目錄

Compiler Fall 2018

There are some materials for the course taught by Yu Zhang from University of Science and Technology of China.

The TAs are Yu-xiang Zhang and Ze-wen Jin. The projects were designed by Huan-qi Cao and Yu Zhang.

Course homepage

编译原理 H 实验环境配置

本课程推荐的实验环境有两种：

- Linux
- Mac with Homebrew installed

Linux 和 Mac 都是不错的 POSIX 环境，也都各自有好用的包管理器、便利的图形化环境。

1. 基本的编译执行环境

为了编译实验代码，你需要安装以下软件：

- CMake 3.5 或更高版本
- 任何支持 C++ 14 的编译器，如 g++，clang
- 任意 Java VM，推荐使用 OpenJDK 8 或 Oracle JRE 1.8/8
- pkg-config
- uuid-dev

1.1. 依赖软件的安装

在 Linux 的发行版上安装这些软件会十分简单，例如, 在 Ubuntu 下，运行如下命令即可：

```
sudo apt install cmake g++ openjdk-8-jdk pkg-config uuid-dev
```

在 Mac 下，你需要使用 clang 作为编译器（随 XCode 安装），单独安装 JVM，然后从 Homebrew 安装其余包：

```
brew install cmake pkg-config ossp-uuid antlr4-cpp-runtime
```

1.2. CMake 的使用

CMake 根据内置的规则和语法来自动生成相关的Makefile 文件进行编译，同时还支持静态库和动态库的构建等。通过 CMake 管理的项目主目录下会有一个 CMakeLists.txt ，其中会包括工程包含哪些子目录等内容；而在每个子目录下，也会包含一个 CMakeLists.txt，用来管理该子目录中相关内容的构建。

编译

编译的时候，可以建立单独的文件夹（如 `build` ），让编译产生的文件和源代码区分出来，以下是一种常用的编译方式, 当前目录为项目源代码目录：

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

编写CMakeLists.txt

在 CMakeLists.txt 中，可以包含如下一些内容：

```
cmake_minimum_required(VERSION 3.1)  # CMake 版本要求
PROJECT(hello)                       # 项目名称

set(CMAKE_CXX_STANDARD 14)           # 设置编译器遵循的C++标准

set(CMAKE_CXX_FLAGS “-O2 -g”)        # 自定义编译选项
```

假设 `args` 代表通过空格或换行分隔的多个参数，你可以

- 通过 `include_directories(args)` 来添加外部库的头文件所在的一组目录 `args` 作为头文件查找目录；
- 通过 `link_directories(args)` 来添加外部链接库的查找目录;
- 通过 `link_libraries(args)` 来添加链接的外部库；
- 通过 `target_link_libraries(hello args)` 来设置生成的可执行目标及其要链接的库
- 通过 `add_library(mylib STATIC ${SRC})` 来创建静态库 `libmylib.a`；如果 `STATIC` 换成 `SHARED` ,则为创建动态库 `libmylib.so`；
- 通过 `add_executable(hello ${SRC})` 来创建可执行文件 `hello`

你可以使用内置宏 `CMAKE_CURRENT_SOURCE_DIR` 来获取当前CMakeLists.txt 所在目录，例如 `${CMAKE_CURRENT_SOURCE_DIR}/..` 表示CMakeLists.txt 所在目录的上一级目录。

2. ANTLR

ANTLR (ANother Tool for Language Recognition)是一个强大的解析器的生成器。在课程实验中，你将利用 ANTLR 工具读入描述要解析的语言词法和语法的文法文件（扩展名是 `.g4` ），生成解析器的源码。ANTLR v4 使用 Java 实现，但是生成的解析器源码可以是 Java、C++、C#、Go、JavaScript、Swift等。

2.1. 依赖环境准备

ANTLR 工具需要 JVM 才能执行；另一方面，为了方便使用 grun，你需要一个能够编译 java 源文件的环境。因此，你需要一个完整的 Java Development Kit。

- 如果你使用 Linux：

  推荐通过包管理器安装 OpenJDK 8。在你的包管理器中通过搜索确定包名，如 Ubuntu 下包名为 `openjdk-8-jdk`，安装之即可。

- 如果你使用 Mac：

  你需要手动安装一个 JDK。

2.2. 获取 ANTLR

你需要从 ANTLR Download 下载 `antlr-4.7.1-complete.jar`。

然后，你需要将该jar包的路径加入到环境变量 `CLASSPATH` 中，即 Bash 中执行

```
export CLASSPATH=".:/path/to/your/antlr-4.7.1-complete.jar:$CLASSPATH"
```

你可以考虑将这一命令加入 `.bashrc`（对于Bash），以省去你每次配置的麻烦。

2.3. antlr4 和 grun 工具

可以定义别名 `antlr4` 表示 ANTLR 工具，即

```
alias antlr4='java org.antlr.v4.Tool'
```

这样，你可以直接使用 `antlr4 your.g4` 来为your.g4 生成解析器源码。

ANTLR 的运行时库中还提供了一个灵活的测试工具 `TestRig`，它可以显示解析器如何匹配输入的许多相关信息。`TestRig` 使用Java的反射机制来调用编译过的解析器。为了方便用户使用，ANTLR 提供了一个 `grun` 工具来使用 `TestRig`。

`grun` 本质上是一个别名，可以定义如下：

```
alias grun='java org.antlr.v4.runtime.misc.TestRig'
```

或

```
alias grun='java org.antlr.v4.gui.TestRig'
```

同样的，你可以将这些别名命令加入到 `.bashrc`，以节省你配置和使用的时间。

3. LLVM 配置

由于 Ubuntu 源中 LLVM 的打包错误、以及此后对 Clang 和 LLVM 的实验需要，这里推荐同学们自行编译 LLVM。

3.1. 获取 LLVM

这里将在当前目录下建立一个名为 `llvm` 的目录，其中包含了 LLVM 和 Clang。

```
wget http://releases.llvm.org/6.0.1/llvm-6.0.1.src.tar.xz
wget http://releases.llvm.org/6.0.1/cfe-6.0.1.src.tar.xz
tar xvf llvm-6.0.1.src.tar.xz
mv llvm-6.0.1-src llvm
tar xvf cfe-6.0.1.src.tar.xz
mv cfe-6.0.1.src llvm/tools/clang
```

3.2. 编译并安装

这里将在当前目录下建立一个名为 `llvm-build` 的目录（和 `llvm` 同级），用于构建 LLVM 和 Clang，并将安装目录配置为和这两个目录同级的 `llvm-install` 目录。

```
mkdir llvm-build && cd llvm-build
cmake ../llvm -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=`cd .. && pwd`/llvm-in
stall
make -j install
```

推荐考虑使用 `make -jn`，其中将 `n` 替换为你的 CPU 逻辑处理器数目减一。例如，在 i5-7287U 上，使用 `make -j3`，编译时间可能不到一个小时。而如果使用单线程，那么编译时间可能超过两个小时。这段时间里你可能需要玩一会儿手机或者去冲一杯咖啡，当然复习一下编译的知识、看一下 LALR 是个不错的选择。

请注意，为了避免你的内存爆炸，这里使用了 **Release** 配置。这一配置下，链接时无需处理调试符号，内存占用大幅度降低。考虑到同学们不会有 Debug 进入 LLVM 中的需求，Release 配置已经足够使用。

但是，我们强烈建议你使用 **Debug** 配置进行编译。一方面是让你体验一下比较大的工程的编译过程，另一方面在这个过程里面会有一点点奇怪又好玩的问题出现，强烈安利！前两年课程的llvm编译，用的不是新版的llvm 6，所以编译的时候大家都遇到了一些很多好玩的问题，得到了刺激的体验！内存占用峰值也超过了12GB。所以真的强烈安利！内存不大的同学也可以尝试一下，看看会有什么有趣的问题，理解一下编译过程、链接过程的各种优化的必要性！

基础实验：构建C1语言的编译器

基础实验包含三阶段。

- Lab1: 基于 ANTLR v4 的 C1 词法与语法分析器
- Lab2: 语义分析器
- Lab3: 面向 LLVM IR 的中间代码生成
- 基于 LLVM OrcJIT 的简单即时编译实现

实验 1 词法与语法分析器

在本实验中，你将通过编写文法描述文件，使用 ANTLR v4 生成词法和语法分析器，并编写一个在 parse tree 上遍历生成 abstract syntax tree (AST) 的 visitor 来实现一个 C1 语言分析器。

C1 语言说明

C1 是本课程实验要实现的一个 C 语言子集。它没有完善的类型系统，只有**int**、**float**和元素为 **int** 或 **float** 类型的一维数组，并附带 const 描述符；它的函数定义没有参数和返回值。C1 语言的文法采用 EBNF (Extended Backus-Naur Form) 表示如下：

```
CompUnit    → [ CompUnit ] ( Decl | FuncDef )
Decl        → ConstDecl
            | VarDecl
ConstDecl    → 'const' BType ConstDef { ',' ConstDef } ';'
BType        → 'int' | 'float'
ConstDef    → Ident '=' Exp
            | Ident '[' [ Exp ] ']' '=' '{' Exp { ',' Exp } '}'
VarDecl        → BType VarDef { ',' VarDef } ';'
VarDef        → Ident
            | Ident '[' Exp ']'
            | Ident '=' Exp
            | Ident '[' [ Exp ] ']' '=' '{' Exp { ',' Exp } '}'
FuncDef        → void Ident '(' ')' Block
Block        →'{' { BlockItem } '}'
BlockItem    → Decl
            | Stmt
Stmt        → LVal '=' Exp ';'
            | Ident '(' ')' ';'
            | Block
            | 'if' '( Cond ')' Stmt [ 'else' Stmt ]
            | 'while' '(' Cond ')' Stmt
            | ';'
LVal        → Ident
            | Ident '[' Exp ']'
Cond        → Exp RelOp Exp
RelOp        → '==' | '!=' | '<' | '>' | '<=' | '>='
Exp            → Exp BinOp Exp
            | UnaryOp Exp
            | '(' Exp ')'
            | LVal
            | Number
Number      → FloatConst
            | IntConst
BinOp        → '+' | '−' | '*' | '/' | '%'
UnaryOp        → '+' | '−'
```

EBNF 中的符号含义

> - [...]: 内包含的为可选项
> - {...}: 内包含的为可重复0次至无数次的项

文法之外，需要补充说明的内容有：

- C1 语言中运算符的优先级及结合性与 C 语言一致
- C1 语言中标识符 `Ident` 规范为 C 语言标识符规范子集，参考 ISO/IEC 9899 第51页关于标识符的定义。规范中，将非终结符 `identifier` 产生式修改为 `identifier-nondigit: nondigit` ，不考虑universal-character-name、other implementation-defined characters 等
- C1 语言中注释规范与 C 语言一致，参考 ISO/IEC 9899 第66页关于注释的定义, 其中

```
// \
this is a two-line comment.

/\
/ this is also a two-line comment actually.
```

- C1 语言中数值常量可以是整型数 `IntConst`，也可以是浮点数 `FloatConst`。整型数参考 ISO/IEC 9899 第54页关于整型常量的定义，在此基础上忽略所有后缀；浮点数参考 ISO/IEC 9899 第57页关于浮点常量的定义，在此基础上上忽略16进制浮点表示和所有后缀。

实验内容与提交要求

在本课程实验中，你将通过编写文法描述文件和相关的处理代码，使用 ANTLR v4，最终为 C1 语言实现一个能输出抽象语法树的 C1 解析器。为便于过程控制与管理，课程实验拆分成三步，分三次提交：

- lab1-1：构造 C1 的词法分析器
- lab1-2：构造 C1 的语法分析器
- lab1-3：构造能生成 AST 的 C1 解析器

你需要对所提供的课程实验软件包中的部分文件进行修改和完善，补充更多的测试例子，撰写和提交介绍课程实验的分析、设计和测试方面的报告。你需要事先阅读本页其他小节的内容，它们将能在一定程度上给你的课程实验提供帮助和指引。如果有疑问请及时提出，我们会公开回答有必要回答的问题。

Lab1-1. C1 的词法分析

主要任务

- 根据 C1 语言 的词法描述，修改并完善 C1 的词法描述文件 `c1recognizer/grammar/C1Lexer.g4`
- 利用 ANTLR v4 节介绍的 `antlr4` 和 `grun` 工具正确构造你的词法分析器
- 编写若干个正确的和错误的C1程序（侧重在词法是否有错误）作为测试用例，测试你的 C1 词法分析器
  - 测试程序置于 `c1recognizer/test/test_cases/lexer` 目录下
  - 正确测试用例文件名前缀为 `pt_`
  - 错误测试用例文件名前缀为 `ft_`
- 在 `c1recognizer/doc` 目录下增加markdown文件描述实验遇到的问题、分析与设计，文件名前缀为 `lab1-1`

重点与难点

- 特别注意行尾的处理，正确地处理 CRLF 和 LF 两种换行
- 你需要仔细编写单行注释和多行注释的词法规则，并构造各种测试例子进行测试
- 你需要仔细考虑数值的多种形式，编写能表达这些形式的词法规则，并构造各种测试例子进行测试

提交说明

你需要参照实验软件包 c1recognizer的目录结构来组织你本次实验的提交文件，目录结构如下。助教会借助脚本实现实验的半自动检查，请一定严格按照本节要求组织项目目录和文件，否则会影响本次实验成绩。

```
- <your repo>
  | c1recognizer          复制自公共仓库的 c1recognizer 项目，请勿遗漏内容。
    | cmake/
>>  | grammar/            修改其中的 C1Lexer.g4
    | include/c1recognizer/
    | src/
>>  | test/test_cases/    增加你的测试程序
>>  | doc/                增加文档描述实验中遇到的问题、分析和设计，文件名前缀为lab1-1
    | 其他已有的文件
```

ANTLR v4 的使用

ANTLR v4 是一个基于 LL(*) 分析方法 [PLDI 2011] 、Adaptive LL(*)分析[OOPSLA 2014]的自上而下分析的LL解析器的生成工具（Parser Generator）。它接收一个或多个 `.g4` 格式的文法描述文件作为输入，然后分析它们并为之生成一套解析器的源程序。生成的解析器依赖于 ANTLR v4 的运行时库，能够执行指定的文法分析过程，并可以通过 ANTLR v4 运行时提供的接口和数据格式来调用与访问。

ANTLR v4 所提倡的思想在于，使用文法描述文件描述语言本身，而不必描述具体实现细节。也就是说，文法描述文件应当只是对语言本身文法的一个规范描述，它不应包括其实现逻辑（虽然 **ANTLR** 具备在文法描述文件中附加程序代码的功能，但并不提倡使用该功能；这一功能是从 **ANTLR v3** 遗留下来的）；直到根据输入生成 **parse tree** 后，才通过用 **ANTLR** 工作语言编写的 **Visitor** 遍历**parse tree** 并生成抽象语法树 **AST**。

文法描述文件（ `.g4` ）的编写

在这里安利 **Visual Studio Code**的**antlr4**插件 ！这在一定程度上可以通过自动的检查来避免一些低级错误的产生。

g4格式文件的作用是描述语言的文法。其中能够定义终结符和非终结符，终结符代表词法记号，非终结符代表语法结构。

ANTLR v4 允许在单个文法描述文件中描述语言的词法和语法，但是本实验会将词法描述和语法描述分离成多个文件。分离的原因主要是为了便于分步进行课程实验、调试和提交，也有助于理解和评分。

在用于本课程实验的初始代码中，给出了全部需要完成的终结符和非终结符，但是只填入了能够完成简单的整数加、减、乘、除、取余表达式解析的文法描述。你需要在本实验中将它完善至能够解析完整的 C1 语言。

词法分析（**Lexer**）

词法分析的文法文件（未完成,等待修改/增加内容）位于课程实验软件包的 `c1recognizer/grammar/C1Lexer.g4` 。它的初始内容如下：

```
lexer grammar C1Lexer;

tokens {
    Comma,
    SemiColon,
    Assign,
    LeftBracket,
    RightBracket,
    LeftBrace,
    RightBrace,
    LeftParen,
    RightParen,
    If,
    Else,
    While,
    Const,
    Equal,
    NonEqual,
    Less,
    Greater,
    LessEqual,
    GreaterEqual,
    Plus,
    Minus,
    Multiply,
    Divide,
    Modulo,

    Int,
    Float,
    Void,

    Identifier,
    IntConst,
    FloatConst
}

LeftParen: '(';
RightParen: ')';

Plus: '+' ;
Minus: '-' ;
Multiply: '*' ;
Divide: '/' ;
Modulo: '%' ;

WhiteSpace: [ \t\r\n]+ -> skip;
```

文法类型和分析器名

在 ANTLR v4 的文法文件格式中，首先应注明文法类型和分析器名：`lexer grammar C1Lexer;` 表示该文法文件描述了一个词法分析器，其名称为 C1Lexer。

记号声明

`tokens{...}` 中包含所有 token 名，以逗号分隔。这里已经列出了你会用到的所有 token。需要注意的是：

- 在 ANTLR 语法文件中，**token** 名需以大写字母开头;
- 空白符等不包含在其中。

你可以看到，在文件最后一行出现的 `WhiteSpace` 并不在这里出现。这是因为它们所代表的文本将被忽略（通过命令 `-> skip` 来表达，后面会有详细解释），不会出现在词法分析器 C1Lexer 输出的 token 流（ANTLR v4 运行时库中的 `TokenStream`）中。

文法规则——词法规则

接下来则是文法规则。在 `lexer grammar` 中，这里出现的规则必须是对词法单元的描述，即对 token 的定义。每条规则由大写字母开头的 token 名开始，后跟一个冒号，然后是对该 token 构成规则的描述。一个 token 可以由多种模式匹配到，各个模式之间以 `|` 分隔，规则以 `;` 结束。

规则的表示

规则的表述与正则表达式大致相同。例如，你可以看到 `'blabla'` 表示相应的字面文本，`[0-9]` 表示相应的单字符可能值，`+` 作为后缀修饰符表示一次或多次出现相应的文本；此外，还有 `?` 作为后缀修饰符表示被修饰文本的可选出现，而 `(somerule)*` 等价于 `((somerule)+)?`。这些都与正则表达式的表示相同。但是需要注意其中的区别也是存在的：在课程实验中主要涉及的区别在于，正则表达式中使用 `[^a-z]` 表示一个非小写字母的字符，而在 ANTLR 中使用记号 `~[a-z]`，其中 `~` 为一个前缀修饰符，用于匹配一个不在此修饰符后描述的字符范围内的字符；这里的“范围”可能不仅由 `[]` 内的规则指定，还可能是单字符的字面值，例如 `~'a'` 表示一个非a的字符。

本次实验中匹配注释是最困难的部分。你可能会用到一些后缀标识符，如 `*?`，它代表 non-greedy many，也就是说，在此处匹配尽可能少的内容使得整条规则得以匹配。其它的简单后缀标识符，包括 `+` 和 `?`，都有它们的 non-greedy 版本，即 `+?` 和 `??`，它们的含义也类似。

命令

每一条规则结尾处可附加一条命令，跟随在 `->` 之后。本次实验中仅会用到命令 `skip`，它表示此条规则匹配到的 token 会被忽略，不计入 `TokenStream` 中。

其他

此外，ANTLR 允许在词法规则中附加操作代码，也允许词法规则中出现嵌套和递归，同时还有匹配模式(mode)的切换和模式栈机制的存在。如果你想了解更多关于 ANTLR Lexer 规则的信息，请参考 Lexer Rules。

实验代码的编译和运行

ANTLR v4 具备独立测试语法文件、以及将parse tree 和分析结果可视化的功能。

为可视化分析图(lab1接下来的阶段会用到可视化分析)，推荐使用 Visual Studio Code 作为编辑器，安装 ANTLR 插件。在一个 `.g4` 文件的编辑窗口中右键，你会看到几个选项，点击即可。

为测试Lexer，你首先需要按照 Environment 页面中的要求配置好 ANTLR v4 的工作环境。在此前提下，按如下步骤完成测试（在 `c1recognizer/grammar` 目录中，以../test/test_cases/simple.c1为例）：

```
# Compile grammar to Java source code
antlr4 *.g4
# Compile Java source code
javac *.java
# Testing lexer
grun C1Lexer tokens -tokens ../test/test_cases/simple.c1
```

参考的输出如下:

```
>> grun C1Lexer tokens -tokens ../test/test_cases/simple.c1
[@0,0:2='int',<'int'>,1:0]
[@1,4:4='i',<Identifier>,1:4]
[@2,6:6='=',<'='>,1:6]
[@3,8:8='0',<IntConst>,1:8]
[@4,9:9=';',<';'>,1:9]
[@5,12:15='void',<'void'>,2:0]
[@6,17:20='main',<Identifier>,2:5]
[@7,21:21='(',<'('>,2:9]
[@8,22:22=')',<')'>,2:10]
[@9,25:25='{',<'{'>,3:0]
[@10,32:32='i',<Identifier>,4:4]
[@11,34:34='=',<'='>,4:6]
[@12,36:36='1',<IntConst>,4:8]
[@13,37:37=';',<';'>,4:9]
[@14,40:40='}',<'}'>,5:0]
[@15,43:42='<EOF>',<EOF>,6:0]
```

`grun` 的命令中你也可以不指定输入文件，采用标准输入流作为输入文本。这时候你需要键入一个 EOF 标志来结束输入：在 Linux 等环境中是 Ctrl-D。