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。

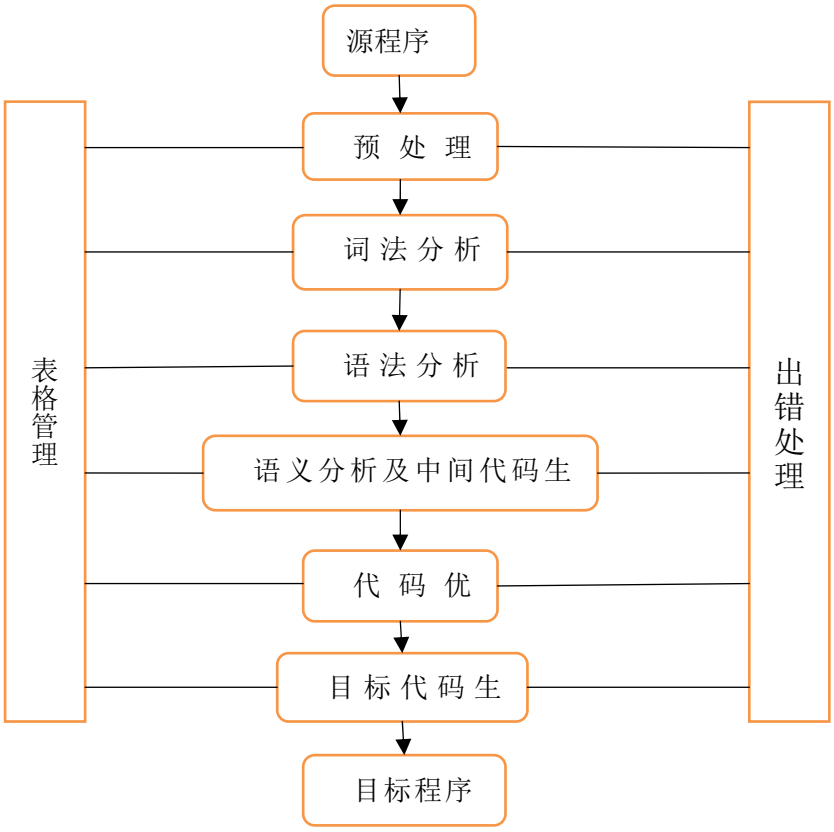


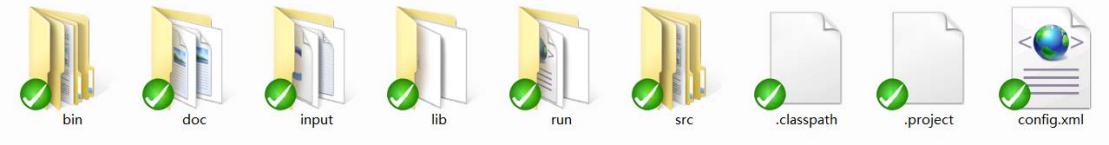
图 1- 1 编译程序典型逻辑结构

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。

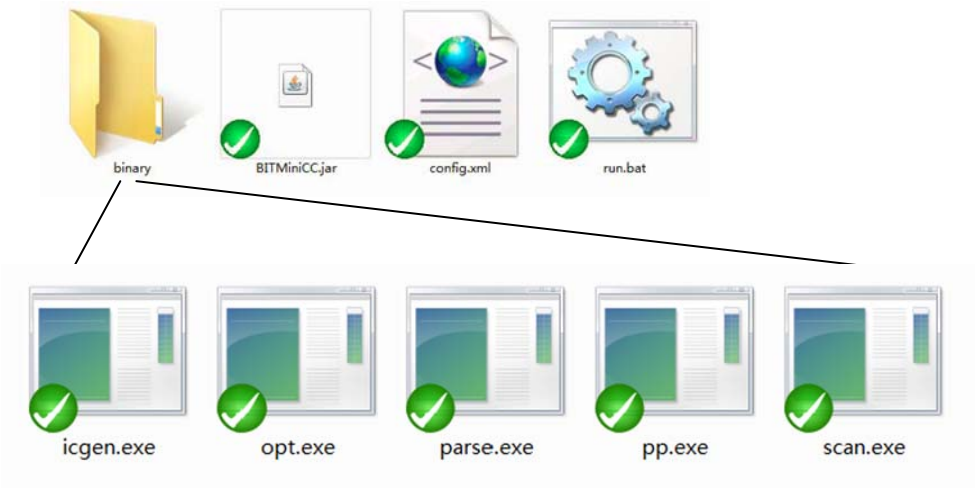
关于每个模块的功能，已经在各个编译原理相关的教材进行了详细的介绍，本书不再赘述。

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

语法分析的任务要求

该实验选择 C 语言的一个子集，基于 BIT-MiniCC 构建 C 语法子集的语法分析器，该语法分析器能够读入 XML 文件形式的属性字符流，进行语法分析并进行错误处理，如果输入正确时输出 XML 形式的语法树，输入不正确时报告语法错误。

需要说明的是，能够分析的输入程序依赖于选用的语法子集，而输出的语法树的结构又与文法的定义密切相关。

可参考扩充的 C 语言文法

如下为 C 语言文法的一个子集：

```
CMPL_UNIT     : FUNC_LIST
FUNC_LIST     : FUNC_DEF FUNC_LIST | ε
FUNC_DEF      : TYPE_SPEC ID ( ARG_LIST ) CODE_BLOCK
TYPE_SPEC     : int | void
PARA_LIST     : ARGUMENT | ARGUMENT , PARA_LIST | ε
ARGUMENT      : TYPE_SPEC ID
CODE_BLOCK    : { STMT_LIST }
STMT_LIST     : STMT STMT_LIST | ε
STMT          : RTN_STMT | ASSIGN_STMT
RTN_STMT      : return EXPR
ASSIGN_STMT   : ID = EXPR
EXPR          : TERM EXPR2
EXPR2         : + TERM EXPR2 | - TERM EXPR2 | ε
TERM          : FACTOR TERM2
TERM2         : * FACTOR TERM2 | / FACTOR TERM2 | ε
FACTOR        : ID | CONST | ( EXPR )
```

读者可以在此基础之上进行文法扩充，包括全局变量声明，循环语句、分支

语句、函数调用语句以及 switch 语句等。要求至少包括局部变量声明语句、赋值语句、返回语句、一种分支语句（if, if-else, switch 等）和一种循环语句（for, while, do-while 等）。

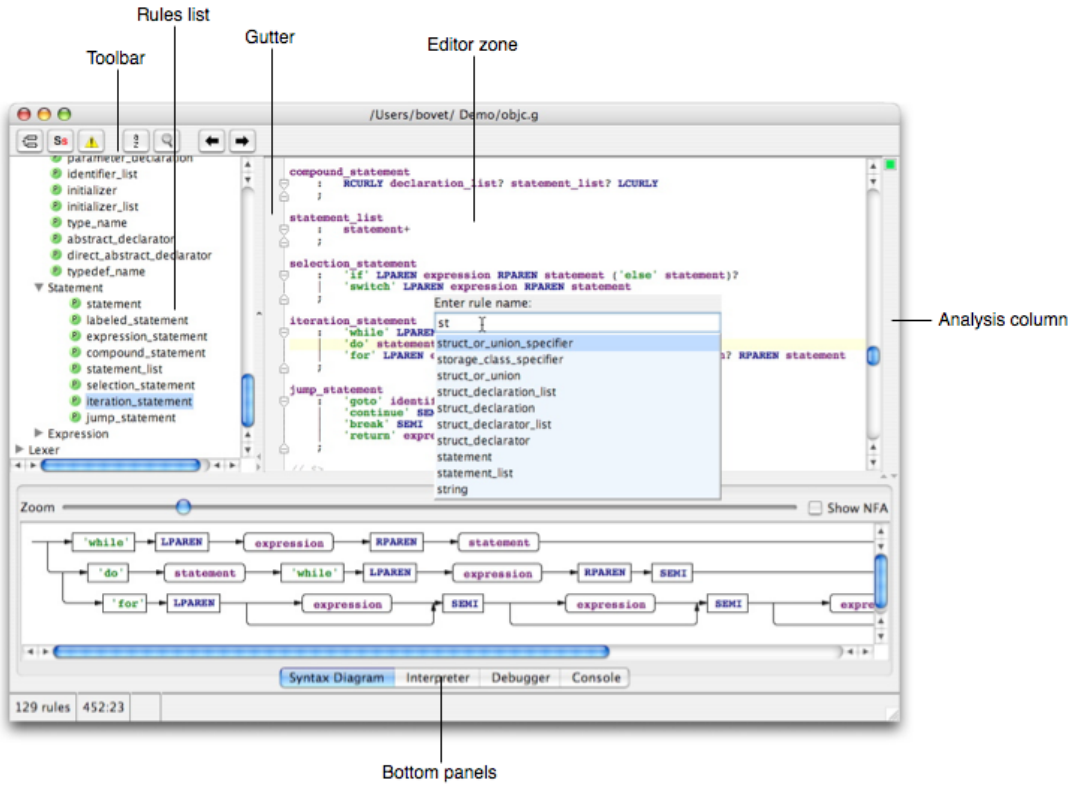
文法设计工具 ANTLRWorks

ANTLRWorks（http://www.antlr3.org/download/antlrworks-1.5.2-complete.jar）是由 Jean Bovet 等人基于 ANTLR 设计并实现的文法设计及检验工具，用户可以使用友好的用户界面编辑、查看、解释执行和调试文法。主要的功能包括：文法编辑及语法高亮显示、动态语法图显示、集成的文法解释器、集成的语言诊断调试器等。

ANTLRWorks 是使用 Java 语言编写的，因此可以在安装了 JRE 1.5 以上版本的多个系统平台上运行。目前该程序打包为一个 jar 包进行分发，因此可以按照如下命令运行：

```
$ java -jar antlrworks.jar
```

ANTLRWorks 为每个文法文件提供了一个编辑窗口，该窗口又进一步分为不同的区域，如下图所示：



主要的部分包括：

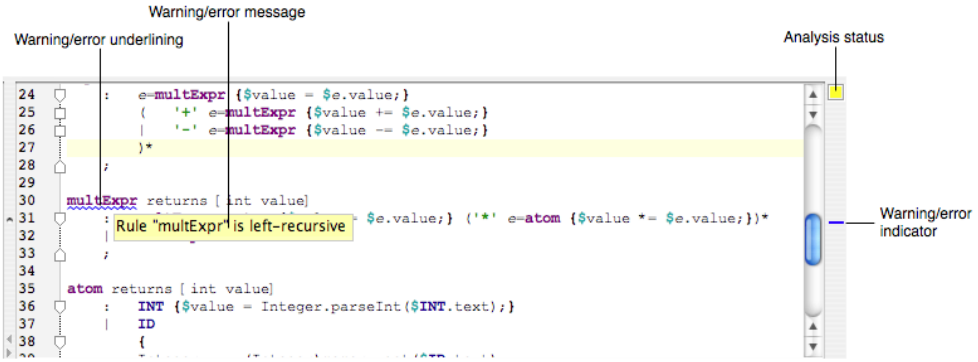
编辑区域：用户可以在该区域编辑文法，能够实时的对输入的文法进行高亮着色显示，并给出文法中的相关错误。在编辑区的左侧显示了规则对应的行号，

规则限制以及调试断点信息；

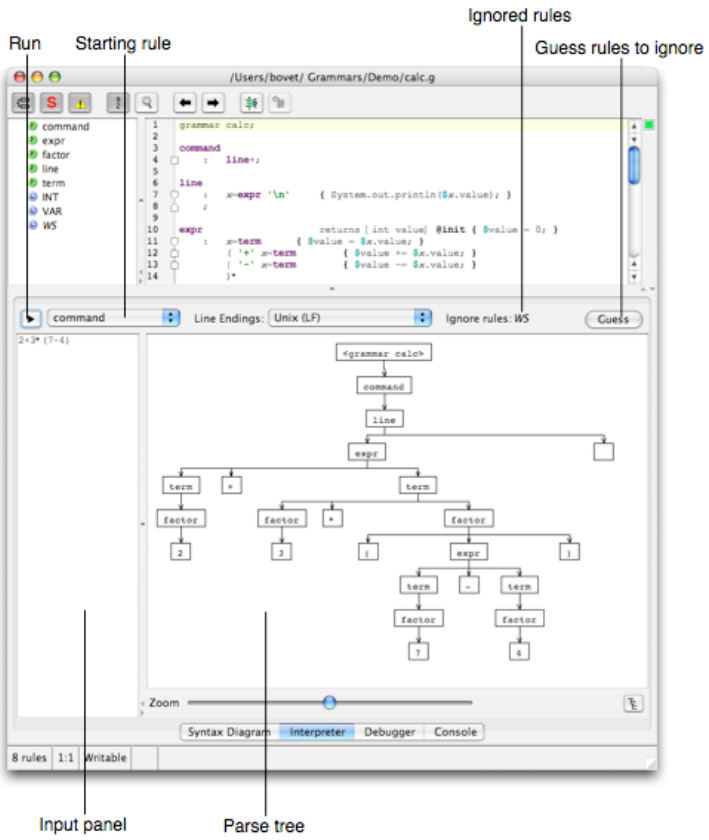
规则列表：所有文法规则的列表区域，给出了所有词法和语法相关的规则，点击相关的规则可以跳转到该规则上，也各异对规则进行分组（右击）。

底部区域：包括语法图、解释器、调试器、控制台等标签窗口。

ANTLRWorks 使用与 Eclipse 类似的方式显示关于文法的错误和警告信息，以左递归文法为例，在文法非终结符的底部以波浪线提示，鼠标放上去之后会给出具体的警告或者错误信息。



ANTLRWorks 最大好处在于可以使用集成的解释器立即对输入的文本进行分析，并不需要提前根据文法生成相应的词法分析器、语法分析器，因此对文法的设计与验证起到即时的支撑。如下所示，点击左下的 Interpreter，再输入框中输入测试代码文本，选择相应的开始产生式，点击运行，即可看到以分析树呈现的解析结果。



例如，在文本框里面输入下面的关于表达式的文法：

```
grammar Expr;

@header {
package test;
import java.util.HashMap;
}

@lexer::header {package test;}

@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}

prog:   stat+ ;

stat:   expr NEWLINE {System.out.println($expr.value);}
    |   ID '=' expr NEWLINE
        {memory.put($ID.text, new Integer($expr.value));}
    |   NEWLINE
    ;

expr returns [int value]
    :   e=multExpr {$value = $e.value;}
        (   '+' e=multExpr {$value += $e.value;}
        |   '-' e=multExpr {$value -= $e.value;}
        )*
```

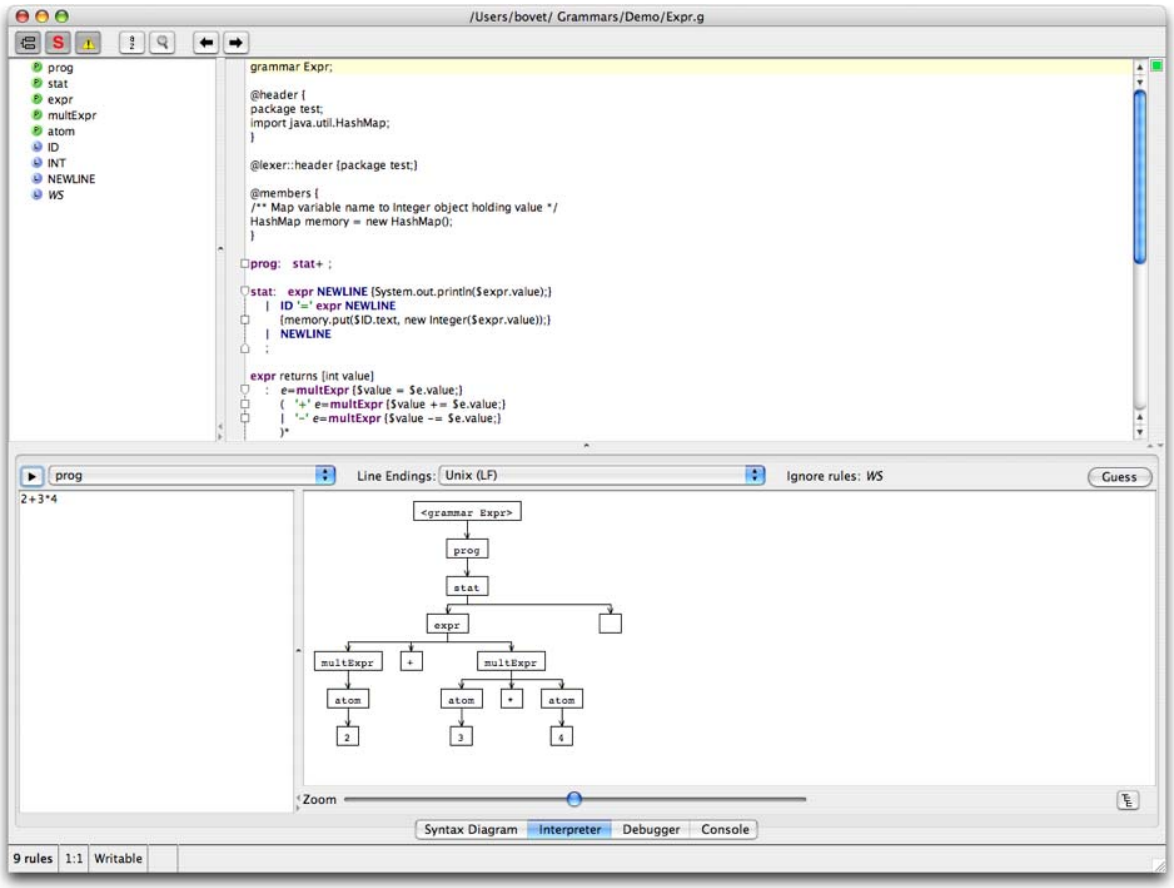
```
    ;

multExpr returns [int value]
    :   e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
    ;

atom returns [int value]
    :   INT {$value = Integer.parseInt($INT.text);}
    |   ID
        {
        Integer v = (Integer)memory.get($ID.text);
        if ( v!=null ) $value = v.intValue();
        else System.err.println("undefined variable "+$ID.text);
        }
    |   '(' e=expr ')' {$value = $e.value;}
    ;

ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t')+ {skip();} ;
```

在输入文本框里面输入测试文本“2+3*4”，并选择开始符号“prog”并点击按钮▶运行解释器，就会看到相应的语法树：



需要注意的是：ANTLRWorks 输入文法时可以采用扩展的 BNF 表示方法，因此可以使用“[]”、“+”等多种扩展符号。

3.3.3. 语义分析

语义分析的任务要求

语义分析阶段的工作主要包括两部分：

（1）基于词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；

（2）将分析树转换为抽象语法树并输出。

其中语义检查的内容主要包括：

（1）变量使用前是否进行了定义；

（2）变量是否存在重复定义；

（3）break 语句是否在循环语句或者 switch 语句中使用；

（4）函数调用的参数个数和类型是否匹配；

（5）运算符两边的操作数的类型是否相容；

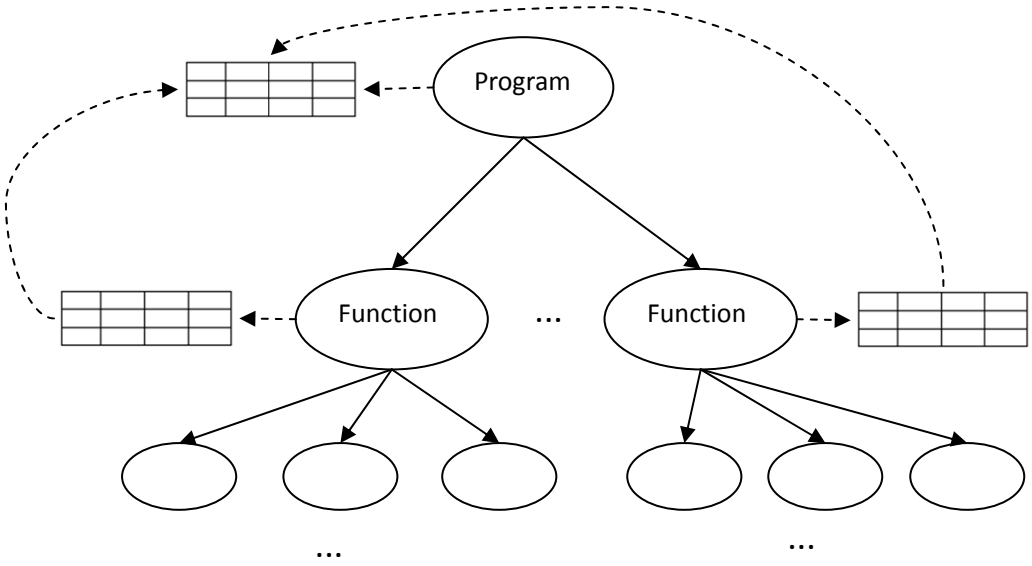
语义检查的前两项（1）和（2）为要求完成内容，而（3）-（5）为可选内容。

符号表的设计与实现

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，因此由语法分析器进行符号表的创建更为合适一些。

针对 C 语言的特点，结合抽象语法树，我们给出如下的参考设计，并在 BIT-MINICC 中进行了实现，在实验中大家可以选择使用，也可以自定义符号表。



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。

分析树与抽象语法树

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，内部戒掉代表程序构造，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。

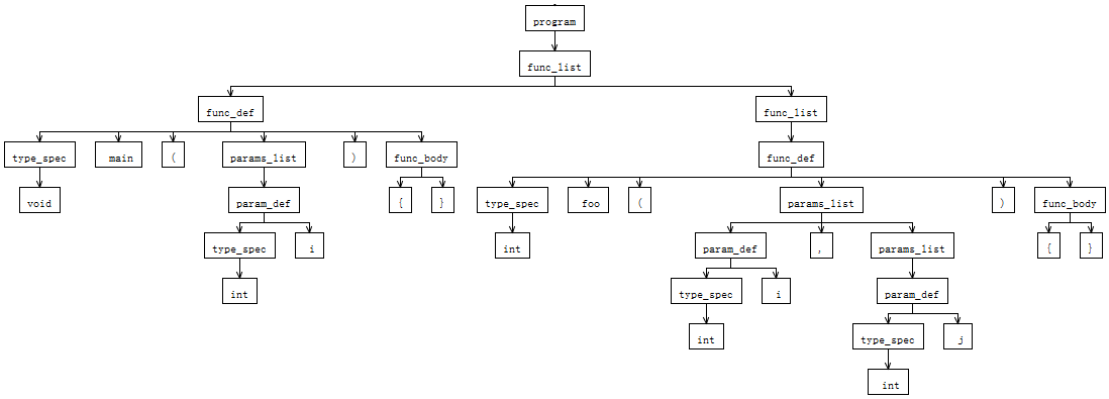
例如对下面忽略函数内部语句的文法：

```
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' | '_';
```

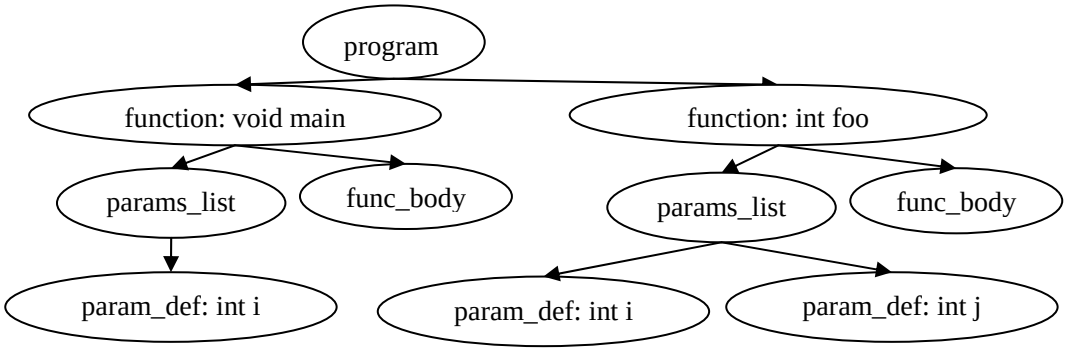
注意：上述文法是按照 ANTLRWorks 的规则编写的，其中非终结符用小写字母，token 用大写字母，在文法中直接出现的终结符号需要用引号括起来。上述文法可以直接输入 ANTLRWorks 进行验证。当输入为如下的程序时：

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

会得到如下的分析树：



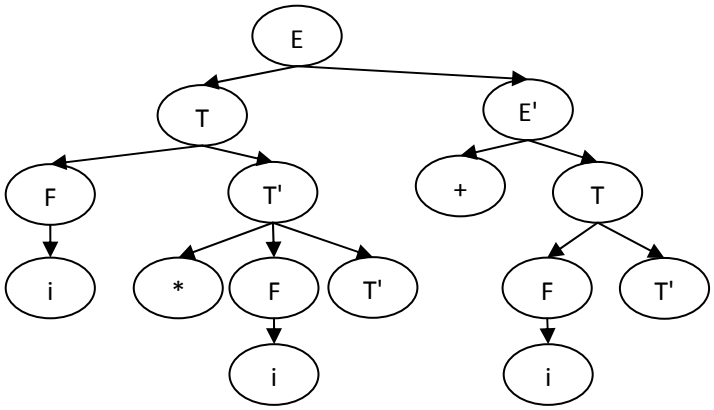
将其转换对应的抽象语法树之后的结果为：



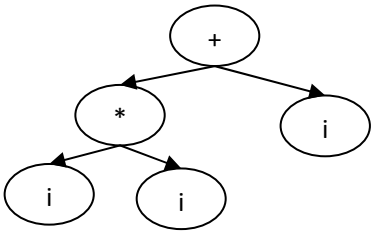
又如下面的文法：

```
E  → T E'
E' → + T E' | - T E' | ε
T  → F T'
T' → * F T' | / F T' | ε
F  → i | ( E )
```

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。

3.3.4. 代码生成

代码生成的任务要求

该实验选择 C 语言的一个子集，基于 BIT-MiniCC 构建 C 语法子集的代码生成模块，该语法分析器能够读入 XML 文件形式的语法分析树，进行寄存器分配，并生成 MIPS 或者 X86 汇编代码。

需要说明的是，生成汇编程序是该部分的基本要求；进一步地，如果生成的是 MIPS 汇编，则要求汇编代码能够在 BIT-MiniCC 集成的 MIPS 模拟器中运行（注意 config.xml 的最后一个阶段“simulating”的“skip”属性配置为“false”）；如果生成的是 X86 汇编，则要求使用 X86 汇编器生成 exe 文件并运行。

MIPS 处理器简介

https://en.wikipedia.org/wiki/MIPS_architecture

https://en.wikipedia.org/wiki/MIPS_architecture_processors

MIPS 是一个精简指令集计算机体系架构，最早期是 32 位，后来增加了 64 位。MIPS 的发展经历了 MIPS I、MIPS II、MIPS III、MIPS IV 和 MIPS V 等几个阶段。MIPS 处理器广泛使用在嵌入式系统中，例如门禁系统和路由器等。最初 MIPS 是面向桌面系统的，后来逐步扩展到工作站和服务器上。

MIPS 只使用 load 和 store 指令访问存储器，其他指令对寄存器进行操作。MIPS I 有 32 个 32 位通用寄存器。寄存器$0 硬编码为 0，只能读不能写；寄存器$31 位链接寄存器，用于存储函数调用时的返回地址。对于整数乘法和除法，则需要使用一对 32 位寄存器：HI 和 LO，MIPS 提供了特殊的指令用于实现在 HI/LO 和通用寄存器之间传输数据。如下为所有 32 个寄存器的编号、别名和用途。

Name	Register	Usage	
$zero	$0	Always 0	(forced by hardware)
$at	$1	Reserved for assembler use	
$v0 – $v1	$2 – $3	Result values of a function	
$a0 – $a3	$4 – $7	Arguments of a function	
$t0 – $t7	$8 – $15	Temporary Values	
$s0 – $s7	$16 – $23	Saved registers	(preserved across call)
$t8 – $t9	$24 – $25	More temporaries	
$k0 – $k1	$26 – $27	Reserved for OS kernel	
$gp	$28	Global pointer	(points to global data)
$sp	$29	Stack pointer	(points to top of stack)
$fp	$30	Frame pointer	(points to stack frame)
$ra	$31	Return address	(used by jal for function call)

如图所示，$at 保留给汇编器使用；$v0-$v1 为存储返回值的寄存器，函数返回值为 32 位时，只需要使用$v0 即可；$a0-$a3 用于函数调用时的参数传递，即当函数实参的前 4 个可以放在这 4 个寄存器里面。$t0-$t9 为临时变量寄存器，当发生函数调用时，如果这些寄存器里面有有用的值，则调用者需要自己保存这些值，被调用者无需关心保存的问题，可以直接使用；相反，$s0-$s7 寄存器里面的值则需要被调用者首先保存了之后才能使用，在使用完之后退出函数之前恢复；$k0-$k1 为操作系统保留使用；$sp 和$fp 分别指向函数栈帧的结束和起始地址。

在 BIT-MINICC 中，需要使用的寄存器包括$0, $v0, $a0-$a3, $t0-$t9, $s0-$s7, $sp, $fp 和$ra。其中寄存器分配阶段主要考虑的是$a0-$a3, $t0-$t9, $s0-$s7 这些寄存器的使用。

MIPS 指令有 3 种格式，分别称为 R 型、I 型和 J 型。具体的编码格式如下图所示。

Type	-31-			format (bits)		-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

其中 R 型为寄存器相关操作，I 型为立即数相关操作，J 型为跳转相关操作。指令查询的快速手册（MIPS32® Instruction Set Quick Reference，https://www.lri.fr/~de/MIPS.pdf）如下所示：

MIPS32® Instruction Set Quick Reference

RD — DESTINATION REGISTER
RS, RT — SOURCE OPERAND REGISTERS
RA — RETURN ADDRESS REGISTER (R31)
PC — PROGRAM COUNTER
ACC — 64-BIT ACCUMULATOR
LO, HI — ACCUMULATOR LOW ($ACC_{31:0}$) AND HIGH ($ACC_{63:32}$) PARTS
± — SIGNED OPERAND OR SIGN EXTENSION
∅ — UNSIGNED OPERAND OR ZERO EXTENSION
:: — CONCATENATION OF BIT FIELDS
R2 — MIPS32 RELEASE 2 INSTRUCTION
DOTTED — ASSEMBLER PSEUDO-INSTRUCTION

PLEASE REFER TO *"MIPS32 ARCHITECTURE FOR PROGRAMMERS VOLUME II: THE MIPS32 INSTRUCTION SET"* FOR COMPLETE INSTRUCTION SET INFORMATION.

Arithmetic Operations		
ADD	R_D, R_S, R_T	$R_D = R_S + R_T$ (overflow trap)
ADDI	R_D, R_S, const16	$R_D = R_S + const16^{\pm}$ (overflow trap)
ADDIU	R_D, R_S, const16	$R_D = R_S + const16^{\pm}$
ADDU	R_D, R_S, R_T	$R_D = R_S + R_T$
CLO	R_D, R_S	R_D = CountLeadingOnes(R_S)
CLZ	R_D, R_S	R_D = CountLeadingZeros(R_S)
LA	R_D, label	R_D = Address(label)
LI	R_D, imm32	R_D = imm32
LUI	R_D, const16	$R_D = const16 << 16$
MOVE	R_D, R_S	$R_D = R_S$
NEGU	R_D, R_S	$R_D = -R_S$
SEB[R2]	R_D, R_S	$R_D = R_{S_{7:0}}^{\pm}$
SEH[R2]	R_D, R_S	$R_D = R_{S_{15:0}}^{\pm}$
SUB	R_D, R_S, R_T	$R_D = R_S - R_T$ (overflow trap)
SUBU	R_D, R_S, R_T	$R_D = R_S - R_T$

Shift and Rotate Operations		
ROTR[R2]	R_D, R_S, bits5	$R_D = R_{S_{bits5-1:0}} :: R_{S_{31:bits5}}$
ROTRV[R2]	R_D, R_S, R_T	$R_D = R_{S_{RT4:0-1:0}} :: R_{S_{31:RT4:0}}$
SLL	R_D, R_S, shift5	$R_D = R_S << shift5$
SLLV	R_D, R_S, R_T	$R_D = R_S << R_{T_{4:0}}$
SRA	R_D, R_S, shift5	$R_D = R_S^{\pm} >> shift5$
SRAV	R_D, R_S, R_T	$R_D = R_S^{\pm} >> R_{T4:0}$
SRL	R_D, R_S, shift5	$R_D = R_S^{\varnothing} >> shift5$
SRLV	R_D, R_S, R_T	$R_D = R_S^{\varnothing} >> R_{T4:0}$

Logical and Bit-Field Operations		
AND	R_D, R_S, R_T	$R_D = R_S \,\&\, R_T$
ANDI	R_D, R_S, const16	$R_D = R_S \,\&\, \text{const16}^{\varnothing}$
EXT[R2]	R_D, R_S, P, S	$R_S = R_{S_{P+S-1:P}}{}^{\varnothing}$
INS[R2]	R_D, R_S, P, S	$R_{D_{P+S-1:P}} = R_{S_{S-1:0}}$
NOP		No-op
NOR	R_D, R_S, R_T	$R_D = \sim(R_S \mid R_T)$
NOT	R_D, R_S	$R_D = \sim R_S$
OR	R_D, R_S, R_T	$R_D = R_S \mid R_T$
ORI	R_D, R_S, const16	$R_D = R_S \mid \text{const16}^{\varnothing}$
WSBH[R2]	R_D, R_S	$R_D = R_{S_{23:16}} :: R_{S_{31:24}} :: R_{S_{7:0}} :: R_{S_{15:8}}$
XOR	R_D, R_S, R_T	$R_D = R_S \oplus R_T$
XORI	R_D, R_S, const16	$R_D = R_S \oplus \text{const16}^{\varnothing}$

Condition Testing and Conditional Move Operations		
MOVN	R_D, R_S, R_T	if $R_T \neq 0$, $R_D = R_S$
MOVZ	R_D, R_S, R_T	if $R_T = 0$, $R_D = R_S$
SLT	R_D, R_S, R_T	$R_D = (R_S{}^{\pm} < R_T{}^{\pm}) \;?\; 1 : 0$
SLTI	R_D, R_S, const16	$R_D = (R_S{}^{\pm} < \text{const16}^{\pm}) \;?\; 1 : 0$
SLTIU	R_D, R_S, const16	$R_D = (R_S{}^{\varnothing} < \text{const16}^{\varnothing}) \;?\; 1 : 0$
SLTU	R_D, R_S, R_T	$R_D = (R_S{}^{\varnothing} < R_T{}^{\varnothing}) \;?\; 1 : 0$

Multiply and Divide Operations		
DIV	R_S, R_T	$L_O = R_S{}^{\pm} / R_T{}^{\pm}$; $H_I = R_S{}^{\pm} \bmod R_T{}^{\pm}$
DIVU	R_S, R_T	$L_O = R_S{}^{\varnothing} / R_T{}^{\varnothing}$; $H_I = R_S{}^{\varnothing} \bmod R_T{}^{\varnothing}$
MADD	R_S, R_T	$Acc \mathrel{+}= R_S{}^{\pm} \times R_T{}^{\pm}$
MADDU	R_S, R_T	$Acc \mathrel{+}= R_S{}^{\varnothing} \times R_T{}^{\varnothing}$
MSUB	R_S, R_T	$Acc \mathrel{-}= R_S{}^{\pm} \times R_T{}^{\pm}$
MSUBU	R_S, R_T	$Acc \mathrel{-}= R_S{}^{\varnothing} \times R_T{}^{\varnothing}$
MUL	R_D, R_S, R_T	$R_D = R_S{}^{\pm} \times R_T{}^{\pm}$
MULT	R_S, R_T	$Acc = R_S{}^{\pm} \times R_T{}^{\pm}$
MULTU	R_S, R_T	$Acc = R_S{}^{\varnothing} \times R_T{}^{\varnothing}$

Accumulator Access Operations		
MFHI	R_D	$R_D = H_I$
MFLO	R_D	$R_D = L_O$
MTHI	R_S	$H_I = R_S$
MTLO	R_S	$L_O = R_S$

Jumps And Branches (Note: One Delay Slot)		
B	off18	PC += off18$^{\pm}$
BAL	off18	R_A = PC + 8, PC += off18$^{\pm}$
BEQ	Rs, Rt, off18	if Rs = Rt, PC += off18$^{\pm}$
BEQZ	Rs, off18	if Rs = 0, PC += off18$^{\pm}$
BGEZ	Rs, off18	if Rs ≥ 0, PC += off18$^{\pm}$
BGEZAL	Rs, off18	R_A = PC + 8; if Rs ≥ 0, PC += off18$^{\pm}$
BGTZ	Rs, off18	if Rs > 0, PC += off18$^{\pm}$
BLEZ	Rs, off18	if Rs ≤ 0, PC += off18$^{\pm}$
BLTZ	Rs, off18	if Rs < 0, PC += off18$^{\pm}$
BLTZAL	Rs, off18	R_A = PC + 8; if Rs < 0, PC += off18$^{\pm}$
BNE	Rs, Rt, off18	if Rs ≠ Rt, PC += off18$^{\pm}$
BNEZ	Rs, off18	if Rs ≠ 0, PC += off18$^{\pm}$
J	addr28	PC = $PC_{31:28}$ :: addr28$^{\varnothing}$
JAL	addr28	R_A = PC + 8; PC = $PC_{31:28}$ :: addr28$^{\varnothing}$
JALR	Rd, Rs	Rd = PC + 8; PC = Rs
JR	Rs	PC = Rs

Load and Store Operations		
LB	Rd, off16(Rs)	Rd = mem8(Rs + off16$^{\pm}$)$^{\pm}$
LBU	Rd, off16(Rs)	Rd = mem8(Rs + off16$^{\pm}$)$^{\varnothing}$
LH	Rd, off16(Rs)	Rd = mem16(Rs + off16$^{\pm}$)$^{\pm}$
LHU	Rd, off16(Rs)	Rd = mem16(Rs + off16$^{\pm}$)$^{\varnothing}$
LW	Rd, off16(Rs)	Rd = mem32(Rs + off16$^{\pm}$)
LWL	Rd, off16(Rs)	Rd = LoadWordLeft(Rs + off16$^{\pm}$)
LWR	Rd, off16(Rs)	Rd = LoadWordRight(Rs + off16$^{\pm}$)
SB	Rs, off16(Rt)	mem8(Rt + off16$^{\pm}$) = $Rs_{7:0}$
SH	Rs, off16(Rt)	mem16(Rt + off16$^{\pm}$) = $Rs_{15:0}$
SW	Rs, off16(Rt)	mem32(Rt + off16$^{\pm}$) = Rs
SWL	Rs, off16(Rt)	StoreWordLeft(Rt + off16$^{\pm}$, Rs)
SWR	Rs, off16(Rt)	StoreWordRight(Rt + off16$^{\pm}$, Rs)
ULW	Rd, off16(Rs)	Rd = unaligned_mem32(Rs + off16$^{\pm}$)
USW	Rs, off16(Rt)	unaligned_mem32(Rt + off16$^{\pm}$) = Rs

Atomic Read-Modify-Write Operations		
LL	Rd, off16(Rs)	Rd = mem32(Rs + off16$^{\pm}$); link
SC	Rd, off16(Rs)	if Atomic, mem32(Rs + off16$^{\pm}$) = Rd; Rd = Atomic ? 1 : 0

Registers		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

Default C Calling Convention (O32)

Stack Management

- The stack grows down.
  - Subtract from $sp to allocate local storage space.
  - Restore $sp by adding the same amount at function exit.
- The stack must be 8-byte aligned.
  - Modify $sp only in multiples of eight.

Function Parameters

- Every parameter smaller than 32 bits is promoted to 32 bits.
- First four parameters are passed in registers $a0–$a3.
  - 64-bit parameters are passed in register pairs:
    - Little-endian mode: $a1:$a0 or $a3:$a2.
    - Big-endian mode: $a0:$a1 or $a2:$a3.
- Every subsequent parameter is passed through the stack.
  - First 16 bytes on the stack are not used.
  - Assuming $sp was not modified at function entry:
    - The 1st stack parameter is located at 16($sp).
    - The 2nd stack parameter is located at 20($sp), etc.
  - 64-bit parameters are 8-byte aligned.

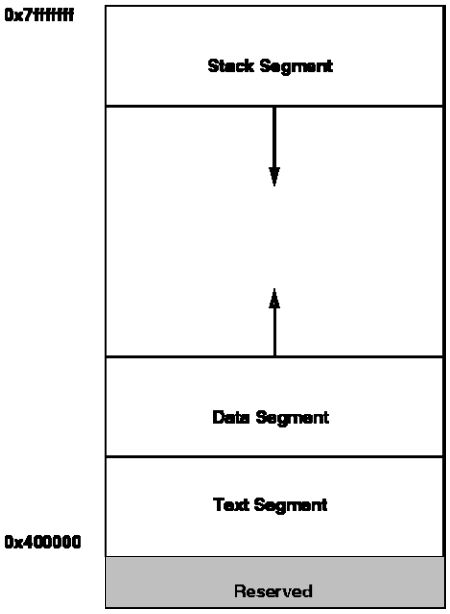
Return Values

- 32-bit and smaller values are returned in register $v0.
- 64-bit values are returned in registers $v0 and $v1:
  - Little-endian mode: $v1:$v0.
  - Big-endian mode: $v0:$v1.

如上图所示为 MIPS 处理器上 C 函数调用的一些约定，主要包括堆栈是从高地址向低地址生长的，$sp 始终指向栈顶。进入函数时，$sp 减掉一定的值为函数分配空间，退出函数后则需要增加同样的值以让$sp 指向调用函数的栈顶。函数栈帧是 8 字节对齐的。另外函数的参数传递也有特殊的要求，前四个参数是通过$a0-$a3 传递的，其他则通过堆栈传递。但是堆栈里面仍然为通过寄存器传递的参数预留了存储空间，因此第一个通过堆栈传递的参数位于 16($sp)，第二个位于 20($sp)，以此类推。返回值为 32 位时放在寄存器$v0 里面，64 位则放在$v0 和$v1 里面。

MIPS32 Virtual Address Space				
kseg3	0xE000.0000	0xFFFF.FFFF	Mapped	Cached
ksseg	0xC000.0000	0xDFFF.FFFF	Mapped	Cached
kseg1	0xA000.0000	0xBFFF.FFFF	Unmapped	Uncached
kseg0	0x8000.0000	0x9FFF.FFFF	Unmapped	Cached
useg	0x0000.0000	0x7FFF.FFFF	Mapped	Cached

以上为 32 位 MIPS 的存储空间分配，其中用户空间为 0x00000000-0x7fffffff。具体的划分如下图所示，因此堆栈栈底地址为 0x7fffffff。



C 语言程序生成的 MIPS 汇编举例

例 1：

READING THE CYCLE COUNT REGISTER FROM C

```
unsigned mips_cycle_counter_read()
{
     unsigned cc;
     asm volatile("mfc0 %0, $9" : "=r" (cc));
     return (cc << 1);
}
```

ASSEMBLY-LANGUAGE FUNCTION EXAMPLE

```
# int asm_max(int a, int b)
# {
#   int r = (a < b) ? b : a;
#   return r;
# }

     .text
     .set     nomacro
     .set     noreorder

     .global  asm_max
     .ent     asm_max

asm_max:
     move     $v0, $a0          # r = a
     slt      $t0, $a0, $a1     # a < b ?
     jr       $ra               # return
     movn     $v0, $a1, $t0     # if yes, r = b

     .end     asm_max
```

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>
```

3.4.2. 语法分析

在 BIT-MiniCC 框架下，可以按照如下步骤完成语法分析实验：

（1）参照 3.3.2 节给出的文法，扩充定义自己希望实现的 C 语言语法子集。参考文法只给出了函数定义以及简单的表达式相关的文法。局部变量声明、分支语句以及循环语句等需要自己进行扩充。主要采用自顶向下的分析方法时，不能有左递归，避免文法产生式的多个候选式存在公共因子。如果出现左递归或者公共因子，则可以通过文法等价变换进行消除。

（2）从递归下降分析方法、LL(1)分析方法、算符优先分析方法、LR 分析方法中选择一种，作为 BIT-MiniCC 框架中设计并实现语法分析器的指导方法。

（3）构建语法分析器，BIT-MiniCC 中已经定义了一个类 CMyMiniCCParser，并定义了 run 方法，实验以该类为主进行。语法分析的输入为词法分析的输出，因此语法分析器首先要读入 xxx.token.xml 文件；在分析的过程中构建语法树；

（4）将语法树输出为 xml 文件；

目前已有的分析方法包括递归下降、LL(1)和 LR 等多种分析方法，可以选择其中的一种实现，递归下降更为直观。

例如，基于框架自带的文法及其对应的实现，当输入为如下的程序时：

```
int main(){
    int a=1;
    a = a * 2;
    return a;
}
```

得到的语法树如下所示：

对应的输出的 xml 的部分内容为：

```
<?xml version="1.0" encoding="UTF-8"?>
<ParserTree name="test.tree.xml">
  <compilationUnit>
    <translationUnit>
      <externalDeclaration>
        <functionDefinition>
          <declarationSpecifiers>
            <declarationSpecifier>
              <typeSpecifier>int</typeSpecifier>
            </declarationSpecifier>
          </declarationSpecifiers>
          <declarator>
            <directDeclarator>
              <directDeclarator>main</directDeclarator>
            <punctuation>(</punctuation>
            <punctuation>)</punctuation>
            </directDeclarator>
          </declarator>
          <compoundStatement>
          <punctuation>{</punctuation>
            <blockItemList>
              <blockItemList>
                <blockItemList>
                  <blockItem>
                    <declaration>
                      <declarationSpecifiers>
                        <declarationSpecifier>
                          <typeSpecifier>int</typeSpecifier>
                        </declarationSpecifier>
                      </declarationSpecifiers>
                      <initDeclaratorList>
                        <initDeclarator>
                          <declarator>
                            <directDeclarator>a</directDeclarator>
                          </declarator>
                        <separator>=</separator>
                          <initializer>
                            <assignmentExpression>
```

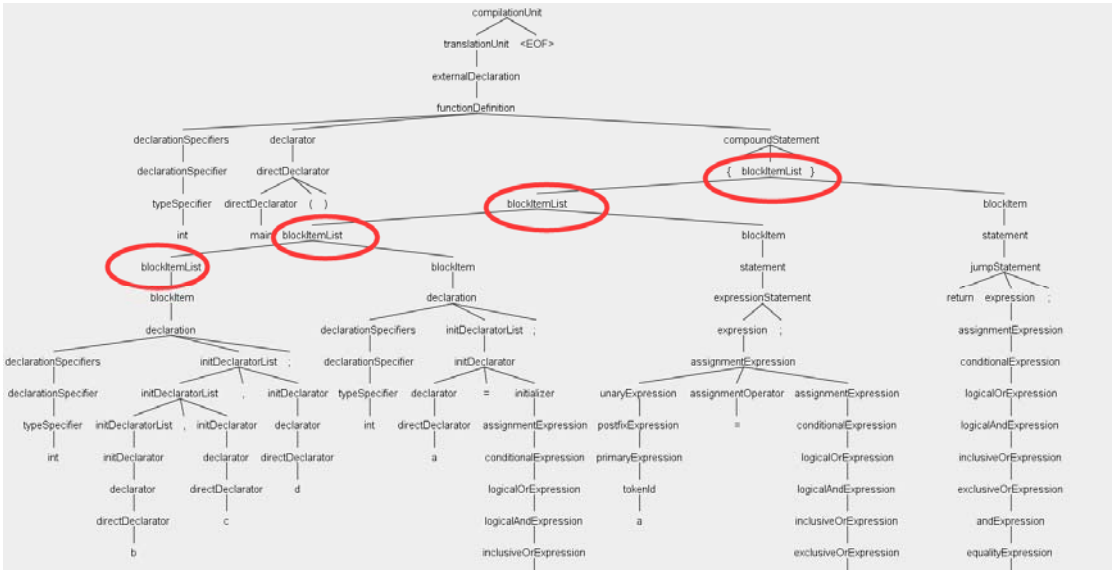
3.4.3. 语义分析

不做要求

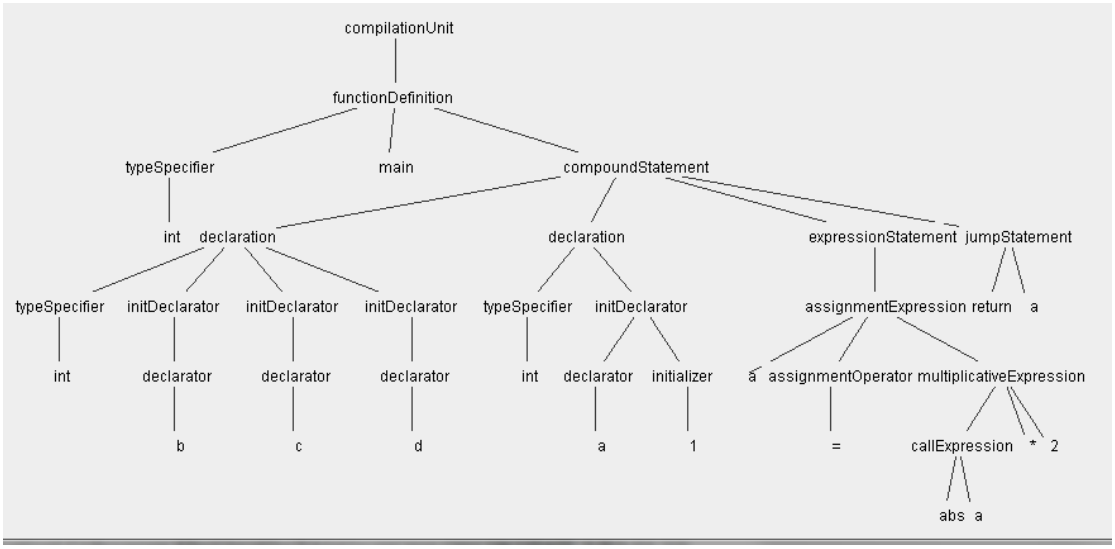
3.4.4. 代码生成

在 BIT-MiniCC 框架下，可以按照如下步骤完成代码生成实验：

（1）在语法分析实验的基础之上，对获得的语法树进行变换，得到抽象语法树（见 3.3.3），变换的过程主要是将左递归导致的左斜树转换为平衡结构。如下图所示为一输出的分析树：



转换后的抽象语法树为：



（2）为每个标识符以及中间临时变量分配寄存器。当变量和临时变量较少时，可以直接用为每个标识符以及中间临时变量分配唯一的标识符。

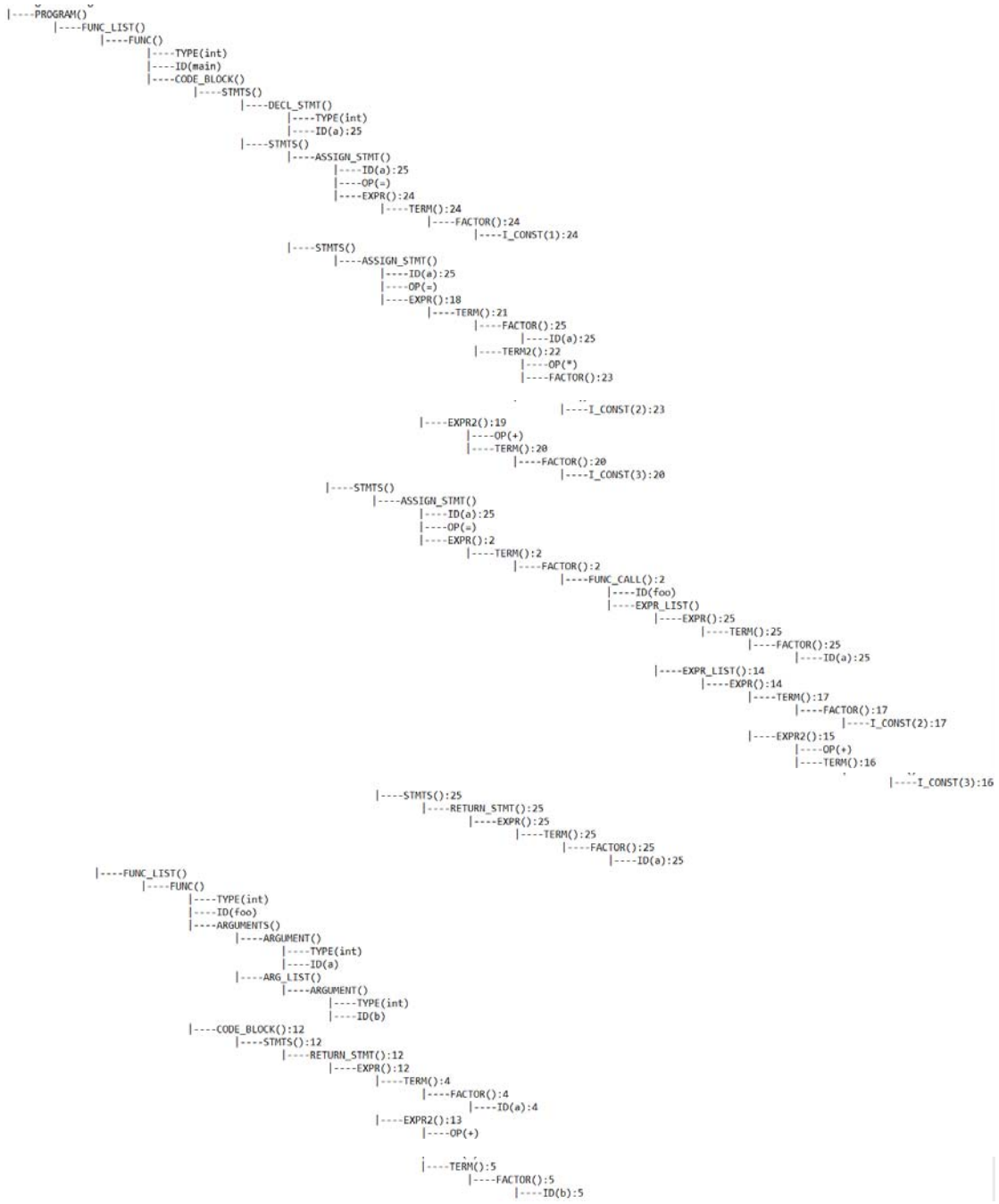
（3）代码生成。根据语法树以及寄存器分配结果，生成相应的代码。

以如下的 C 语言代码为例：

```
int main(){
    int a;
    a = 1;
    a = a * 2 + 3;
    a = foo(a, 2+3);
    return a;
}

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

采用递归下降分析法，获得的语法树及其寄存器分配结果为：



对应的生成的 MIPS 汇编代码为：

```
        .data

        .text

__init:
        # setup the base address of stack
        lui  $sp, 0x8000
        addi $sp, $sp, 0x0000

        # allocate stack frame for main function
        addiu $sp, $sp, -64
        jal __main
        # make a system call and terminate the program
        li $v0, 10
        syscall
        nop

__main:
        li $t8, 1
        move $t9, $t8
        li $s7, 2
        mul $s6, $t9, $s7
        li $s4, 3
        add $s3, $s6, $s4
        move $s2, $s3
        move $t9, $s2
        move $a0, $t9
        li $s1, 2
        li $s0, 3
        add $t7, $s1, $s0
        move $t6, $t7
        move $a1, $t6

        # allocate stack frame for the callee
        addiu $sp, $sp, -64
        sw $ra, 0($sp)

        jal __foo

        # restore the $ra and deallocate the stack frame for the callee
        lw $ra, 0($sp)
        addiu $sp, $sp, 64
        move $t9, $v0
        move $v0, $t9
        jr $ra


__foo:
        add $t5, $a0, $a1
        move $t4, $t5
        move $v0, $t4
        jr $ra
```

3.4.5. 模拟运行

略

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：

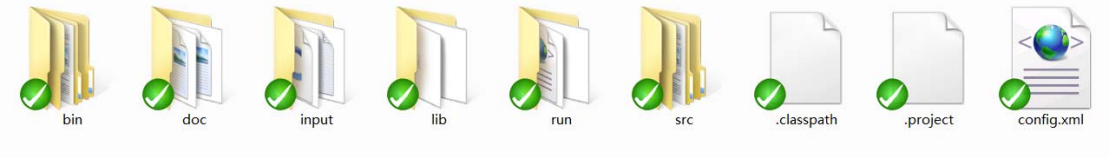
实验报告放置在 doc 目录下，应包括如下内容：

- 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

3.5.2. 语法分析

本实验要求提交语法分析器实现源码，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

3.5.3. 语义分析

无

3.5.4. 代码生成

本实验要求提交代码生成模块实现源码，也可以是整个编译器的源码，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

3.5.5. 模拟运行

无

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。