Project 2 - Parser

2025 年 9 月 29 日

1 项目要求

在 Project 1 中，你已经完成了 Splc 的词法符号定义。
你将在 Project 2 中：

1. 基于已给出的语法规则，完成 Splc 的 ANTLR 语法文件（`.g4`）的撰写，并确保所有**语法规则与词法规则的名称**都与本文档中的要求**完全一致**；
2. 特别关注并处理**表达式（expression）**规则中**操作符的优先级与结合性**；
3. 使用 ANTLR 的 **Visitor 模式**，实现对常量表达式（定义会在下面给出）的求值。
4. 处理 ANTLR 在语法分析阶段检测到的语法错误（对于可能出现的语法错误形式我们将在下面给出）。

1.1 扩展要求

从本次 Project 开始，我们将引入扩展部分。

- 扩展部分的内容会比基础部分更具挑战性，扩展部分的内容会在文档中高亮列出。
- **如果你不完成扩展部分，你将无法获得满分**。不完成扩展部分不会影响你完成所有 Project 的基础部分。
- 扩展部分在本课程的项目中也是连续的。后续的 Project 的扩展部分可能会依赖以前 Project 的扩展部分。
- 扩展部分在每个 Project 之间都是独立计分的，在后续的 Project 中移除对扩展部分的支持不影响前序 Project 的分数。也就是说，你可以选择在前面的 Project 中完成较为简单的扩展任务；如果你发现在后续的 Project 中完成扩展部分过于困难，你可以选择不完成后续 Project 的扩展部分，这样不会影响你前面 Project 的分数。

1.1.1 扩展 1: 结构体

你需要额外完成 蓝色标记 的部分。

类型说明符 (`specifier`) 需要支持结构体类型的定义和声明。全局定义 (`globalDef`) 需要支持全局结构体声明。

1.1.2 扩展 2: 指针

该扩展依赖于扩展 1。你需要额外完成 品红色标记 的部分。

一个变量声明 (`varDec`) 原先只需要定义成简单变量声明和数组声明两种形式，现在你还需要支持指针的声明。需要注意的是，指针的声明可以与数组声明嵌套使用，例如 `int **p[10];` 是合法的。

在解决 `*` 和 `[]` 的结合顺序时：`[]` 优先级比 `*` 更高；也就是说，数组优先于指针与 Identifier 结合；`()` 可以用来改变结合顺序。

例如，`int *a[10]` 的解释是 *an array of 10 elements, which are pointers to int*，`int (*a)[10]` 的解释是 *a pointer to a 10-element array of int*，`struct sign *(*var[5])[10]` 的解释是 *a 5-element array of pointers to 10-element arrays of pointers to struct sign*。

参考资料：Declarators and variable declarations (C) - Microsoft Learn, Interpreting More Complex Declarators (C) - Microsoft Learn

为了简化，我们移除了 C 语言标准中的函数指针（即一个 declarator 只能是 varDec）。如果你想挑战更高难度把它加回来，你可能需要修改本项目所要求的语法规则，这会破坏掉我们的自动评测。所以，如果你决定挑战这一部分，请在评测前联系我们，我们将为你提供更多的参考资料并使用手动评测。**我们非常欢迎你挑战这一部分！**

2 语法规则 (Parser Rules)

2.1 全局定义 (globalDef)

函数定义	: specifier Identifier LPAREN funcArgs RPAREN LBRACE statement* RBRACE
全局变量定义	: specifier varDec SEMI
全局结构体声明	: specifier SEMI

2.2 类型说明符 (specifier)

整型	: INT
字符型	: CHAR
结构体	: 'struct' Identifier
完整结构体	: 'struct' Identifier LBRACE (specifier varDec SEMI)* RBRACE

2.3 变量声明 (varDec)

简单变量声明	: Identifier
数组声明	: varDec LBRACK Number RBRACK
指针	: STAR varDec
结合顺序	: LPAREN varDec RPAREN

2.4 函数参数 (funcArgs)

参数列表	: (specifier varDec (COMMA specifier varDec)*)?

2.5 语句 (statement)

代码块	: LBRACE statement* RBRACE
局部变量定义	: specifier varDec (ASSIGN expression)? SEMI
If 语句	: IF LPAREN expression RPAREN statement (ELSE statement)?
While 语句	: WHILE LPAREN expression RPAREN statement
返回语句	: RETURN expression SEMI
表达式语句	: expression SEMI

2.6 表达式 (expression)

我们将定义 expression 的基础形式 (Primary Expression)，以及包含操作符 (Operator) 的递归定义。

操作符分为 Suffix Unary Operators (后缀一元运算符，形如 `expression SuffixUnaryOp`)、Prefix Unary Operators (前缀一元运算符，形如 `PrefixUnaryOp expression`) 和 Binary Operators (二元运算符，形如 `expression BinaryOp expression`)。

对于包含操作符的递归定义，以下给出的产生式并不符合它们应该在 `Splc.g4` 中出现的顺序。参照 C 语言标准 C Operator Precedence - cppreference.com 处理操作符的优先级和结合性。

- expression 的基础形式：

标识符	`: Identifier`
数字常量	`: Number`
字符常量	`: Char`
括号	`: LPAREN expression RPAREN`

- Suffix Unary Operators (后缀一元运算符):

后缀自增	`: expression INC`
后缀自减	`: expression DEC`
函数调用[1]	`: Identifier LPAREN (expression (COMMA expression)*)? RPAREN`
数组访问	`: expression LBRACK expression RBRACK`
结构体访问	`: expression '.' Identifier`
结构体指针访问	`: expression '->' Identifier`

- Prefix Unary Operators (前缀一元运算符)，右结合的:

前缀自增	`: INC expression`
前缀自减	`: DEC expression`
一元正号	`: PLUS expression`
一元负号	`: MINUS expression`
逻辑非	`: NOT expression`
取地址	`: '&' expression`
解引用	`: STAR expression`

- Binary Operators (二元运算符):

减法	`: expression MINUS expression`
逻辑或	`: expression OR expression`
赋值	`: expression ASSIGN expression`
等于	`: expression EQ expression`
乘法	`: expression STAR expression`
小于等于	`: expression LE expression`
逻辑与	`: expression AND expression`
大于	`: expression GT expression`

[1] 你可能会疑惑为什么函数调用出现在 Suffix Unary Operators 中：这里的 Operator 实际上是 `()`，这条规则中开头的 **Identifier** 实际上应该是 **expression**，我们将其简化处理了。

除法　　: expression DIV expression
不等　　: expression NEQ expression
加法　　: expression PLUS expression
取模　　: expression MOD expression
小于　　: expression LT expression
大于等于 : expression GE expression

3 常量表达式求值

3.1 常量表达式的定义

constexpr (**常量表达式**) 是 **expression** 中特殊的一类，在本项目中被严格定义为：

- **整数常量**是 **constexpr**；
- 对于 **括号**，若其中的 **expression** 是 **constexpr**，则该括号所构成的 **expression** 也是 **constexpr**；
- 对于 **一元正号、一元负号**，如果其操作数是 **constexpr**，则该 **expression** 也是 **constexpr**；
- 对于 **二元运算**，如果运算符是 **加、减、乘、除、取模**之一，且其两个操作数都是 **constexpr**，则该 **expression** 也是 **constexpr**；
- 其他所有形式的 **expression** 都不是 **constexpr**。

注意：在本项目中，**变量、函数调用、数组访问**都不被视为常量表达式，即使它们的计算值在运行时是常量。你可以认为此处的 **constexpr** 定义是一个简化版本，目的是让你专注于处理基本的算术表达式。

- **是常量表达式的例子**：
  - `1 + 2 * 3`
  - `-100 / (2 % 3)`
  - `42`
- **不是常量表达式的例子**：
  - `j + 2 * 3` (因为包含了变量 `j`)
  - `myFunc(1, 2)` (因为包含了函数调用)
  - `a[0] + 3` (因为包含了数组访问)

3.2 常量表达式的求值

你的任务是找出在 **局部变量定义**中出现的 **expression**，判断它是否是一个 **constexpr**，如果是，则计算其数值。

局部变量定义: `varDec (ASSIGN expression)? SEMI`
特别注意以下两点：

- **必须是完整的表达式**：你的目标是处理赋值号（`ASSIGN`）右侧的整个表达式，不能只计算其中的一部分。
  - 对于 `int i = 1 + 2 * 3;`，你需要处理的是完整的 `1 + 2 * 3`。
  - 对于 `int j = (1 + 2) * k;`，由于表达式中包含变量 `k`，因此不是我们要处理的常量表达式，你**也不用处理**其中的子表达式 `1 + 2`。
- **识别所有符合条件的语句**：你的程序需要能够处理源文件中所有符合条件的变量定义与赋值语句，并逐一打印出计算结果。

4 语法错误处理

在编译器理论中，实现完备且高效的语法错误处理与恢复机制是一项极富挑战性的任务，目前业界尚未形成统一的最佳实践。为了降低本项目的复杂度，我们将聚焦于一个定义明确的子问题。

4.1 项目约束与简化假设

为保证任务的可行性，我们设定以下两条核心约束：

- **单一错误原则**：每个独立的测试用例保证最多只包含一处语法错误。
- **特定错误类型**：项目中需要处理的语法错误仅限定为一种，即“缺失词法符号 (Missing Symbol)”。

4.2 错误类型详解：缺失词法符号

“缺失词法符号”错误的具体特征如下：

- **错误定义**：在程序代码的某个位置，缺失了单个本应存在的词法符号 (Token)。
- **发生位置**：缺失可能发生在代码的任意位置。
- **缺失对象**：任意类型的词法符号都可能成为缺失的对象。

4.3 任务要求

你需要实现一个能够准确诊断上述错误的程序。具体来说，程序需要完成以下两个目标：

- **识别符号名称**：识别出缺失词法符号的具体名称（例如 `SEMICOLON`, `RPAREN` 等）。
- **定位出现行号**：定位到该词法符号应当出现的行号（注意：行数从 0 开始计数）。
  - **行号确定规则**：为避免歧义，缺失符号的行号被定义为其前一个有效词法符号（除去那些 skip 的词法符号）所在的行号。
  - **示例说明**：在下面的代码片段中，结尾处缺失了一个右大括号 `}`。按照规则，其前一个有效符号是分号 `;`，因此，缺失的 `}` 的行号应被判定为 **分号**所在的行号。

```
int main() {
    int x = 10;
```

4.4 指定输出格式

当检测到错误时，程序必须严格按照以下格式打印错误信息，不能有任何多余的字符。

```
Missing Symbol '< 词法符号的名称 >' at Line < 该词法符号应该出现在的行数 >
```

4.4.1 格式示例

例如，如果检测到第 15 行（0-indexed）缺失了一个分号（假设其词法符号名称为 `SEMICOLON`），则程序应输出：

```
Missing Symbol 'SEMICOLON' at Line 15
```

4.4.2 错误处理提示

在处理语法错误时，你可以考虑以下两种策略：

- **利用默认机制**：对于多数情况（如缺失符号），ANTLR 的默认错误处理机制已足够强大。你的任务仅是自定义一个 `ErrorListener`，从它提供的错误信息中提取关键内容（如期望的符号、行号），并按指定格式打印出来。
- **定义错误规则**：在少数情况下，ANTLR 的默认处理可能会将简单错误复杂化，产生一连串的连锁错误。此时，你可以在 `.g4` 语法文件中为已知的、特定的错误模式专门定义一条语法规则，然后利用**语法动作 (Action)** `{...}` 在匹配到该错误模式时，直接执行 Java 代码来打印你想要的、更精确的错误信息。

5 初始代码

初始代码位于 `https://github.com/sqlab-sustech/CS323-Compilers-2025F-Projects` 的 `project2-base` 分支。请参照 `Project Zero`，在你 project1 代码基础上合并来自 `project2-base` 分支的代码。

你的核心任务是：

- **编写 `Splc.g4` 文件**：这是本次项目的最主要任务。你需要根据项目文档的要求，正确实现所有语法规则（将你 project 1 当中的词法规则也复制进来），妥善处理表达式的优先级和结合性（按 C 语言的标准）。
- **完善 `ConstExprVisitor.java`**：实现 `constexpr` 的识别与求值，对非 `constexpr` 返回 `null`。
- **完善 `Project2ErrorListener.java`**：实现指定格式的语法错误报告。

你可以随意修改 `Main.java` 文件，但是不要修改 `framework` 包下的文件。

6 评分

本次 Project 满分 100 分，其中基础部分 90 分，扩展 1 结构体 6 分，扩展 2 指针 4 分。

我们将运行 `framework.project2.Grader`，并比对你的程序输出与我们的标准输出，完全一致则视为通过测试。`framework` 将负责打印输出，你不需要关心输出格式上的细节。

6.1 评分细则

对于基础部分，系统共提供 45 个测试样例，每个样例 2 分，总分 90 分。其中包括 40 个语法正确的测试样例和 5 个语法错误的测试样例：

- **语法正确的样例 (40 个)**：

  对于每一个正确的样例，你的程序必须**同时满足**以下两个条件才能获得 2 分，否则该样例为 0 分：

  - 程序必须成功解析并通过测试，不报告任何语法错误。
  - 程序打印的 **语法树**以及 **常量表达式求值结果**必须与标准答案完全一致。任何一项不一致，该样例都不得分。

- **语法错误的样例 (5 个)**：

  对于每一个错误的样例，你的程序必须准确捕获错误，并打印出唯一的错误信息，才能得分。

对于扩展部分，系统将分别提供 6 个和 4 个测试样例，每个样例 1 分，样例保证语法正确，标准同上述正确样例。

在初始代码中，我们提供了一些将被用于评分的测试用例。但是，它们并不涵盖所有可能的情况。你需要自行设计更多测试用例来验证你的程序。

6.2 提交

请参照 Project Zero，使用 `make handin` 提交。请注意，**提交脚本只会提交被 git 追踪的文件**。