高级编程工具

曹东刚

caodg@sei.pku.edu.cn

Linux 程序设计环境

http://c.pku.edu.cn/

内容提要

yacc

- Yacc: "Yet Another Compiler Compiler"

  是 Unix 系统中标准的生成语法分析程序的程序

  - 基于 BNF 范式
  - 生成 C 语言代码
  - 生成的语法分析程序需要一个词法分析器 (lexical analyzer)

- 多种 Yacc 工具, 如 GNU 的 bison

lex

- lex 是 Unix 系统中标准的生成词法分析器的程序
- IEEE POSIX P1003.2 对 Lex 和 Yacc 进行了标准化
- 多种 Lex 工具, 如 GNU 的 flex

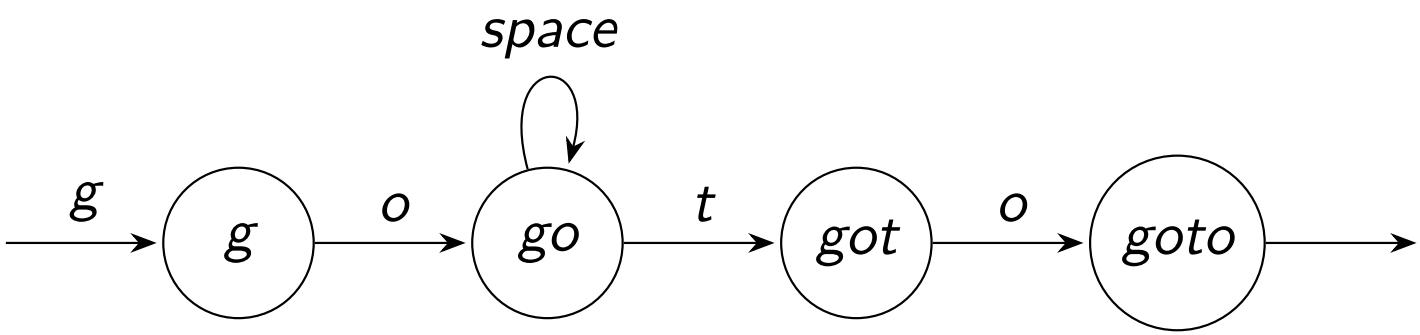
flex

- flex 基于有限状态识别机生成词法分析器
- flex 输入文件通过若干规则描述词法分析器
  - 规则：正则表达式和对应的 C 代码
- flex 缺省生成一个名为 'lex.yy.c' 的 C 文件，该文件的 'yylex()' 函数扫描输入文本
- 编译该文件并和库 libfl.a/libl.a 链接，生成可执行文件
- 该文件执行时，对遇到的每一个规则，执行对应的 C 代码

Finite States Recognizers

FSR 是一种有限自动机, 定义了若干接受状态 (accepting states). 如果存在一条路径从初始状态到接受状态, 则称输入可接受.

例: 识别程序文本中的 “go to”, “goto”



生成的词法分析程序工作过程 —1

- 读取输入, 对输入字符串搜寻匹配的模式
  - 若发现多个匹配模式, 则选择匹配最长字符串的那个
  - 若模式匹配字符串长度相同, 则选择第一个模式

生成的词法分析程序工作过程 —2

- 确定匹配模式后
  - 全局字符串指针 “yytext” 指向匹配字符串，全局变量 “yyleng” 定义了字符串长度
  - 执行该模式的关联动作
- 若没有发现匹配模式，则执行缺省动作：将输入复制到输出

flex 输入文件规格

flex 输入文件由 3 部分组成

- 定义
- 规则
- 用户代码

每部分之间由 %% 分隔

flex 输入文件模板

flex 模板

```
%{
user-file-prologue
%}
definitions
%%
%{
user-yylex-prologue
%}
regular-expression-1 action-1
...
%%
user-epilogue
```

flex 输入文件例子

flex 示例

```
1   %{
2   int num_lines = 0, num_chars = 0;
3   %}
4   %%
5   \n      ++num_lines; ++num_chars;
6   .       ++num_chars;
7
8   %%
9   main()
10  {
11      yylex();
12      printf( "# of lines = %d, # of chars = %d\n",
13              num_lines, num_chars );
14  }
```

定义部分

为了简化输入文件, 可以在定义部分声明若干名字定义, 以及若干开始条件. 名字定义的形式为

name definition

其中

name: [a-zA-Z_][a-zA-Z0-9_-]*

definition: 从 name 后第一个非空白字符开始至行尾

可通过{name} 引用 definition. 例:

```
1  DIGIT    [0-9]
2  ID       [a-z][a-z0-9]*
3  FLOAT    {DIGIT}+"."{DIGIT}*
```

规则部分和用户代码部分

- 规则部分定义了若干“模式–动作”对

  ```
  pattern    action
  ```

  其中“pattern”前面不能有缩进,“action”必须和“pattern”在同一行
- 用户代码部分被复制到生成的文件‘lex.yy.c’中
  - 在此部分可以定义调用词法分析器/被词法分析器调用的函数,如“main”函数
  - 如果没有用户代码,在此部分连同前面的%% 无需声明

最简单的 flex 定义

```
%%
```

prologue

在定义部分和规则部分可以定义序言 (prologue)

prologue: 缩进文本或 '%{' 与 '}%' 之间的文本

'%{' 与 '}%' 不能缩进

- 定义部分的序言常用于声明全局变量, 规则动作要用到的 头文件等

  定义部分的序言被 flex 复制到生成的文件 'lex.yy.c' 中
- 规则部分的序言在规则部分开始声明

  规则部分的序言定义词法分析器用到的局部变量, 词法分析器的 初始化动作等

规则

规则部分使用扩展的正则表达式

x	匹配字符 'x'
.	匹配除换行外的任意单个字符
[xyz]	匹配方括号中的任意字符
[abj-oZ]	匹配字符 'a', 'b', 从 'j' 到 'o' 的字符, 或 'Z'
[^A-Z]	不匹配从 'A' 到 'Z' 的字符
[^A-Z\n]	不匹配从 'A' 到 'Z' 的字符以及换行符
{name}	引用定义部分定义的模式
r*	r 是正则表达式, 匹配 r 任意多次
r+	匹配 r 1 次或多次
r?	匹配 r 0 次或 1 次

规则 (cont.)

r{2,5}	匹配 r 2 次到 5 次
r{2,}	匹配 r 2 次以上
r{4}	匹配 r 4 次
\x	ANSI-C 控制字符或 'x'
\0	字符 NUL
\123	8 进制字符
\x2a	16 进制字符, 值为 2a
(r)	模式组合
rs	正则表达式 r 后面跟随正则表达式 s
r|s	或者 r 或者 s
^r	匹配一行开头的 r
r$	匹配一行结尾的 r, 等价于 "r/\n"
r/s	匹配 r 仅当 r 后面紧跟 s. 这种模式称为 "trailing context"

开始条件中的模式

<s>r	匹配开始条件 s 中的 r
<s1,s2,s3>r	匹配开始条件 s1, s2, 或者 s3 中的 r
<*>r	匹配任意开始条件中的 r
<<EOF>>	文件结束符
<s1,s2><<EOF>>	处于开始条件 s1 或 s2 时的文件结束符

动作

- 动作可以为空，此时对输入字符串不处理
- 如果动作中包含 '{'，则与之匹配的 '}' 之前的内容都是动作内容
- 如果动作只有一个竖线 '|'，表示和下一条规则的动作相同
- 动作可包含任意 C 代码，包括返回语句，用于从yylex() 中返回一个结果
  - 'yylex()' 被调用时，从上次的输入位置开始扫描，直到文件结束或执行了返回动作 return

几条特殊指令

- ECHO：将 yytext 复制到词法分析器的输出
- BEGIN start-condition：让词法分析器处于start-condition 状态
- REJECT：让词法分析器选择第二符合条件 (second best) 的规则
  - 匹配输入字符串，或者输入字符串的前缀

特殊指令: 例 1

```
1  %{
2  int word_count = 0;
3  %}
4  %%
5  frob        special(); REJECT;
6  [^ \t\n]+   ++word_count;
```

结果: 对 'frob' 执行 'special()', 并统计单词个数

特殊指令: 例 2

```
1  %%
2  a          |
3  ab         |
4  abc        |
5  abcd       ECHO; REJECT;
6  .|\n       /* eat up any unmatched character */
```

对于输入字符串 “abcd”, 输出结果为 “abcdabcaba”

开始条件

flex 提供了一种条件激活规则的机制:

当规则的模式以<sc> 为前缀时, 则只有当扫描程序处于“sc” 状态时才会激活该规则. 例:

```
<STRING>[^"]*        { /* eat up string ... */ }
```

- 开始条件在定义部分, 非缩进, 以%s 或者 %x 声明
- 开始条件用 “BEGIN condition” 激活, 直到下一个 BEGIN 指令

包含性与排他性开始条件

- %s：包含性开始条件，没有声明开始条件的规则也会被激活
- %x：排他性开始条件，只有声明了对应开始条件的规则会激活
- <*> 匹配任何开始条件

开始条件示例

```
1  %s example
2  %%
3
4  <example>foo   do_something();
5  bar            something_else();
```

等价于

```
1  %x example
2  %%
3
4  <example>foo   do_something();
5  <INITIAL,example>bar   something_else();
```

开始的例子: goid.l –1

goid.l

```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  const char *yylval = NULL;
5  enum token_e { token_goto = 1, token_identifier };
6  %}
7
8  %%
9  go" "*to    return token_goto;
10 [a-zA-Z_][a-zA-Z0-9_]* { yylval = yytext;
11                         return token_identifier;
12                         }
13 [^ \t\n]*
```

开始的例子: goid.l –2

goid.l

```
14  %%
15  int main (void)
16  {
17      enum token_e token;
18      while ((token = yylex ()))
19          if (token == token_goto)
20              printf ("Saw a goto.\n");
21          else
22          if (token == token_identifier )
23              printf ("Saw an identifier (%s).\n",
24                yylval);
25          printf ("End of file.\n");
26      return 0;
27  }
```

开始的例子: 编译运行

```
1  $ flex -o goid.c goid.l
2
3  $ gcc -Wall -W goid.c -lfl -o goid
4    goid.c:976: warning: `yyunput' defined but not used
5
6  $ echo 'gotoo goto go to go tooo' | ./goid
```

内容提要

make

make 是 Unix 环境大型软件开发的重要工具.

- 自动管理、检查文件之间的倚赖关系
- 自动判断哪些文件要重新编译, 调用外部程序进行处理
  - 根据文件的修改时间
- 常用于编译源文件生成目标文件, 将目标文件链接成可执行文件或库

makefile

- 用文件 “makefile” 或 “Makefile” 描述倚赖和动作，动作由 shell 执行
- 命令 make 解释 “makefile”
- GNU make

hello 的 makefile

```
1 hello: hello.c
2 	gcc hello.c -o hello
```

编译:

```
$ make
gcc hello.c -o hello
```

目标和倚赖

makefile 由如下的一系列规则组成

```
1  target1 target2 target3 : prerequisite1, prerequisite2
2      ↹command1
3      ↹command2
```

目标和倚赖说明

- 目标 (target): 要做的事情, 要生成的文件
- 倚赖 (prerequisite): 在生成目标前, 其所有倚赖必须存在
- 命令 (command): 根据依赖生成目标的 shell 命令. 命令前必须是缩进 (tab)
- makefile 中的第一个规则称为缺省目标 (goal)

工作过程

- 如果在命令行给出了目标，则 make 找到该目标的规则；否则执行缺省目标
- 对于每个规则，首先查看所有的倚赖和目标
  - 若某个依赖有规则，则首先处理该依赖的规则
  - 若某个依赖的时间比目标新，则执行命令更新目标
  - 命令由 shell 执行，若执行错误，则中止处理

示例 –1: lexerl.l

lexerl.l

```
1  %{
2  int fee_count = 0;
3  int fie_count = 0;
4  int foe_count = 0;
5  int fum_count = 0;
6  %}
7  %%
8  fee fee_count++;
9  fie fie_count++;
10 foe foe_count++;
11 fum fum_count++;
12 .
13 \n
```

示例 –1: count_words.c

count_words.c

```
1  #include <stdio.h>
2
3  extern int fee_count, fie_count, foe_count, fum_count;
4  extern int yylex( void );
5
6  int main( int argc, char ** argv )
7  {
8      yylex();
9      printf( "%d %d %d %d\n", fee_count, fie_count,
10         foe_count, fum_count );
11     exit( 0 );
12 }
```

示例 –1: makefile

makefile

```
1  count_words: count_words.o lexer.o -lfl
2      gcc count_words.o lexer.o -lfl -o count_words
3
4  count_words.o: count_words.c
5      gcc -c count_words.c
6
7  lexer.o: lexer.c
8      gcc -c lexer.c
9
10 lexer.c: lexer.l
11     flex -t lexer.l > lexer.c
```

规则

- 显式规则 (explicit rule): makefile 中显式声明的规则, 如

  `vpath.o variable.o: make.h config.h dep.h`

- 模式规则 (pattern rule): 用通配符取代显式的文件名, 跟 Bourne sh 相同, 如

  `~ * ? [...] [^...]`

- 隐式规则 (implicit rule): make 内置的模式规则或后缀规则
- 在 GNU make 中, 后缀规则可被模式规则代替

phony 目标

phony 目标: 只是标记一个要执行的命令脚本, 并不表示一个文件. 如

```
1 clean:
2     rm -f *.o lexer.c
```

为了避免正好有个文件名为 “clean”

```
1 .PHONY: clean
2 clean:
3     rm -f *.o lexer.c
```

标准的 phony 目标

目标	功能
all	执行所有任务
install	根据编译生成的二进制代码，将应用安装到系统中
clean	删除所有生成的二进制代码
distclean	删除所有不在原始源文件包中的文件
info	根据 Texinfo 文件生成 GNU 的 info 文件
check	执行该应用的测试脚本

phony 示例

```
1  $(Program): build_msg $(OBJECTS) $(LIBDEP)
2      $(RM) $@
3      $(CC) $(LDFLAGS) -o $(Program) $(OBJECTS) $(LIBS)
4      ls -l $(Program)
5      size $(Program)
6  .phony: build_msg
7  build_msg:
8      @printf "#\n# Building $(Program) \n#\n"
```

空目标

空目标的目的与 phony 类似, 用于执行控制命令.

技巧: 利用一个空的目标文件

```
1  prog: size prog.o
2      $(CC) $(LDFLAGS) -o $@ $^
3
4  size: prog.o
5      size $^
6      touch size
```

变量

在 makefile 中可以定义变量: Name = Value

随后通过$(Name) 或${Name} 访问

make 的自动变量

$@	目标文件名
$%	档案文件 (库) 的成员
$<	第一个依赖文件的文件名
$?	所有比目标文件新的倚赖文件名列表, 以空格分隔
$^	所有倚赖文件名列表, 以空格分隔
$+	和$^ 类似, 包含重复文件名
$*	目标文件名去除后缀后的部分

示例

makefile

```
1  count_words: count_words.o counter.o lexer.o -lfl
2      gcc $^ -o $@
3  count_words.o: count_words.c
4      gcc -c $<
5  counter.o: counter.c
6      gcc -c $<
7  lexer.o: lexer.c
8      gcc -c $<
9  lexer.c: lexer.l
10     flex -t $< > $@
```

make 的替代工具 —1

- ant, 应用于 Java 文件开发, 使用 XML 格式的描述文件
- cook, 很强大, 语法与 C 类似
- dmake, 分布 make, Sun 公司用来编译 StarOffice, Solaris
- jam, 与 make 相似, 但是进行了增强
- mk, 为 Plan 9 开发, 据说去掉了 make 的缺点, 比 make 强大易用

make 的替代工具 —2

- MPW Make, 为 Mac OS Classic 开发, 和 Unix make不兼容
- Module::Build, 用于编译和安装其它 Perl 模块
- NAnt, 类似于ant, 用于.NET 平台
- maven, 支持模块化编译, 支持网络下载依赖