编译原理 – 词法分析程序实验报告

615220324-许睿

1. 实验目的

本实验需要实现一个简单的词法分析程序.

- 选择一个熟悉的语言的子集作为本实验的测试语言，程序用于分析该语言的词法
- 列出该语言子集的文法定义，并给出转化为 NFA 和 DFA 的过程以及结果
- 编写程序并用改语言子集写若干个测试程序，将词法分析的结果打印出来

本实验已完成的任务:

- 实现了标识符，无符号整型数，保留字，分隔符和运算符的分析
- 实现了/**/注释的忽略
- 给出上述文法，其中四则运算和关系运算只给出状态机，用于后续语法分析

运行环境:

- OS: Windows 10
- CMake: 不低于 3.10
- 实现语言: C++
  - C++版本: std17 及以上
- IDE: CLion2022.2

2. 实验过程

作为测试，本实验采用仅含有如下保留字的 `miniC` 语言:

Table0. `miniC` 中使用的保留字

RESERVED WORD	DESCRIPTION
INT	miniC 的基本数据类型(无符号整型)
VOID	用函数的返回值类型
IF-ELSE	条件分支语句
WHILE	循环分支语句
RETURN	用于返回函数值

此外该实验定义的 `miniC` 还支持两个无符号整型数的`+,-,*,/`二元运算，两个无符号整型数的`<,>,==,<=,>=,==,!=`二元关系运算(运算规则在语法分析中体现)，变量的声明和定义，函数的声明和定义，注释的忽略. 下面给出 `miniC` 的文法定义规则:

a) miniC 的文法定义

标识符的文法定义, NFA 和 DFA

- **正规文法和正规式** 标识符要求开头字符为英文字母的大小写或下划线，不能以数字开头，中间可以出现任意长度的单词或数字或下划线.

```
identifier = letter (letter | digit)*
letter -> [a-z] | [A-Z] | _
digit -> 0 | nonzero-digit
nonzero-digit -> [1-9]
```

- **正规式转化为 NFA** 根据以上正规文法的定义，可以经过如 **Fig1** 转化得到 NFA:

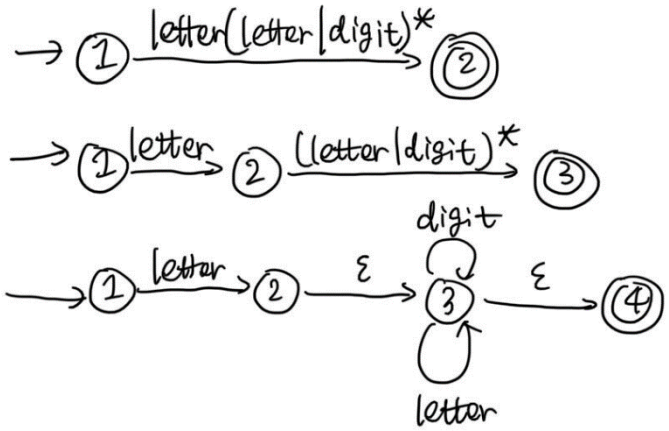


Fig1.标识符的非确定有限状态机推导过程

- **NFA 转化为 DFA** 将最后得到的 NFA 确定化：首先求 DFA 中的相应状态如表 **table 1** 所示

Table1. DFA 中相应的转化表

	Letter	**Digit**
S0={1}	**{2,3,4}**	**-**
S1={2,3,4}	**{3,4}**	**{3,4}**
S2={3,4}	**{3,4}**	**{3,4}**

根据状态转换表得到相应的 DFA 如 **Fig2** 所示

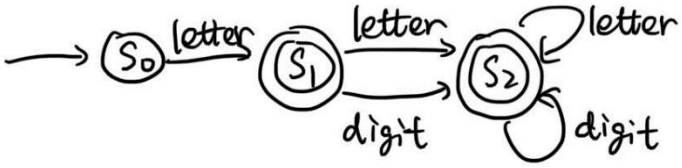


Fig2. 标识符对应的 DFA

该 DFA 可以经过最小化后得到 **Fig3** 所示的最小化 DFA

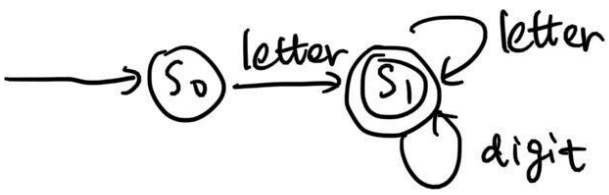


Fig3. 标识符对应的最小 DFA

无符号常量的文法定义, NFA 和 DFA

- **正规文法和正规式** 无符号常量要求以非 0 数字开头，并后接数字或空

```
digits -> digit | nonzero-digit digit*
       -> 0 | nonzero-digit (nonzero-digit | 0)*
```

- **正规式转化为 NFA** 根据以上文法得到 NFA

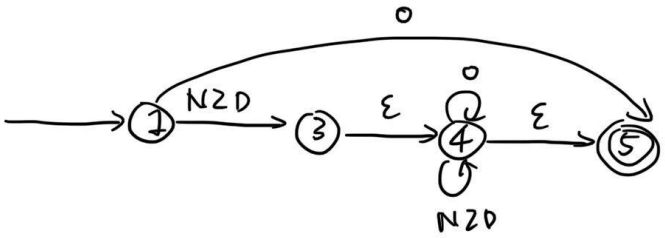


Fig4. 无符号常量的 NFA(其中 NZD 表示 nonzero-digit)

- **NFA 转化为 DFA** 将 NFA 转化为 DFA，先列表:

Table2. DFA 中相应的转化表

	0	NZD
S0={1}	{5}	{3,4,5}
S1={3,4,5}	{4,5}	{4,5}
S2={4,5}	{4,5}	{4,5}
S3={5}	{5}	{5}

根据状态转化表得到如下 DFA(**Fig5**)

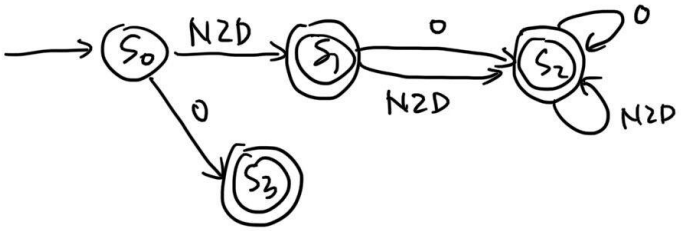


Fig5. 无符号常量的 DFA

该 DFA 经过最小化后得到 **Fig6** 所示的最小化 DFA

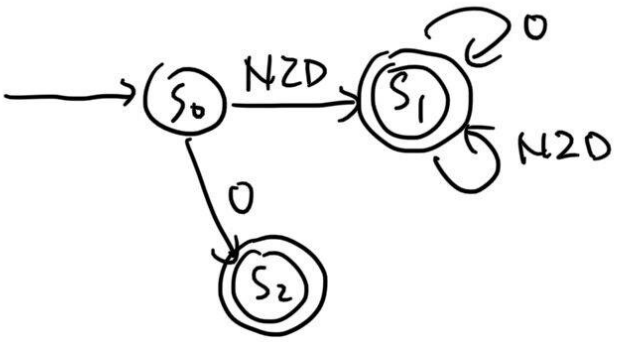


Fig6. 无符号常量的最小化 DFA

四则运算的文法定义, NFA 和 DFA

- **正规文法和正规式**

该语法的正规文法为:

```
expr -> expr (+ | -) term | term
term -> term (* | /) factor | factor
factor -> (expr) | digits | identifier
```

将其转化为正规式得到:

```
expr -> factor ((* | /) factor))* ((+ | -) factor ((* | /) factor)* )*
```

- **正规式转化为 NFA** 根据上述正规式，可以通过 **Fig7.**所示的转化将其化为 NFA

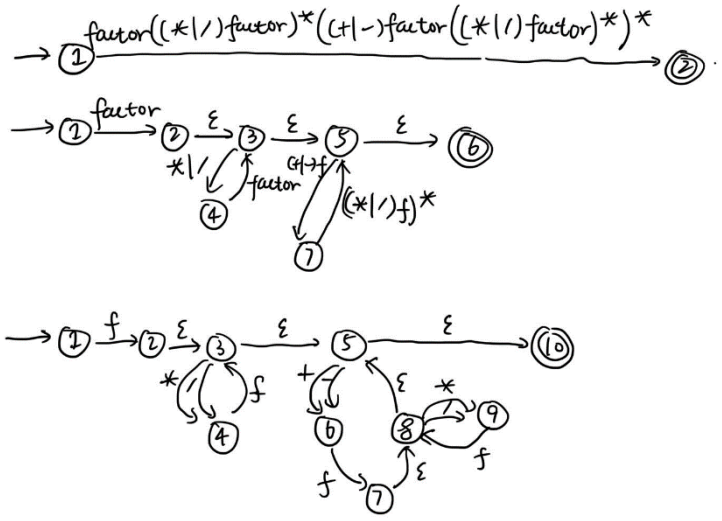


Fig7. 四则运算的 NFA

- **NFA 转化为 DFA** 求ε-闭包后得到相应的 DFA 的状态转移矩阵:

Table3. DFA 相应的状态转移矩阵

	Factor	*	/	+	-
S0={1}	{2,3,5,10}	-	-	-	-
S1={2,3,5,10}	-	{4}	{4}	{6}	{6}
S2={4}	{3,5,10}	-	-	-	-
S3={6}	{5,7,8,10}	-	-	-	-
S4={3,5,10}	-	{4}	{4}	{6}	{6}
S5={5,7,8,10}	-	{9}	{9}	{6}	{6}
S6={9}	{8.5.10}	-	-	-	-
S7={5,8,10}	-	{9}	{9}	{6}	{6}

将以上的状态转移矩阵化为 DFA 如 **Fig8** 所示

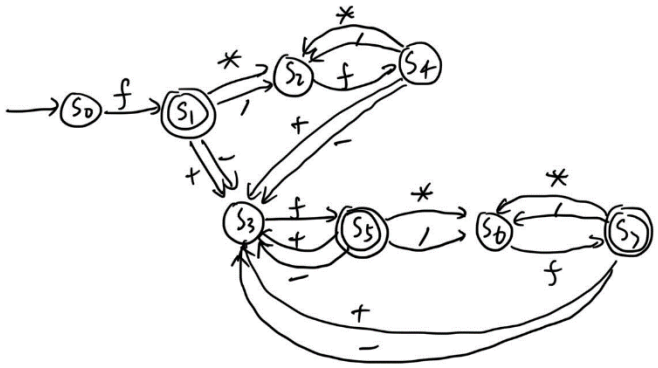


Fig8. 四则运算对应的 DFA

将上述的 DFA 经过最小化得到 **Fig9** 所示的最小化 DFA

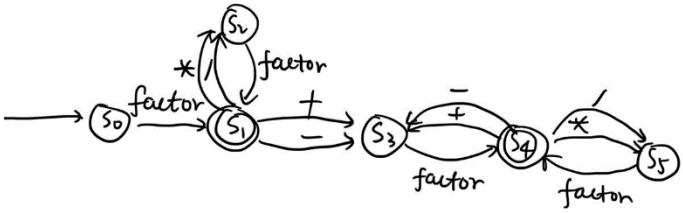


Fig9. 四则运算所对应的最小化 DFA

关系运算的文法定义, NFA 和 DFA

- **正规文法和正规式**

正规文法:

```
uneqExpr -> expr | uneqExpr < assignExpr | uneqExpr > assignExpr
eqExpr -> expr | uneqExpr = assignExpr | uneqExpr ! assignExpr
assignExpr -> expr | = expr
```

正规式:

```
uneqExpr -> expr | uneqExpr < assignExpr | uneqExpr > expr | = expr
eqExpr -> expr | uneqExpr = assignExpr | uneqExpr ! expr | = expr
```

- **正规式转化为 NFA** 根据上述的正规式求出如下 **Fig10** 所示的 NFA

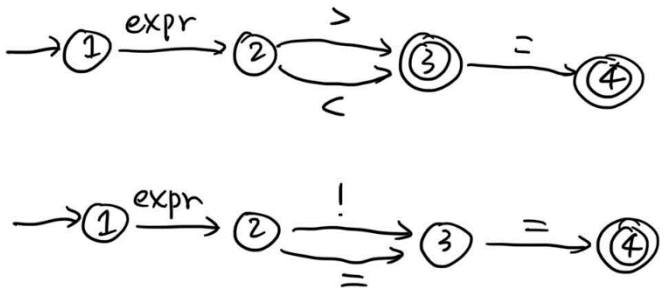


Fig10. 关系运算的 NFA

- **NFA 转化为 DFA**

可以证明，上述 **Fig10** 所示的 NFA 同时也是最小的 DFA.

//型注释的文法定义, NFA 和 DFA**

- **正规文法和正规式**

正规文法:

```
cmt -> / cmtStart
cmtStart -> \* doc
doc -> [^\*]* doc | \* cmtEnd
cmtEnd -> \* cmtEnd | [^\*/] doc | /
```

正规式:

```
/ * ([^*]* | [*^/])* (*)+ /
```

- **正规式转化为 NFA** 根据上述正规式，可以画出如下 **Fig11** 所示的 NFA

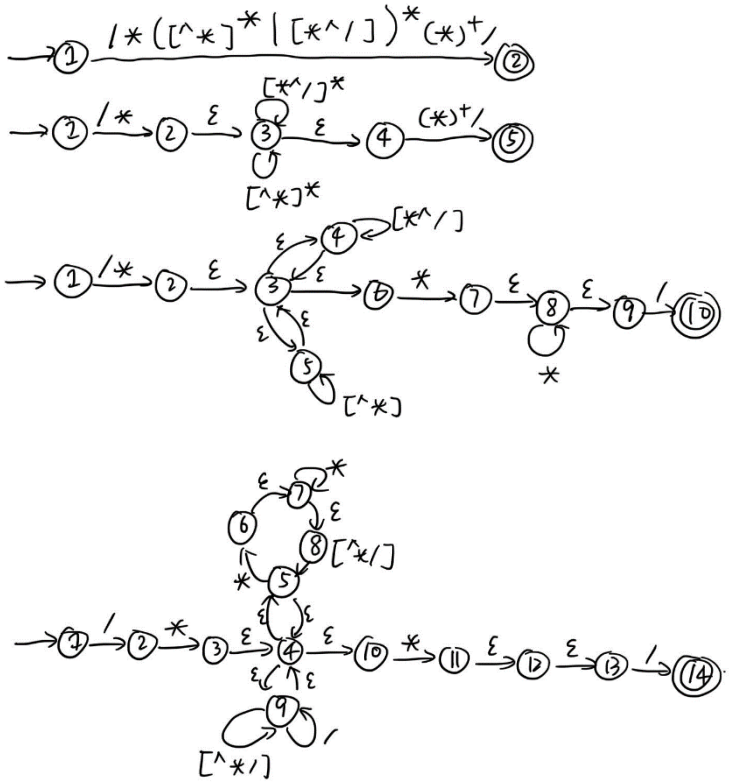


Fig11. /**/注释所对应的 NFA

- **NFA 转化为 DFA** 根据上述的 NFA 画出对应的 DFA 的状态转移矩阵

Table4. DFA 对应的状态转移矩阵

	/	*	[^*/]
S0={1}	{2}	-	-
S1={2}	-	{3,4,5,9,10}	-
S2={3,4,5,9,10}	{4,5,9,10}	{6,7,8,11,12,13}	{4,5,9,10}
S3={4,5,9,10}	{4,5,9,10}	{6,7,8,11,12,13}	{4,5,9,10}
S4={6,7,8,11,12,13}	{14}	{7,8,12,13}	{4,5,9,10}
S5={7,8,12,13}	{14}	{7,8,12,13}	{4,5,9,10}
S6={14}	-	-	-

根据上述状态转移矩阵做出对应的 DFA 如下图 **Fig12** 所示

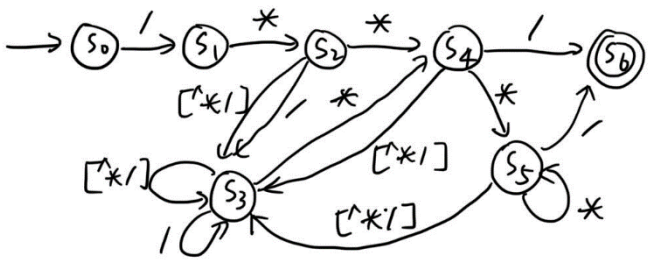


Fig12. /**/注释对应的 DFA

将其最小化后得到 **Fig13.**所示的最小化 DFA

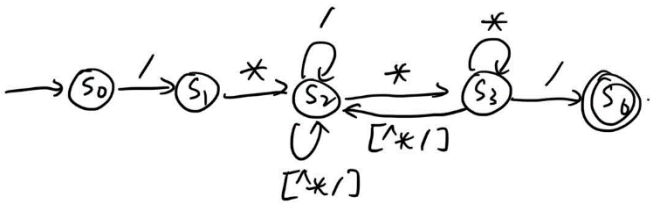


Fig13. /**/注释所对应的最小化 DFA

3. 词法分析程序

为实现该词法分析程序，本项目为每一类 lexeme 创建一个状态机并单独储存在一个头文件中，因此目录结构如下:

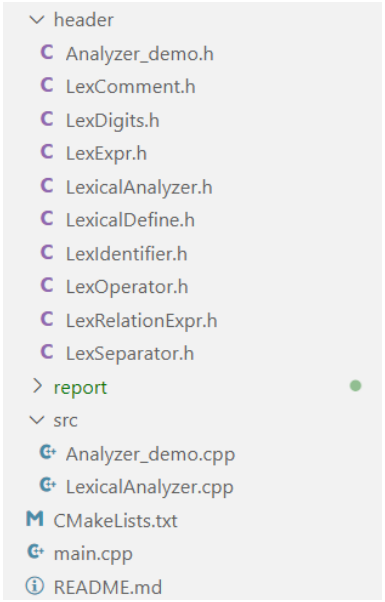


Fig14. 目录结构

其中 LexicalAnalyzer.h 为该词法分析函数，实现思路如下:

- 将状态设置为开始状态
- 对于测试文件中的每个字符，遍历每个状态机（此处使用的是单线程遍历），得到结束状态或错误状态
- 直到文件结束

(具体代码如图 **Fig15** 所示)

```
StateType state = StateType::START;
std::vector<char> buffer;
std::vector<dfa> DFA;
DFA.push_back(dfaIdentifier);
DFA.push_back(dfaDigits);
DFA.push_back(dfaComment);
DFA.push_back(dfaSeparator);
DFA.push_back(dfaOperator);

int line_num = 0;

while (read_p < file_size) {
    for (auto &cur_dfa: DFA) {
        cur_dfa(file_in_str, state, buffer, read_p);
        switch (state) {···
    }
}

file_write.close();
```

Fig15. 词法分析代码

下面具体介绍每个状态机的实现

标识符状态机(LexIdentifier.h):

Fig3.各状态对应如下: S0: ENTER_ID; S1: IN_ID/END_ID; 错误: ERROR

首先是 ENTER_ID -> IN_ID: 该部分的处理是先读取一个非空字符 (若是空字符则继续读取并保持当前状态), 若该字符是字母或下划线则进入 IN_ID 状态等待 ID 结束, 若是其它字符(如数字)则将整个状态机置为结束, 等待试探下一个状态机.

```
case StateType::ENTER_ID:
    if (std::isblank(c) || c == '\r' || c == '\n') {
        state = StateType::ENTER_ID;
    } else if (std::isalpha(c) || c == '_') {
        buf.push_back(c);
        state = StateType::IN_ID;
    } else {
        file_read.seekg((long long) -sizeof(char), std::ios::cur);
        read_p = file_read.tellg();
        state = StateType::END;
    }
    break;
```

其次是 IN_ID -> END_ID: 根据标识符的状态机, 在 IN_ID 状态时, 若此时读取的字符是数字, 字母或下划线则保持当前状态, 若读到了空白符, 分隔符或者运算符, 则代表标识符结束, 进入 END_ID 状态, 若不符合上述条件则进入 ERROR 状态, 等待后续处理.

```
case StateType::IN_ID:
    if (std::isalpha(c) || std::isdigit(c) || c == '_') {
        buf.push_back(c);
        state = StateType::IN_ID;
    } else if (std::isblank(c) || isSeparator(c) || isOperator(c)) {
        if (isSeparator(c) || isOperator(c)) {
            file_read.seekg((long long) -sizeof(char), std::ios::cur);
            read_p = file_read.tellg();
        }
        state = StateType::END_ID;
    } else {
        // ERROR
        state = StateType::ERROR;
    }
    break;
```

无符号整数的状态机(LexDigits.h)

Fig6.各状态对应如下: S0: ENTER_DIGITS; S1, S2: IN_DIGITS/END_DIGITS; 错误: ERROR

ENTER_DIGITS -> IN_DIGITS: 分两种情况，如果此时的非空字符为 0，则直接进入 END_DIGITS; 若此时的非空字符为非零数，则进入 IN_DIGITS 状态; 若不符合上述任一描述则当前状态机遍历结束.

```
case StateType::ENTER_DIGITS:
    if (std::isblank(c) || c == '\r' || c == '\n') {
        state = StateType::ENTER_DIGITS;
    } else if (std::isdigit(c)) {
        buf.push_back(c);
        state = StateType::IN_DIGITS;
        if ((int) c == 0) {
            state = StateType::END_DIGITS;
        }
    } else {
        file_read.seekg((long long) -sizeof(char), std::ios::cur);
        read_p = file_read.tellg();
        state = StateType::END;
    }
```

IN_DIGITS -> END_DIGITS: 若当前字符为数字则保持当前 IN_DIGITS，若出现了空白符，分隔符或运算符，则进入 END_DIGITS，否则进入错误状态.

```
case StateType::IN_DIGITS:
    if (std::isdigit(c)) {
        buf.push_back(c);
        state = StateType::IN_DIGITS;
    } else if (std::isblank(c) || isSeparator(c) || isOperator(c)) {
        if (isSeparator(c) || isOperator(c)) {
            file_read.seekg((long long) -sizeof(char), std::ios::cur);
            read_p = file_read.tellg();
        }
        state = StateType::END_DIGITS;
    } else {
        // ERROR
        state = StateType::ERROR;
    }
```

//型注释的状态机 (LexComment.h)**

Fig13. 各状态对应如下: S0: ENTER_CMT; S1: CMT_S1; S2: INNER_CMT_S2; S3: CMT_S3; S4: END_CMT

ENTER_CMT -> CMT_S1: 若读取的非空字符为'/'，则进入 CMT_S1，表示正在等待'*'; 若读取的为其它字符，则结束当前状态机.

```
case StateType::ENTER_CMT:
    if (std::isblank(c) || c == '\r' || c == '\n') {
        state = StateType::ENTER_CMT;
    } else if (c == '/') {
        // overlook the comment, no need to input into buf
        state = StateType::CMT_S1;
    } else {
        file_read.seekg((long long) -sizeof(char), std::ios::cur);
        read_p = file_read.tellg();
        state = StateType::END;
    }
```

CMT_S1 -> INNER_CMT_S2: 当前状态表示正在等待'*'字符，若当前读取的字符为'*'，则进入 INNER_CMT_S2 状态，表示注释的内部，应该被忽略不被存储; 若当前读取的字符为其它字符则结束当前状态机，并将读取的指针往回 2 个字符.

```
case StateType::CMT_S1:
    if (c == '*') {
        state = StateType::INNER_CMT_S2;
    } else {
        file_read.seekg((long long) -2 * sizeof(char), std::ios::cur);
        read_p = file_read.tellg();
        state = StateType::END;
    }
```

INNER_CMT_S2 -> CMT_S3: 若在注释的内部出现了'*'则立刻进入 CMT_S3 状态; 若为其他字符, 则保持当前状态.

CMT_S3 -> INNER_CMT_S2/END_CMT: 若刚读取了字符'*', 则当前状态为 CMT_S3, 表示等待注释结束. 若当前读取的字符为'*', 则依然保持现在状态, 等待注释结束; 若当前字符为除了'*'和'/'的其他字符, 则回到 INNER_CMT_S2 状态, 表示仍然处于注释内部; 若当前字符为'/', 则表示注释结束, 进入 END_CMT 状态:

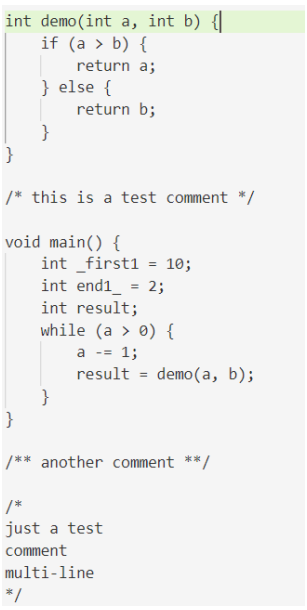
```
case StateType::INNER_CMT_S2:
    if (c == '*') {
        state = StateType::CMT_S3;
    } else {
        state = StateType::INNER_CMT_S2;
    }
    break;
case StateType::CMT_S3:
    if (c == '/') {
        state = StateType::END_CMT;
    } else if (c == '*') {
        state = StateType::CMT_S3;
    } else {
        state = StateType::INNER_CMT_S2;
    }
```

分隔符和操作符等较简单的 DFA 见附件具体代码.

4. 测试

测试 1: 包括标识符, 保留字, 分隔符, 运算符, 单行注释, 多行注释和无符号整数:

测试程序:



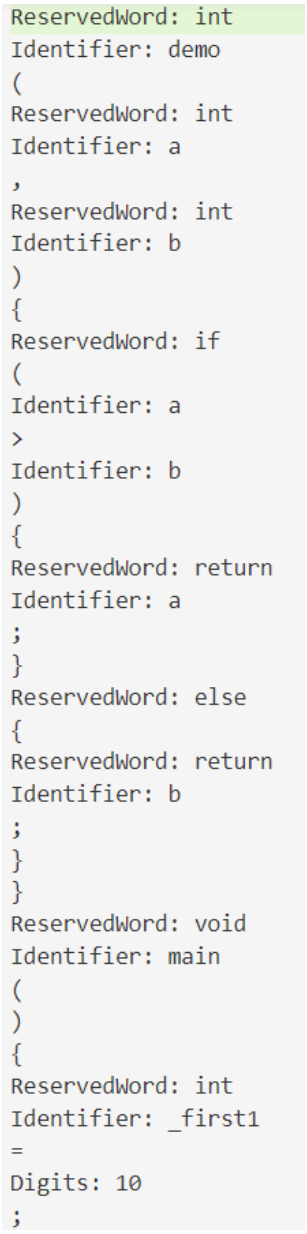
```
int demo(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

/* this is a test comment */

void main() {
    int _first1 = 10;
    int end1_ = 2;
    int result;
    while (a > 0) {
        a -= 1;
        result = demo(a, b);
    }
}

/** another comment **/

/*
just a test
comment
multi-line
*/
```

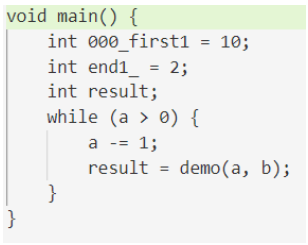


```
ReservedWord: int
Identifier: demo
(
ReservedWord: int
Identifier: a
,
ReservedWord: int
Identifier: b
)
{
ReservedWord: if
(
Identifier: a
>
Identifier: b
)
{
ReservedWord: return
Identifier: a
;
}
ReservedWord: else
{
ReservedWord: return
Identifier: b
;
}
}
ReservedWord: void
Identifier: main
(
)
{
ReservedWord: int
Identifier: _first1
=
Digits: 10
;
ReservedWord: int
Identifier: end1_
=
Digits: 2
;
ReservedWord: int
Identifier: result
;
ReservedWord: while
(
Identifier: a
>
Digits: 0
)
{
Identifier: a
-
=
Digits: 1
;
Identifier: result
=
Identifier: demo
(
Identifier: a
,
Identifier: b
)
;
}
}
```

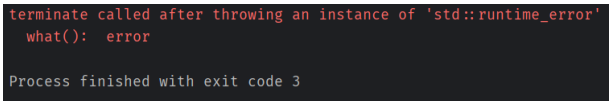
(a) 测试程序　　(b) 词法分析测试结果

Fig16. 词法分析测试 1 (可以看到含下划线，数字的标识符都被正确识别，没有保留字被错误识别为标识符，无符号整数被正确识别，所有注释均被忽略，分隔符和运算符都单独打印)

测试 2: 标识符命名出错



```
void main() {
    int 000_first1 = 10;
    int end1_ = 2;
    int result;
    while (a > 0) {
        a -= 1;
        result = demo(a, b);
    }
}
```



```
terminate called after throwing an instance of 'std::runtime_error'
  what():  error

Process finished with exit code 3
```

(a) 测试程序　　(b) 词法分析结果

Fig17. 词法分析测试 2 (可以看到标识符错误时程序抛出异常)

测试 3: 整型数出错

```
void main() {
    int _first1 = 0;
    int end1_ = 02;
    int result;
    while (a > 0) {
        a -= 1;
        result = demo(a, b);
    }
}
```

```
terminate called after throwing an instance of 'std::runtime_error'
  what():  error

Process finished with exit code 3
```

(a) 测试程序　　(b) 词法分析结果

Fig18. 词法分析测试 3 (可以看到由于第二个整型定义错误，词法分析程序抛出异常)

5. 实验总结

经过该次实验的训练，我学习并实践了由语言所定义的文法到 NFA, NFA 转化为 DFA 以及 DFA 的最小化的过程，更加熟悉了课堂内容. 除此之外，通过编写简单的词法分析程序，我明白了如何用代码实现状态机中状态的转换，并能对该实验定义的 C 语言子集进行初步的词法分析.