基础课程实践项目

C1 语言编译器的构造

张昱　　2015 年秋

目录

P1 预备阶段

1、实验环境熟悉

安装 Linux, LLVM，使用 GCC/Clang 来编译 C 程序，并运行编译生成的可执行程序。初步熟悉 Makefile，了解编译器的过程。

2、熟悉 C1 语言的语言特征

C1 语言的特征参见附 2。用 C1 语言编写 3 个以上的程序，其中至少有解决数组排序问题的程序，并使用 GCC/Clang 编译。

3、编译过程的初步了解

阅读教材 11.1 节和 LLVM 的资料，了解预处理、编译、连接的含义和命令，并使用。

4、提交的目录结构要求

各次课程设计将按如下目录结构提交到学生自己的 git 库的 master 分支(缺省的分支就是 master):

P1

|----README	提交内容说明，包含如何编译运行所提交的内容等
|----src/	存放源程序文件（*.h, *.c）和 Makefile
|----doc/	实验设计文档
|----bin /	帮助快速编译和运行实验内容的 shell 脚本文件
|----test/	存放测试程序

P2 词法分析

1、理解 PL/0 编译器的词法分析过程

PL/0 编译器的词法分析由 src/pl0.c 中的 getsym( )函数来负责完成。

2、扩展 PL/0 的词法

扩展 PL/0 编译器以支持：

1）由/*开始后跟 0 个或多个字符、再以*/结尾的多行注释；

2）以 0 开头后跟 0~7 这 8 种数字组成的八进制数；

3）以 0x 或 0X 开头后跟 0~9、A~F、a~f 组成的十六进制数；

4）参照 odd，增加对取模运算符(%和 mod 两种表示都需支持)的单词识别，"%"和"mod"识别成符号常量 mod（需要在 pl0.h 中给出定义）；

取模运算符的使用格式如下

a mod 3　　或者　a % 3

5）增加对数组的支持，数组变量的声明和使用如下：

```
const m=3;
var a[m], b[4];
b[3] = 4;
```

其中对前 3 点的支持只需要修改词法分析部分（即 getsym( )的实现）即可；而对第 4、5 点的支持则还涉及对后续的语法分析、P-code 代码生成以及解释器的修改。在本次实验中，针对第 4、5 点，先只完成对取模运算符和左、右方括号的单词识别，识别出的符号用符号

常量 mod、lbracket、rbracket。

3、词法的形式描述

给出扩展后的 PL/0 语言的词法规范（即用正规式描述语言中合法的单词）。

4、学习使用词法分析器的生成工具 Flex

阅读 Flex manual (http://flex.sourceforge.net/manual)和一个简单的表达式语言的词法分析例子(http://staff.ustc.edu.cn/~yuzhang/compiler/proj/flex-examples.zip)，了解 Flex 的输入词法规范文件的格式，以及 Flex 生成的词法分析器的接口形式和实现。你可以从 flex-2.5.35.tar.gz (http://prdownloads.sourceforge.net/flex/flex-2.5.35.tar.gz?download) 的 examples 目录下获得更多 使用 Flex 的例子。

5、用 Flex 生成扩展后的 PL/0 的词法分析器

用 Flex 生成 PL/0 的词法分析器 getsym( )，修改 PL/0 编译器源代码 src/pl0.c，使得 PL/0 编译器能调用 Flex 生成的词法分析器来进行词法分析。

注意：

1) 要求用宏 LEXERGEN 和条件编译来控制词法分析器的选择，即编译器是调用原有 pl0.c 中的 getsym( )还是调用 Flex 生成的词法分析器；

2）能快速编译和运行使用 1）中任意一种方式（原有 pl0.c 中的 getsym( )或 Flex 生成的词法分析器）构造的 PL/0 编译器。

提示：(1) 在 pl0.c 中做如下改动：

#ifndef LEXERGEN void getch( ){...} void getsym( ) {...} #endif	在 main()中，进行分析之前增加： #ifdef LEXERGEN extern FILE * yyin=infile; #endif

(2) pl0.lex

```
...
void getsym() {
    sym = yylex();
}
```

(3) Makefile

```
# This Makefile requires GNU make.
SHELL = /bin/sh
CC = gcc
PROGRAM = pl0
OBJDIR = ../bin
BINDIR = ../bin
SRCDIR = src
OBJS = ${OBJDIR}/pl0.o
CFLAGS = -m32 -I../include
CCOMPILE = ${CC} ${CFLAGS} -c
LEX      = flex -i -I
```

```
default:: ${PROGRAM}

pl0: ${OBJS}
	@mkdir -p ${BINDIR}
	${CC} ${CFLAGS} -o ${BINDIR}/${PROGRAM} ${OBJS}

${OBJDIR}/%.o: %.c
	@mkdir -p ${OBJDIR}
	${CCOMPILE} -o $@ $<

pl0lex:	pl0.lex
	@mkdir -p ${BINDIR}
	$(LEX)	pl0.lex
	$(CC) -I../include -DLEXERGEN lex.yy.c pl0.c -o ${BINDIR}/${PROGRAM} -ll

clean:
	-rm -f ${OBJDIR}/*.o lex.yy.c
```

6、用扩展的 PL/0 语言编写如下程序，并用扩展的 PL/0 编译器进行词法分析，要求能输出识别出的一个个单词：

1) 对给定的一组整数进行排序

2) 判断一个数是否是素数

7、P2 提交的目录结构要求

按如下目录结构提交到学生自己的 git 库的 master 分支(缺省的分支就是 master):

```
P2
  |----README        提交内容说明，包含如何编译运行所提交的内容等
  |----src/          存放源程序文件（*.h, *.c）和 Makefile
  |----doc/          实验设计文档
  |----bin/          帮助快速编译和运行实验内容的 shell 脚本文件
  |----test/         存放测试程序
```

附 1：参考资料

- PL/0 源程序：http://staff.ustc.edu.cn/~yuzhang/compiler/proj/pl0.zip
  《编译原理实践教程》：http://staff.ustc.edu.cn/~yuzhang/compiler/old_pl0project.pdf
- 词法分析器的生成工具 Lex 的变种 Flex：
  Flex manual：http://flex.sourceforge.net/manual
  flex 源码：http://flex.sourceforge.net/
- LALR 分析器的生成工具 Yacc 的变种 Bison：
  Bison manual：http://www.gnu.org/software/bison/manual/
  Bison 源码：http://ftp.gnu.org/gnu/bison/
- 运用 Bison 构造编译器的示例代码包 bison-examples
  http://staff.ustc.edu.cn/~yuzhang/compiler/proj/bison-examples.tar.gz
- ANSI-C 语言的 lex 词法规范和 Yacc 文法规范
  http://www.quut.com/c/ANSI-C-grammar-l.html
  http://www.quut.com/c/ANSI-C-grammar-y.html
- 一个可变目标的 C 编译器 LCC(ftp://ftp.cs.princeton.edu/pub/packages/lcc/)：它采用递归下降的语法分析。《A Retargetable C Compiler: Design and Implementation（可变目标 C 编译器——设计与实现）》是对 LCC 的设计与实现的详细解释。
- LLVM（Low Level Virtual Machine）：http://llvm.org/
  Clang：http://clang.llvm.org/

附 2：C1 语言的特征

C1 语言是一个类 C 的小型实验语言。在数据类型上，只支持整数类型和一维整型数组类型。一个 C1 语言程序文件包含若干常量定义、全局变量声明、无参函数定义，其中必须包含一个名为 main 的函数，作为程序执行的主函数。函数体中可以有常量定义、变量声明和语句。C1 语言有赋值语句、函数调用语句、复合语句、条件语句、循环语句和空语句。由于上面这些语言概念已为大家熟知，因此不再进行语义解释。下面用习题3.7 所介绍的扩展方式来给C1语言的文法。

CompUnit → [CompUnit] (Decl | FuncDef)

CompUnit代表一个C1程序文件，其中必须包含一个名为main的函数定义FuncDef

这里假设一个C1程序只存放在一个C1程序文件中

Decl → ConstDecl | VarDecl

ConstDecl → **const int** ConstDef {, ConstDef} ';'

ConstDef → **ident** '=' **number** | **ident** '[' [**number**] ']' '=' '{' **number** { ',' **number** } '}'

ConstDecl常量声明：可以声明一个或多个整型常量、整型常量数组，如：

const int a[]={2,3};

声明了长度为2的常量数组a，a[0]、a[1]的值分别为2和3；数组在声明时若未指明长度，则长度为初始化表达式中值的个数。在程序中，不能对常量重新赋值。

VarDecl → **int** Var{, Var}';'

Var → **ident** | **ident** '[' **number** ']'

VarDecl 变量声明：可以声明整型和一维整型数组变量，

FuncDef → **void ident** '(' ')' Block

FuncDef 函数定义：无参函数，且必须有一个名字为main 的函数

Block →'{' { BlockItem } '}'

BlockItem → Decl | Stmt

Block语句块：表示函数体或者复合语句，其中可以包含0个或多个声明或语句

Stmt →LVal '=' Exp ';' | **ident** '(' ')' ';' | Block | **if** Cond Stmt [**else** Stmt]

| **while** Cond Stmt | ';'

LVal → **ident**| **ident** '[' Exp ']'

Cond → **odd** Exp | Exp RelOp Exp

RelOp →"==" | "!=" | '<' | '>' | "<=" | ">="

Exp → Exp BinOp Exp | UnaryOp Exp | '(' Exp ')' | LVal | **number**

BinOp → '+' | '−' | '*' | '/' | '%'

UnaryOp → '+' | '−'

运算符的优先级和结合性与C语言中的一样

附 2：Bison 生成的分析器源码中使用的一些表及其含义

以下内容来自 Bison-2.4.1 源码中 src/tables.h 里的注释：

- YYTRANSLATE = vector mapping yylex's token numbers into bison's token numbers.
- YYTNAME = vector of string-names indexed by bison token number.
- YYTOKNUM = vector of yylex token numbers corresponding to entries in YYTNAME.
- YYRLINE = vector of line-numbers of all rules. For yydebug printouts.
- YYRHS = vector of items of all rules. This is exactly what RITEMS contains. For yydebug and for semantic parser.
- YYPRHS[R] = index in YYRHS of first item for rule R.
- YYR1[R] = symbol number of symbol that rule R derives.
- YYR2[R] = number of symbols composing right hand side of rule R.
- YYSTOS[S] = the symbol number of the symbol that leads to state S.
- YYDEFACT[S] = default rule to reduce with in state s, when YYTABLE doesn't specify something else to do. Zero means the default is an error.
- YYDEFGOTO[I] = default state to go to after a reduction of a rule that generates variable NTOKENS + I, except when YYTABLE specifies something else to do.
- YYPACT[S] = index in YYTABLE of the portion describing state S. The lookahead token's type is used to index that portion to find out what to do. If the value in YYTABLE is positive, we shift the token and go to that state. If the value is negative, it is minus a rule number to reduce by. If the value is zero, the default action from YYDEFACT[S] is used.
- YYPGOTO[I] = the index in YYTABLE of the portion describing what to do after reducing a rule that derives variable I + NTOKENS. This portion is indexed by the parser state number, S, as of before the text for this nonterminal was read. The value from YYTABLE is the state to go to if the corresponding value in YYCHECK is S.
- YYTABLE = a vector filled with portions for different uses, found via YYPACT and YYPGOTO.
- YYCHECK = a vector indexed in parallel with YYTABLE. It indicates, in a roundabout way, the bounds of the portion you are trying to examine. Suppose that the portion of YYTABLE starts at index P and the index to be examined within the portion is I. Then if YYCHECK[P+I] != I, I is outside the bounds of what is actually allocated, and the default (from YYDEFACT or YYDEFGOTO) should be used. Otherwise, YYTABLE[P+I] should be used.
- YYFINAL = the state number of the termination state.
- YYLAST (= high) the number of the last element of YYTABLE, i.e., sizeof (YYTABLE) - 1.

附 3：bison-examples 的说明

本压缩包围绕 2 个小型编程语言：L-expr（简单的表达式语言）、L-asgn（赋值语句序列语言），给出如何利用 Flex 和 Bison 构造编译器，其中涉及程序位置跟踪、符号表的管理、抽象语法树的构造、错误信息管理等。

1、L-expr 语言的文法

```
input    ::= ε | input line
line     ::=  EOL | expr EOL
expr     ::=  NUMBER
              | expr PLUS expr          # PLUS  – '+'加号
              | expr MINUS expr         # MINUS – '-'减号
              | expr MULT expr          # MULT  – '*'乘号
              | expr DIV expr           # DIV   – '/' 除号
              | MINUS expr              # MINUS – '-'负号
              | expr EXPON expr         # EXPON – '^'乘幂
              | LB expr RB              # LB, RB – '(', ')' 左右括号
```

2、L-asgn 语言的文法

```
input    ::= ε | input line
line     ::=  EOL
              | asgnexp ';' EOL
asgnexp  ::= IDENTIFIER ASGN exp    # ASGN  - '='赋值号
expr     ::=  NUMBER
              | IDENTIFIER          # IDENTIFIER – 标识符
              | expr PLUS expr      # PLUS  – '+'加号
              | expr MINUS expr     # MINUS – '-'减号
              | expr MULT expr      # MULT  – '*'乘号
              | expr DIV expr       # DIV   – '/' 除号
              | MINUS expr          # MINUS – '-'负号
              | expr EXPON expr     # EXPON – '^'乘幂
              | LB expr RB          # LB, RB – '(', ')' 左右括号
```

在 L-asgn 语言中，标识符代表变量，标识符需要先赋值再使用。如果程序使用了未赋值的变量，则需要给出警告并默认初值为 0。

3、bison-examples.zip 的文件目录结构说明

```
|- README 对本压缩包的说明文档
|- Makefile      用于构造语言的词法、语法分析器并将它们编译成可执行文件
|- run.sh        以./run.sh x y 命令启动 bin 子目录下的 x 编译器编译 test 子目录下的
                 y 程序
|- config        存放语言的词法和语法文法文件
   |- expr.lex   L-expr 的 Flex 词法规范。
                 以-oexpr.lex.c 选项执行 flex，会根据该文件生成词法分析器 expr.lex.c
   |- expr.y     L-expr 的 Yacc 语法规范，它利用 Yacc 提供的优先级声明来消除文法
```

的二义性。该语法规范内嵌有边分析边对表达式求值的语义动作。
以-b expr -o expr.tab.c 选项执行 Bison，会根据该文件生成语法分析器 expr.tab.c 以及词法、语法分析都需使用的符号声明头文件 expr.tab.h

|- expr1.y L-expr 的 Yacc 语法规范，它通过增加文法非终结符，将文法改写成无二义的文法。可以用 Bison 处理该文件，生成 expr1.tab.h(.c)。该语法规范内嵌有边分析边对表达式求值的语义动作。

|- exprL.y L 属性定义举例：L-expr 的 Yacc 语法规范，在规则中嵌入打印行号的语义动作，行号通过全局变量传递。

|- exprL1.y L 属性定义举例：L-expr 扩展语言略(每行开始增加行号)的 Yacc 语法规范，在规则中嵌入打印源程序中所给行号的语义动作。

|- midrule.y L 属性定义举例：使用$<val>N 引用语义值栈中任意位置的语义值，N 可以为正整数、0、负整数，<val>为所引用的语义值的类型。

|- asgn.lex L-asgn 的 Flex 词法规范。

|- asgn.y L-asgn 的 Yacc 语法规范。该语法规范内嵌有边分析边对赋值语句求值的语义动作。

|- asgn1.y L-asgn 的 Yacc 语法规范，它与 asgn.y 的区别在于引入一个表示双目运算符的非终结符 op。

|- asgn2ast.y L-asgn 的 Yacc 语法规范。该语法规范内嵌有边分析边构造抽象语法树的语义动作。

|- asgn_err.lex L-asgn 的 Flex 词法规范，它与 asgn.lex 的区别在于引入统计尚未匹配的左括号数的全局量 lparen_num，以便在词法分析的过程中进行括号匹配跟踪。

|- asgn_err.y L-asgn 的 Yacc 语法规范。它与 asgn_err.lex 协作，支持对不合法的 L-asgn 程序的分析，提供错误恢复以及错误信息的记录并在分析结束时集中输出。

|- include 存放用户自定义的头文件

|-util.h 存放一些实用的宏、类型声明、函数声明，如动态分配相关的宏定义(如 NEW, NEW0)；线性表、线性表迭代器的数据结构的定义以及相关接口函数的声明。

|-common.h 编译器需要使用的一些公共的声明和定义，如符号表的结构、错误信息的结构、错误容器（或称错误工厂）的结构、语法树及其结点的结构定义，相关函数的声明等等

|-op.h 保存所处理的运算符及其对应的文本信息的配置文件

|-errcfg.h 保存所处理的错误类别及其对应的提示信息的配置文件

|- src 存放源程序文件

|-list.c 线性表的接口函数的实现（采用链式结构存储）、线性表迭代器 Iterator 的接口函数的实现

|-symtab.c 符号表的接口函数的实现（采用链地址结构的 Hash 表存储）

|-error.c 错误信息及其管理的接口函数的实现

|-ast.c 抽象语法树相关的接口函数的实现

|- expr.1ex.c, expr.tab.h, expr.tab.c, expr1.tab.c
利用 Bison 和 Flex 处理 config 目录下的 expr.lex, expr.y, expr1.y 生成的词法、语法分析器的源代码文件。

|- asgn.1ex.c, asgn.tab.h, asgn.tab.c, asgn1.tab.c, asgn2ast.tab.c

利用 Bison 和 Flex 处理 config 目录下的 asgn.lex, asgn.y, asgn1.y, asgn2ast.y 生成的词法、语法分析器的源代码文件。

|- asgn_err.1ex.c, asgn_err.tab.h, asgn_err.tab.c

利用 Bison 和 Flex 处理 config 目录下的 asgn_err.lex, asgn_err.y 生成的词法、语法分析器的源代码文件。

|- bin 存放可执行文件

|- expr, expr1, asgn, asgn1, asgn2ast, asgn_err

执行 make 或者 make expr (或 expr1, asgn, asgn1, asgn2ast, asgn_err) 后得到的 L-expr、L-asgn 语言的编译器的可执行文件

|- test 存放测试程序

|- expr.in 一个合法的 L-expr 程序

|- asgn.in 一个合法的 L-asgn 程序

|- asgn_err.in 一个不合法的 L-asgn 程序

4、如何使 Flex 和 Bison 生成的分析器能跟踪程序位置

bison-examples 中 config/asgn.lex 和 asgn.y 给出了如何使生成的分析器能跟踪程序位置的示例。这里简述要点：

1) Bison 生成的分析器缺省地用 YYLTYPE 结构类型（在生成的*.tab.h 中定义）来存储位置，其中包含 first_line、first_column、last_line、last_column 四个 int 域；并且引入 YYLTYPE 类型的全局变量 yylloc 保存当前记号的位置信息。

2) 在 asgn.lex 的声明段，增加如下代码：

```
%{int yycolumn = 1;

#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno; \
    yylloc.first_column = yycolumn; yylloc.last_column = yycolumn+yyleng-1; \
     yycolumn += yyleng;
%}
%option yylineno
```

3) 在 asgn.lex 的规则段，在处理一个新行时增加对 yycolumn 修改的语义动作；

4) Bison 会按缺省的位置跟踪方法为生成的分析器跟踪位置信息，见 Bison manual 3.6 Tracking Locations。在 asgn.y 的语义动作中，可以通过@$.first_line、@1.last_column 来访问位置信息， 其中@$表示产生式左部符号(LHS, left-hand side)对应的位置信息，@1 表示产生式右部符号(RHS, right-hand side) 对应的位置信息。

5、利用预编译指令灵活处理诸如错误类别等的可配置信息

在 bison-examples 中使用预编译指令灵活处理错误类别以及操作符等可配置的信息，并根据实际的配置信息生成相关类型和全局变量等。这里以操作符为例，简述处理方法：

1) include/op.h 为操作符的配置文件，其中每个操作符对应于如下形式的一行条目

```
opxx(PLUS, " + " )
```

2) 在 include/common.h 中有如下代码段：

```
enum {
#define opxx(a, b) OP_##a,
#include "op.h"
        OPLAST
```

};

该代码段将由 op.h 里的配置信息产生枚举常量，如 PLUS 对应的为 OP_PLUS

3) 在 src/ast.c 中有如下代码段：

```
char *opname[]={
#undef opxx
#define opxx(a, b) b,
#include "op.h"
        "Undefined Op"
};
```

该代码段将由 op.h 里的配置信息来初始化操作符名称数组 opname 的取值。

4) 用户可以在 op.h 中增加更多的操作符，而无须在修改枚举类型和 opname 的定义。

有关错误类型和对应的错误提示信息可以在 include/errcfg.h 中配置，在 include/common.h 和 src/error.c 中有相应的枚举类型、错误信息数组的定义。

6、引入宏 NEW 和 NEW0 来简化对各类结点的动态创建和清 0 操作的代码编写

```
#define NEW(p)     ((p) = malloc(sizeof *(p)))
#define NEW0(p)   memset(NEW(p), 0, sizeof *(p))
```

则若要构建结点类型为 A 的结点，则可以：

```
A *p, *q;
NEW(p);          // 创建 A 类型的结点，使 p 指向 A 类型的结点
NEW0(q);         // 创建 A 类型的结点并将所创建的结点清 0，使 q 指向该结点
```

7、引入宏 NEW 和 NEW0 来简化对各类结点的动态创建和清 0 操作的代码编写

1）引入设计模式来创建各类 AST 结点、错误信息结点等

2）引入“工厂模式”这一常用设计模式来创建各类 AST 结点、错误信息结点等

8、在 Yacc 文法描述文件中处理 L 属性定义

1）在.y 文件中，翻译规则可以内嵌语义动作代码，如 config/exprL.y 中有如下片段：

```
input    : …
         | input
         { lineno ++;
           printf("Line %d:\t", lineno);
         }
           line
```

每个用一对花括号括起的内嵌语义动作也视为一个文法符号（匿名符号），故上例中右部 input {…} line 共计 3 个文法符号，line 对应的语义值通过$3 引用。

2）可以在翻译规则的内嵌语义动作代码中设置该文法符号对应的语义值，如 config/exprL1.y 中有如下片段：

```
line    : …
        | NUMBER { …
          $<val>lineno = $1;
          //$<val>$ = $1;…
        }[lineno]
```

```
exp EOL { …
  printf("Line %d: %g\n", (int)$<val>lineno, $3);
  …
}
```

其中：

(1) [lineno]为嵌入在 NUMBER 和 exp 之间的语义动作命名,从而该语义动作对应的文法符号名为 lineno。若上述代码片段中去掉[lineno]，则上述内嵌的语义动作对应的文法符号是匿名的。

(2) $<val>lineno = $1; 是在设置 lineno 文法符号的语义值，<val>是 lineno 文法符号的语义值类型。$<val>lineno 也可以换成$<val>$。后者可以在未指定语义动作对应的文法符号名时引用该语义值单元。

(3) 规则最右边的语义动作中的$<val>lineno 是在引用第 2 个文法符号（即 lineno）的语义值，这里$<val>lineno 可以换成$<val>2。

3)可以在一个翻译规则的语义动作代码中使用存储在栈中任意固定相对位置的语义值，如 config/midrule.y 中有如下片段：

```
exp: a_1 a_2 { $<val>$ = 3; } { $<val>$ = $<val>3 + 1; } a_5
     sum_of_the_five_previous_values
    {
       USE (($1, $2, $<foo>3, $<foo>4, $5));
       printf ("%d\n", $6);
    }
sum_of_the_five_previous_values:
    {
       $$ = $<val>0 + $<val>-1 + $<val>-2 + $<val>-3 + $<val>-4;
    }
```

其中，sum_of_the_five_previous_values 文法符号对应的规则中$<val>0、$<val>-1、$<val>-2、$<val>-3、$<val>-4 分别表示栈中 a_5、{ $<val>$ = $<val>3 + 1; }、{ $<val>$ = 3; }、a_2、a_1 文法符号的语义值。

附 5：LLVM IR 及相关工具链简介

LLVM（Low Level Virtual Machine）是一个编译器框架，包含众多的开发支持工具和子项目（比如 Clang），它旨在研究支持任何静态和动态语言的动态编译技术。LLVM 提供一种类 RISC 的低级虚拟指令集 LLVM IR（intermediate represention）。

LLVM IR 是自包含的、语言无关和机器无关的语言。所谓**自包含**是指当你写一个前端时，只需要了解如何将源语言转换成 IR；而当你开发针对 IR 的分析和优化时，可以从 IR 中获取所有需要的信息。LLVM IR 对机器相关的细节进行了抽象，它允许用户使用数量无限的虚拟寄存器。LLVM IR 有**三种等价的表示形式**：一种是文本格式（文件扩展名为.ll），一种是存在于内存中的形式，一种是保存在磁盘上的二进制格式（bitcode，文件扩展名为.bc，它主要便于即时编译器快速加载）。这三种形式完全等价，通过工具 llvm-as 和 llvm-dis 可以实现三种形式的相互转换。下面简要介绍 LLVM IR 语言的特征，更详细的内容请参见 http://llvm.org/docs/LangRef.html。

标识符：LLVM IR 的标识符有全局和局部两种基本类型。全局标识符（函数、全局变量）以'@'字符开始，局部标识符（寄存器名、类型）以'%' 字符开始。

LLVM IR 程序的高层结构：LLVM IR 的程序由 Module 组成，每个 Module 是输入源程序的一个翻译单元，由函数、全局变量和符号表条目等组成。

你可以阅读 http://llvm.org/docs/LangRef.html 来了解 LLVM 语言的类型、常量、指令等的定义形式和特点。你可以首先阅读 http://llvm.org/docs/GettingStarted.html 来快速了解 LLVM 系统。阅读 http://www.aosabook.org/en/llvm.html 来了解 LLVM 编译框架的设计。图 1 给出了一个简单的 C 代码和它对应的 LLVM IR。

C 代码：

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}

unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

LLVM IR 代码：

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

图 1　LLVM IR 代码示例

LLVM 工具链：LLVM 系统包含一系列的工具链。你可以使用 clang 前端来编译 C 程序得到可执行文件：

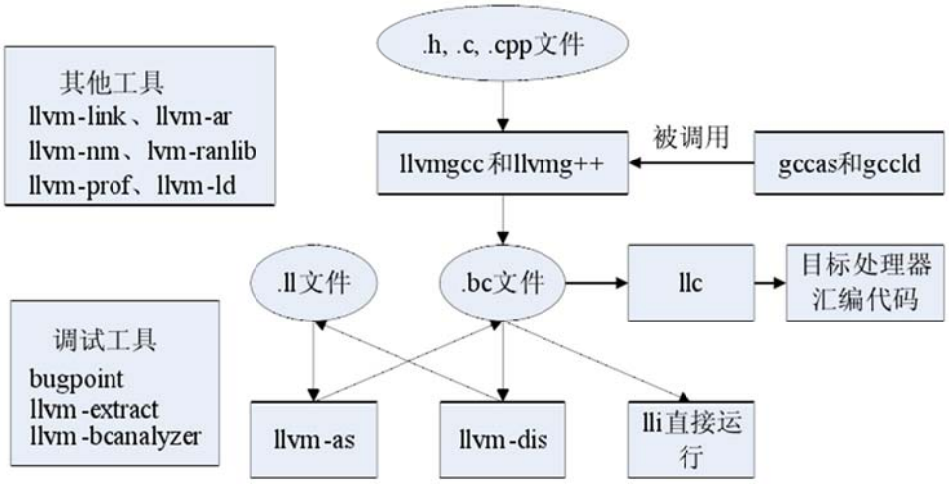

图 2　LLVM 工具链图

```
% clang hello.c -o hello
```

也可以利用 clang 将 C 程序编译成 LLVM IR 的二进制形式文件(bitcode)，其中-emit-llvm 选项可以和-S 或-c 选项一起用于为 C 程序产生 LLVM 的.ll 文件或.bc 文件：

```
% clang -O3 -emit-llvm hello.c -c -o hello.bc
```

你可以调用 LLVM 即时编译器 lli 来运行一个 LLVM 程序：

```
% lli hello.bc
```

你可以使用 llvm-dis 来查看 LLVM 汇编码：

```
% llvm-dis < hello.bc | less
```

你可以利用 LLC 代码生成器将 LLVM 程序编译成本地汇编码：

```
% llc hello.bc -o hello.s
```

你可以利用 gcc 等将本地汇编程序汇编成目标程序：

```
% gcc hello.s -o hello.native
% ./hello.native
```