**《汇编与编译原理》课程实验**

**简易编译器的实现说明文档**

1. **小组成员**

组员 1：朴灿彬 - 2018080116

组员 2：岳坤 - 2020010878

组员 3：王展鹏 - 2020010876

1. **开发环境**

开发平台：**Windows** 开发语言：**C**

词法分析工具：**flex** 语法分析工具：**bison**

语言转换： **C++ ---> python**

支持的源程序：回文检测(**src/palindrome.cpp**)、排序(**src/rank.cpp**)

1. **实现原理**
2. **词法分析**

我们使用的是已有的 flex/bison 工具，WIN\_FLEX\_BISON。

（Windows 适配版，链接：https://sourceforge.net/projects/winflexbison/）

如课程上介绍，我们编写了包括正则定义、匹配规则在内的 flex 输入文件（.l 格式）。其中，正则定义部分包括了常见的 C++语言类型符（int/char/指针…）、保留字（include/return…）、操作符（算数操作符与逻辑操作符）、分隔符（;/{}…）、标识符与立即数等。

经 flex 工具、C 编译器处理后生成的可执行文件（即词法分析器）将同目录下的”input.cpp”文件作为输入的源文件。生成token流。

下面是部分token的正则定义：

SINGLESPACE             [" "]

DELIM                   [\r\t\n]

WHITESPACE              ({SINGLESPACE}|{DELIM})+

LETTER                  [A-Za-z]

DIGIT                   [0-9]

VTYPE\_INT               int

VTYPE\_CHAR              char

VTYPE\_STRING            string

VTYPE\_INTP              {VTYPE\_INT}{WHITESPACE}\*\\*{WHITESPACE}\*

1. **语法分析**

如课程上介绍，我们编写了如下文件：

1) 包括正则定义、匹配规则在内的 flex 输入文件（.l 格式，文件名为”mylp.l”）。每当语法分析函数 yyparse()调用 yylex()时匹配当前 token 并从”tree\_node.h”中调用 create\_leaf()函数生成语法分析树中的叶子结点，作为当前 token 返回给 yyparse()。

2) 包括语法分析功能、报错功能、main 函数（调用 yyparse()函数）定义在内的 bison 输入文件(.y 格式，文件名为”mylp.y”)，其中的%token（yylval 变量）的类型均为自定义的 tree\_node 结构体指针，每当匹配到正确的转换规则时，从”tree\_node.h”中调用create\_innernode()/create\_emptynode()函数以生成内部/空结点(表示从ε规约)并规约给原非终结符。

3) 包括语法分析树结点结构体声明 & 定义、相关构造函数及打印函数声明 & 定义、输出文件指针声明在内的 c 语言文件(文件名为”tree\_node.h/c”)。输出的语法分析树格式采用 json 类型，并且在输出文件中实现了行数对齐的美化功能，方便查看语法分析树的结构。

有三种结点类型如下：

- 内部结点（根节点也属于该类型）

{

    “type”: “xxx”, <--均为小写，代表当前非终结符名称“content”: “”, <--内部结点无内容

    “childs”: [...] <--子结点列表

}

- 空结点（表示空推导）

{

    “type”: “xxx”,  <--均为小写，代表当前非终结符名称“content”: “”,  <--空结点无内容、无子结点

}

- 叶子结点（表示任何词法分析器返回的 token，即终结符）

{

    “type”: “XXX”,  <--均为大写，代表当前终结符名称

    “content”: “xxx”, <--表示当前终结符的代表的内容(yytext)，可能为空

    <--叶子结点无子节点

}

1. **语义处理和代码生成**

为了实现由语法分析树到python代码的功能，我们自己编写了generate\_code()函数，用于逐个分析抽象语法树中出现的非终结符的产生形式，并且生成目标代码。

generate\_code()函数按照语法树的节点类型划分，对每一个非终结符的每一个产生式都进行分别的分析和处理，例如generate\_include\_expr()函数负责分析c++代码中的头文件包含部分。这些函数一共有21个，包含了大部分c++程序代码中的常见语句。

generate\_code()函数采用递归调用的方式遍历语法树，节省了代码量。每次函数接收两个参数：tree\_node结点类型和int类型，分别表示将要分析的目标结点和代码块层级（用于实现python代码的缩进语法）。

代码生成函数处理抽象语法树时，分别分析对应产生式中的非终结符和终结符，调用结点属性，根据c++代码和python代码的特性将语句重组、补充完整。例如以下代码是将for循环逻辑转化为适合python语法的代码：

void generate\_for\_conditions(struct tree\_node \*nd, int ts)

{

  fprintf(outfp, "%s in range(", nd->childs[6]->content);

  generate\_code(nd->childs[2], ts);

  fprintf(outfp, ", ");

  generate\_code(nd->childs[4]->childs[2], ts);

  fprintf(outfp, ", ");

  if (strcmp(nd->childs[7]->childs[0]->type, "OP\_PLUSPLUS") == 0)

    fprintf(outfp, "1)");

  else if (strcmp(nd->childs[7]->childs[0]->type, "OP\_MINUSMINUS") == 0)

    fprintf(outfp, "-1)");

}

1. **实现功能**
2. **include指令：**编译器可以识别c++头文件包含语句，并且将常用的标准库头文件转换为对应的python标准库模块（例如<cmath>头文件转换为math模块），便于编译使用相对应的函数。
3. **函数调用语句：**编译器可以识别出函数调用语句。
4. **变量的定义：**编译器识别c++中变量的声明和定义语句，并转换为对应的python代码。
5. **逻辑和运算表达式：**编译器支持布尔逻辑表达式和运算表达式生成。
6. **标准输入输出流语句：**编译器支持cin/cout语句的实现（通过python内置函数）。
7. **一维动态数组：**编译器可以识别一维整型动态数组的声明定义。
8. **for循环：**编译器支持for循环的编译运行。
9. **if-else分支：**编译器可以实现if-else语句。
10. **代码缩进：**编译器可以根据c++代码的结构生成语法正确的python代码块缩进格式。
11. **难点和创新点**
12. **Include语句转化：**在生成头文件时，我们找到了一些具有相似作用的c++库文件和python库模块：<unistd.h>和os模块、<cstdlib>和sys模块、<cmath>和math模块、<ctime>和time模块、<random>和random模块，这些常用的支持文件可以产生对应关系，便于实现常用函数的转化和调用。
13. **Python代码块缩进实现：**python依靠代码块缩进解释代码层级逻辑，我们为了实现缩进逻辑，在generate\_code()函数的参数中传入一个整数，用于记录每次递归调用时代码块的层级，并且在每个需要换行的语句前打印制表符，这样就实现了缩进逻辑。
14. **小组分工**

朴灿彬：词法分析和语法分析代码编写，测试样例编写，测试脚本编写

王展鹏：语义处理和代码生成的部分代码编写，展示ppt制作，小组展示

岳坤：代码生成的部分函数编写，实验文档编写