第16章 TinyC 编译器

在第 14 章中，完成了 TinyC 前端，可以将 TinyC 源程序编译成中间代码 Pcode；在第 15 章中，完成了 TinyC 后端，可以将改写后的中间代码 Pcode 翻译、汇编并链接成可执行程序；现在，是时候将二者结合起来形成最终的 TinyC 编译器了。

16.1 改进 TinyC 前端

上一章的 TinyC 后端中，为了降低 Pcode 命令的翻译难度，对 arg / var / ENDFUNC 命令的格式进行了改写，因此需要改进 TinyC 前端，使之能生成能被 TinyC 后端所识别的新格式 Pcode 命令。具体来说，对于下面这段源程序 `test.c` ：

```
int main() {
    int a;
    a = 3;
    print("sum = %d", sum(4, a));
    return 0;
}

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

改进后 TinyC 前端需要生成一个 Pcode 文件 `test.pcode` ：

```
FUNC @main:
    main.var a

    push 3
    pop a

    push 4
    push a
    $sum
    print "sum = %d"

    ret 0
```

```
ENDFUNC@main

FUNC @sum:
    sum.arg a, b
    sum.var c

    push a
    push b
    add
    pop c

    ret c
ENDFUNC@sum
```

以及一个宏文件 `test.funcmacro` ：

```
; ==== begin function `main` ====
%define main.varc 1

%MACRO main.var main.varc
    %define a [EBP - 4*1]
    SUB ESP, 4*main.varc
%ENDMACRO

%MACRO ENDFUNC@main 0
    LEAVE
    RET
    %undef a
%ENDMACRO
; ==== end function `main`   ====

; ==== begin function `sum` ====
%define sum.argc 2
%define sum.varc 1

%MACRO $sum 0
    CALL @sum
    ADD ESP, 4*sum.argc
    PUSH EAX
%ENDMACRO

%MACRO sum.arg sum.argc
    %define a [EBP + 8 + 4*sum.argc - 4*1]
    %define b [EBP + 8 + 4*sum.argc - 4*2]
%ENDMACRO

%MACRO sum.var sum.varc
```

```
    %define c [EBP - 4*1]
    SUB ESP, 4*sum.varc
%ENDMACRO

%MACRO ENDFUNC@sum 0
    LEAVE
    RET
    %undef a
    %undef b
    %undef c
%ENDMACRO
; ==== end function `sum`  ====
```

在第 14 章的 TinyC 前端 1.0 版的 parser.y 的基础上，针对函数定义、参数定义以及变量定义的语句进行改写，改进后的语法分析文件 **parser.y**：

```
%{

#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>

void init_parser(int argc, char *argv[]);
void quit_parser();

extern FILE* yyin;
FILE *asmfile, *incfile;
#define BUFSIZE 256

#define out_asm(fmt, ...) \
    {fprintf(asmfile, fmt, ##__VA_ARGS__); fprintf(asmfile, "\n");}

#define out_inc(fmt, ...) \
    {fprintf(incfile, fmt, ##__VA_ARGS__); fprintf(incfile, "\n");}

void file_error(char *msg);

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])

int argc = 0, varc = 0;
char *cur_func_name, *args[128], *vars[128];
void write_func_head();
void write_func_tail();

#define _BEG_FUNCDEF(name)  (cur_func_name = (name))
#define _APPEND_ARG(arg)    (args[argc++] = (arg))
#define _APPEND_VAR(var)    (vars[varc++] = (var))
#define _WRITE_FUNCHEAD     write_func_head
#define _END_FUNCDEF        write_func_tail

#define YYSTYPE char *

%}

%token T_Void T_Int T_While T_If T_Else T_Return T_Break T_Continue
%token T_Print T_ReadInt T_Le T_Ge T_Eq T_Ne T_And T_Or
%token 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 '!'

%%

Start:
    Program                         { /* empty */ }
;

Program:
    /* empty */                     { /* empty */ }
|   Program FuncDef                 { /* empty */ }
;

FuncDef:
    T_Int  FuncName Args Vars Stmts EndFuncDef
|   T_Void FuncName Args Vars Stmts EndFuncDef
;
```

```
FuncName:
    T_Identifier                    { _BEG_FUNCDEF($1); }
;

Args:
    '(' ')'                         { /* empty */ }
|   '(' _Args ')'                   { /* empty */ }
;

_Args:
    T_Int T_Identifier              { _APPEND_ARG($2); }
|   _Args ',' T_Int T_Identifier    { _APPEND_ARG($4); }
;

Vars:
    _Vars                           { _WRITE_FUNCHEAD(); }
;

_Vars:
    '{'                             { /* empty */ }
|   _Vars Var ';'                   { /* empty */ }
;

Var:
    T_Int T_Identifier              { _APPEND_VAR($2); }
|   Var ',' T_Identifier            { _APPEND_VAR($3); }
;

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

EndFuncDef:
    '}'                             { _END_FUNCDEF(); }
;

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

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

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

IfStmt:
    If '(' Expr ')' Then '{' Stmts '}' EndThen EndIf
                                    { /* empty */ }
|   If '(' Expr ')' Then '{' Stmts '}' EndThen T_Else '{' Stmts '}'
                                    { /* empty */ }
;

If:
    T_If            { _BEG_IF; out_asm("_begIf_%d:", _i); }
;

Then:
    /* empty */     { out_asm("\tjz _elIf_%d", _i); }
;

EndThen:
    /* empty */     { out_asm("\tjmp _endIf_%d\n_elIf_%d:", _i, _i);
;

EndIf:
    /* empty */     { out_asm("_endIf_%d:", _i); _END_IF; }
;

WhileStmt:
    While '(' Expr ')' Do '{' Stmts '}' EndWhile
                    { /* empty */ }
;

While:
    T_While         { _BEG_WHILE; out_asm("_begWhile_%d:", _w); }
;

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

EndWhile:
    /* empty */     { out_asm("\tjmp _begWhile_%d\n_endWhile_%d:",
                                            _w, _w); _END_WHILE;
```

```
;

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

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

ReturnStmt:
    T_Return ';'            { out_asm("\tret"); }
|   T_Return Expr ';'       { out_asm("\tret ~"); }
;

PrintStmt:
    T_Print '(' T_StringConstant PrintIntArgs ')' ';'
                            { out_asm("\tprint %s", $3); }
;

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

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

```
ReadInt:
    T_ReadInt '(' T_StringConstant ')'
                            { out_asm("\treadint %s", $3); }
;

CallExpr:
    T_Identifier Actuals
                            { out_asm("\t$%s", $1); }
;

Actuals:
    '(' ')'
|   '(' _Actuals ')'
;

_Actuals:
    Expr
|   _Actuals ',' Expr
;

%%

int main(int argc, char *argv[]) {
    init_parser(argc, argv);
    yyparse();
    quit_parser();
}

void init_parser(int argc, char *argv[]) {
    if (argc < 2) {
        file_error("Must provide an input source file!");
    }

    if (argc > 2) {
        file_error("Too much command line arguments!");
    }

    char *in_file_name = argv[1];
    int len = strlen(in_file_name);

    if (len ⩽ 2 || in_file_name[len-1] ≠ 'c' \
            || in_file_name[len-2] ≠ '.') {
        file_error("Must provide an '.c' source file!");
    }

    if (!(yyin = fopen(in_file_name, "r"))) {
        file_error("Input file open error");
    }
```

```
    char out_file_name[BUFSIZE];
    strcpy(out_file_name, in_file_name);

    out_file_name[len-1] = 'a';
    out_file_name[len]   = 's';
    out_file_name[len+1] = 'm';
    out_file_name[len+2] = '\0';
    if (!(asmfile = fopen(out_file_name, "w"))) {
        file_error("Output 'asm' file open error");
    }

    out_file_name[len-1] = 'i';
    out_file_name[len]   = 'n';
    out_file_name[len+1] = 'c';
    if (!(incfile = fopen(out_file_name, "w"))) {
        file_error("Output 'inc' file open error");
    }
}

void file_error(char *msg) {
    printf("\n*** Error ***\n\t%s\n", msg);
    puts("");
    exit(-1);
}

char *cat_strs(char *buf, char *strs[], int strc) {
    int i;
    strcpy(buf, strs[0]);
    for (i = 1; i < strc; i++) {
        strcat(strcat(buf, ", "), strs[i]);
    }
    return buf;
}

#define _fn (cur_func_name)

void write_func_head() {
    char buf[BUFSIZE];
    int i;

    out_asm("FUNC @%s:", _fn);
    if (argc > 0) {
        out_asm("\t%s.arg %s", _fn, cat_strs(buf, args, argc));
    }
    if (varc > 0) {
        out_asm("\t%s.var %s", _fn, cat_strs(buf, vars, varc));
    }
```

```
    out_inc("; ==== begin function `%s` ====", _fn);
    out_inc("%%define %s.argc %d", _fn, argc);
    out_inc("\n%%MACRO $%s 0\n"
            "   CALL @%s\n"
            "   ADD ESP, 4*%s.argc\n"
            "   PUSH EAX\n"
            "%%ENDMACRO",
            _fn, _fn, _fn);
    if (argc) {
        out_inc("\n%%MACRO %s.arg %s.argc", _fn, _fn);
        for (i = 0; i < argc; i++) {
            out_inc("\t%%define %s [EBP + 8 + 4*%s.argc - 4*%d]",
                        args[i], _fn, i+1);
        }
        out_inc("%%ENDMACRO");
    }
    if (varc) {
        out_inc("\n%%define %s.varc %d", _fn, varc);
        out_inc("\n%%MACRO %s.var %s.varc", _fn, _fn);
        for (i = 0; i < varc; i++) {
            out_inc("\t%%define %s [EBP - 4*%d]",
                        vars[i], i+1);
        }
        out_inc("\tSUB ESP, 4*%s.varc", _fn);
        out_inc("%%ENDMACRO");
    }
}

void write_func_tail() {
    int i;

    out_asm("ENDFUNC@%s\n", _fn);

    out_inc("\n%%MACRO ENDFUNC@%s 0\n\tLEAVE\n\tRET", _fn);
    for (i = 0; i < argc; i++) {
        out_inc("\t%%undef %s", args[i]);
    }
    for (i = 0; i < varc; i++) {
        out_inc("\t%%undef %s", vars[i]);
    }
    out_inc("%%ENDMACRO");
    out_inc("; ==== end function `%s`   ====\n", _fn);

    argc = 0;
    varc = 0;
}
```

```
void quit_parser() {
    fclose(yyin); fclose(asmfile); fclose(incfile);
}
```

词法分析文件 scanner.l 不变，和第 14 章的 TinyC 前端 1.0 版的相同。

将以上 scanner.l, parser.y, test.c 三个文件放在同一目录，输入以下命令生成 TinyC 前端 tcc-frontend :

```
flex scanner.l
bison -vdty parser.y
gcc -o tcc-frontend lex.yy.c y.tab.c
```

再输入:

```
./tcc-frontend test.c
```

将利用 tcc-frontend 编译 test.c ，生成 Pcode 文件 test.asm 以及宏文件 test.inc 。对比一下前面的 test.pcode 和 test.funcmacro 文件，二者几乎是一模一样的。

16.2 TinyC 编译器

现在可以将 TinyC 前端和 TinyC 后端整合起来了。新建一个空的 tinyc 目录，然后 cd 到此目录，之后新建一个 sources 目录，然后将以下 7 个文件放到 sources 目录下:

- scanner.l ， 词法分析文件，和上一节相同;
- parser.y ， 语法分析文件，和上一节相同;
- pysim.py ， Pcode 模拟器（ python 程序），和第 4 章相同;
- tio.c ， 库函数文件，和上一章最后一节相同;
- macro.inc ， NASM 宏文件，和上一章最后一节相同;
- tcc ， 编译 TinyC 源程序的脚本文件;

`pysimulate` ， 模拟运行 Pcode 的脚本文件。

然后在 tinyc 目录下新建一个脚本文件 `build.sh` ，内容如下：

```
mkdir -p release
flex sources/scanner.l
bison -vdty sources/parser.y
gcc -o release/tcc-frontend lex.yy.c y.tab.c
rm -f y.* lex.*
gcc -m32 -c -o tio.o sources/tio.c
ar -crv release/libtio.a tio.o > /dev/null
rm -f tio.o
cp sources/macro.inc sources/pysim.py sources/tcc sources/pysimulate
chmod u+x release/tcc release/pysimulate
export PATH=$PATH:$PWD/release
echo "export PATH=\$PATH:$PWD/release" >> ~/.bashrc
```

在终端输入 **source build.sh** 将编译生成 TinyC 前端 tcc-frontend 、库文件 libtio.a ，并放在 release 目录下，同时将 macro.inc, pysim.py, pysimulate, tcc 这四个文件拷贝至 release 目录，最后，将 release 目录输出到 PATH 环境变量中。现在，在终端输入 tcc filename.c 就可以利用 TinyC 编译成可执行程序了，而输入 pysimulate filename.asm -da 则可以用 Pcode 模拟器单步调试中间代码 Pcode 了。

让我们来测试一下第一章的示例代码 `test.c` 吧，将其放在当前目录，然后在终端输入 tcc test.c ，将生成一个 test-c-build 目录，此目录中包含了中间代码文件 test.asm 、函数定义宏文件 test.inc 、目标文件 test.o 、最终的可执行文件 test 。可以输入 test-c-build/test 来运行可执行文件，也可以输入 pysimulate test-c-build/test.asm -da 用 Pcode 模拟器单步调试中间代码。

脚本文件 tcc 首先调用 tcc-frontend 将输入文件（假设为 test.c ）编译为 test.asm 和 test.inc ，然后调用 nasm ，将 test.asm 、test.inc 和 macro.inc 三个文件一起汇编成 test.o ，最后调用 ld 将 test.o 和 libtio.a 一起链接为最终的可执行程序 test 。 tcc 的内容如下：

```
#!/usr/bin/env bash
```

```
if [ $# ≠ 1 ];
then
    echo "Usage: $0 <filename>"
    exit 1
fi

if ! [ -f $1 ];
then
    echo "Error: File $1 does NOT exists."
    exit 1
fi

tccdir=$(dirname $0)
filename=${1%.*}
fileext=${1##*.}
objdir=$filename-$fileext-build

"$(dirname $0)/tcc-frontend" $1
nasm -f elf32 -P"$tccdir/macro.inc" -P"$filename.inc" -o "$filename.
ld -m elf_i386 -o "$filename" "$filename.o" -L"$tccdir" -ltio
mkdir -p "$objdir"
mv "$filename.asm" "$filename.inc" "$filename.o" "$filename" "$objdi
```

脚本文件 pysimulate 将调用 python 和 pysim.py 文件，模拟运行输入的 Pcode 文件，其内容如下：

```
#!/usr/bin/env bash

if [[ ($# ≠ 1) && ($# ≠ 2) ]];
then
    echo "Usage: $0 <filename> [-da]"
    exit 1
fi

if ! [ -f $1 ];
then
    echo "Error: File $1 does NOT exists."
    exit 1
fi

python "$(dirname $0)/pysim.py" $1 $2
```

下面来测试一下第 14 章最后的测试文件包 `samples.zip` ，将其解包至 samples 目录，再在当前目录新建一个脚本文件 `testall.sh` ，内容如下：

```
for src in $(ls samples/*.c)
do
        filename=${src%.*}
        fileext=${src##*.}
        filenakedname=${filename##*/}
        objdir=$filename-$fileext-build

        clear
        echo build \"$src\" and run
        echo
        tcc "$src"
        "$objdir/$filenakedname"
        echo
        echo press any key to continue...
        read -n 1
done
```

最后在终端输入 bash testall.sh 将对所有文件进行编译、运行。

至此 TinyC 编译器全部完成。

第 16 章完