

第14章 TinyC 前端

上一章介绍了 bison 中一些常用功能的使用方法，bison 是一个非常强大的语法分析工具，读者还可以阅读一下 bison 的文档进行更深入的学习。本章介绍如何利用 flex 和 bison 实现 TinyC 编译器的前端，建议读者先复习一下 **第 5 章 手工编译 TinyC**，再来看本章的代码。

14.1 第 0.1 版

首先对上一章的雏形版本稍微升级一下，增加变量声明和 print 语句，一共有 5 个文件：

词法分析文件： `scanner.l`

```
%{
#define YYSTYPE char *
#include "y.tab.h"
int cur_line = 1;
void yyerror(const char *msg);
void unrecognized_char(char c);
#define _DUPTXT {yylval = strdup(yytext);}
%}

/* note \042 is '"' */

OPERATOR      ([-/+*()=,;])
INTEGER       ([0-9]+)
STRING        (\042[^\042\n]*\042)
IDENTIFIER    ([_a-zA-Z][_a-zA-Z0-9]*)
WHITESPACE    ([ \t]*)

%%
{OPERATOR}    { return yytext[0]; }
"int"         { return T_Int; }
"print"       { return T_Print; }

{INTEGER}     { _DUPTXT; return T_IntConstant; }
{STRING}      { _DUPTXT; return T_StringConstant; }
{IDENTIFIER}  { _DUPTXT; return T_Identifier; }

{WHITESPACE}  { /* ignore every whitespace */ }
\n            { cur_line++; }
```

```

.          { unrecognized_char(yytext[0]); }
%%

int yywrap(void) {
    return 1;
}

void unrecognized_char(char c) {
    char buf[32] = "Unrecognized character: ?";
    buf[24] = c;
    yyerror(buf);
}

void yyerror(const char *msg) {
    printf("Error at line %d:\n\t%s\n", cur_line, msg);
    exit(-1);
}

```

语法分析文件: [parser.y](#)

```

%{
#include <stdio.h>
#include <stdlib.h>
void yyerror(const char*);
#define YYSTYPE char *
%}

%token T_StringConstant T_IntConstant T_Identifier T_Int T_Print

%left '+' '-'
%left '*' '/'
%right U_neg

%%

S:
    Stmt                                { /* empty */ }
|   S Stmt                            { /* empty */ }
;

Stmt:
    VarDecl ';'                       { printf("\n\n"); }
|   Assign                            { /* empty */ }
|   Print                             { /* empty */ }
;

```

```

VarDecl:
    T_Int T_Identifier      { printf("var %s", $2); }
|   VarDecl ',' T_Identifier { printf(", %s", $3); }
;

Assign:
    T_Identifier '=' E ';'  { printf("pop %s\n\n", $1); }
;

Print:
    T_Print '(' T_StringConstant Actuals ')' ';'
                                { printf("print %s\n\n", $3); }
;

Actuals:
    /* empty */              { /* empty */ }
|   Actuals ',' E            { /* empty */ }
;

E:
    E '+' E                  { printf("add\n"); }
|   E '-' E                  { printf("sub\n"); }
|   E '*' E                  { printf("mul\n"); }
|   E '/' E                  { printf("div\n"); }
|   '-' E %prec U_neg        { printf("neg\n"); }
|   T_IntConstant            { printf("push %s\n", $1); }
|   T_Identifier              { printf("push %s\n", $1); }
|   '(' E ')'                { /* empty */ }
;

%%

int main() {
    return yyparse();
}

```

makefile 文件: `makefile`

```

OUT          = tcc
TESTFILE     = test.c
SCANNER      = scanner.l
PARSER       = parser.y

CC           = gcc
OBJ          = lex.yy.o y.tab.o
TESTOUT      = $(basename $(TESTFILE)).asm

```

```
OUTFILES = lex.yy.c y.tab.c y.tab.h y.output $(OUT)

.PHONY: build test simulate clean

build: $(OUT)

test: $(TESTOUT)

simulate: $(TESTOUT)
    python pysim.py $<

clean:
    rm -f *.o $(OUTFILES)

$(TESTOUT): $(TESTFILE) $(OUT)
    ./$(OUT) < $< > $@

$(OUT): $(OBJ)
    $(CC) -o $(OUT) $(OBJ)

lex.yy.c: $(SCANNER) y.tab.c
    flex $<

y.tab.c: $(PARSER)
    bison -vdtty $<
```

测试文件: `test.c`

```
int a, b, c, d;
a = 1 + 2 * ( 2 + 2 );
c = 5;
d = 10;
b = c + d;

print("a = %d, b = %d, c = %d, d = %d", a, b, c, d);
```

Pcode 模拟器: `pysim.py` , 已经在第 4 章中介绍了。

这个版本在上一章的雏形版本的基础上, 进行了以下扩充:

词法分析文件中:

增加了 `T_StringConstant`, `T_Int`, `T_Print` 类型的 `token` , 以及相应的正则表达式;

增加了一个 `_DUPTXT` 宏，表示 `yylval = strdup(yytext)`。

语法分析文件中：

增加了 `VarDecl` 和 `Print` 两个非终结符以及相应的产生式。

本版本的语法分析文件中，同样要注意源文件的解析过程中各产生式的折叠顺序以及相应的 `Pcode` 生成顺序。

`makefile` 里面是编译和测试这个程序的命令，在终端输入 `make` 后，将编译生成可执行文件 `tcc`，然后输入 `make test`，（相当于 `"./tcc < test.c > test.asm"`），将输出 `test.asm` 文件，内容如下：

```
var a, b, c, d

push 1
push 2
push 2
push 2
add
mul
add
pop a

push 5
pop c

push 10
pop d

push c
push d
add
pop b

push a
push b
push c
push d
print "a = %d, b = %d, c = %d, d = %d"
```

可以看出 test.c 文件里的所有语句都被转换成相应的 Pcode 了。再用 Pcode 模拟器运行一下这些 Pcode，在终端输入 “make simulate”（相当于 “python pysim.py test.asm”），将输出：

```
a = 9, b = 5, c = 10, d = 15
```

14.2 第 0.5 版

在第 0.1 版的基础上升级，增加函数定义及调用语句、注释等功能，一共有 5 个文件：

词法分析文件： `scanner.l`

```
%{
#define YYSTYPE char *
#include "y.tab.h"
int cur_line = 1;
void yyerror(const char *msg);
void unrecognized_char(char c);
void unterminate_string();
#define _DUPTXT {yylval = strdup(yytext);}
%}

/* note \042 is '"' */
WHITESPACE      ([ \t\r\a]+)
SINGLE_COMMENT1  ("//[^\n]*")
SINGLE_COMMENT2  ("#[^\n]*")
OPERATOR        ([+*-/%=,;!<>(){}])
INTEGER         ([0-9]+)
IDENTIFIER       ([_a-zA-Z][_a-zA-Z0-9]*)
UNTERM_STRING   (\042[^\042\n]*)
STRING          (\042[^\042\n]*\042)

%%

\n              { cur_line++; }
{WHITESPACE}    { /* ignore every whitespace */ }
{SINGLE_COMMENT1}{ /* skip for single line comment */ }
{SINGLE_COMMENT2}{ /* skip for single line comment */ }

{OPERATOR}      { return yytext[0]; }
"int"           { return T_Int; }
"void"          { return T_Void; }
"return"        { return T_Return; }
```

```

"print"                { return T_Print; }

{INTEGER}              { _DUPTXT return T_IntConstant; }
{STRING}               { _DUPTXT return T_StringConstant; }
{IDENTIFIER}           { _DUPTXT return T_Identifier; }

{UNTERM_STRING}        { untermiante_string(); }
.                       { unrecognized_char(yytext[0]); }

%%

int yywrap(void) {
    return 1;
}

void unrecognized_char(char c) {
    char buf[32] = "Unrecognized character: ?";
    buf[24] = c;
    yyerror(buf);
}

void untermiante_string() {
    yyerror("Unterminate string constant");
}

void yyerror(const char *msg) {
    fprintf(stderr, "Error at line %d:\n\t%s\n", cur_line, msg);
    exit(-1);
}

```

语法分析文件: [parser.y](#)

```

%{
#include <stdio.h>
#include <stdlib.h>
void yyerror(const char*);
#define YYSTYPE char *
%}

%token T_Int T_Void T_Return T_Print T_IntConstant
%token T_StringConstant T_Identifier

%left '+' '-'
%left '*' '/'
%right U_neg

```

%%

Program:

```
    /* empty */          { /* empty */ }
|   Program FuncDecl    { /* empty */ }
;
```

FuncDecl:

```
    RetType FuncName '(' Args ')' '{' VarDecls Stmts '}'
                                { printf("ENDFUNC\n\n"); }
;
```

RetType:

```
    T_Int                  { /* empty */ }
|   T_Void                 { /* empty */ }
;
```

FuncName:

```
    T_Identifier           { printf("FUNC @%s:\n", $1); }
;
```

Args:

```
    /* empty */           { /* empty */ }
|   _Args                 { printf("\n\n"); }
;
```

_Args:

```
    T_Int T_Identifier     { printf("arg %s", $2); }
|   _Args ',' T_Int T_Identifier
                                { printf(", %s", $4); }
;
```

VarDecls:

```
    /* empty */           { /* empty */ }
|   VarDecls VarDecl ';'  { printf("\n\n"); }
;
```

VarDecl:

```
    T_Int T_Identifier     { printf("var %s", $2); }
|   VarDecl ',' T_Identifier
                                { printf(", %s", $3); }
;
```

Stmts:

```
    /* empty */           { /* empty */ }
|   Stmts Stmt            { /* empty */ }
;
```



```

Stmt:
    AssignStmt          { /* empty */ }
|   PrintStmt          { /* empty */ }
|   CallStmt           { /* empty */ }
|   ReturnStmt         { /* empty */ }
;

AssignStmt:
    T_Identifier '=' Expr ';'
                                { printf("pop %s\n\n", $1); }
;

PrintStmt:
    T_Print '(' T_StringConstant PActuals ')' ';'
                                { printf("print %s\n\n", $3); }
;

PActuals:
    /* empty */              { /* empty */ }
|   PActuals ',' Expr        { /* empty */ }
;

CallStmt:
    CallExpr ';'              { printf("pop\n\n"); }
;

CallExpr:
    T_Identifier '(' Actuals ')'
                                { printf("$%s\n", $1); }
;

Actuals:
    /* empty */              { /* empty */ }
|   Expr PActuals            { /* empty */ }
;

ReturnStmt:
    T_Return Expr ';'         { printf("ret ~\n\n"); }
|   T_Return ';'              { printf("ret\n\n"); }
;

Expr:
    Expr '+' Expr             { printf("add\n"); }
|   Expr '-' Expr             { printf("sub\n"); }
|   Expr '*' Expr             { printf("mul\n"); }
|   Expr '/' Expr             { printf("div\n"); }
|   '-' Expr %prec U_neg      { printf("neg\n"); }
|   T_IntConstant             { printf("push %s\n", $1); }

```

```
| T_Identifier      { printf("push %s\n", $1); }
| CallExpr         { /* empty */ }
| '(' Expr ')'     { /* empty */ }
;
```

%%

```
int main() {
    return yyparse();
}
```

makefile 文件: `makefile`, 和第 0.1 版本中唯一不同的只有 “python pysim.py \$< -a” 那一行有一个 “-a”。

测试文件: `test.c`

```
// tiny c test file
```

```
int main() {
    int a, b, c, d;

    c = 2;
    d = c * 2;

    a = sum(c, d);
    b = sum(a, d);
    print("c = %d, d = %d", c, d);
    print("a = sum(c, d) = %d, b = sum(a, d) = %d", a, b);

    return 0;
}

int sum(int a, int b) {
    int c, d;
    return a + b;
}
```

Pcode 模拟器: `pysim.py`, 已经在第 4 章中介绍了。

这个版本在第 0.1 版本的基础上, 进行了以下扩充:

词法分析文件中:

增加了 T_Void 和 T_Return 类型的 token , 以及相应的正则表达式;

增加了单行注释的过滤功能; 增加了一个错误处理函数: `unterminate_string` , 该函数可以检查出未结束的字符串 (不匹配的双引号) 的词法错误。

语法分析文件中:

增加了 Program, FuncDecl, Args, Actuals, CallExpr 等非终结符以及相应的产生式, 请注意各产生式的折叠顺序以及相应的 Pcode 生成顺序。

`makefile` 里面是编译和测试这个程序的命令, 内容和第 0.1 版的基本一样, 但增加了一些变量以便于扩充, 另外, `"python pysim.py..."` 那一行最后的命令行参数是 `"-a"` 。在终端输入 `make` 后, 将编译生成可执行文件 `tcc` , 然后输入 `make test` , (相当于 `"./tcc < test.c > test.asm"`) , 将输出 `test.asm` 文件, 内容如下:

```
FUNC @main:
var a, b, c, d

push 2
pop c

push c
push 2
mul
pop d

push c
push d
$sum
pop a

push a
push d
$sum
pop b

push c
push d
print "c = %d, d = %d"
```

```
push a
push b
print "a = sum(c, d) = %d, b = sum(a, d) = %d"

push 0
ret ~

ENDFUNC

FUNC @sum:
arg a, b

var c, d

push a
push b
add
ret ~

ENDFUNC
```

可以看出 test.c 文件里的所有语句都被转换成相应的 Pcode 了。再用 Pcode 模拟器运行一下这些 Pcode，在终端输入 “make simulate”（相当于 “python pysim.py test.asm -a”，注意最后有一个 “-a”），将输出：

```
c = 2, d = 4
a = sum(c, d) = 6, b = sum(a, d) = 10
```

有兴趣的读者还可以使用 “python pysim.py test.asm -da” 来逐句运行一下这个 Pcode 文件。

14.3 第 1.0 版

继续在第 0.5 版的基础上升级，增加 if 和 while 语句、比较运算符和逻辑运算符以及 readint 命令，就形成了完整的 TinyC 前端。一共有 7 个文件：

词法分析文件： [scanner.l](#)

```

%{
#define YYSTYPE char *
#include "y.tab.h"
int cur_line = 1;
void yyerror(const char *msg);
void unrecognized_char(char c);
void unterminate_string();
#define _DUPTXT {yyval = strdup(yytext);}
%}

/* note \042 is '"' */
WHITESPACE      ([ \t\r\a]+)
SINGLE_COMMENT1  ("//[\"^[\\n]*")
SINGLE_COMMENT2  ("#[\"^[\\n]*")
OPERATOR        ([+*-/%=,;!<>(){}])
INTEGER         ([0-9]+)
IDENTIFIER      ([_a-zA-Z][_a-zA-Z0-9]*)
UNTERM_STRING   (\\042[^\042\\n]*)
STRING          (\\042[^\042\\n]*\\042)

%%

\n              { cur_line++; }
{WHITESPACE}    { /* ignore every whitespace */ }
{SINGLE_COMMENT1}{ /* skip for single line comment */ }
{SINGLE_COMMENT2}{ /* skip for single line comment */ }

{OPERATOR}      { return yytext[0]; }
"int"           { return T_Int; }
"void"          { return T_Void; }
"return"        { return T_Return; }
"print"         { return T_Print; }
"readint"       { return T_ReadInt; }
"while"         { return T_While; }
"if"            { return T_If; }
"else"          { return T_Else; }
"break"         { return T_Break; }
"continue"      { return T_Continue; }
"≤"             { return T_Le; }
"≥"             { return T_Ge; }
"="             { return T_Eq; }
"≠"             { return T_Ne; }
"&&"           { return T_And; }
"||"           { return T_Or; }

{INTEGER}       { _DUPTXT return T_IntConstant; }
{STRING}        { _DUPTXT return T_StringConstant; }

```

```

{IDENTIFIER}      { _DUPTXT return T_Identifier; }

{UNTERM_STRING}   { unterminate_string(); }
.                 { unrecognized_char(yytext[0]); }

%%

int yywrap(void) {
    return 1;
}

void unrecognized_char(char c) {
    char buf[32] = "Unrecognized character: ?";
    buf[24] = c;
    yyerror(buf);
}

void unterminate_string() {
    yyerror("Unterminate string constant");
}

void yyerror(const char *msg) {
    fprintf(stderr, "Error at line %d:\n\t%s\n", cur_line, msg);
    exit(-1);
}

```

语法分析文件: [parser.y](#)

```

%{
#include <stdio.h>
#include <stdlib.h>
void yyerror(const char*);
#define YYSTYPE char *

int ii = 0, itop = -1, istack[100];
int ww = 0, wtop = -1, wstack[100];

#define _BEG_IF      {istack[++itop] = ++ii;}
#define _END_IF      {itop--;}
#define _i           (istack[itop])

#define _BEG_WHILE   {wstack[++wtop] = ++ww;}
#define _END_WHILE   {wtop--;}
#define _w           (wstack[wtop])

%}

```

```
%token T_Int T_Void T_Return T_Print T_ReadInt T_While
%token T_If T_Else T_Break T_Continue T_Le T_Ge T_Eq T_Ne
%token T_And T_Or T_IntConstant T_StringConstant T_Identifier
```

```
%left '='
%left T_Or
%left T_And
%left T_Eq T_Ne
%left '<' '>' T_Le T_Ge
%left '+' '-'
%left '*' '/' '%'
%left '!'
```

```
%%
```

```
Program:
```

```
    /* empty */          { /* empty */ }
|   Program FuncDecl    { /* empty */ }
;
```

```
FuncDecl:
```

```
    RetType FuncName '(' Args ')' '{' VarDecls Stmts '}'
                                { printf("ENDFUNC\n\n"); }
;
```

```
RetType:
```

```
    T_Int          { /* empty */ }
|   T_Void         { /* empty */ }
;
```

```
FuncName:
```

```
    T_Identifier    { printf("FUNC @%s:\n", $1); }
;
```

```
Args:
```

```
    /* empty */          { /* empty */ }
|   _Args                { printf("\n\n"); }
;
```

```
_Args:
```

```
    T_Int T_Identifier    { printf("\targ %s", $2); }
|   _Args ',' T_Int T_Identifier
                                { printf(", %s", $4); }
;
```

```
VarDecls:
```

```
    /* empty */          { /* empty */ }
```

```
|   VarDecls VarDecl ';'          { printf("\n\n"); }
;

VarDecl:
    T_Int T_Identifier      { printf("\tvar %s", $2); }
|   VarDecl ',' T_Identifier { printf(", %s", $3); }
;

Stmts:
    /* empty */           { /* empty */ }
|   Stmts Stmt            { /* empty */ }
;

Stmt:
    AssignStmt             { /* empty */ }
|   PrintStmt              { /* empty */ }
|   CallStmt               { /* empty */ }
|   ReturnStmt             { /* empty */ }
|   IfStmt                 { /* empty */ }
|   WhileStmt              { /* empty */ }
|   BreakStmt              { /* empty */ }
|   ContinueStmt           { /* empty */ }
;

AssignStmt:
    T_Identifier '=' Expr ';'     { printf("\tpop %s\n\n", $1); }
;

PrintStmt:
    T_Print '(' T_StringConstant PActuals ')' ';' { printf("\tprint %s\n\n", $3); }
;

PActuals:
    /* empty */           { /* empty */ }
|   PActuals ',' Expr      { /* empty */ }
;

CallStmt:
    CallExpr ';'             { printf("\tpop\n\n"); }
;

CallExpr:
    T_Identifier '(' Actuals ')' { printf("\t$%s\n", $1); }
;
```


Actuals:

```
    /* empty */          { /* empty */ }
|   Expr PActuals        { /* empty */ }
;
```

ReturnStmt:

```
    T_Return Expr ';'      { printf("\tret ~\n\n"); }
|   T_Return ';'          { printf("\tret\n\n"); }
;
```

IfStmt:

```
    If TestExpr Then StmtBlock EndThen EndIf
                                { /* empty */ }
|   If TestExpr Then StmtBlock EndThen Else StmtBlock EndIf
                                { /* empty */ }
;
```

TestExpr:

```
    '(' Expr ')'           { /* empty */ }
;
```

StmtsBlock:

```
    '{' Stmt '}'          { /* empty */ }
;
```

If:

```
    T_If                  { _BEG_IF; printf("_begIf_%d:\n", _i); }
;
```

Then:

```
    /* empty */          { printf("\tjz _eIf_%d\n", _i); }
;
```

EndThen:

```
    /* empty */          { printf("\tjmp _endIf_%d\n_eIf_%d:\n", _i, _i)
;
```

Else:

```
    T_Else                { /* empty */ }
;
```

EndIf:

```
    /* empty */          { printf("_endIf_%d:\n\n", _i); _END_IF; }
;
```

WhileStmt:

```
    While TestExpr Do StmtBlock EndWhile
```

```

        { /* empty */ }
;

While:
    T_While          { _BEG_WHILE; printf("_begWhile_%d:\n", _w); }
;

Do:
    /* empty */      { printf("\tjz _endWhile_%d\n", _w); }
;

EndWhile:
    /* empty */      { printf("\tjmp _begWhile_%d\n_endWhile_%d:\n\n"
                           _w, _w); _END_WHILE; }
;

BreakStmt:
    T_Break ';'      { printf("\tjmp _endWhile_%d\n", _w); }
;

ContinueStmt:
    T_Continue ';'   { printf("\tjmp _begWhile_%d\n", _w); }
;

Expr:
    Expr '+' Expr     { printf("\tadd\n"); }
|   Expr '-' Expr     { printf("\tsub\n"); }
|   Expr '*' Expr     { printf("\tmul\n"); }
|   Expr '/' Expr     { printf("\tdiv\n"); }
|   Expr '%' Expr     { printf("\tmod\n"); }
|   Expr '>' Expr     { printf("\tcmpgt\n"); }
|   Expr '<' Expr     { printf("\tcmplt\n"); }
|   Expr T_Ge Expr    { printf("\tcmpge\n"); }
|   Expr T_Le Expr    { printf("\tcmple\n"); }
|   Expr T_Eq Expr    { printf("\tcmpeq\n"); }
|   Expr T_Ne Expr    { printf("\tcmpne\n"); }
|   Expr T_Or Expr    { printf("\tor\n"); }
|   Expr T_And Expr   { printf("\tand\n"); }
|   '-' Expr %prec '!' { printf("\tneg\n"); }
|   '!' Expr          { printf("\tnot\n"); }
|   T_IntConstant     { printf("\tpush %s\n", $1); }
|   T_Identifier       { printf("\tpush %s\n", $1); }
|   ReadInt            { /* empty */ }
|   CallExpr           { /* empty */ }
|   '(' Expr ')'       { /* empty */ }
;

ReadInt:

```

```
T_ReadInt '(' T_StringConstant ')'
                { printf("\treadint %s\n", $3); }
;

%%

int main() {
    return yyparse();
}
```

makefile 文件: `makefile` , 内容和 第 0.5 版是一样的。

测试文件: `test.c` , 就是第二章的示例源程序。

```
#include "for_gcc_build.hh" // only for gcc, TinyC will ignore it.

int main() {
    int i;
    i = 0;
    while (i < 10) {
        i = i + 1;
        if (i == 3 || i == 5) {
            continue;
        }
        if (i == 8) {
            break;
        }
        print("%d! = %d", i, factor(i));
    }
    return 0;
}

int factor(int n) {
    if (n < 2) {
        return 1;
    }
    return n * factor(n - 1);
}
```

测试文件包: `samples.zip` , 包含了 7 个测试文件。

测试脚本: `test_samples.sh` 。

Pcode 模拟器: `pysim.py` 。

这个版本在第 0.1 版本的基础上，进行了以下扩充：

词法分析文件中：

增加了 T_Void 和 T_Return 类型的 token，以及相应的正则表达式。

语法分析文件中：

增加了 IfStmt, WhileStmt, BreakStmt, ContinueStmt, ReadInt 等非终结符以及相应的产生式，请注意各产生式的折叠顺序以及相应的 Pcode 生成顺序；

增加了比较运算符、逻辑运算符，以及相应的优先级；

在 Declarations 段，增加了几个全局变量和宏：

```
int ii = 0, itop = -1, istack[100];
int ww = 0, wtop = -1, wstack[100];

#define _BEG_IF      {istack[++itop] = ++ii;}
#define _END_IF      {itop--;}
#define _i           (istack[itop])

#define _BEG_WHILE   {wstack[++wtop] = ++ww;}
#define _END_WHILE   {wtop--;}
#define _w           (wstack[wtop])
```

这些全局变量和宏配合后面的 if/while 语句产生式中的 action 使用，是该文件中的最精妙的部分，它们的作用是：在生成 if 和 while 语句块的 Pcode 的过程中，给相应的 Label 进行编号。它们给每个 if 语句块和每个 while 语句块一个唯一的编号，使不同的 if/while 语句块的 jmp 不相互冲突。其中 _i 永远是当前的 if 语句块的编号，_w 永远是当前的 while 语句块的编号；ii/ww 永远是当前解析到的 if/while 语句块的总数。

将以上所有文件都放在当前目录，在终端直接输入 make test，将自动编译生成 TinyC 前端：tcc，并自动调用 tcc 将 test.c 编译成 test.asm 文件，内容如下，和第 5 章的手工编译的结果差不多吧：

```
FUNC @main:
    var i

    push 0
    pop i

_begWhile_1:
    push i
    push 10
    cmplt
    jz _endWhile_1
    push i
    push 1
    add
    pop i

_begIf_1:
    push i
    push 3
    cmpeq
    push i
    push 5
    cmpeq
    or
    jz _elif_1
    jmp _begWhile_1
    jmp _endIf_1
_elif_1:
_endIf_1:

_begIf_2:
    push i
    push 8
    cmpeq
    jz _elif_2
    jmp _endWhile_1
    jmp _endIf_2
_elif_2:
_endIf_2:

    push i
    push i
    $factor
    print "%d! = %d"

    jmp _begWhile_1
_endWhile_1:
```

```

        push 0
        ret ~

ENDFUNC

FUNC @factor:
    arg n

    _begIf_3:
        push n
        push 2
        cmplt
        jz _eIf_3
        push 1
        ret ~

        jmp _endIf_3
    _eIf_3:
    _endIf_3:

        push n
        push n
        push 1
        sub
        $factor
        mul
        ret ~

ENDFUNC

```

再输入 “make simulate”，将输出：

```

1! = 1
2! = 2
4! = 24
6! = 720
7! = 5040

```

和第二章中用 gcc 编译并运行此文件的结果完全一样。

再把测试文件包里的所有源文件全部测试一遍，将 [samples.zip](#) 解压到 samples 目录下，测试脚本 [test_samples.sh](#) 将分别调用 tcc 和 gcc 编译测试文件包中的每一个文件，并分别使用 pysim.py 和 操作系统 运行编译得到的目标文件，内容如下：

```
for src in $(ls samples/*.c)
do
    clear
    file=${src%%.c}
    echo build with tcc
    ./tcc < $file.c > $file.asm
    python pysim.py $file.asm -a
    echo
    echo build with gcc
    gcc -o $file $file.c
    ./$file
    echo
    echo press any key to continue...
    read -n 1
done
```

在终端输入 `bash ./test_samples.sh`，将分别输出一系列的结果，典型输出如下，可以看到 `gcc` 和 `tcc` 编译运行的结果完全一致。

```
build with tcc, the output are:
The first 10 number of the fibonacci sequence:
fib(1)=1
fib(2)=1
fib(3)=2
fib(4)=3
fib(5)=5
fib(6)=8
fib(7)=13
fib(8)=21
fib(9)=34
fib(10)=55
```

```
build with gcc, the output are:
The first 10 number of the fibonacci sequence:
fib(1)=1
fib(2)=1
fib(3)=2
fib(4)=3
fib(5)=5
fib(6)=8
fib(7)=13
fib(8)=21
fib(9)=34
fib(10)=55
```

至此 TinyC 前端完成。

