第13章 用 bison 做语法分析

上一章介绍了 LR(1) 的构造过程，也给出了非常完整的构造流程，但用代码来实现这个构造流程还是很不容易的，不过已经有了非常优秀的工具可以利用了，那就是 bison ，本章介绍如何使用 bison 进行语法分析。

13.1 bison 简介

bison 是一个语法分析器的生成器，网址为：

http://www.delorie.com/gnu/docs/bison/bison_toc.html

bison 和 flex 配合使用，它可以将用户提供的语法规则转化成一个语法分析器。简单来说，我们在上一章中，从语法的 4 个产生式开始，通过一系列的复杂的构造流程，最终得到了一个动作表，然后又利用这个动作表去解析句子。 bison 就是干这个事情的，它读取用户提供的语法的产生式，生成一个 C 语言格式的 LALR(1) 动作表，并将其包含进一个名为 **yyparse** 的 C 函数，这个函数的作用就是利用这个动作表来解析 token stream ，而这个 token stream 是由 flex 生成的词法分析器扫描源程序得到。

13.2 bison 示例 1

上面这段话可能不太容易理解，还是来看一个简单的例子吧。首先安装 bison ，在终端输入：

```
sudo apt-get install bison
```

安装完成后，新建一个文本文件，输入以下内容：

```
%{
#include "y.tab.h"
%}

%%
[0-9]+          { yylval = atoi(yytext); return T_NUM; }
[-/+*()\n]      { return yytext[0]; }
.               { return 0; /* end when meet everything else */ }
%%

int yywrap(void) {
```

```
    return 1;
}
```

将此文件另存为 **calc.l** 。注意此文件中的 **%%** 、 **%{** 、 **%}** 的前面不能有任何空格。

再新建一个文本文件，输入以下内容：

```
%{
#include <stdio.h>
void yyerror(const char* msg) {}
%}

%token T_NUM

%left '+' '-'
%left '*' '/'

%%

S   :   S E '\n'        { printf("ans = %d\n", $2); }
    |   /* empty */     { /* empty */ }
    ;

E   :   E '+' E         { $$ = $1 + $3; }
    |   E '-' E         { $$ = $1 - $3; }
    |   E '*' E         { $$ = $1 * $3; }
    |   E '/' E         { $$ = $1 / $3; }
    |   T_NUM           { $$ = $1; }
    |   '(' E ')'       { $$ = $2; }
    ;

%%

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

将此文件另存为 **calc.y** 。注意此文件中的 **%%** 、 **%{** 、 **%}** 的前面也不能有任何空格。

将前面两个文件都放在终端的当前目录，再在终端输入：

```
bison -vdty calc.y
```

此时可以发现终端下多了三个文件： y.tab.h, y.tab.c, y.output 。

再在终端输入：

```
flex calc.l
```

此时终端下又多了一个文件： lex.yy.c 。

最后将 y.tab.c 和 lex.yy.c 一起编译并运行一遍：

```
gcc -o calc y.tab.c lex.yy.c
./calc
```

然后在终端输入算术表达式并回车：

```
1+2+3
ans = 6
2*(2+7)+8
ans = 26
```

可以发现回车后，终端会自动输出算术表达式的结果。这个程序就是一个简单的支持加、减、乘、除以及括号的整数计算器。想象一下，如果用 C 语言手工编写一个同样功能的程序，那代码量肯定很大吧。

下面再来详细的解释一下 calc.l 和 calc.y 代码。

calc.l 文件就是一个词法分析器（或者说扫描器），在第 8 章中已经介绍了 flex 的语法。该扫描器扫描标准输入（键盘），将其分割为一个个的 token ，从其代码中可以看出，它将整数扫描为一个 T_NUM 型的 token ，而将 “-/+*()n” 这些字符扫描为一个单字符 token （其 token_type 的值就是该字符的 ASCII 码），任何其他字符都会被扫描为一个值为 0 的 token 。

再来看 calc.y 文件，这个就是 bison 的自定义语法文件，其格式和 flex 分词模式文件的格式非常相似，共分为 4 段，如下：

```
%{
Declarations
%}
Definitions
%%
Productions
%%
User subroutines
```

其中的 Declarations 段和 User subroutines 和 flex 文件中是一样的， bison 会将这些代码原样的拷贝到 y.tab.c 文件中； Definitions 段和 flex 中的功能也差不多，也是在这个段定义一些 bison 专有的变量，稍后再解释这个文件中

的这个段里的代码；最重要的是 Productions 段，这里面是用户编写的语法产生式，这个文件里定义的产生式用常规的方式书写的格式如下：

```
S -> S E \n | ε
E -> E + E | E - E | E * E | E / E | T_NUM | ( E )
```

bison 里面 ":" 代表一个 "->" ，同一个非终结符的不同产生式用 "|" 隔开，用 ";" 结束表示一个非终结符产生式的结束；每条产生式的后面花括号内是一段 C 代码、这些代码将在该产生式被应用时执行，这些代码被称为 action ，产生式的中间以及 C 代码内部可以插入注释（稍后再详细解释本文件中的这些代码）；产生式右边是 ε 时，不需要写任何符号，一般用一个注释 /* empty */ 代替。

bison 会将 Productions 段里的第一个产生式的左边的非终结符（本文件中为 S ）当作语法的起始符号，同时，为了保证起始符号不位于任何产生式的右边， bison 会自动添加一个符号（如 S' ）以及一条产生式（如 S' -> S ），而将这个新增的符号当作解析的起始符号。

产生式中的非终结符不需要预先定义， bison 会自动根据所有产生式的左边来确定哪些符号是非终结符；终结符中，单字符 token （ token type 值和字符的 ASCII 码相同）也不需要预先定义，在产生式内部直接用单引号括起来就可以了（如本文件中的 'n', '+', '-' 等），其他类型的 token 则需要预先在 Definitions 段中定义好，如本文件中的 **token T_NUM**， bison 会自动为这种 token 分配一个编号，再写到 **y.tab.h** 文件中去，打开该文件，可以看到如下代码：

```
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
    enum yytokentype
    {
        T_NUM = 258
    };
#endif
/* Tokens.  */
#define T_NUM 258
```

因此在 calc.l 文件中包含此文件就可以使用 T_NUM 这个名称了。

可以在 Definitions 段定义符号的优先级，本文件中，定义了各运算符的优先级，如下：

```
%left '+' '-'
%left '*' '/'
```

其中的 **%left** 表明这些符号是左结合的。同一行的符号优先级相同，下面行的符号的优先级高于上面的。

bison 会将语法产生式以及符号优先级转化为一个 C 语言的 LALR(1) 动作表，并输出到 y.tab.c 文件中去，另外，还会将这个动作表以及语法中的相关要素以可读的文字形式输出到 y.output 文件中去，该文件中内容如下：

```
Grammar

    0 $accept: S $end

    1 S: S E '\n'
    2  | %empty

    3 E: E '+' E
    4  | E '-' E
    5  | E '*' E
    6  | E '/' E
    7  | T_NUM
    8  | '(' E ')'

......

State 0

    0 $accept: . S $end

    $default  reduce using rule 2 (S)

    S  go to state 1


State 1

    0 $accept: S . $end
    1 S: S . E '\n'

    $end   shift, and go to state 2
    T_NUM  shift, and go to state 3
    '('    shift, and go to state 4

    E  go to state 5

......
```

上面 state x 等表示一个状态以及该状态里面的所有形态，以及该状态的所有动作，由于采用了 LALR(1) 方法构造动作表，因此这些状态里的形态数量和用 LR(1) 法构造的有所不同。

bison 将根据自定义语法文件生成一个函数 **int yyparse (void)** （在 y.tab.c 文件中），该函数按 LR(1) 解析流程对词法分析得到的 token stream 进行解析，每当它需要读入下一个符号时，它就执行一次 **x = yylex()** ，每当它要执行一个折叠动作时，这个折叠动作所应用的产生式后面的花括号里面的 C 代码将被执行，执行完后才将相应的状态出栈。

若 token stream 是符合语法的，则解析过程中不会出错， yyparse 函数将返回 0 ，否则，该函数会在第一次出错的地方终止，并调用 yyerror 函数，然后返回 1 。

yyparse 函数不仅维持一个状态栈，它还维持一个符号属性栈，当它执行 shift 动作时，它除了将相应的状态压入状态栈之外，还会将一个类型为 YYSTYPE （默认和 int 相同）、名为 yylval 的全局变量的数值压入到属性栈内，而在 reduce 动作时，可以用 **$1, $2, ... $n** 来引用属性栈的属性，reduce 动作不仅将相应的状态出栈，还会将同样数量的属性出栈，这些属性和 reduce 产生式的右边的符号是一一对应的，同时，用 **$$** 代表产生式左边的终结符，在 reduce 动作里可以设置 $$ 的值，当执行 goto 动作时，除了将相应的状态入栈，还会将 $$ 入栈。

以本程序为例：

（1） 当执行 yylex 函数时（在 calc.l 文件里），在扫描到一个完整的整数后， **yylval = atoi(yytext)** 将被执行，并返回 T_NUM ；

（2） 当 yyparse 执行 x = yylex() 后，将压入 yylval 的值，如果返回的 x 是 T_NUM，那这个符号已经和（1）中的 atoi(yytext) 这个值绑定起来了；

（3） 当 yyparse 执行 reduce E -> T_NUM 以及后面的 goto 动作时， **$$ = $1** 被执行，$1（绑定到 T_NUM 的值）将出栈，$$（=$1）将入栈，故符号 E 也被绑定了一个数值；

（4） 当 yyparse 执行 reduce E -> E - E 以及后面的 goto 动作时， **$$ = $1 - $3** 被执行，同时 $1 ~ $3 将出栈， $$ 将入栈，相当于左边的 E 被绑定了右边的两个 E 的值的差；

（5） 当 yyparse 执行 reduce S -> S E n 以及后面的 goto 动作时， **printf(“ans = %dn”, $2)** 被执行，于是绑定到 E 的数值被打印出来。

以下为 yyparse 函数的基本解析流程：

（1） 将初始状态 state0 压入栈顶；

（2） 执行 x = yylex() ，有两种情况：

（2.1） x 不是终结符：输入不合语法，执行 deny 操作，调用 yyerror 函数，返回 1；

（2.2） x 是终结符：转到（3）；

（3） 置 X = x；

（4） 设栈顶状态为 I ，有以下五种情况：

（4.1） M[I, X] 为 shift I’ 动作：执行 shift I’ 操作：

将 I’ 压入栈顶，将 yylval （可能在（2）中被赋值）压入属性栈，转到（2）；

（4.2） M[I, X] 为 goto I’ 动作：执行 goto I’ 操作：

将 I’ 压入栈顶，将 $$ （可能在（4.3）中被赋值）压入属性栈，转到（3）；

（4.3） M[I, X] 为 reduce A -> X1 X2 ... Xn ： 执行 reduce(A, n) 操作，具体步骤为：

（4.3.1） 执行相应产生式 A -> X1 X2 ... Xn 后面的 C 代码；

（4.3.2） 将栈顶及以下 n 个状态出栈，将属性栈顶及以下 n 个属性出栈，置 X = A ；

（4.3.3） 转到（4）；

（4.4） M[I, X] 为 ACCEPT ： 执行 accept 操作，返回 0 ；

（4.5） M[I, X] 为空白：执行 deny 操作，调用 yyerror 函数，返回 1。

以上流程只是基本流程， bison 会对以上流程进行一些优化以加快解析速度，但大体的流程、相关动作执行的先后顺序以及栈的操作方式是和上面描述的一样的。

以上流程中：如果在第（2）步（ yylex 函数内、也就是 flex 文件的 action 中）对 yylval 赋值，那么这个值将和 yylex 返回的终结符绑定；如果在第（4.3）步中（也就是 bison 文件的 action 中）对 $$ 进行赋值，那么这个值将和此 action 的产生式的左边的非终结符绑定；而 bison 文件的 action 中，可以用 $1, $2, ..., $n 来引用和此 action 的产生式右边的第 1 ~ n 个符号所绑定的值。

13.3 bison 示例 2

再来看一个稍微复杂一点的示例，一共有 4 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);
%}

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

%%
{OPERATOR}      { return yytext[0]; }
{INTEGER}       { yylval = strdup(yytext); return T_IntConstant; }
{IDENTIFIER}    { yylval = strdup(yytext); return T_Identifier; }
{WHITESPACE}    { /* ignore every whitespcace */ }
\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_IntConstant T_Identifier

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

%%

S   :   Stmt
    |   S Stmt
    ;

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

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**

```
CC = gcc
OUT = tcc
OBJ = lex.yy.o y.tab.o
SCANNER = scanner.l
```

```
PARSER = parser.y

build: $(OUT)

run: $(OUT)
	./$(OUT) < test.c > test.asm

clean:
	rm -f *.o lex.yy.c y.tab.c y.tab.h y.output $(OUT)

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

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

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

测试文件： **test.c**

```
a = 1 + 2 * ( 2 + 2 );
b = c + d;
e = f + 7 * 8 / 9;
```

这个示例对第一个示例进行了一些扩充。

词法分析文件中：

增加了 T_Identifier 类型的 token ，整数类型的 token 名改为了 T_IntConstant ；

增加了一个全局变量 cur_line ，表示扫描所在的位置（行）；

增加了错误处理函数 unrecognized_char 和 yyerror 函数，前者在扫描到非法字符时被执行，并将相关信息传递给后者，后者则打印出错误信息以及当前位置，并退出程序；

第二行增加了 **define YYSTYPE char *** ，上一节中说过了，全局变量 yylval 的类型是 YYSTYPE ，而 YYSTYPE 默认为 int ，添加了这一行后， YYSTYPE 变成了 char * ，这样 yylval 的类型就变成了 char * 了；

当扫描到完整的整数或标识符时， **yylval = strdup(yytext)** 被执行，扫描到的字符串被拷贝一份传给

了 yylval ，到语法分析时，这个字符串将被绑定到终结符 T_IntConstant 或 T_Identifier 上面。

语法分析文件中：

语法规则有所扩充，增加了终结符 T_Identifier 和非终结符 Stmt ， Stmt 可表示一个赋值表达式；

符号优先级那一段，增加了 **%left U_neg** ，表示增加一个符号，该符号为左结合，且优先级在 * 和 / 之上，这个符号在 **'-' E %prec U_neg** 那一行被使用，%prec 命令可以给一个产生式定义一个优先级，"%prec U_neg" 表示这个产生式的优先级和 U_neg 一样（而不是等于产生式中最右边的、定义了优先级的符号的优先级），当出现 shift/reduce 冲突时，将利用 U_neg 的优先级和 lookahead 的优先级比较，然后根据比较结果来选择动作；

大部分产生式的后面的 action 都是执行一个 printf 函数，表示这些产生式被 reduce 时所打印的字符串。

makefile 里面是编译这个程序的命令，在终端输入 make 后，将编译生成可执行文件 tcc ，然后用 test.c 文件来测试一下：

```
./tcc < test.c > test.asm
```

test.asm 文件中的输出内容如下：

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

push c
push d
add
pop b

push f
push 7
push 8
mul
push 9
```

```
div
add
pop e
```

可以看出 test.c 文件里的所有赋值表达式都被转换成相应的 Pcode 了，是不是很神奇？这个程序相当于我们的 TinyC 前端的一个雏形了。在这个雏形前端中请注意源文件的解析过程中各产生式折叠的先后顺序，中间代码就是按照产生式折叠的顺序生成的。

第 **13** 章完