基于有限自动机的词法分析器

基于 C 语言

软件学院

陈睿 141250013

141250013@smail.nju.edu.cn

2016.10.27

目录

1. 目标

主要目标是实现对自定义的程序语言的词法分析器程序构造。我选择了 C 语言当中部分具有代表性的子集，定义了一部分正则表达式，实现了一个简单的词法分析器。同本实验，可以更好地理解从 RE->NFA->DFA 的过程，并应用到具体的程序语言词法分析中。

2. 内容描述

本报告主要描述了一个简单的词法分析器构造过程，包括构思、实现过程中的理论推导、具体的核心算法和数据结构的描述、最终成品、测试结果展示以及在其中遇到的困难与解决方法，还有最后的总结和感受。

3. 构思与方法

从整体思路来看，首先需要确定 C 语言的子集，然后对定义的语言子集进行分类，写出各类别的正则表达式，根据正则表达式手动构建有限自动机，最后根据有限自动机实现代码。

具体来看，代码部分比较简单，有如下步骤：

1. 读入一段基于定义语言的 C 语言代码
2. 逐个分析读入的字符，并记录产生的 token
3. 输出结果

4. 假设

4.1. 实验环境

1. Mac OS

2. Clion + MinGW (gcc)

4.2.C 语言子集定义

符号	单目符号	( ) , ; [ ] { } 等
	双目符号	+= == -= != ++ -- >= <=
	注释	// /*...*/
	空白符	\n \t space
字符串	标识符	字母开头，后面可以加任意数量的字母和数字
	保留字	else if int double char return void while for
数字	整数	0 或者[1-9]开头，后面加任意数字
	浮点数	[0-9].开头，后面加只至少一个数字

5. 有关自动机描述

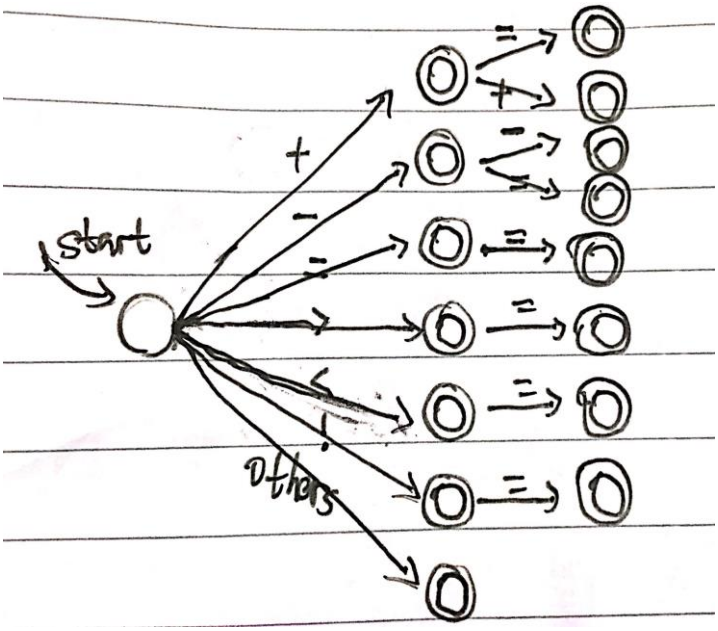
5.1.基于语言分类的正则表达式

符号	**单目符号**	single_char->(,|{|}|(|)|;|...)
	双目符号	double_char->(+=|==|-=|>=|<=|++|--|!=)
	注释	comment_single->//[.*]\n comment_multi->/*(([~*~/]|[*~/]|[~*/])*)*/

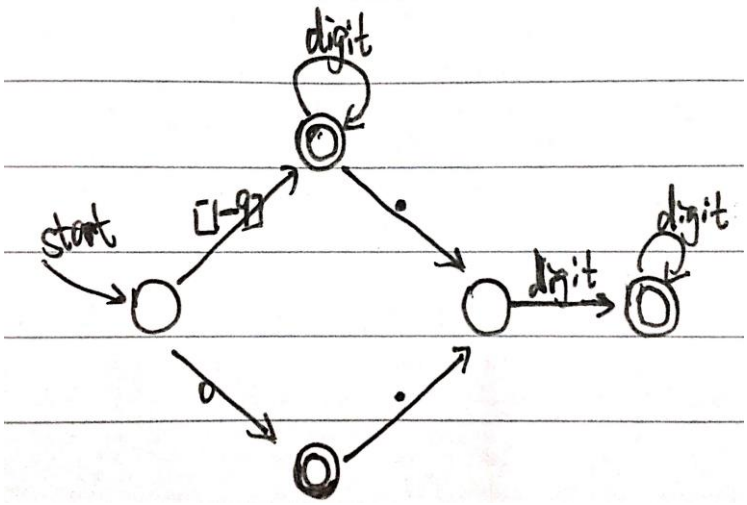
	空白符	blank->(\n|\t|space)
字符串	标识符	char->[a-zA-Z] digit->[0-9] str->char(char|digit)*
	保留字	reserve_word->(else|if|int|double|char|return|void|while|for)
数字	整数	digit->[0-9] integer->([1-9](digit)*|0)
	浮点数	digit->[0-9] double->digit.digit(digit)*

5.2.基于正则表达式构造有限自动机

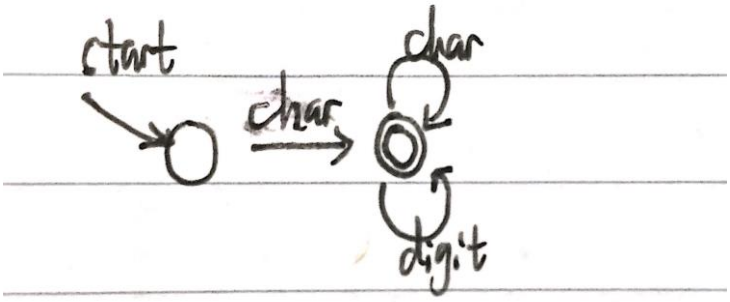
1. 符号



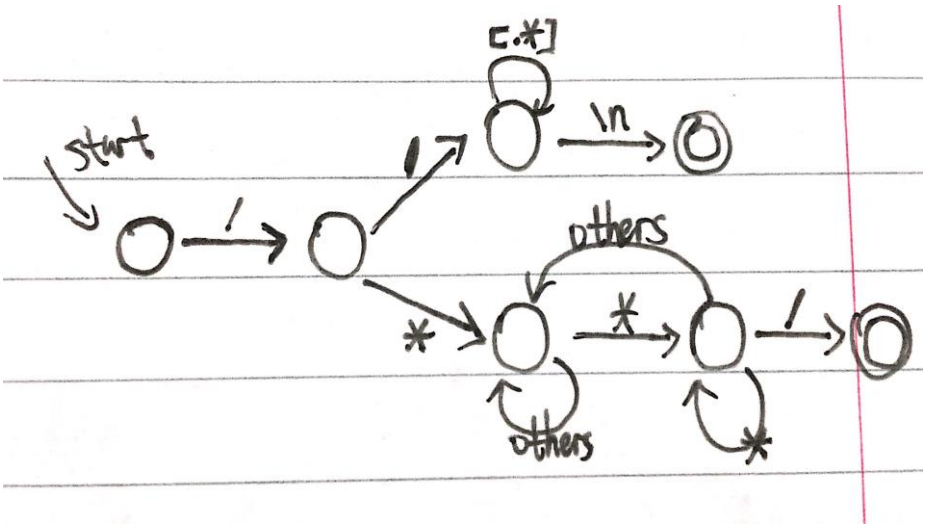
2. 数字



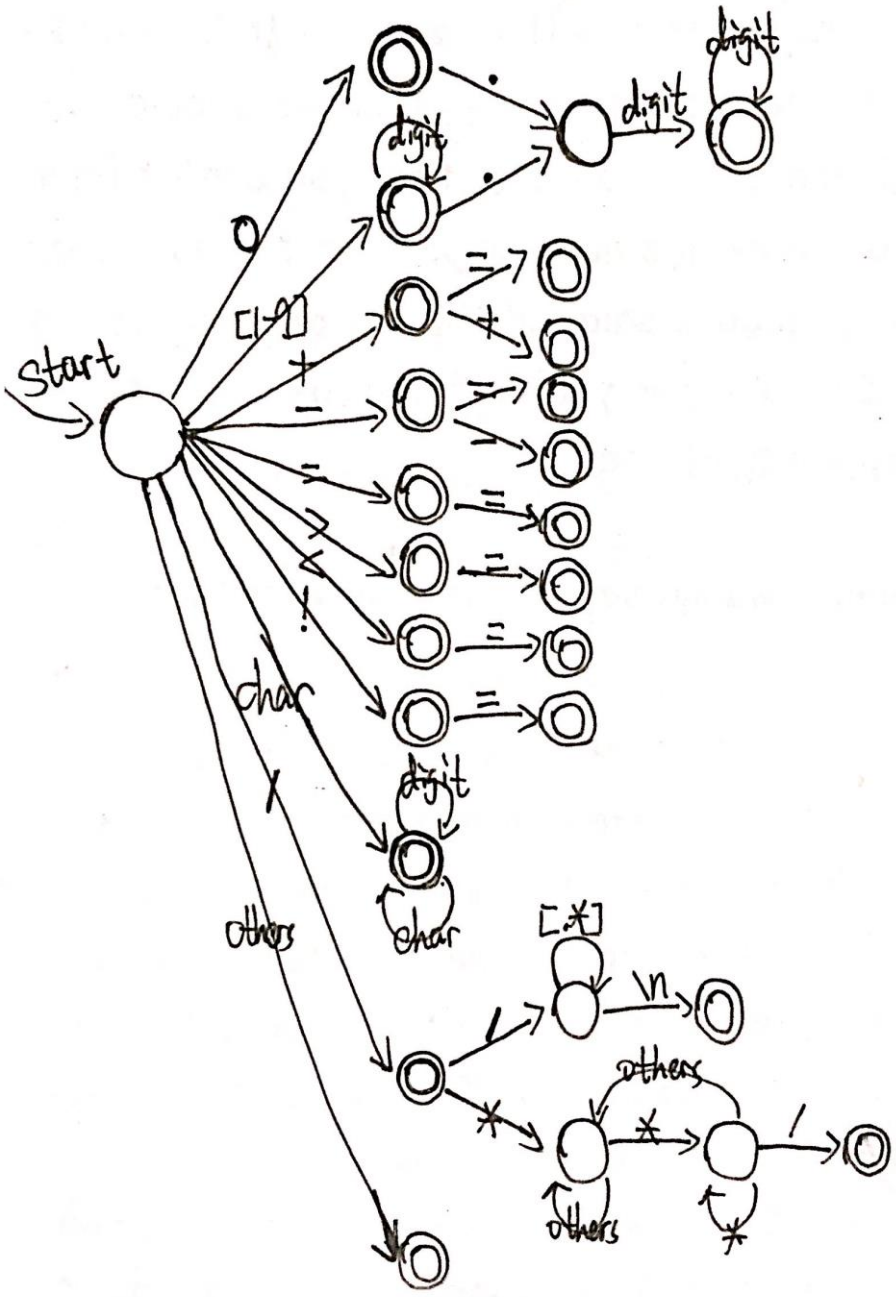
3. 标识符



4. 注释



5. 总有限状态机



6. 数据结构描述

1. Token

```
struct Token{
    int type;
    string name;
}token[max_tokens];
```

2. States

```
//state types
const int SIGN = 1;
const int NUM = 2;
const int ZERO = 12;
const int DOUBLE_START = 7;
const int DOUBLE = 11;
const int STR = 3;
const int RES = 4;
const int NUL = 5;
const int COM_SINGLE = 8;
const int COM_MULTI = 9;
const int COM_MULTI_OUT = 10;
const int COM_END = 6;
string getTypeName(int type);
```

7. 核心算法描述

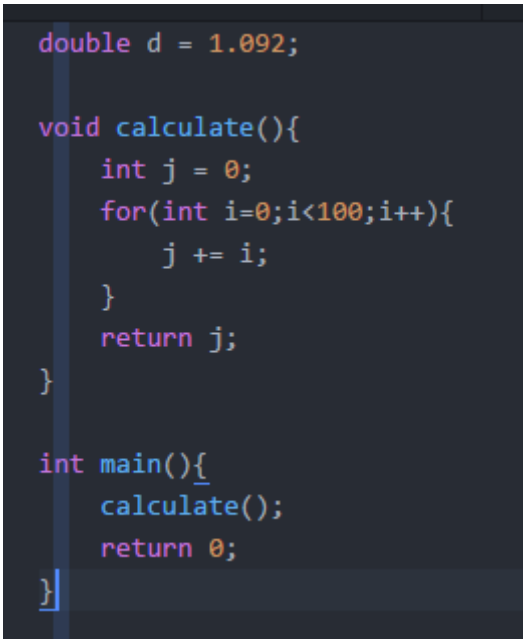
1. Analyze 函数负责依据有限自动机，进行逐字分析输入的语言，并保存 Token

```
void analyze(string s){
    int p = 0;
    string current_token = "";
    m_state = NUL;
    bool in_com = false; //判断是否在注释中
    while (p <= s.length()) {
        int type = getType(s[p]);
        if (type == BLANK && m_state != NUL && !in_com){ //空白符
            addToken(m_state, current_token);
            current_token = "";
            m_state = NUL;
        } else{
            switch (m_state) {
```

8. 测试用例

8.1. Test 1

1. 测试输入



```
double d = 1.092;

void calculate(){
    int j = 0;
    for(int i=0;i<100;i++){
        j += i;
    }
    return j;
}

int main(){
    calculate();
    return 0;
}
```

2. 测试输出

```
/Users/raychen/Library/Caches/CLion2016.1/cmake/generated/c_lex:
<RESERVE_WORD, double>
<ID, d>
<SYMBOL, =>
<DOUBLE, 1.092>
<SYMBOL, ;>
<RESERVE_WORD, void>
<ID, calculate>
<SYMBOL, (>
<SYMBOL, )>
<SYMBOL, {>
<RESERVE_WORD, int>
<ID, j>
<SYMBOL, =>
<INTEGER, 0>
<SYMBOL, ;>
<RESERVE_WORD, for>
<SYMBOL, (>
<RESERVE_WORD, int>
<ID, i>
<SYMBOL, =>
<INTEGER, 0>
<SYMBOL, ;>
<ID, i>
<SYMBOL, <>
<INTEGER, 100>
```

```
<INTEGER, 100>
<SYMBOL, ;>
<ID, i>
<SYMBOL, ++>
<SYMBOL, )>
<SYMBOL, {>
<ID, j>
<SYMBOL, +=>
<ID, i>
<SYMBOL, ;>
<SYMBOL, }>
<RESERVE_WORD, return>
<ID, j>
<SYMBOL, ;>
<SYMBOL, }>
<RESERVE_WORD, int>
<ID, main>
<SYMBOL, (>
<SYMBOL, )>
<SYMBOL, {>
<ID, calculate>
<SYMBOL, (>
<SYMBOL, )>
<SYMBOL, ;>
<RESERVE_WORD, return>
<INTEGER, 0>
<SYMBOL, ;>
<SYMBOL, }>

Process finished with exit code 0
```

8.2. Test 2

1. 测试输入

```
/* author: raychen
   date: 2016-10-28
   description: test */

// definition
double d1 = 10.24;
double d2 = 0.157;
int i1 = 0;
int i2 = 123;
// calculation
double calculation(){
    while (i1 > i2){
        i1++;
        i2--;
    }
    double ans = d1 + d2;
    ans += i2;
    ans -= i1;
    return ans;
}
// main
int main(){
    double ans = calculation();
}
```

2. 测试输出

```
/Users/raychen/Library/Caches/CLion2016.1/cmake/gen
<COMMENTS_MULTI, /* author: raychen
   date: 2016-10-28
   description: test */>
<COMMENTS_SINGLE, // definition>
<RESERVE_WORD, double>
<ID, d1>
<SYMBOL, =>
<DOUBLE, 10.24>
<SYMBOL, ;>
<RESERVE_WORD, double>
<ID, d2>
<SYMBOL, =>
<DOUBLE, 0.157>
<SYMBOL, ;>
<RESERVE_WORD, int>
<ID, i1>
<SYMBOL, =>
<INTEGER, 0>
<SYMBOL, ;>
<RESERVE_WORD, int>
<ID, i2>
<SYMBOL, =>
<INTEGER, 123>
<SYMBOL, ;>
<COMMENTS_SINGLE, // calculation>
<RESERVE_WORD, double>
<ID, calculation>
<SYMBOL, (>
<SYMBOL, )>
<SYMBOL, {>
<RESERVE_WORD, while>
<SYMBOL, (>
<ID, i1>
<SYMBOL, >>
<ID, i2>
<SYMBOL, )>
<SYMBOL, {>
<ID, i1>
<SYMBOL, ++>
<SYMBOL, ;>
<ID, i2>
<SYMBOL, -->
<SYMBOL, ;>
<SYMBOL, }>
<RESERVE_WORD, double>
```

```
<SYMBOL, }>
<RESERVE_WORD, double>
<ID, ans>
<SYMBOL, =>
<ID, d1>
<SYMBOL, +>
<ID, d2>
<SYMBOL, ;>
<ID, ans>
<SYMBOL, +=>
<ID, i2>
<SYMBOL, ;>
<ID, ans>
<SYMBOL, -=>
<ID, i1>
<SYMBOL, ;>
<RESERVE_WORD, return>
<ID, ans>
<SYMBOL, ;>
<SYMBOL, }>
<COMMENTS_SINGLE, // main>
<RESERVE_WORD, int>
<ID, main>
<SYMBOL, (>
<SYMBOL, )>
<SYMBOL, {>
<RESERVE_WORD, double>
<ID, ans>
<SYMBOL, =>
<ID, calculation>
<SYMBOL, (>
<SYMBOL, )>
<SYMBOL, ;>
<SYMBOL, }>

Process finished with exit code 0
```

9. 问题与解决

1. 保留字判断，如何从标识符中区分出保留字？

解决：对于判断一个标识符是不是保留字，我采取了先当成一般标识符去识别，得到 token 后再去判断该标识符是否为保留字，减少了中间判断过程。

2. 整体代码结构，判断流程、状态转换比较复杂。如何解决？

解决：我采取了先对当前状态进行判断，在对应各状态内对各种 input 的类型进行处理。层次比较清晰，状态的进一步转换也易懂。

10. 总结与感受

通过本次实验，我更深入理解了编译原理课程所讲内容，词法分析过程，同时为写词法分析生成器奠定基础。