

用于 C 语言的错误处理预编译器

施海昕, 余 堃

(电子科技大学 计算机科学与工程学院, 四川 成都 610054)

(sinwooo@126.com)

摘 要:根据编译原理,设计并实现了一个用于 C 语言的错误处理预编译器。通过扫描程序代码,它可以找到特定格式的注释,并从中提取错误名称、错误信息和错误号,最终生成对应的错误处理代码。这个预编译器改进了 C 语言错误处理机制,提高了软件开发效率。

关键词:有限自动机;状态转换图;词法分析;语法分析

中图分类号: TP314 **文献标识码:** A

Precompiler handling errors for C language

SHI Hai-xin, SHE Kun

(College of Computer Science and Engineering, University of Electronic Science and Technology of China,
Chengdu Sichuan 610054, China)

Abstract: According to the principle of compiling, a precompiler handling errors for C language was designed and realized. Through scanning the source code, it can could find the specific format comments, and get error names, error messages and error numbers from them. Finally it generates generated the corresponding source codes handling errors. The precompiler improved improves the mechanism of handling errors in C language, and increased increases software development efficiency.

Key words: finite automaton; state transition diagram; lexical analysis; syntax analysis

C++、Java 等语言支持异常处理机制,可以很方便地进行错误处理;而 C 语言却不支持异常处理。用 C 语言进行错误处理,需要手工书写 #define 宏定义或 enum 枚举变量把错误名称和错误号绑定在一起,书写一个 GetErrMsg() 函数把错误名称和错误信息绑定在一起。在商业程序中,错误名称的数量往往很大。手工书写冗长的错误处理代码,既效率低下又容易出现拼写错误。另一方面,在程序中使用错误名称的时候,常常需要在后面注释上错误信息和/或错误号,以增加程序的可读性,这就让人想到能否把这些注释看成错误名称的定义。为了解决这些矛盾,本文开发一个错误处理的预编译器。

1 任务描述

错误处理预编译器的任务是扫描程序代码,找到其中特定格式的注释,并从中提取错误名称、错误信息和错误号,最终生成对应的错误处理代码。

我们把关于错误名称的注释定义为以下格式:

```
/* deferr( ERR, ErrMsg [, ErrNo] ) */
```

其中,deferr 是保留字,表示此注释为预编译器可识别的注释;ERR 是错误名称,标识符的格式;ErrMsg 是错误信息,字符串的格式;ErrNo 是错误号,数字常数,可以省略,省略后由预编译器自动生成一个值。

2 有限自动机与正则表达式

2.1 有限自动机

有限自动机是识别 3 型语言(正则语言)的数学方法。它可用作描述在输入串中识别模式的过程,因此能用作构造

词法分析器。有限自动机又分为确定的有限自动机和不确定有限自动机两种。

2.1.1 确定的有限自动机的定义

确定的有限自动机(Deterministic Finite Automaton, DFA),是一个五元组 $Md = (\Sigma, S, s_0, F, \delta)$,其中:

Σ 是一个有限字母表,它的每个一元素称为一个输入符号;

S 是一个有限状态集,它的每一个元素称为一个状态;

$s_0 \in S$, 是唯一的初始状态,简称初态;

F 是 S 的子集,是终止状态,简称终态集;

δ 是 $S \times \Sigma \rightarrow S$ 上的一个映射,称为状态转换函数。如 $\delta(s_i, a) = s_j$, 其中 $s_i, s_j \in S, a \in \Sigma$, 表示在 s_i 状态下,输入字符为 a 时,状态变为 s_j 。

2.1.2 不确定的有限自动机的定义

不确定的有限自动机(Nondeterministic Finite Automaton, NFA),是一个五元组 $Mn = (\Sigma, S, s_0, F, \delta)$,其中 Σ, S, s_0, F 同确定的有限自动机, δ 是 $S \times \Sigma \rightarrow 2^S$ 上的一个映射,即:

$$\delta(s_i, a) = \{s_{i1}, s_{i2}, \dots\}$$

其中 $s_i, s_{i1}, s_{i2}, \dots \in S, \{si1, si2, \dots\} \in 2^S, a \in \Sigma$, 表示在 s_i 状态下,输入字符为 a 时,状态变为 s_{i1} 或 s_{i2} 或 \dots 。

2.2 有限自动机的表示

有限自动机的一种常用表示方法是状态转换图。对于有限自动机 FA,用 m 个节点表示 FAM 的 m 个状态,如果有 $\delta(s_i, a) = \{s_j\}$, 则用有向边连接两个节点 s_i 和 s_j , 有限边上标记输入字符 a , 这样构成的图称为状态转换图。

状态转换图只有唯一的一个初始状态节点和若干个(可

收稿日期:2005-04-13;修订日期:2005-07-11

作者简介:施海昕(1980-),男,硕士研究生,主要研究方向:信息安全、编译理论; 余堃(1967-),男,副教授,主要研究方向:网络计算、信息安全和中间件技术。

以是 0 个) 终止状态节点。初始状态的节点用 \Rightarrow 来标记, 终止状态的节点用双圈来表示。

2.3 正则表达式和正则集

有限自动机接受的语言可以用正则表达式 (Regular Expression) 来描述, 它所表示的字符串集为正则集, 与正则文法产生的正则语言是相同的语言类, 因此正则表达式与正则文法有相同的表达能力, 两者是等价的。而正则表达式给出了字符串的简洁结构表示, 因此通常用正则表达式来描述字符串的词法结构, 在利用正则表达式与有限自动机之间的等价变换, 构造出能识别符合词法结构的字符串的有限自动机, 这边是编译程序中词法分析器的形式化的构造方法。

对于给定的字母表, 有:

1) ϵ 和 Φ 是 Σ 上的正则表达式, 它们所表示的正则集分别为 $\{\epsilon\}$ 和 Φ ;

2) 对任一 $a \in \Sigma$, a 是 Σ 上的正则表达式, 它所表示的正则集为 $\{a\}$;

3) 如果 R 和 S 是 Σ 上的正则表达式, 它们所表示的正则集分别为 $L(R)$ 和 $L(S)$, 则:

$(R \mid S)$ 也是 Σ 上的正则表达式, 它所表示的正则集为 $L(R) \cup L(S)$;

$(R \cdot S)$ 也是 Σ 上的正则表达式, 它所表示的正则集为 $L(R)L(S)$;

$(R)^*$ 也是 Σ 上的正则表达式, 它所表示的正则集为 $(L(R))^*$;

4) 仅有限次使用规则 1)、2)、3) 得到的表达式, 是 Σ 上的正则表达式, 它所表示的集合是 Σ 上的正则集。

上面是用递归方法定义了 Σ 上的正则表达式和正则集, 其中规则 (1) 和 (2) 为基始规则, (3) 为递归规则, (4) 为界限规则。正则表达式之间的运算 “ \mid ” 为 “或”, “ \cdot ” 为 “连接”, “ * ” 为 “闭包”。

3 有限自动机的实现

在构造词法分析的有限自动机之前, 首先需要考虑如何用代码实现一个有限自动机, 也就是说如何把一个有限自动机从状态转换图形式变为代码形式。我们采用的具体方法如下:

1) 把状态转换图的各节点都定义为 StateType 类型的枚举变量的值, 并定义一个表示当前状态的变量 state, 一个表示输入字符的变量 c。

2) 程序流程是用外层的 while 语句判断是否到终止状态, 如果没有到就继续。用内层的 switch - case 语句描述每一个状态转换函数。

例如, 一个不确定的有限自动机用状态转换如图 1 所示。

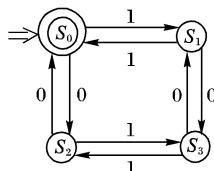


图1 不确定的有限自动机的状态转换图
使用上面介绍的具体方法转换成的代码就是:

```
enum StateType { S0, S1, S2, S3 };
StateType state = S0;
char c;
do {
    c = GetNextChar();
    switch( state ) {
        case S0:
            if ( c == 0 )
                state = S2;
            if ( c == 1 )
```

```
                state = S1;
            break;
        case S1:
            if ( c == 0 )
                state = S3;
            if ( c == 1 )
                state = S0;
            break;
        case S2:
            if ( c == 0 )
                state = S0;
            if ( c == 1 )
                state = S3;
            break;
        case S3:
            if ( c == 0 )
                state = S1;
            if ( c == 1 )
                state = S2;
            break;
    }
} while( state != S0 );
```

4 词法分析

通常一种程序语言中定义的单词种类包括标识符 (identifier)、保留字 (reserved word)、常数 (literal)、运算符 (operator)、界符等。C 语言中定义了属于这五种类型的大量的单词, 但是由于预编译器只识别我们自定义的注释, 因此预编译器处理的单词集只是 C 语言中定义的单词集的一个真子集。

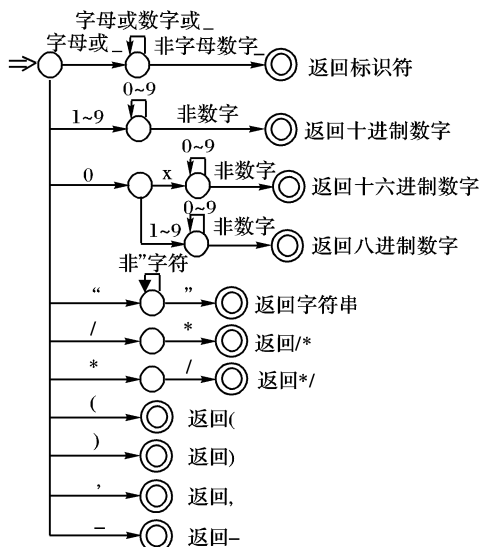


图2 预编译器词法分析的状态转换图

我们可以对自定义的注释格式进行分解:

```
/* deferr( ERR, ErrMsg, ErrNo) */
```

其中: / * 为界符; deferr 为保留字; (为界符; ERR 为标识符; , 为界符; ErrMsg 为字符串; , 为界符; ErrNo 为数字常数;) 为界符; * / 为界符。

通过对自定义的注释格式进行分析, 可以发现预编译器的词法分析需要识别的单词集包括: 界符 / *、界符 (、界符、界符)、界符 * /、保留字 deferr、标识符、字符串、数字常数。另外, 由于数字常数有可能是负数, 所以单词集还应该包括运算符 -。

预编译器词法分析的状态转换如图 2 所示。

其中, 标识符返回后还需要查保留字表, 如果是保留字表中的单词则为保留字, 否则为标识符。另外, 这个有限自动机无法识别的其余单词都被标记为未知单词。

根据这个状态转换图, 我们使用将状态转换图变为代码

