第3章. 基于框架的编译课程实验

3.1 BIT-MiniCC

3.1.1. 相关背景

编译原理课程是计算机学科的一门核心专业必修课程，课程通过介绍编译系统构造的理论基础、构造方法和实现技术，不仅让学生掌握编译器的工作过程，而且让学生掌握语法分析、语义分析、中间代码生成、代码优化和目标代码生成的基本原理和方法，并具备设计一个基本编译系统的能力。该课程是理论与实践结合的典型课程，通过理论学习指导实践，通过实践进一步加深对理论知识的理解，也是在本科阶段培养学生动手能力的非常重要的环节。

然而，在日常的教学过程中，如何快速有效地掌握编译原理的基本理论和方法成为了学生们的一大困难，很多学生都反映编译原理课程晦涩难懂，学习了很多的理论却无法融会贯通，导致对编译原理的理解不够深入，无法将其理论和方法转化为一个可以运行的编译器。经过多年的实践和总结，发现由于以下多种原因，学生往往无法按照预定的步骤完成相应的课程实验：

- **从理论到实践的距离**：学生虽然在课堂教学过程中掌握了基本理论和方法，但是要从头设计一个编译器时，却不知从何处下手。主要原因在于，工业级的编译器是一个大规模的复杂软件系统，不太可能成为大部分学生编译器的一个工具，学生缺少轻量级、模块化的编译器实现作为参考；
- **从前端到后端的距离**：编译器典型框架结构中包括了词法分析、语法分析、语义分析和中间代码生成、代码优化、目标代码生成等多个阶段。阶段之间按照顺序衔接工作，前端分析模块的实现质量将直接影响后端的设计和实现。目前的现状是大部分学生的前端实现无法满足后端综合的要求，导致后端直接无法实施。
- **从理想到现实的距离**：编译器的开设一般在计算机专业三年级，这个阶段学生面临多个核心专业课的学习，又要考虑参加竞赛、实验室项目，以及面临出国和就业的压力，因此没有足够的时间实现如此大型的软件项目，时间不足是导致学生无法完成整个工作流程的一个重要影响因素。

为了解决上述问题，北京理工大学编译原理课程组教师根据实际教学工作的需要，设计并实现了一个小型的 C 语言编译器框架。该框架将编译器的工作流程划分为多个阶段，并为每个阶段设计并实现了一个内嵌的参考实现。该参考实现是黑盒的，并不对学生开放，但是学生可以运行每个模块，并查看到相应的输入和输出。学生可以使用自己的模块实现替换框架内嵌的模块，也可以部分使用内嵌模块，部分使用自己设计的模块。框架不仅为学生提供了一个参考实现，也很好的解决了上述三个“距离”问题。

BIT-MiniCC 设计了规范的中间文件，全部使用 XML 文件表示，另外框架本身使用 Java 语言实现，因此具有较好的跨平台特性，但是框架并没有限定学生实现自己的模块所使用的语言，学生可以使用 Java、C 或者 Python 实现自己的模

块并进行替换操作，运行框架并查看结果，极大地提高了框架的灵活性和兼容性。

BIT-MiniCC 集成了 MIPS 等处理器的模拟器，基于框架生成的汇编代码，可以直接在模拟其中汇编并运行，通过模拟器，学生能够更直观的观测到程序运行的过程，以及处理器内部的变化过程。经过验证后的程序，能够与其他课程进行很好的衔接，例如：将生成的二进制程序下载到自己设计的基于 FPGA 的 CPU 上运行。

综上所述，该框架的主要特点在于：

（1）该框架将 C 语言的编译过程划分为多个阶段，相互衔接的模块通过 XML 文件进行数据交换。这样设计的目的是使学生能够更好的了解每个阶段的工作原理，输入数据和输出数据。通过观察模块之间的交互和衔接，能够更直观的了解编译器的工作过程。

（2）该框架包含了编译器各个阶段的内部实现，学生可以直接运行该框架，查看多个阶段之间的输入和输出。内部集成的各个模块的源代码是不可见的，仅供学生自己实现各个模块时参考。

（3）在使用框架的过程中，学生可以自由选择使用内部集成的模块或者自己设计的模块。其原因及好处在于，编译器各个阶段是互相依赖的，如果前面部分实现不好，后续工作较难进行，但是基于该框架，进行后端的实验时，可以直接选择使用内部集成的前段模块，从而节省了时间。

（4）该框架集成了 MIPS 的汇编器和模拟器 MARS，生成代码后可以直接调用该模块对生成的代码进行验证。如果验证成功，则可以与体系结构和组成相关课程实验进行衔接，将生成的代码在自己设计的目标系统上运行。

（5）学生可以定义新的指令，并将高级语言程序翻译为新指令。

3.1.2. 框架结构

尽管编译程序的处理过程十分复杂，不同的编译程序实现方法也各不相同，但任何编译程序的基本功能都是类似的，其基本逻辑功能以及必要的模块大致相同，即都要经过预处理、词法分析、语法分析、语义分析和中间代码生成、代码优化、目标代码生成这些步骤，并在这些步骤中贯穿着表格管理和出错处理的功能。图 1- 1 给出了一般编译程序的典型逻辑结构。

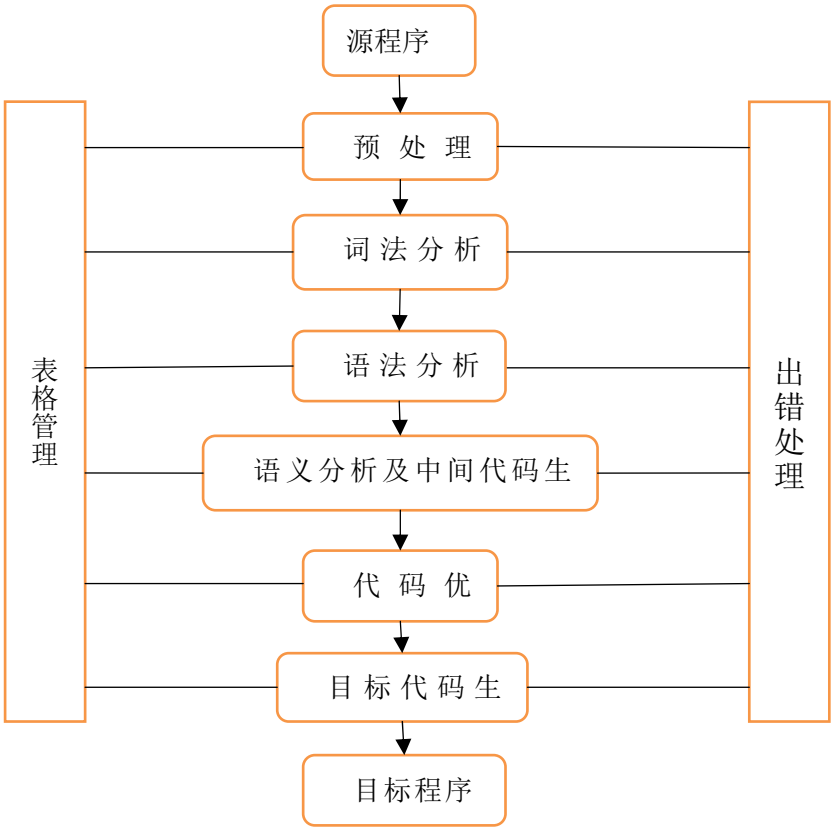


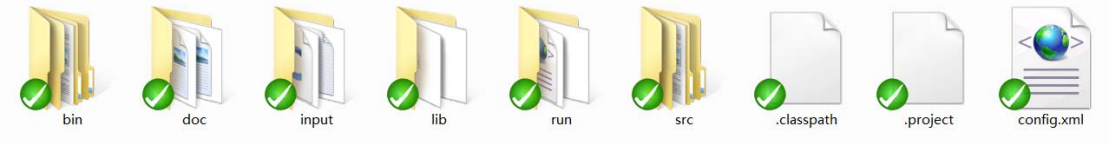
图 1- 1 编译程序典型逻辑结构

如图 1- 1 所示，一般编译器的编译过程分为预处理、词法分析、语法分析、语义分析及中间代码生成、代码优化、目标代码生成几个阶段。其中，表格管理和出错管理两个模块可以在编译的任何阶段被调用，以便辅助完成编译功能。

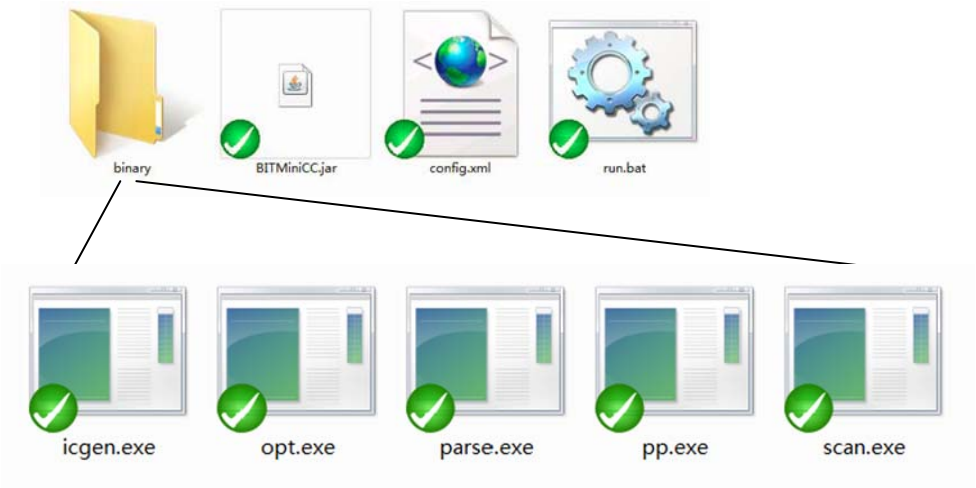
关于每个模块的功能，已经在各个编译原理相关的教材进行了详细的介绍，本书不再赘述。

3.1.3. 框架使用方法

框架基于 Java 语言开发，框架的运行需要 Java 运行时环境(JRE)。目前项目中配置为 JRE 1.7，但是也可修改为更高版本 Java。项目目录如下所示：



其中 bin 是框架代码编译后的 class 文件目录，doc 目录为项目相关的文档描述，input 为框架运行时测试源程序的输入目录，lib 为为框架用到的相关的 Java 库程序，src 为框架源代码部分（不包含内嵌模块），run 为框架导出之后运行目录，该目录如下所示：



图中的 binary 目录下为学生使用 C 语言设计并实现的编译器的各个模块。

其中 BITMiniCC.jar 为 Eclipse 导出的可运行的 jar 文件，config.xml 为控制框架运行的配置文件，run.bat 为 Windows 环境中运行的 bat 文件，binary 文件夹下包含了使用其他语言编写的编译器各个模块。如果使用 Java 语言编写各个模块，则相应的程序已经在 BITMiniCC.jar 文件中包含，如果使用 C 语言或者 Python。

由于学生可以选择使用 Java、C/C++和 Python 来实现相应的模块，并且考虑到编译器从预处理到模拟执行的运行时间比较长，在实际开发中可能仅仅关注某一个模块，因此框架使用了如下的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
- <config name="config.xml">
  - <phases>
    - <phase>
        <phase name="pp" path="" type="java" skip="false"/>
        <phase name="scanning" path="" type="java" skip="false"/>
        <phase name="parsing" path="" type="java" skip="false"/>
        <phase name="semantic" path="" type="java" skip="false"/>
        <phase name="icgen" path="" type="java" skip="false"/>
        <phase name="optimizing" path="" type="java" skip="false"/>
        <phase name="codegen" path="" type="java" skip="false"/>
        <phase name="simulating" path="" type="java" skip="false"/>
      </phase>
    </phases>
</config>
```

配置文件按照程序编译和模拟运行的各个阶段进行划分，从上往下依次为预处理（pp）、词法分析（scanning）、语法分析（parsing）、语义分析（semantic）、中间代码生成（icgen）、优化（optimizing）、代码生成（codegen）和模拟运行（simulating），阶段在每个 phase 的 name 属性中进行标注。除此之外，每个阶段 phase 还可以指定 type 和 path，分别是实现的语言和相应的可执行程序的路径。如果是使用 Java 语言实现的，可以直接在框架项目中完成，因此不需要指定路径，如果是使用 C/C++或者 Python 实现的，则需要指定相应可执行程序的路径。例如部分使用 C/C++语言实现的配置文件可以按照如下的方式对框架进行配置。

```
<?xml version="1.0" encoding="UTF-8" ?>
- <config name="config.xml">
  - <phases>
    - <phase>
        <phase skip="false" type="binary" path="D:\projects\bit-minicc\bit-minic-clean\run\binary\pp.exe" name="pp" />
        <phase skip="false" type="binary" path="D:\projects\bit-minicc\bit-minic-clean\run\binary\scan.exe" name="scanning" />
        <phase skip="false" type="java" path="" name="parsing" />
        <phase skip="false" type="java" path="" name="semantic" />
        <phase skip="false" type="java" path="" name="icgen" />
        <phase skip="false" type="java" path="" name="optimizing" />
        <phase skip="false" type="java" path="" name="codegen" />
        <phase skip="false" type="java" path="" name="simulating" />
      </phase>
    </phases>
  </config>
```

编译器框架以 jar 包的形式发布，通过命令行调用，调用时需要传入一个命令行参数，代表需要处理的 C 源程序的路径。如下所示：

java -jar BITMiniCC.jar test.c

在 Windows 环境中，也可以运行同一个目录下 bat 文件，如下所示：

run.bat test.c

运行结果如下图所示：

```
D:\projects\bit-minicc\bit-minic-clean\run>run.bat D:\projects\bit-minicc\bit-minic-clean\input\test
.c

D:\projects\bit-minicc\bit-minic-clean\run>java -jar BITMiniCC.jar D:\projects\bit-minicc\bit-minic-
clean\input\test.c
Start to compile ...
1. PreProcess finished!
2. LexAnalyse finished!
3. Parse finished!
4. Semantic finished!
5. Intermediate code generate not finished!
6. Optimize not finished!
OP: return
7. Code generate finished!
8. Simulate not finished!
Compiling completed!
```

3.2 实验目的

ACM/IEEE-CS 计算学科 2013 新教程（简称 CS2013 教程）中，根据计算学科的迅速发展和变化，根据学校的定位和培养目标，亦强调计算机学科学生除了掌握本学科领域重要的知识和技能，还要有领域拓宽和终身学习的能力。CS2013 教程比 CC2001 对计算学科涉及的知识领域的凝练在深度和广度上都有较大变化。它将计算学科划分为 18 个知识领域[1]，编译原理与设计课程涉及的知识直接关联到 ACM/IEEE-CS2013 许多知识领域，诸如算法和复杂性、计算科学、架构与组织、系统的基础、离散结构、编程语言、并行和分布式计算、软件工程等。因此若本课程教学计划仍然沿用传统的教学模式而不进行改革，将难以支撑课程改革和学科发展的需求，难以胜任研究型大学的培养目标。

编译原理是计算机科学与技术专业的主干课程之一，在计算机本科教学中占有重要地位。该课程具有较强的理论性和实践性，但在教学过程中容易偏重于理论介绍而忽视实验环节，使学生在学习过程中普遍感到内容抽象，不易理解。该

实验的目的是指导和帮助学生通过实践环节深入理解与编译实现有关的形式语言理论基本概念，掌握编译程序构造的一般原理、基本设计方法和主要实现技术，并通过运用自动机理论解决实际问题，从问题定义、分析、建立数学模型和编码的整个实践活动中逐步提高软件设计开发的能力。

3.3 实验内容

3.3.1. 词法分析

该实验以 C 语言作为源语言，构建 C 语言的词法分析器，对于给定的测试程序，输出 XML 格式的属性字符流。词法分析器的构建按照 C 语言的词法规则进行。C 语言的发展尽力了不同给的阶段，早期按照 C99 标准进行编程和编译器的实现，2011 年又对 C 语言规范进行了修订，形成了 C11（又称 C1X）。下面以 C1X 为基准，对 C 语言的词法规则进行简要的描述。

C 语言的**关键字**包括如下单词：

```
auto            * if              unsigned
break             inline          void
case              int             volatile
char              long            while
const             register        _Alignas
continue          restrict        _Alignof
default           return          _Atomic
do                short           _Bool
double            signed          _Complex
else              sizeof          _Generic
enum              static          _Imaginary
extern            struct          _Noreturn
float             switch          _Static_assert
for               typedef         _Thread_local
goto              union
```

C 语言**标识符**的定义如下：

identifier:

identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:

nondigit
universal-character-name
other implementation-defined characters

nondigit: one of

```
_  a  b  c  d  e  f  g  h  i  j  k  l  m
   n  o  p  q  r  s  t  u  v  w  x  y  z
   A  B  C  D  E  F  G  H  I  J  K  L  M
   N  O  P  Q  R  S  T  U  V  W  X  Y  Z
```

digit: one of

```
0  1  2  3  4  5  6  7  8  9
```

C 语言**整型常量**的定义如下：

integer-constant:

decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:

nonzero-digit
decimal-constant digit

octal-constant:

0
octal-constant octal-digit

hexadecimal-constant:

hexadecimal-prefix hexadecimal-digit
hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: one of

0x 0X

nonzero-digit: one of

`1 2 3 4 5 6 7 8 9`

octal-digit: one of

`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of

`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:

unsigned-suffix $long\text{-}suffix_{opt}$
unsigned-suffix *long-long-suffix*
long-suffix $unsigned\text{-}suffix_{opt}$
long-long-suffix $unsigned\text{-}suffix_{opt}$

unsigned-suffix: one of

`u U`

long-suffix: one of

`l L`

long-long-suffix: one of

`ll LL`

C 语言**浮点型常量**定义如下：

floating-constant:
decimal-floating-constant
hexadecimal-floating-constant

decimal-floating-constant:
fractional-constant *exponent-part*$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence *exponent-part* *floating-suffix*$_{opt}$

hexadecimal-floating-constant:
hexadecimal-prefix *hexadecimal-fractional-constant*
binary-exponent-part *floating-suffix*$_{opt}$
hexadecimal-prefix *hexadecimal-digit-sequence*
binary-exponent-part *floating-suffix*$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
+ **-**

digit-sequence:
digit
digit-sequence *digit*

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{opt}$ **.**
hexadecimal-digit-sequence
hexadecimal-digit-sequence **.**

binary-exponent-part:
p *sign*$_{opt}$ *digit-sequence*
P *sign*$_{opt}$ *digit-sequence*

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence *hexadecimal-digit*

floating-suffix: one of
f **l** **F** **L**

C 语言**字符常量**定义如下：

character-constant:

' *c-char-sequence* '
L' *c-char-sequence* '
u' *c-char-sequence* '
U' *c-char-sequence* '

c-char-sequence:

c-char
c-char-sequence c-char

c-char:

any member of the source character set except
the single-quote ', backslash \, or new-line character
escape-sequence

escape-sequence:

simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence
universal-character-name

simple-escape-sequence: one of

\' \" \? \\
\a \b \f \n \r \t \v

C 语言**字符串字面量**定义如下：

string-literal:

encoding-prefix$_{opt}$ " *s-char-sequence*$_{opt}$ "

encoding-prefix:

u8
u
U
L

s-char-sequence:

s-char
s-char-sequence s-char

s-char:

any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

C 语言**运算符**和**界限符**定义如下：

```
[   ]   (   )   {   }   .   ->
++   --   &   *   +   -   ~   !
/   %   <<   >>   <   >   <=   >=   ==   !=   ^   |   &&   ||
?   :   ;   ...
=   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
,   #   ##
<:   :>   <%   %>   %:   %:%:
```

3.3.2. 语法分析

3.3.3. 语义分析

3.3.4. 代码生成

3.3.5. 模拟运行

3.4 实验过程与方法

3.4.1. 词法分析

从 github 下载 BIT-MINICC 框架，下载网址为：https://github.com/jiweixing/bit-minic-compiler；编写测试程序，并使用内置的词法分析器对输入进行测试，观察词法分析器的输入和输出；选择实现的语言（Java、C 或者 Python 等），设计实现自己的 C 语言词法分析器。

例如如下的测试程序：

```
int main(int a, int b){      return a + b;}
```

输出的 XML 格式的属性字流为：

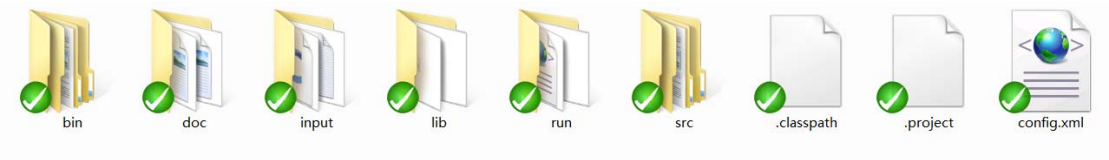
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="test.l">
  <tokens>
    <token>
      <number>1</number>
      <value>int</value>
      <type>keyword</type>
      <line>1</line>
      <valid>true</valid>
    </token>
    <token>
      <number>2</number>
      <value>main</value>
      <type>identifier</type>
      <line>1</line>
      <valid>true</valid>
    </token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
    <token>
  </tokens>
</project>
```

3.5 实验提交内容

3.5.1. 词法分析

本实验要求提交词法分析器实现源码，C/C++需提供对应的可执行程序（不需要编译的中间文件），Java 提供编译后的 class 文件或者 jar 包，每个人提交一份实验报告。

提交目录如下所示：



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

3.6 其他

已有框架使用的 Java JRE 的版本可能与每个同学机器上的并不相同，建议根据自己的实际情况进行配置。

框架还在完善的过程中，github 版本将不定期进行更新，如有问题请及时与我们联系。