

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

Make a Parser: From Source Code to AST

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

为什么要生成抽象语法树

编写一个完全用 Yacc 语法分析器的语义动作短语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，最好将语法问题(语法分析)与语义问题(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语法分(parse tree)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

- 输入 TeaPL 的源代码
- 输出生成的 AST（通过提供的打印函数输出）

Lex 和 Yacc 的分工

- 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 生成 AST 中的节点（进行语法分析）
- Yacc 解决语法中的冲突（规定优先级和结合性）
- 如何获取 Yacc 生成的 AST（用全局变量）

```
Program: ProgramElementList
{
    root = A_Program($1);
    $$ = A_Program($1);
}
```

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

Test

- 执行 `make` 以运行 `tests` 目录下的测试用例