1.1语义分析

1.1.1. CST 与 AST

具体语法树（Concrete Syntax Tree）又称为分析树（Parse Tree），是源代码的抽象语法结构的树状表示，树中的每个结点表示源码中的一种结构，给出了具体的文法如何将属性字流组合起来构成相应的语法成份。抽象语法树（Abstract syntax Tree）又称为语法树（Syntax Tree），给出了语义上非常重要，且会对编译器的输出产生实际影响的语法部分，并不会表示出真实语法的每个细节，例如，每个语句后面的分号，包围语句块的大括号，表达式中的圆括号等，取而代之的是层次化的树状结构。

在使用抽象语法树表示表达式时，通常内部结点代表操作符，对应的子结点表示操作数。实际上，每个语句都可以将语句构造映射为操作符，有意义的语义成份映射为操作数，因此采用与表达式类似的树状表示方法。在语法树中，内部节点代表程序构造，而在分析树中，内部结点代表非终结符。在程序设计语言的文法中，许多非终结符表示程序的构造，而其他部分是辅助的部分，例如采用递归文法表示多个函数的 funcList。在语法树中，这些辅助成份不再需要，因此在从分析树到抽象语法树的转换过程中可以直接去掉，也可以在语法分析的过程中跳过辅助成份，直接构造语法树。

例如对下面忽略函数内部语句的文法:

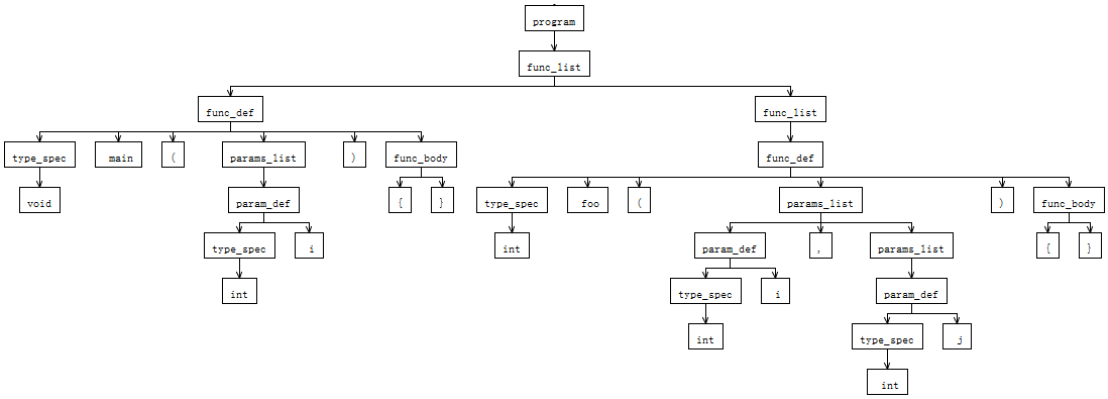
```
program       : func_list ;
func_list     : func_def func_list | func_def  ;
func_def      : type_spec ID '(' params_list ')' func_body;
type_spec     : 'int' | 'void'    ;
params_list   : param_def  | param_def ',' params_list ;
param_def     : type_spec ID ;
func_body     : '{' '}' ;
ID            : LETTER (LETTER|'0'..'9')*;
fragment LETTER : 'A'..'Z' | 'a'..'z' | '_';
```

注意: 上述文法是按照 ANTLRWorks 的规则编写的，其中非终结符用小写字母，token 用大写字母，在文法中直接出现的终结符号需要用引号括起来。上述

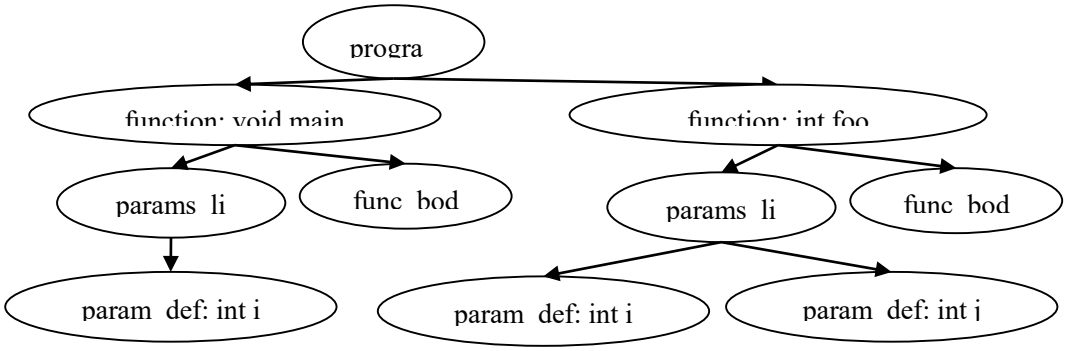
文法可以直接输入 ANTLRWorks 进行验证。当输入为如下的程序时:

```
void main(int i){
}
int foo(int i, int j){
}
```

会得到如下的分析树:



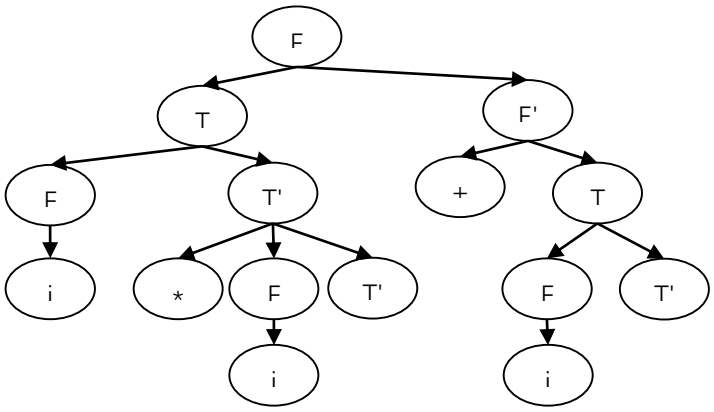
将其转换对应的抽象语法树之后的结果为:



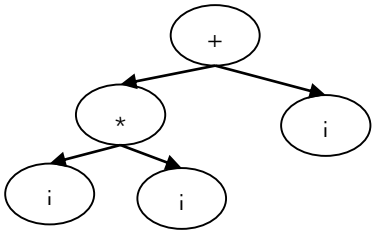
又如下面的文法:

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid -\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid /\ F\ T' \mid \varepsilon \\
F &\rightarrow i \mid (\ E\ )
\end{aligned}
$$

当输入为$=i*i+i 时，得到的分析树如下所示:



对应的抽象语法树表示为:

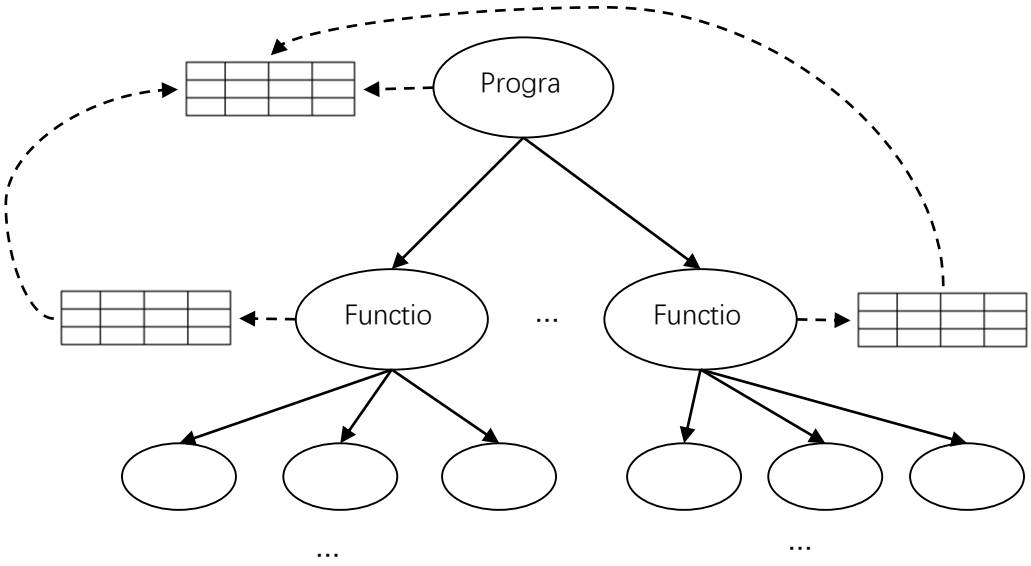


对比发现，抽象语法树更加简洁，方便后端进行处理。

1.1.2. 符号与符号表

在语义分析和代码生成过程中，需要频繁的查询各种符号信息，例如 C 语言中的全局变量、函数、函数局部变量和跳转标号等。由于 C 语言不允许函数嵌套，因此以 C99 为标准，当只允许在函数开始位置声明变量时，符号表的管理则比较简单。但是由于每类符号需要收集的信息各不相同，因此，可能需要设计多种不同的表格。为了方便在不同的范围进行符号查找，可以采用层次化的符号表。

符号表中的符号信息是在编译分析阶段逐步收集的，并在综合阶段生成代码时使用。由于语法分析阶段能够获得程序的语法结构，因此基于语法树进行符号表的创建更为合适一些。针对 C 语言的特点，当不考虑函数内部的代码块时，我们给出如下的参考设计，在实验中大家可以给出自己的设计。



在这个方案中，使用同一个符号表存储各类不同的符号，每个函数对应一个符号表，用以存储函数范围内可见的符号，整个程序有一个全局的符号表，用以存储全局范围内可见的符号信息，从函数符号表到全局符号表有一个引用。查找符号时，首先从函数内部开始查找，如果找不到，则需要去全局符号表中查询。该方案进行多层次扩展，也可以适用于允许函数嵌套定义的程序设计语言。

符号表里面每个符号的信息包括名称、类型、存储位置，以及其他的一些信息。符号表中的信息通常在语法分析阶段建立，在语义分析和代码生成阶段频繁查找和使用。

1.1.3. Scope 的概念

在计算机出现的早期，机器内存非常小，程序的大小以及程序处理的数据也非常小，所以所有的数据都保存在内存里，程序的任意部分都可以访问所有的数据。但是随着计算机以及运行在上面的程序变得越来越复杂，经常发现一部分程序需要使用的数据会被其他不相关部分莫名地修改。另外，也没有很好的办法判断哪一部分程序对数据的访问是正确的，哪一部分程序对数据的访问是错误的。为此，大家改进了程序设计语言，达到相应的代码管理控制自己相应的数据，不同部分互不影响。

Scope 是指程序中某一区域变量和方法对程序中其他区域的可见性。Scope 的含义和重要性对不同的语言是不一样的。Scope 的范围可能小到一个表达式，

大到整个程序。最简单的就是只有一个全局的 Scope，最基本的模块化 scope 是一个两层的 scope，即整个程序范围内的全局 Scope 和函数内部的局部 Scope。更复杂的模块化编程则允许多个层次的 Scope，内层 Scope 里面的符号对外层是不可见的。以 C 语言为例，函数内部是可以包括 block scope，另外还有文件范围内的；函数式编程语言则能将 scope 限制在单个表达式内。

1.1.4. 实验目的

（1）熟悉 C 语言的语义规则，了解编译器语义分析的主要功能；

（2）掌握语义分析模块构造的相关技术和方法，设计并实现具有一定分析功能的 C 语言语义分析模块；

（3）掌握编译器从前端到后端各个模块的工作原理，语义分析模块与其他模块之间的交互过程。

1.1.5. 实验内容

语义分析阶段的工作为基于语法分析获得的分析树构建符号表，并进行语义检查。如果存在非法的结果，请将结果报告给用户，其中语义检查的内容主要包括：

- 变量使用前是否进行了定义；
- 变量是否存在重复定义；
- break 语句是否在循环语句中使用；
- 函数调用的参数个数和类型是否匹配；
- 函数使用前是否进行了定义或者声明；
- 运算符两边的操作数的类型是否相容；
- 数组访问是否越界；
- goto 的目标是否存在；
- …

本次语义检查的前（1）-（3）为要求完成内容，而其余为可选内容。

1.1.6. 实验过程与方法

在语法分析实验的基础上，构建符号表，并基于符号表和抽象语法树进行语义检查。要求对于给定的程序，输出检测到的语义错误信息。要求输出信息中包含错误编号，且错误编码与下表一致，错误信息可以自行定义。

序号	错误原因	错误编号	示例
1	变量使用前是否进行了定义	ES01	0_var_not_defined.c
2	变量是否存在重复定义	ES02	1_var_defined_again.c
3	break 语句是否在循环语句中使用	ES03	2_break_not_in_loop.c
4	函数调用的参数个数和类型是否匹配	ES04	3_func_arg_not_match.c
5	运算符两边的操作数的类型是否相容	ES05	4_opnd_not_match.c
6	数组访问是否越界	ES06	5_arrayaccess_out_of_bound.c
7	goto 的目标是否存在	ES07	6_goto_target_not_exist.c
8	函数是否以 return 结束	ES08	7_func_lack_of_return.c

1.1.7. 实验提交内容

本实验要求提交语义分析器实现源码（项目 src 下所有文件），以及对应的 config.xml 和 classpath 两个文件；C/C++需提供对应的可执行程序（不需要编译的中间文件），每个人提交一份实验报告。

实验报告放置在 doc 目录下，应包括如下内容:

- 实验目的和内容
- 实现的具体过程和步骤
- 运行效果截图

- 实验心得体会