

Assignment 1: Make a Parser: From Source Code to AST

Make a Parser: From Source Code to AST

- 词法分析和语法分析
- 为什么要生成抽象语法树
- 关于本次实验的代码实现
 - 介绍 AST 的定义
 - Lex 和 Yacc 的分工

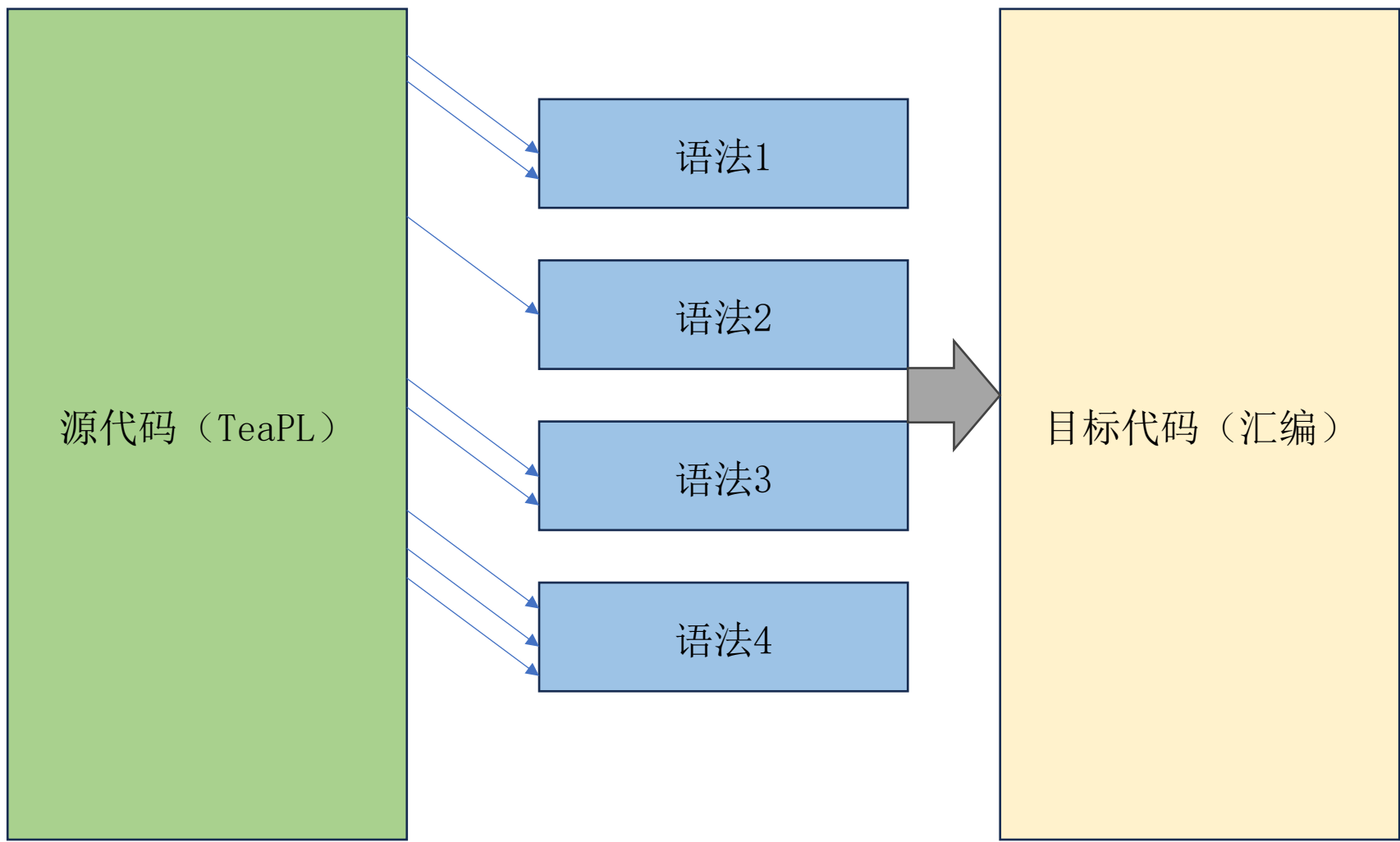
词法分析和语法分析

为了将一个程序从一种语言翻译成另一种语言，编译器必须首先把程序的各种成分拆开，并搞清其结构和含义，然后再用另一种方式把这些成分组合起来。编译器的前端执行分析，后端进行合成。

分析一般分为以下 3 种：

- **词法分析**：将输入分解成一个个独立的词法符号，即“单词符号”(token)，简称单词¹。
- **语法分析**：分析程序的短语结构。
- **语义分析**：推算程序的含义。

词法分析器以字符流作为输入，生成一系列的名字、关键字和标点符号，同时抛弃单词之间的空白符和注释。程序中每一点都有可能出现空白符和注释；如果让语法分析器来处理它们就会使得语法分析过于复杂，这便是将词法分析从语法分析中分离出去的主要原因。



为什么要生成抽象语法树

编写一个完全用 Yacc 中的语义动作来实现的编译器是有可能的，但这种编译器很难阅读和维护，并且这种方法限制了编译器只能完全按语法分析的顺序来处理程序。

为了有利于模块化，最好将语法分析和语义分析 (类型检查和翻译成机器代码) 分开处理。达到此目的一种方法是由语法分析器生成抽象语法树即一种数据结构，编译器在较后阶段可对其进行遍历。

抽象语法 (abstract syntax) 起到了在语法分析器和编译器(或其他序分析工具，如依赖关系分析器)的较后阶段之间建立一个清晰接口的作用。抽象语法树传递源程序的短语结构，其中已解决了所有语法分析问题，但不带有任何语义解释。

本次实验

- 进行词法分析和语法分析
- 输入 TeaPL 的源代码
- 输出生成的 AST（通过提供的打印函数输出）

AST 的定义: TeaplAst.h

- 基本上和语法定义一一对应
- 不对应的地方有注释说明

Lex 和 Yacc 的分工

Lex 进行词法分析

- Lex 负责识别终结符
- Lex 负责处理注释
- Lex 中维护位置信息（便于后续进行类型检查）

Yacc 进行语法分析，并执行语义动作生成AST

- Yacc 根据语法规则生成 AST 中的节点（进行语法分析）

Test

- 执行 `make` 以运行 `tests` 目录下的测试用例