产物，如记号序列、语法树（字符串形式）。

第三章 开发环境

我们的开发环境为 Linux 以及 Windows，使用 C++ 作为开发语言，语言标准采用 C++20。开发过程中使用到了以下工具：

- 编译器：GCC/Clang
- 调试工具：GDB
- 构建工具：CMake
- 第三方库：GoogleTest(后端测试), react-codemirror(文本框), react-hot-toast(通知), jest(前端测试)
- 版本控制：Git, Github
- 文档生成：Doxygen
- 编辑器：VSCode, Vim, CLion
- 前端：React, Next.js
- 后端：Golang
- CI/CD：Github Actions
- 静态分析：clang-tidy
- 代码格式化：clang-format
- 报告撰写： \LaTeX

第四章 总体设计

本编译程序采用**面向对象**的设计原则。每个功能模块对应一个类，并为每个类定义清晰的职责和接口。通过面向对象的封装、继承和多态特性，实现了模块间的低耦合高内聚，确保了代码的可维护性、可扩展性和可重用性。

4.1 词法分析

词法分析器由一个名为 `Lexer` 的类实现。该类接收一个 `string` 类型的输入，并提供 `public` 方法来输出 `Token` 数组。

为实现这一功能，我们首先需要定义相关的数据结构。

4.1.1 数据结构设计

1. `TokenType` 枚举。该枚举列出所有可能的 `Token` 类型，具体类型在需求分析中已经介绍。

2. `Token` 类。该类表示一个记号。

- 成员：
 - 记号类型；
 - 记号值（仅标识符和字面量有值，其他类型为空）；
 - 记号所在行号。
 - 由于 `Token` 类仅用于表示记号且不需要复杂方法，这些成员被设为 `public`。
- 方法：重载两个构造函数以创建不同类型的记号（含/不含记号值）。

3. `Lexer` 类。

- 成员：
 - 源代码字符串 `src` ；
 - `Token` 序列，设计为 `std::vector<Token>` 。
- 方法：
 - 构造函数以接收 `string` 类型的源代码，存储于 `src` 中；
 - 提供 `scan()` 方法进行词法分析；
 - 提供 `getTokens()` 方法获取分析结果；
 - 额外提供 `printTokens()` 方法用来打印词法分析结果，方便调试。

4.1.2 算法描述

`scan()` 函数会不断调用一个内部函数 `scanNextToken()` 来获取下一个记号并添加到 `Token` 数组中,直到字符串结尾。

`scanNextToken()` 函数通过读取下一个字符来确定并生成相应的记号,根据读取到的字符,执行不同的处理逻辑:

- 空格、制表符和回车符:跳过,不做处理。
- 换行符:增加行号计数。
- 特定符号(如`'=',';','(',')','.'`):添加相应的标记。
- 左花括号:跳过注释内容,直到右花括号。
- 数字:调用 `processNumber()` 处理数字。
- 字母:调用 `processKeywordsAndIdentifiersAndAlphaOps()` 处理关键字、标识符以及字母组成的运算符。
- 单引号:调用 `processString()` 处理字符串。
- 其他字符:抛出未知字符异常。

以下是数字、字母、单引号的处理逻辑:

- `processNumber()`:读取连续的数字字符,直到遇到非数字字符或到达源代码末尾。如果遇到小数点,继续读取后续的数字字符,处理浮点数。在读取整数部分或小数部分后,检查是否有科学计数法的符号(`'e'`),并进一步处理。最后,将读取到的数字字面量作为 `Token` 添加到 `Token` 流中。
- `processKeywordsAndIdentifiersAndAlphaOps()`:读取字母和数字的组合,直到遇到非字母或非数字字符。然后判断所读取的单词是否为关键字,或是 `div`、`mod`、`and`、`or`、`not` 这五个由字母组成的运算符。
- `processString()`:读取字符直到字符串结束,并将其作为一个字符串类型的 `Token` 添加到 `Token` 流中。字符串不能跨行,且必须以单引号(`'`)开头和结尾。如果在遇到第二个单引号之前遇到了换行符或源代码结束,则抛出错误:字符串未闭合。

`scan()` 会捕获抛出的异常并记录下来,然后继续执行,以便识别出尽可能多的错误。当到达字符串结尾时,如果存在错误,则输出所有错误信息并终止编译过程;否则在记号数组的末尾添加 `END_OF_FILE` 标记。这个标记并非 Pascal-S 语言的一部分,而是为了方便后续的语法分析而加入的标记。

4.2 语法分析

4.2.1 数据结构说明

1. `Parser` 类用于解析一段 `Token` 流,以生成一棵 `AST`。
2. `ASTNode` 类用于存储 `AST` 的节点,其包含若干子类,会在下面详细介绍。
3. `BasicType` 枚举类中包含 Pascal-S 的四种基本类型,分别是 `integer`、`real`、`boolean`、`char`。

4.2.2 详细设计说明

Parser 类

Parser 类的构造函数接受一个类型为 `std::vector<Token>` 的记号流。Parser 类中包含下面两个成员：

- `std::vector<Token> tokens`；保存即将分析的记号流。
- `std::size_t current = 0`；记录此时分析到 `tokens` 的哪一项。

另外，类中包含下列基本的通用方法，用于识别记号：

- `match()` 方法用于判断接下来的一个记号是否是某个类型的记号，如果是，则返回该记号并使记号流前进一位
- `check()` 方法用于判断接下来的一个记号是否是某个类型的记号，只返回 `true` 或 `false`。
- `isEnd()` 方法用于判断是否到达记号流的末尾。
- `forward()` 让记号流前进一位。
- `backward()` 让记号流后退一位。
- `getToken()` 获取当前记号。
- `consume()` 尝试接收某一类记号，如果下一个记号不是我们想要接收的，则进行报错，否则返回下一个记号并使记号流前进一位。

Parser 类中另外包含若干方法，这些识别函数用于进行语法分析，每个识别函数对应下面的一个 `ASTNode` 的子类，比如 `Program` 对应 `program()` 识别函数，以此类推。每个识别函数内包含不同的错误识别机制，以便提供更加精确的报错信息。同时可以构造出一棵灵活的 AST，便于后面进行语义分析和代码生成。

`ASTNode` 类包含一个基本方法，是：

- `accept()` 用于接受一个 `visitor`，以便遍历树结构。

`ASTNode` 类派生出了若干子类，分别是：

Program 类

用于存储整个 `program` 的信息，包含下列方法和成员（我们采用自解释的命名方法，不再赘述函数和变量的功能，另外每个成员有对应的 `set` 和 `get` 方法，也不再赘述）：

- `program_id`，`std::string` 类型。
- `parameters`，`std::vector<ParameterPtr>` 类型。
- `const_declaration`，`std::vector<ConstDeclPtr>` 类型。
- `var_declaration`，`std::vector<VarDeclPtr>` 类型。
- `subprograms`，`std::vector<SubprogramPtr>` 类型。
- `statements`，`std::vector<StatementPtr>` 类型。

Subprogram 类

用于存储一个子程序，包含下列方法和成员：

- `id` , `std::string` 类型。
- `parameters` , `std::vector<ParameterPtr>` 类型。
- `ret_type` , `BasicType` 类型，对应返回值的类型，如果是 procedure (`is_func` 为 `false`) 则忽略该值。
- `is_func` , `bool` 类型。
- `const_declaration` , `std::vector<ConstDeclPtr>` 类型。
- `var_declaration` , `std::vector<VarDeclPtr>` 类型。
- `compound_statement` , `CompoundStatementPtr` 类型。

Parameter 类

用于存储一个参数的信息，包含下列方法和成员：

- `has_var` , `bool` 类型，标记该参数是否带有 `var` 。
- `id_list` , `std::vector<std::string>` 类型。
- `basic_type` , `BasicType` 类型。

ConstDecl 类

用于存储一个常量的定义，包含下列方法和成员：

- `id` , `std::string` 类型。
- `value` , `std::string` 类型。

VarDecl 类

用于存储一个变量的定义，包含下列方法和成员：

- `id_list` , `std::vector<std::string>` 类型。
- `type` , `TypePtr` 类型。

Type 类

用于存储一个类型，包含下列方法和成员：

- `periods` , `std::vector<std::pair<std::string, std::string>>` 类型，若 `vector` 的 `size` 为 0，说明这不是一个数组类型。
- `basic_type` , `BasicType` 类型。

Variable 类

用于存储一个变量，包含下列方法和成员：

- `id` , `std::string` 类型。
- `expr_list` , `std::vector<ExpressionPtr>` 类型，若 `vector` 的 `size` 为 0，说明这不是一个数组变量。

Statement 类

用于存储一条语句。

NullStatement 类

是 `Statement` 类的子类，表示一条空语句，即一条只含有`;`的语句。

Assign 类

是 `Statement` 类的子类，用于存储一条赋值语句，包含下列方法和成员：

- `left` , `VariablePtr` 类型。
- `right` , `ExpressionPtr` 类型。

ProcedureCall 类

是 `Statement` 的子类，用于存储一个函数调用语句，包含下列方法和成员：

- `id` , `std::string` 类型。
- `parameters` , `std::vector<ExpressionPtr>` 类型。

CompoundStatement 类

是 `Statement` 的子类，一个子块，包含下列方法和成员：

- `statements` , `std::vector<StatementPtr>` 类型。

If 类

是 `Statement` 的子类，用于存储一个条件判断语句，包含下列方法和成员：

- `condition` , `ExpressionPtr` 类型。
- `then_statement` , `StatementPtr` 类型。
- `else_statement` , `StatementPtr` 类型。

For 类

是 `Statement` 的子类，用于存储一个 `for` 循环语句，包含下列方法和成员：

- `id` , `std::string` 类型。
- `lb` , `ExpressionPtr` 类型。
- `rb` , `ExpressionPtr` 类型。
- `body` , `StatementPtr` 类型。

While 类

是 `Statement` 的子类，用于存储一个 `while` 循环语句，包含下列方法和成员：

- `condition` , `ExpressionPtr` 类型。
- `body` , `StatementPtr` 类型。

Read 类

是 `Statement` 的子类，用于存储一个 `read` 语句，包含下列方法和成员：

- `variables` , `std::vector<VariablePtr>` 类型。

Write 类

是 `Statement` 的子类，用于存储一个 `write` 语句，包含下列方法和成员：

- `expressions` , `std::vector<ExpressionPtr>` 类型。

Break 类

是 `Statement` 的子类，用于存储一个 `break` 语句。

FactorType 枚举类

表示 `Factor` 的类型。

Factor 类

用于存储一个 `Factor` 成分，包含下列方法和成员：

- `type` , `FactorType` 类型。当 `Factor` 是变量、函数调用、表达式之一时，`value` 为 `ASTNodePtr`，否则为 `std::string`。
- `value` , `std::variant<std::string, ASTNodePtr>` 类型。

Term 类

用于存储一个 `Term` 成分，包含下列方法和成员：

- `factors` , `std::vector<std::pair<TokenType, FactorPtr>>` 类型。
- `firstFactor` , `FactorPtr` 类型。

SimpleExpression 类

用于存储一个 SimpleExpression 成分，包含下列方法和成员：

- terms , std::vector<std::pair<TokenType, TermPtr>> 类型。
- firstTerm , TermPtr 类型。

Expression 类

用于存储一个 Expression 成分，包含下列方法和成员：

- left, SimpleExpressionPtr 类型。
- right , std::optional<std::pair<RelOp, SimpleExpressionPtr>> 类型。

以上的 SomethingPtr 是指指向 Something 这个类的智能指针,即 std::unique_ptr<Something>。

4.3 语义分析

4.3.1 数据结构设计

1. 符号表项 SymbolEntry：一个作为符号表中项的类，用于记录符号的各种属性。
2. 符号表 SymbolTable：一个用于管理编程语言符号信息的核心类，实现了典型的符号表功能。
3. 作用域栈 ScopeStack：一个用于管理 pascal-s 语言作用域层次的工具类，采用栈结构实现嵌套作用域的动态管理。
4. 错误处理 SemanticError：一个语义错误管理类。
5. 类型 Type：一个类型描述类，用于表示程序中不同类型（基础类型、数组类型、函数或过程类型）的结构和元信息。
6. 语义分析接口 SemanticAnalyzer：一个基于访问者模式（Visitor Pattern）的语义分析组件类，负责对抽象语法树（AST）进行深度遍历。

4.3.2 详细设计说明

SemanticError 类

有如下枚举

- enum class ErrorType {
 DUPLICATE_DEFINITION, // 重复定义
 UNDEFINED_SYMBOL, // 未定义符号
 SCOPE_VIOLATION, // 作用域违规
 VAR_PARAM_ERROR, // var 参数错误
 TYPE_MISMATCH, // 类型不匹配
 ARRAY_INDEX_OUT_OF_BOUNDS, // 数组越界

```
    CONSTANT_ASSIGNMENT, // 常量赋值 OTHER_ERROR // 其他错误  
};
```

此外还有一个静态方法

- `report()` 用于报告错误

SymbolEntry 类

有如下成员：

- `std::string name`; 标识符名称;
- `std::shared_ptr<Type> type`; 类型指针;
- `int scope_level`; 作用域层级;
- `bool is_constant`; 是否为常量;
- `ConstValue const_value`; 常量值;
- `bool is_reference`; 是否为引用参数 (var 参数);
- `int line_number`; 源码行号;
- `std::string returnType`; 返回值类型 (returnType)。

SymbolTable 类

有如下成员：

- `std::unordered_map<std::string, SymbolEntryPtr> entries`; 记录符号条目中的一项;
- `int current_scope`; 记录当前作用域层级;
- `std::weak_ptr<SymbolTable> parent`; 父作用域符号表。

此外类中包含如下方法

- `insert()` 方法, 用于向符号表中插入一项;
- `lookup()` 方法, 用于在当前作用域查找符号;
- `recursiveLookup()` 方法, 用于递归向上搜索父作用域;
- `getCurrentScope()` 方法, 用于获取当前作用域层级。

ScopeStack 类

有如下成员：

- `std::vector<SymbolTablePtr> stack`; 核心的作用域栈, 用来实现嵌套作用域。

此外类中包含如下方法

- `push()` 用来创建并进入新作用域;

- `pop()` 退出当前作用域；
- `current()` 用来获取当前作用域指针；
- `empty()` 检查栈是否为空。

Type 类

有如下成员：

- `TypeKind kind`；存储具体类型的详细信息；
- `std::variant<BasicType, ArrayType, CallableType> type_data`；用来存储具体类型的详细信息。

此外类中包含如下方法

- `createBasicType()` 用来创建简单的类型，表示简单的数据类型（如整数、浮点数等）；
- `createArrayType()` 用来创建多维数组，存储数组的维度范围、元素类型以及数组的总大小；
- `createCallableType()` 用来创建函数或过程的类型，存储返回类型、参数列表、是否嵌套以及局部作用域等信息。

SemanticAnalyzer 类

有如下方法

- `getGlobalSymbolTable()` 用于获取分析结果的符号表；
- `hasError()` 用于检查是否有语义错误；
- 重载了 `visit` 方法，用于接收传进 `visit` 方法的语法符号。

4.4 语义分析表

4.4.1 数据结构说明

语义分析阶段会输出以下含语义信息的数据结构：

- `symbol_table`：符号表，存储了所有的符号信息，包括变量、常量、函数等。
- `is_function_return`：标识一个赋值语句是否是函数返回值。
- `read_fmt_specifier`：标识一个读入语句的格式说明符。
- `write_fmt_specifier`：标识一个输出语句的格式说明符。
- `is_factor_function_call`：标识一个因子是否是函数调用。
- `is_factor_with_not_number`：标识一个 `with not` 类型的因子是否为数字。
- `is_var_param`：标识一个参数是否是引用参数。
- `params_name`：标识一个函数的参数名称。

4.5 代码生成

4.5.1 数据结构说明

`Generator` 类用于分析语法分析产生的 AST 和语义分析阶段产生的语义信息表，以生成目标 C 语言代码，具体实现见下文。`ASTNode` 类用于存储语法树节点，其具体实现已在语法分析模块中介绍。

4.5.2 详细设计说明

算法方面，采用深度优先的方法遍历整棵 AST（起始符号为根节点的 `program`），并且根据下面所示的翻译方案，自底向上逐步合并求出每个节点的综合属性 `result`，最后在根节点对 `result` 即最终生成的 C 语言代码进行输出。

与其他模块的接口方面，本模块作为整个编译器的最后一环，接收语义分析模块传递的 AST（使用重写的 `visit` 函数接收根节点的 `ASTNodePtr`）以及分析完毕的的语义分析表部分内容。

`Generator` 类包含以下 `private` 成员：

- `bool` 类型的 `flag`，用于记录当前在翻译的 `variable` 是否是数组
- `std::string` 类型的 `result`，用于存储翻译得到的 C 语言代码；
- `int` 类型的 `now_level`，用于记录作用域层级，方便在最后的代码中添加相应缩进。
- `int` 类型的 `single_procedure`，用于记录当前表达式嵌套了几层 `procedure_call`，判断是否需要加括号
- `std::vector<std::string>` 类型的 `tmp_periods`，用于临时记录数组的多个维度信息
- `std::vector<std::pair<std::string, std::vector<std::string>>>` 类型的 `arr_bias`，用于记录 Pascal-S 代码定义的数组的起始下标
- `std::map<Assign *, bool>` 类型的 `is_function_return`，用于标识一个赋值语句是否是函数返回值
- `std::map<Read *, std::string>` 类型的 `read_fmt_specifier`，用于将 `read` 语句映射到对应的 `fmt specifier`
- `std::map<Write *, std::string>` 类型的 `write_fmt_specifier`，用于将 `write` 语句映射到对应的 `fmt specifier`
- `std::map<Factor *, bool>` 类型的 `is_factor_function_call`，用于标识一个因子是否是函数调用
- `std::map<Factor *, bool>` 类型的 `is_factor_with_not_number`，用于标识一个 `with_not` 类型的因子是否是数字
- `std::map<std::string, std::vector<bool>>` 类型的 `is_var_param`，用于将子程序名映射到布尔值数组，表示参数是否为引用类型
- `std::map<std::string, std::vector<std::string>>` 类型的 `params_name`，用于将子程序名映射到参数名列表

- `std::string` 类型的 `now_in_which_subprogram`, 用于记录当前正在访问的子程序
- `CompoundStatement *` 类型的 `main_compound_statement`, 用于记录当前正在访问的 `main` 复合语句
- `void` 类型的函数 `addIndent()`, 用于根据 `now_level` 添加缩进

程序设计采用 Visitor 模式, Generator 类继承自 Visitor 基类, 其中包含 20 个不同的继承自 Visitor 的 public 函数, 分别对应不同的 ASTNode 类型, 由语法树的 accept 方法做为接口进行调用。

此外还需要有一个 public 的 getResult 方法用于得到 private 的 str 变量并完成输出。

下面给出对于微调后文法的 result 翻译方案简述, 每条翻译方案会实现在对应的 visit 函数中 (有部分不好表述的内容细节, 如何时需要在变量名前加 * 或 &、不同数据类型对应不同的 and or 符号翻译、数组下标不从 0 开始的处理等, 详见代码实现):

- `Program.result = "\#include<stdio.h>\n" + ConstDecl.result + VarDecl.result + Subprogram.result + "int main(){\n" + Statement.result + "}\n"`
- `Subprogram.result = Retype.result`(若 `isfunc` 为 `false` 则此处为 `void`) + `id.result` + "(" + `Parameter.result`(多个则用逗号分隔开) + "){\n" + `ConstDecl.result` + `VarDecl.result` + " 额外要加上一个 `_return` 为结尾的变量, 类型与函数相同, 作为返回值" + `Statement.result` + ")\n";
- `Parameter.result = type.result + idlist.tr`(用逗号分隔) 如果 `type` 为数组则特殊处理, 在 `id` 后面加上 `[]` 分隔的数组每一维大小; 还需要注意判断是否为 `var` 以判断是否加上 * 符号
- `ConstDecl.result = "const" + basic_type + id + "=" + value + ";\n"`
- `VarDecl.result = type.result + idlist.tr`(用逗号分隔) 如果 `type` 为数组则特殊处理, 在 `id` 后面加上 `[]` 分隔的数组每一维大小 + ";\n"
- `Type.result = basic_type` 由于父节点根据 `periods` 的值进行判断, 故此处无需考虑数组
- `Variable.result = id.result` 后面根据 `expr_list` 的值讨论, 若不为空则是数组, 则 + `[]` 的 `exprlist` 中每一个字串, 代表数组的若干维大小, 否则结束。注意需要查询 `arr_bias` 来减去起始下标。
- `Statement.result = NullStatement.result + Assign.result + ProcedureCall.result + If.result + For.result + While.result + Read.result + Write.result + "\n"`
- `NullStatement.result = ""` 即空字符串
- `Assign.result = variable.result + "=" + Expression.result`
- `ProcedureCall.result = id.result + "(" + parameters.result` ” 此处注意检查是否为 `var`, 以判断是否需要加上 & 符号” + ");\n"
- `If.result = "if(" + Expression.result + "){\n" + Statement.result + "}\n"` 如果有 `else` 则 + `"else{\n" + Expression.result + "}\n"`

- `For.result = "for(" + id.result + "=" + lb + ";" + id.result + "<=" + rb + ";" + id.result + "++){\\n" + Statement.result + "}\\n"`
- `While.result = "While(" + Expression.result + ")\\n" + Statement.result + "}\\n"`
- `Read.result = "scanf(" + \% 某种变量对应的输入格式, 此处需要查符号表得到 type + "," + variables.result + ");\\n"` 如果读入变量有多个则逗号空格隔开
- `Write.result = "printf(" + \%variable 或 expression 对应类型, 此处需要查符号表 + "," + Expression.result + ");\\n"` 如果多个则逗号分隔
- `Break.result = "break;\\n"`
- `FactorType` 无需翻译, 只是用于下面翻译 `factor` 时指明类型
- `Factor.result = value`(此处需要根据 `factor` 的类型判断 `value` 的内容以什么形式读出, 若 `factor` 为表达式则 `Expression.result` 否则为 `Variable.result`)
- `Term.result = Factor.result` 如果为多个因子 + `"* / % & && 其中之一` (可能需要根据 `factor` 类型决定) + `Factor.result`
- `SimpleExpression.result = Term.result` 如果为多个项拼成则 + `"+ - || | 其中之一` (可能需要根据 `term` 类型决定) + `Term.result`
- `Expressions.result = SimpleExpression.result` 如果为多个简单表达式拼成则 + `"> = < <= >= <> 其中之一"` + `SimpleExpression.result`

4.6 用户接口

4.6.1 浏览器交互界面

浏览器交互界面采用 React 框架开发, 凭借其高效的虚拟 DOM 和组件化结构, 能够快速更新界面并保持 UI 一致性, 从而显著提升用户体验。

输入输出框使用 `react-codemirror` 文本编辑器组件, 支持代码的实时语法高亮。输入框为可编辑状态, 供用户输入源代码; 输出框为只读状态, 用于显示编译结果。

前后端通过 RESTful API 进行通信。当用户点击“编译”按钮时, 系统会以 JSON 格式发送 POST 请求至后端, 格式为 `"code": "<source code>", "mode": <mode number>`, 其中 `<source code>` 为用户输入的源代码, `<mode number>` 是一个表示结果类型的数字, 0 表示记号流, 1 表示语法树, 2 表示 C 代码。后端返回的响应也为 JSON 格式:

- 若编译成功, HTTP 状态码为 200, 响应格式为 `"result": "<result code>"`, 其中 `<result code>` 为编译后的代码。
- 若编译失败, HTTP 状态码为 422, 响应格式为 `"message": "<error message>"`, 其中 `<error message>` 为具体的错误信息。此时, 输出框的语法高亮功能将被禁用。

为了实现历史记录功能, 每次编译请求都会被存储在浏览器的 `localStorage` 中。历史记录包含源代码和编译结果, 界面通过表格形式展示, 用户可以轻松查看之前的编译任务, 无需重复操作。点击“显示历史记录”按钮时, 历史记录表格将在可见与不可见状态之间切换。

此外，浏览器界面还支持缓存功能：如果用户提交的源代码与之前相同，系统将直接使用缓存结果，而无需再次向后端发送请求，从而减少网络开销并提升效率。

通知功能使用 `react-hot-toast` 库实现，提供了简洁优雅的用户提示。例如，编译成功、失败或复制结果时，系统会弹出相应的通知。

亮色/暗色模式切换功能基于 CSS 媒体查询实现。界面的背景色、字体颜色以及编辑器的配色方案会根据系统的夜间模式设置自动调整，以满足不同用户的使用偏好。

数据结构

设计了 `HistoryItem` 类，用于存储编译历史记录的一项，从而实现历史记录以及缓存功能。`HistoryItem` 类包含以下成员：

- `mode`：存储结果类型的数字，0 表示记号流，1 表示语法树，2 表示 C 代码。
- `input`：存储源代码的字符串。
- `output`：存储编译结果的字符串。

4.6.2 命令行界面

命令行界面使用 Golang 实现。Golang 不仅适合构建高性能的命令行工具，还能轻松开启 Web 服务，其内置的 `net/http` 库能够高效处理 HTTP 请求和响应。

通过使用 `kong` 库解析命令行参数，命令行界面提供了以下功能选项：

- `browser`：启动浏览器交互界面，用户可以通过浏览器访问编译器。
- `cli`：启动命令行交互界面，用户可以通过命令行输入 Pascal-S 源代码进行编译。
 - `-t` 或 `--terminator`：指定终止符，用户可以自定义输入源代码的结束标志。
- `file`：指定输入文件路径，用户可以通过文件输入 Pascal-S 源代码进行编译。
 - `-i` 或 `--input`：指定输入文件路径，用户可以通过文件输入 Pascal-S 源代码进行编译。
 - `-o` 或 `--output`：指定输出文件路径，用户可以将编译结果输出到指定文件。
- `-h` 或 `--help`：显示帮助信息，详细说明各选项的使用方法。
- `-s` 或 `--show`：显示编译过程中生成的中间产物，例如记号序列或语法树（以字符串形式）。

通过 `.env` 文件设置监听端口，默认端口为 8080。用户可以通过环境变量 `PORT` 自定义端口号，使用了 `godotenv` 来解析 `.env` 文件。

在 `browser` 模式下，命令行界面会输出当前监听的端口号以及相关日志信息（如接收请求和发送响应的记录）。后端程序在接收到请求后，会调用编译器执行编译操作，并将结果返回给前端。

在 `cli` 模式下，用户可以直接在命令行中输入 Pascal-S 源代码。为标识输入结束，用户需输入指定的终止符（默认为 `EOF`，可自定义）。程序随后调用编译器完成编译，并将结果直接输出到命令行。

在 `file` 模式下，用户可以指定输入文件路径，程序会读取文件内容并进行编译。编译结果输出到用户自定义输出路径中。

第五章 程序清单

程序包括三个文件夹: `compiler`、`compiler_browser` 和 `compiler_server`。其中 `compiler` 文件夹包含编译器的核心代码, `compiler_browser` 文件夹包含浏览器交互界面的代码, `compiler_server` 文件夹包含命令行交互界面的代码。

5.1 `compiler` 文件夹

5.1.1 文件夹结构

```
compiler
├── CMakeLists.txt
├── include
│   ├── ast.hpp
│   └── ...
├── src
│   ├── ast.cpp
│   └── ...
├── test
│   ├── CMakeLists.txt
│   ├── lexer_test.cpp
│   └── parser_test.cpp
├── third_party
│   ├── CMakeLists.txt
│   └── googletest
├── build
├── compile_commands.json
├── Doxyfile
├── doc
│   ├── html
│   │   ├── index.html
│   │   └── ...
│   └── latex
├── LICENSE
└── ...
```

5.1.2 编译器核心代码

以下列出了一些核心的代码文件：

- `ast.h`：定义了抽象语法树的节点类，包括程序、子程序、参数、常量声明、变量声明、类型、变量、语句等。
- `lexer.h` & `lexer.cpp`：定义且实现了词法分析器，负责将源代码转换为记号流。
- `parser.h` & `parser.cpp`：定义且实现了语法分析器，负责将记号流转换为抽象语法树。
- `semantic_analyzer.h` & `semantic_analyzer.cpp`：定义且实现了语义分析器，负责检查语法树的语义正确性，并输出含有语义信息的数据结构供后续代码生成使用。
- `generator.h` & `generator.cpp`：定义且实现了代码生成器，负责将语法树转换为 C 语言代码。
- `main.cpp`：程序的入口文件，负责输入数据并调用各个模块进行编译。
- `main_submit.cpp`：提交到 OJ 的主函数。
- `test_lexer.cpp`：测试词法分析器的单元测试。
- `test_parser.cpp`：测试语法分析器的单元测试。

5.1.3 代码质量

使用 `clang-format` 进行代码格式化，确保代码风格一致。

使用 `clang-tidy` 进行静态代码分析，检查潜在的错误和不规范的代码。

使用 `clang-tidy` 约束命名规范，确保变量、函数和类名符合命名规则。命名规范如下：

- 类名：使用驼峰命名法（`CamelCase`）。
- 枚举名：使用驼峰命名法（`CamelCase`）。
- 函数名：使用小驼峰命名法（`camelBack`）。
- 全局常量名：使用全大写字母（`UPPER_CASE`）。
- 成员变量名：使用小写字母加下划线（`lower_case`）。
- 变量名：使用小驼峰命名法（`camelBack`）。

5.2 compiler_browser 文件夹

5.2.1 文件夹结构

```
compiler_browser
├── src
│   └── app
│       ├── layout.tsx
│       └── page.tsx
```

```
├── global.css
├── favicon.ico
├── components
│   ├── button.tsx
│   ├── container.tsx
│   └── history.tsx
├── docs
├── eslint.config.mjs
├── jest.config.ts
├── next.config.ts
├── next-env.d.ts
├── node_modules
├── package.json
├── pnpm-lock.yaml
├── postcss.config.mjs
├── public
├── README.md
├── tailwind.config.ts
├── test
│   ├── button.test.tsx
│   ├── container.test.tsx
│   ├── history.test.tsx
│   ├── page.test.tsx
│   └── tsconfig.json
└── ...
```

5.2.2 浏览器交互界面

以下列出了一些核心的代码文件：

- `src/app/layout.tsx`：定义了浏览器交互界面的布局，包括头部、输入框、输出框和按钮等。
- `src/app/page.tsx`：定义了浏览器交互界面的主要页面逻辑，包括编译按钮的点击事件处理。
- `src/app/_components/button.tsx`：定义了编译按钮的组件。
- `src/app/_components/container.tsx`：定义了输入输出框的容器组件。
- `src/app/_components/history.tsx`：定义了历史记录组件。
- `src/app/global.css`：定义了浏览器交互界面的全局样式，包括亮色/暗色模式的切换。
- `src/app/favicon.ico`：浏览器交互界面的图标。
- `src/docs`：包含了浏览器交互界面的文档。
- `test/*`：包含了浏览器交互界面的单元测试。
- `package.json`：定义了浏览器交互界面的依赖项和脚本。

5.2.3 代码质量

使用 `prettier` 进行代码格式化，确保代码风格一致。

使用 `eslint` 进行静态代码分析，检查潜在的错误和不规范的代码。

5.3 `compiler_server` 文件夹

5.3.1 文件夹结构

```
compiler_server
├── browser.go
├── cli.go
├── COMPILER
├── file.go
├── go.mod
├── go.sum
├── main.go
└── run.go
```

5.3.2 命令行交互界面

以下列出了一些核心的代码文件：

- `main.go`：命令行交互界面的入口文件，负责解析命令行参数并启动相应的模式。
- `cli.go`：实现了命令行交互界面的逻辑，包括输入源代码、编译和输出结果。
- `browser.go`：实现了浏览器交互界面的逻辑，包括启动 Web 服务和处理 HTTP 请求。
- `file.go`：实现了文件输入输出的逻辑。
- `run.go`：实现了调用预编译好的编译器二进制文件的逻辑。
- `go.mod`：定义了命令行交互界面的依赖项。
- `go.sum`：定义了命令行交互界面的依赖项的版本信息。
- `COMPILER`：预编译好的编译器二进制文件。

5.3.3 代码质量

使用 `go fmt` 进行代码格式化，确保代码风格一致。

第六章 测试报告

我们将测试分为了：平台测试、单元测试和图形化界面实际运行测试，确保了程序的正确性和鲁棒性。

测试环境为：

OS: Ubuntu 22.04.5 LTS x86_64

Host: 82JW Lenovo Legion R70002021

Kernel: 6.8.0-59-generic

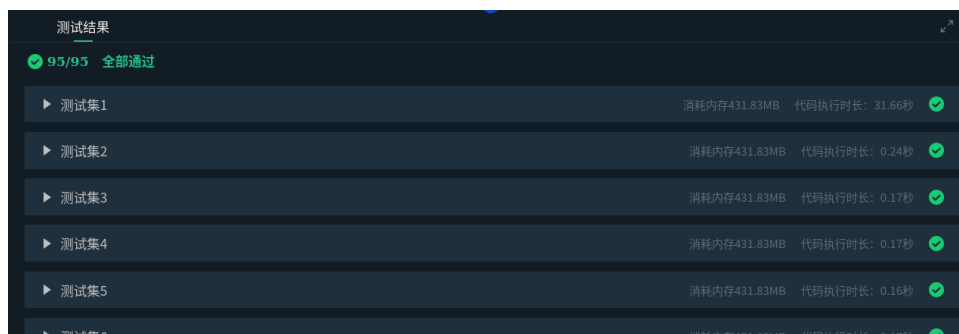
CPU: AMD Ryzen 7 5800H with Radeon Graphics (16) @ 4.463GHz

GPU: NVIDIA GeForce RTX 3050 Mobile

Memory: 5663MiB / 15841MiB

6.1 平台测试

我们通过了头歌平台的所有测试用例，测试结果如图 6.1。



测试结果		
95/95 全部通过		
▶ 测试集1	消耗内存431.83MB 代码执行时长: 31.66秒	✓
▶ 测试集2	消耗内存431.83MB 代码执行时长: 0.24秒	✓
▶ 测试集3	消耗内存431.83MB 代码执行时长: 0.17秒	✓
▶ 测试集4	消耗内存431.83MB 代码执行时长: 0.17秒	✓
▶ 测试集5	消耗内存431.83MB 代码执行时长: 0.16秒	✓
▶ 测试集6	消耗内存431.83MB 代码执行时长: 0.17秒	✓

图 6.1: 头歌平台测试结果

6.2 单元测试

6.2.1 词法分析器单元测试

我们对词法分析器进行了单元测试。测试内容包括：

1. **关键字**：词法分析器是否能正确识别所有的关键字。
2. **数字**：词法分析器是否能正确识别所有的数字，包括整数和浮点数。
3. **分隔符**：词法分析器是否能正确识别所有的分隔符。

4. **运算符**：词法分析器是否能正确识别所有的运算符。
5. **关系运算符**：词法分析器是否能正确识别所有的关系运算符。
6. **标识符**：词法分析器是否能正确识别所有的标识符。
7. **字符常量**：词法分析器是否能正确识别所有的字符常量。
8. **注释**：词法分析器是否能正确识别所有的注释。
9. **一般情况**：词法分析器是否能正确识别一般情况下的记号。
10. **未知字符**：词法分析器是否能正确处理未知字符。
11. **未闭合的字符常量**：词法分析器是否能正确处理未闭合的字符常量。
12. **超长标识符**：词法分析器是否能正确处理超长的标识符。
13. **多个异常情况**：词法分析器是否能正确处理多个异常情况。
14. **正常情况下的异常情况**：词法分析器是否能正确处理正常情况下的异常情况。
15. **科学计数法**：词法分析器是否能正确识别科学计数法。

测试方法为：为词法分析器输入一段字符串，调用词法分析器的 `scan()` 方法进行词法分析，然后调用词法分析器的 `getTokens()` 方法获取分析结果，最后将分析结果与预期结果进行比较。

我们使用了 Google Test 框架进行单元测试，测试代码位于 `test/lexer_test.cpp` 文件中。测试结果如图 6.2。

```
~/projects/compiler/compiler/build/test (main*) » ./lexer_test
Running main() from /home/lyh/projects/compiler/compiler/third_party/googletest/googletest/src/gtest_main.cc
[=====] Running 15 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 15 tests from LexerTest
[ RUN      ] LexerTest.Keywords
[ OK       ] LexerTest.Keywords (0 ms)
[ RUN      ] LexerTest.Number
[ OK       ] LexerTest.Number (0 ms)
[ RUN      ] LexerTest.Delimiters
[ OK       ] LexerTest.Delimiters (0 ms)
[ RUN      ] LexerTest.Operators
[ OK       ] LexerTest.Operators (0 ms)
[ RUN      ] LexerTest.RelationalOperators
[ OK       ] LexerTest.RelationalOperators (0 ms)
[ RUN      ] LexerTest.Identifier
[ OK       ] LexerTest.Identifier (0 ms)
[ RUN      ] LexerTest.CharLiteral
[ OK       ] LexerTest.CharLiteral (0 ms)
[ RUN      ] LexerTest.Comment
[ OK       ] LexerTest.Comment (0 ms)
[ RUN      ] LexerTest.General
[ OK       ] LexerTest.General (0 ms)
[ RUN      ] LexerTest.UnknownChar
[ OK       ] LexerTest.UnknownChar (0 ms)
[ RUN      ] LexerTest.UnclosedCharLiteral
[ OK       ] LexerTest.UnclosedCharLiteral (0 ms)
[ RUN      ] LexerTest.VeryLongIdentifier
[ OK       ] LexerTest.VeryLongIdentifier (0 ms)
[ RUN      ] LexerTest.MultipleExceptions
[ OK       ] LexerTest.MultipleExceptions (0 ms)
[ RUN      ] LexerTest.NormalWithException
[ OK       ] LexerTest.NormalWithException (0 ms)
[ RUN      ] LexerTest.ScientificNotation
[ OK       ] LexerTest.ScientificNotation (0 ms)
[-----] 15 tests from LexerTest (0 ms total)
[-----] Global test environment tear-down
[=====] 15 tests from 1 test suite ran. (0 ms total)
[ PASSED ] 15 tests.
```

图 6.2: 词法分析器单元测试结果

6.2.2 语法分析器单元测试

我们对语法分析器进行了单元测试。测试内容包括：

1. **ABareProgram**：语法分析器能否正确分析一个最简单的程序。
2. **SubprogramTest_Procedure**：语法分析器能否正确分析一个子程序（过程）。
3. **SubprogramTest_Function**：语法分析器能否正确分析一个子程序（函数）。
4. **ParameterTest**：语法分析器能否正确分析参数。
5. **ConstDeclarationTest**：语法分析器能否正确分析常量声明。
6. **VarDeclarationTest**：语法分析器能否正确分析变量声明。
7. **TypeTest**：语法分析器能否正确分析类型。
8. **TypeTest_Array**：语法分析器能否正确分析数组类型。
9. **VariableTest**：语法分析器能否正确分析变量。
10. **VariableTest_Array**：语法分析器能否正确分析数组变量。
11. **AssignTest**：语法分析器能否正确分析赋值语句。
12. **ProcedureCallTest**：语法分析器能否正确分析过程调用。
13. **CompoundStatementTest**：语法分析器能否正确分析复合语句。
14. **IfStatementTest**：语法分析器能否正确分析 if 语句。
15. **IfStatementTest_WithElse**：语法分析器能否正确分析带 else 的 if 语句。
16. **ForStatementTest**：语法分析器能否正确分析 for 语句。
17. **WhileStatementTest**：语法分析器能否正确分析 while 语句。
18. **ReadStatementTest**：语法分析器能否正确分析读语句。
19. **WriteStatementTest**：语法分析器能否正确分析写语句。
20. **BreakStatementTest**：语法分析器能否正确分析 break 语句。
21. **FactorTest**：语法分析器能否正确分析因子。
22. **FactorTest_Parentheses**：语法分析器能否正确分析带括号的因子。
23. **FactorTest_Number**：语法分析器能否正确分析数字因子。
24. **FactorTest_Boolean**：语法分析器能否正确分析布尔因子。
25. **FactorTest_Function**：语法分析器能否正确分析函数因子。
26. **FactorTest_WithNot**：语法分析器能否正确分析带 not 的因子。
27. **FactorTest_WithUminus**：语法分析器能否正确分析带负号的因子。

28. **TermTest**: 语法分析器能否正确分析项。
29. **SimpleExpressionTest**: 语法分析器能否正确分析简单表达式。
30. **ExpressionTest**: 语法分析器能否正确分析表达式。
31. **MatchTest**: 语法分析器的 `match()` 方法能否正确匹配记号。
32. **CheckTest**: 语法分析器的 `check()` 方法能否正确检查记号。
33. **IsEndTest**: 语法分析器的 `isEnd()` 方法能否正确判断是否到达记号流的末尾。
34. **GetTokenTest**: 语法分析器的 `getToken()` 方法能否正确获取下一个记号。
35. **ForwardTest**: 语法分析器的 `forward()` 方法能否正确前进记号流。
36. **BackwardTest**: 语法分析器的 `backward()` 方法能否正确后退记号流。
37. **ConsumeTest**: 语法分析器的 `consume()` 方法能否正确消费记号。
38. **ConsumeTest_Fail**: 语法分析器的 `consume()` 方法能否正确处理消费失败的情况。
39. **ARealProgram**: 语法分析器能否正确分析一个完整的程序。
40. **ErrorHandling1**、**ErrorHandling2**、**ErrorHandling3**: 语法分析器能否正确处理错误情况。

测试方法为:为语法分析器输入一个 `std::vector<Token>`,调用其 `program()`、`subprogram()` 等方法进行语法分析,由于除了 `program()` 方法外,其他方法均为私有方法,我们使用了 Google Test 框架的 `FRIEND_TEST` 宏,这可以使得测试函数能够访问到私有成员变量和方法。

我们实现了一个 `ParserTester` 类,继承自 `Visitor` 类,并实现了其 `visit` 方法,思路为:向一个 `std::vector<std::string>` 中添加每一个节点的信息,以一种类似深度优先遍历的方式遍历语法树,这样可以保证形成的字符串数组与语法树的结构一致且唯一。然后将这个字符串数组与预期结果进行比较,判断语法分析器是否正确。

我们使用了 Google Test 框架进行单元测试,测试代码位于 `test/parser_test.cpp` 文件中。测试结果如图 6.3。

6.2.3 补充测试

对于 `Expression`、`SimpleExpression`、`Term`、`Factor` 这些较为复杂和关键的成分,我们补充了一些测试,测试代码位于 `test/parser_test_supplement.cpp` 文件中。测试结果如图 6.4。

6.2.4 语义分析器测试

语义分析阶段的主要任务为:为代码生成阶段提供必要的语义信息;识别并报告语义错误。为了测试语法分析器,我们构造了几个常见的包含语义错误的代码样例,位于 `semantic_errors` 文件夹中。

测试内容包括:

1. **数组越界**: 语法分析器应该识别数组越界并报错。
2. **重定义常量**: 语法分析器应该识别常量重定义并报错。


```

[ OK ] ParserTest.FactorTest_Function (0 ms)
[ RUN ] ParserTest.FactorTest_WithNot
[ OK ] ParserTest.FactorTest_WithNot (0 ms)
[ RUN ] ParserTest.FactorTest_WithUminus
[ OK ] ParserTest.FactorTest_WithUminus (0 ms)
[ RUN ] ParserTest.TermTest
[ OK ] ParserTest.TermTest (0 ms)
[ RUN ] ParserTest.SimpleExpressionTest
[ OK ] ParserTest.SimpleExpressionTest (0 ms)
[ RUN ] ParserTest.ExpressionTest
[ OK ] ParserTest.ExpressionTest (0 ms)
[ RUN ] ParserTest.MatchTest
[ OK ] ParserTest.MatchTest (0 ms)
[ RUN ] ParserTest.CheckTest
[ OK ] ParserTest.CheckTest (0 ms)
[ RUN ] ParserTest.IsEndTest
[ OK ] ParserTest.IsEndTest (0 ms)
[ RUN ] ParserTest.GetTokenTest
[ OK ] ParserTest.GetTokenTest (0 ms)
[ RUN ] ParserTest.ForwardTest
[ OK ] ParserTest.ForwardTest (0 ms)
[ RUN ] ParserTest.BackwardTest
[ OK ] ParserTest.BackwardTest (0 ms)
[ RUN ] ParserTest.ConsumeTest
[ OK ] ParserTest.ConsumeTest (0 ms)
[ RUN ] ParserTest.ConsumeTest_Fail
[ OK ] ParserTest.ConsumeTest_Fail (0 ms)
[ RUN ] ParserTest.ARealProgram
[ OK ] ParserTest.ARealProgram (0 ms)
[ RUN ] ParserTest.ErrorHandling1
[ OK ] ParserTest.ErrorHandling1 (0 ms)
[ RUN ] ParserTest.ErrorHandling2
[ OK ] ParserTest.ErrorHandling2 (0 ms)
[ RUN ] ParserTest.ErrorHandling3
[ OK ] ParserTest.ErrorHandling3 (0 ms)
[-----] 42 tests from ParserTest (1 ms total)

[-----] Global test environment tear-down
[=====] 42 tests from 1 test suite ran. (1 ms total)
[ PASSED ] 42 tests.

```

图 6.3: 语法分析器单元测试结果

```

~/projects/compiler/compiler (main*) » ./build/test/parser_test_supplement
Running main() from /home/lyh/projects/compiler/compiler/third_party/googletest/googletest/src/gtest_main.cc
[=====] Running 5 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 5 tests from ParserTestSupplement
[ RUN ] ParserTestSupplement.Factor
[ OK ] ParserTestSupplement.Factor (0 ms)
[ RUN ] ParserTestSupplement.Term
[ OK ] ParserTestSupplement.Term (0 ms)
[ RUN ] ParserTestSupplement.SimpleExpression
[ OK ] ParserTestSupplement.SimpleExpression (0 ms)
[ RUN ] ParserTestSupplement.Expression
[ OK ] ParserTestSupplement.Expression (0 ms)
[ RUN ] ParserTestSupplement.ExpressionComplex
[ OK ] ParserTestSupplement.ExpressionComplex (0 ms)
[-----] 5 tests from ParserTestSupplement (0 ms total)

[-----] Global test environment tear-down
[=====] 5 tests from 1 test suite ran. (0 ms total)
[ PASSED ] 5 tests.

```

图 6.4: 语法分析器补充测试结果

3. **重定义变量**：语法分析器应该识别变量重定义并报错。
4. **重定义函数**：语法分析器应该识别函数重定义并报错。
5. **重复参数**：语法分析器应该识别函数参数重复定义并报错。
6. **尝试修改常量**：语法分析器应该识别尝试修改常量并报错。
7. **未定义数组**：语法分析器应该识别未定义的数组并报错。
8. **未定义函数**：语法分析器应该识别未定义的函数并报错。
9. **未定义变量**：语法分析器应该识别未定义的变量并报错。
10. **函数参数作用域外使用**：语法分析器应该识别函数参数在函数外使用并报错。
11. **函数作用域外使用变量**：语法分析器应该识别函数定义的变量在函数外使用并报错。
12. **局部作用域外使用变量**：语法分析器应该识别局部作用域外使用变量并报错。

测试方法为：编写了一个脚本 `test_semantic_errors.sh`，运行后会将输出的报错结果存到 `output_for_semantic_errors` 文件夹中，人工检查输出的报错结果是否与预期结果一致。

经检查，所有的语义错误均能被正确识别并报错。图 6.5 为部分测试结果。

```
~/projects/compiler/compiler (main) » ./test_semantic_errors.sh > /dev/null 2>&1  
  
~/projects/compiler/compiler (main) » cat output_for_semantic_errors/array_index_out_of_boundary.log  
[Semantic Error] Array out of bounds: Array index 0 is out of range [1..5] for array 'arr' at dimension 1  
[Semantic Error] Array out of bounds: Array index 6 is out of range [1..5] for array 'arr' at dimension 1  
  
~/projects/compiler/compiler (main) » cat output_for_semantic_errors/redefined_constants.log  
[Semantic Error] Redefinition: identifier 'pi' is already defined in the current scope  
  
~/projects/compiler/compiler (main) » cat output_for_semantic_errors/try_modify_constant.log  
[Semantic Error] Constant assignment error: cannot to constants 'pi' assign values  
  
~/projects/compiler/compiler (main) » cat output_for_semantic_errors/undefined_array.log  
[Semantic Error] Undefined symbol: Use undeclared variables 'arr'
```

图 6.5: 语义分析器测试结果

6.2.5 图形化界面实际运行测试

我们对图形化界面进行了实际运行测试。测试内容包括：

1. **编译成功**：输入正确的 Pascal-S 代码，编译器应该能够成功编译并输出 C 代码。
2. **编译失败**：输入错误的 Pascal-S 代码，编译器应该能够识别错误并输出错误信息。
3. **历史记录功能**：编译器应该能够正确保存历史记录，并在用户请求时显示。
4. **亮色/暗色模式切换**：浏览器界面应该能够正确切换亮色和暗色模式。
5. **代码高亮**：浏览器界面应该能够正确高亮 Pascal-S 和 C 代码。

图 6.6、6.7、6.8、6.9 为部分测试截图。



图 6.6: 编译成功



图 6.7: 编译失败



图 6.8: 历史记录功能

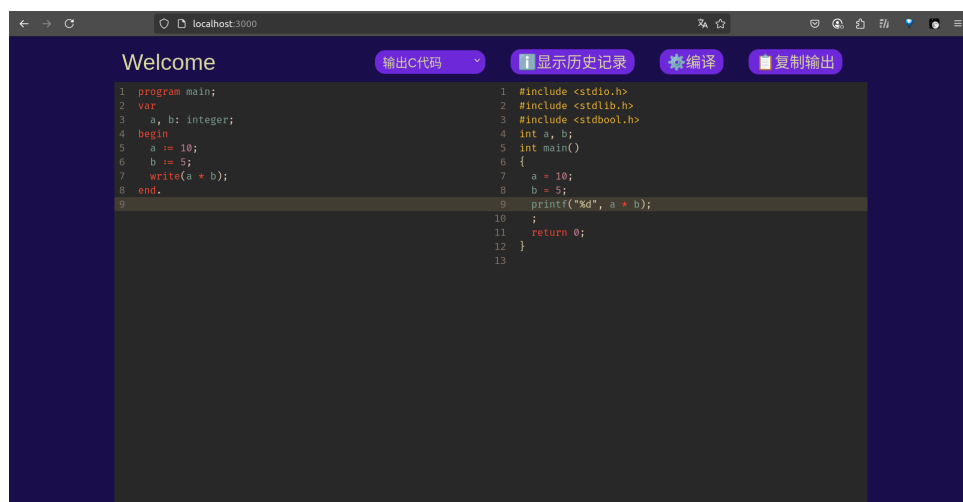


图 6.9: 亮色/暗色模式切换

第七章 实验总结

在本次实验中，我们设计并实现了一个将 Pascal-S 语言翻译成 C 语言的翻译器。Pascal-S 是 Pascal 语言的一个子集，具有相对简单但完整的语法结构。在实验的过程中，我们对编译、翻译的过程有了更加深刻的认识，提升了设计、编码的能力以及团队合作的能力，收获了完成项目的宝贵经验，并对 Pascal-S 和 C 语言的特性有了更进一步的认识。

翻译器采用经典的编译器设计架构，包含下面四个主要模块：

1. 词法分析器 (Lexer)
2. 语法分析器 (Parser)
3. 语义分析器 (Semantic Analyzer)
4. 代码生成器 (Code Generator)

7.1 实验亮点

我们的实现有如下亮点：

1. **语法拓展**：我们额外添加了对科学计数法的支持，允许用户在 Pascal-S 代码中使用科学计数法表示浮点数。
2. **自主实现**：我们自主实现了词法分析器、语法分析器、语义分析器和代码生成器，避免了使用现成的工具（如 Flex 和 Bison），加深了对编译原理的理解。
3. **多种交互方式**：我们实现了多种交互方式，包括浏览器、命令行、文件三种方式，用户可以根据自己的需求选择合适的方式进行编译。
4. **多种编译模式**：我们实现了多种编译模式，可以输出不同的中间产物，包括记号流、语法树、C 语言代码等，方便用户进行调试和分析。
5. **图形化用户界面**：我们实现了一个图形化用户界面，采用 B/S 架构，用户可以通过浏览器进行交互，支持亮色/暗色模式切换、代码高亮显示、历史记录功能等。
6. **代码质量**：我们使用了 clang-format、clang-tidy、prettier 和 eslint 等工具进行代码格式化和静态分析，确保代码风格一致且符合规范。

7.2 各模块总结

下面介绍了每个模块完成的具体功能。

7.2.1 词法分析模块

- 实现了有限状态自动机 DFA 来识别 Pascal-S 的各种词法单元；
- 能够识别关键字（如 `begin`, `end`, `if`, `then` 等）、标识符、常量、运算符和分隔符；
- 处理了对空白字符的过滤；
- 输出统一的记号（Token）流供语法分析器使用。

7.2.2 语法分析模块

- 采用递归下降分析法实现，对语法成分进行了整理，去除了语法生成式中对于语法分析冗余的成分。
- 实现了语法错误检测机制；
- 生成抽象语法树（AST）给语义分析器使用。

7.2.3 语义分析模块

- 实现了符号表管理，处理作用域和生命周期；
- 进行了类型检查和补充了强制类型转换；
- 使用符号表对变量的声明以及使用进行了检查，提供了详细的错误信息；
- 为代码生成阶段补充必要的语义信息；
- 生成了包含语义信息的数据结构给代码生成器使用。

7.2.4 代码生成模块

- 遍历 AST 生成等效的 C 语言代码；
- 处理了 Pascal-S 和 C 语言之间的语法差异，根据 C 语言的惯用写法对代码结构进行了部分调整；
- 实现了变量声明、控制结构、过程调用等的转换规则；
- 生成可读性较好的 C 代码，包含适当的缩进和换行。

7.2.5 用户接口模块

- 实现了图形化用户界面，支持浏览器和命令行两种交互方式；
- 支持输入 Pascal-S 代码，编译并输出 C 代码；
- 支持历史记录功能，能够保存用户的编译记录；
- 支持亮色/暗色模式切换；
- 支持代码高亮显示；
- 支持文件输入输出。

7.3 问题及解决方案

在实验过程中，并不是一路顺风顺水，我们遇到了一些困难，但是在我们的努力下都得到了充分的解决，其中包含：

1. **语法差异**：Pascal-S 中如 for 循环、函数定义都与 C 语言的语法结构不一致。我们通过建立语法树，提取语法中的关键信息，然后再重新进行对应的代码生成来解决；
2. **作用域管理**：Pascal 的嵌套作用域与 C 的扁平作用域不同，但是在 Pascal-S 中嵌套深度不超过 2，这对我们的设计提供了便利。我们将 program 中的变量放在全局作用域中，subprogram 则放在局部作用域中，和 Pascal-S 中的作用域形成对应；
3. **函数返回值**：Pascal-S 中函数的返回值不通过 return 类似的语法来提供，而是通过对函数名的赋值语句提供，这为编译时提供了额外的困难，对于赋值语句在查找符号表时如果查找不到，还需要考虑左值是否是函数名称而不是变量名的情况。这一点在语法分析过程中没有办法得到确定，我们通过在语义分析阶段使用映射变量记录这种语句，从而解决该问题；
4. **函数调用**：Pascal-S 中函数调用并不强制需要一对小括号，当函数没有形参时，Pascal-S 允许仅仅使用该函数的名字来进行调用，这导致在语法分析时无法确定该成分是变量还是函数。我们将这个问题保留到语义分析进行解决，而不给语法分析添加额外的任务，明确了代码的功能，通过巧妙的设计降低了编码实现的难度。

然而我们必须指出，在项目推进过程中我们遇到的另一大类障碍，其根源在于实验指导书本身。我们发现指导书中存在多处表述不清、前后矛盾甚至明显错误的内容。这些问题直接导致了我们在理解需求、设计方案以及调试代码等多个环节上投入了远超预期的额外时间进行反复验证、试错和修正。这不仅严重打乱了我们的开发计划，也极大地消耗了我们的精力，对项目进度造成了实质性的负面影响。我们恳切希望相关教师能够重视这些问题，对实验指导书进行彻底的审查和修订，以确保后续学生能够在一个更清晰、准确的指导下进行学习和实践，避免不必要的困扰和时间浪费。

7.4 实验成果

- 成功实现了 Pascal-S 到 C 语言的基本翻译功能，翻译出的代码结构基本保持一致，有良好的可读性；
- 能够处理包括变量声明、赋值、控制结构、过程定义和调用等高级语言核心特性；
- 生成的 C 代码可在标准 C 编译器（如 gcc）下正确编译运行；
- 通过了多组测试用例的验证，包括简单程序、嵌套控制结构、递归过程以及复杂的综合用例；
- 实现了图形化界面，支持浏览器和命令行、文件 3 种交互方式。

7.5 实验收获

通过本次实验，我们收获了许多能力和经验，每位成员的体会如下：

7.6 改进方向

我们深知该翻译器仍有许多不足，以下是我们目前想到的可改进方向：

1. 增加更多 Pascal 特性的支持（如文件操作、记录类型等）；
2. 优化生成的 C 代码质量和效率；
3. 添加更完善的错误处理和恢复机制；
4. 实现中间代码优化环节。

附录 A 代码文档

我们使用 Doxygen 生成了代码文档，文档包含了所有的类、函数和变量的详细说明。位于 `compiler/doc/html` 文件夹中，可以通过打开 `compiler/doc/html/index.html` 文件查看。如图 A.1所示。

我们还为前端代码生成了代码文档，位于 `compiler_browser/docs` 文件夹中，可以通过打开 `compiler_browser/docs/index.html` 文件查看。如图 A.2所示。



图 A.1: 编译器代码文档

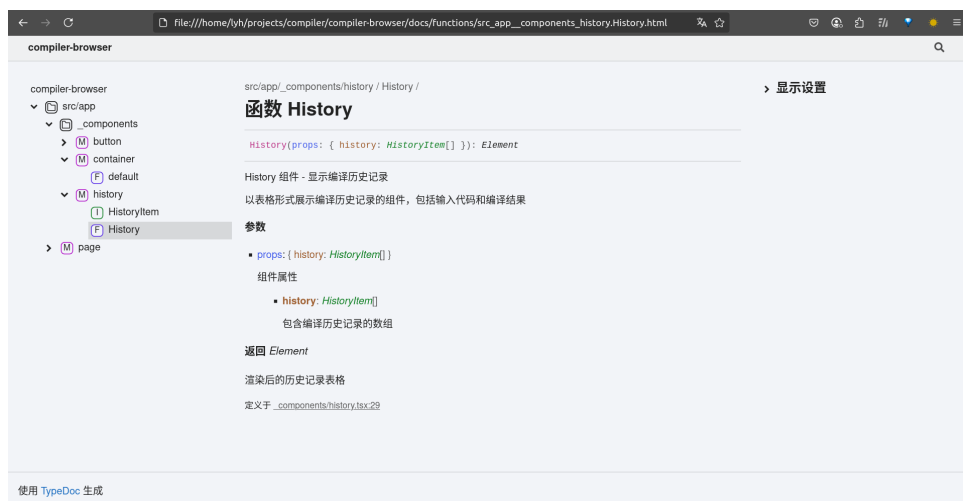


图 A.2: 前端代码文档

附录 B Pascal-S 文法

我们使用的文法，使用 EBNF 范式表示。对指导书做了修正和补充。

```
programstruct ::= program_head ";" program_body "."
program_head ::= "program" id ["(" idlist ")"]
idlist ::= id {"", " id"}
program_body ::= const_declarations var_declarations
               subprogram_declarations compound_statement
const_declarations ::= ["const" const_declaration_list ";"]
const_declaration_list ::= single_const_declaration
                        {";" single_const_declaration}
single_const_declaration ::= id "=" const_value
const_value ::= [sign] num | "'" character "'" | "true" | "false"
              | "'" string_literal "'"
var_declarations ::= ["var" var_declaration_list ";"]
var_declaration_list ::= single_var_declaration {";" single_var_declaration}
single_var_declaration ::= idlist ":" type
type ::= basic_type | "array" "[" period "]" "of" basic_type
basic_type ::= "integer" | "real" | "boolean" | "char"
period ::= dimension_range {"", " dimension_range"}
dimension_range ::= digits ".." digits
subprogram_declarations ::= {subprogram ";"}
subprogram ::= subprogram_head ";" subprogram_body
subprogram_head ::= ("procedure" id formal_parameter)
                  | ("function" id formal_parameter ":" basic_type)
formal_parameter ::= ["(" [parameter_list] ")"]
parameter_list ::= parameter {";" parameter}
```

```

    parameter ::= var_parameter | value_parameter

    var_parameter ::= "var" value_parameter

    value_parameter ::= idlist ":" basic_type

    subprogram_body ::= const_declarations var_declarations compound_statement

    compound_statement ::= "begin" statement_list "end"

    statement_list ::= statement { ";" statement }

    statement ::= eps
                | variable assignop expression
                | func_id assignop expression
                | procedure_call
                | compound_statement
                | "if" expression "then" statement else_part
                | "for" id assignop expression "to" expression "do" statement
                | "while" expression "do" statement
                | "read" "(" variable_list ")"
                | "write" "(" expression_list ")"
                | "break"

    variable_list ::= variable { "," variable }

    variable ::= id id_varpart

    id_varpart ::= "[" expression_list "]"

    func_id ::= id

    procedure_call ::= id "(" [expression_list] ")"

    else_part ::= ["else" statement]

    expression_list ::= expression { "," expression }

    expression ::= simple_expression [relop simple_expression]

    simple_expression ::= term { addop term }

    term ::= factor { mulop factor }

    factor ::= [sign] num
            | variable
            | "(" expression ")"
            | id "(" [expression_list] ")"
            | "not" factor
            | sign factor
            | "true" | "false"

```

Operators:

`assignop` ::= ":"

`relop` ::= "=" | "<>" | "<" | "<=" | ">" | ">="

`addop` ::= "+" | "-" | "or"

`mulop` ::= "*" | "/" | "div" | "mod" | "and"

`sign` ::= "+" | "-"

Lexical Primitives (Conceptual):

`id` ::= `letter` {`letter` | "_"}

`num` ::= `digits` ["." `digits`] [`scale_factor`] (unsigned integer or real)

`digits` ::= `digit` {`digit`}

`scale_factor` ::= ("E" | "e") [`sign`] `digits`

`letter` ::= a..z, A..Z

`digit` ::= 0..9

`character` ::= ? any single printable character (excluding ') ?

`string_literal` ::= `character` {`character`}

`eps` ::= ? an empty string ?