

词法分析

不论是从标准输入中读取还是从文件中解析，我们的源程序总是以**字符流**的形式存在。例如，对于下面这个简单的源程序：

```
1  int main() {
2      int var = 8 + 2;
3      if (var != 10) {
4          printf("error!");
5      } else {
6          printf("correct!");
7      }
8      return 0;
9  }
```

尽管从我们的视角来看，源程序是结构化的，包含了顺序、分支、循环等多种结构；同时通过空格、换行等方法，我们还可以将程序中的各个成分清晰地区分开来。但是从编译器的角度看，源程序不带有任何结构，只是由字符组成的序列（字符串）：

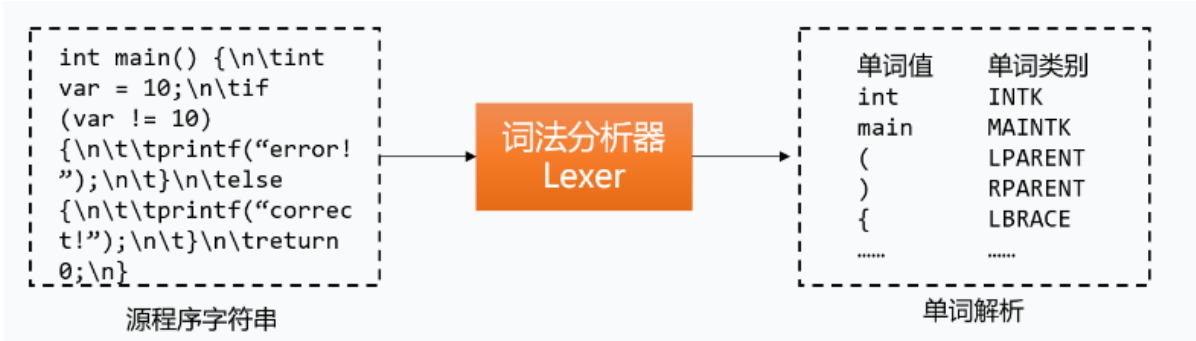
```
1  int main() {\n\tint var = 8+2;\n\tif (var != 10)
   {\n\t\tprintf("error!");\n\t}\n\telse
   {\n\t\tprintf("correct!");\n\t}\n\treturn 0;\n}
```

这导致了一个问题，虽然字符是构成字符串的基本单元，却每个字符自身却不一定具备特定的含义。这些字符需要组成一个单词才能传达出对于程序来说有意义的信息。因此，实现编译器的第一步就是要将这样的线性字符串分割成一个个单词，便于后续分析。在编译器中，这一阶段被称为**词法分析**。

一、词法分析作用

词法分析器作为编译器的第一部分，承担的任务就是通过扫描输入的源程序字符串，将其分割成一个个**单词**。如图所示，经过词法分析器的处理，我们将字符序列转换为单词序列。对于每个单词，我们至少应当记录单词的**取值**及其**类别信息**。另外，编译器在词法分析阶段也可能记录单词在源代码中的**位置信息**，例如源文件路径、行号、列号等等。这些额外信息对于编译器的错误处理十分重要。得当的错误定位能够为代码的编写者提供极大便利。

在源程序中，还有一些字符并不会影响程序的语法语义，如换行符 `\n`、注释等，这些符号自然也不会被词法分析器解析为单词，但词法分析器也需对这些符号进行适当处理，如忽略跳过、记录行号列号等等。



二、词法分析器的接口

在上一小节中，我们了解了词法分析阶段的主要作用，在本小节中，我们将基于此介绍词法分析器 (Lexer)。

首先，词法分析的输出是单词序列，因而我们需要定义单词的数据结构，称作 `Token`。使用 Java 语言 (cpp 类似，不再赘述)，我们所实现的 `Token` 类包括三个字段：`tokenType`、`tokenContent` 和 `lineNum`，分别对应了单词的类型、取值和位置。

```
1 public class Token extends Node {
2     private final String tokenContent;
3     private final TokenType tokenType;
4     private final int lineNum;
5
6     public Token(String tokenContent, TokenType tokenType, int lineNum) {
7         this.tokenContent = tokenContent;
8         this.tokenType = tokenType;
9         this.lineNum = lineNum;
10    }
11 }
```

尽管本课程不是 C 语言的语法课，但还是有必要解释一下 `TokenType` 中的可疑代码。这是被称为 X-Macro 的技术，用于处理为枚举附加额外属性的问题。由于 C++ 中的枚举只是一个数值，所以无法像 Java 一样为枚举值添加额外属性。例如，我们可以在 Java 中为 `TokenType` 枚举定义 `toString` 方法：

```
1 public enum TokenType {
2     IDNET("identifier"),
3     NUMBER("number"),
4     // ...
5     EOF("eof");
6
7     private String value;
8
9     TokenType(String value) {
10        this.value = value;
11    }
12
13    @Override
14    public String toString() {
15        return value;
16    }
17 }
```

但 C++ 无法做到。因此在 C++ 中实现类似的 `toString` 方法就需要另外的方法。这里碍于篇幅，就不再详述。

接着，我们考虑词法分析器 `LexerAnalyzer` 类的输入。考虑到本课程不会对一个非常大的文件进行输入再编译，便于操作与理解，我们采用直接把代码源文件作为一整个**字符串**进行读取。请注意，如果今后有志于从事编译相关工作，编译大文件甚至大项目是必然的，那么就要考虑将源程序视为**字符流**而非字符串，该分析器词法分析器只在我们需要单词时才会进行单词的解析。

为此，我们需要在 `Lexer` 初始化时传入源文件的 `String` 作为参数，其中代码具体实现功能请自行实现，如使用 `StringBuffer` 类实现 `readFile`。

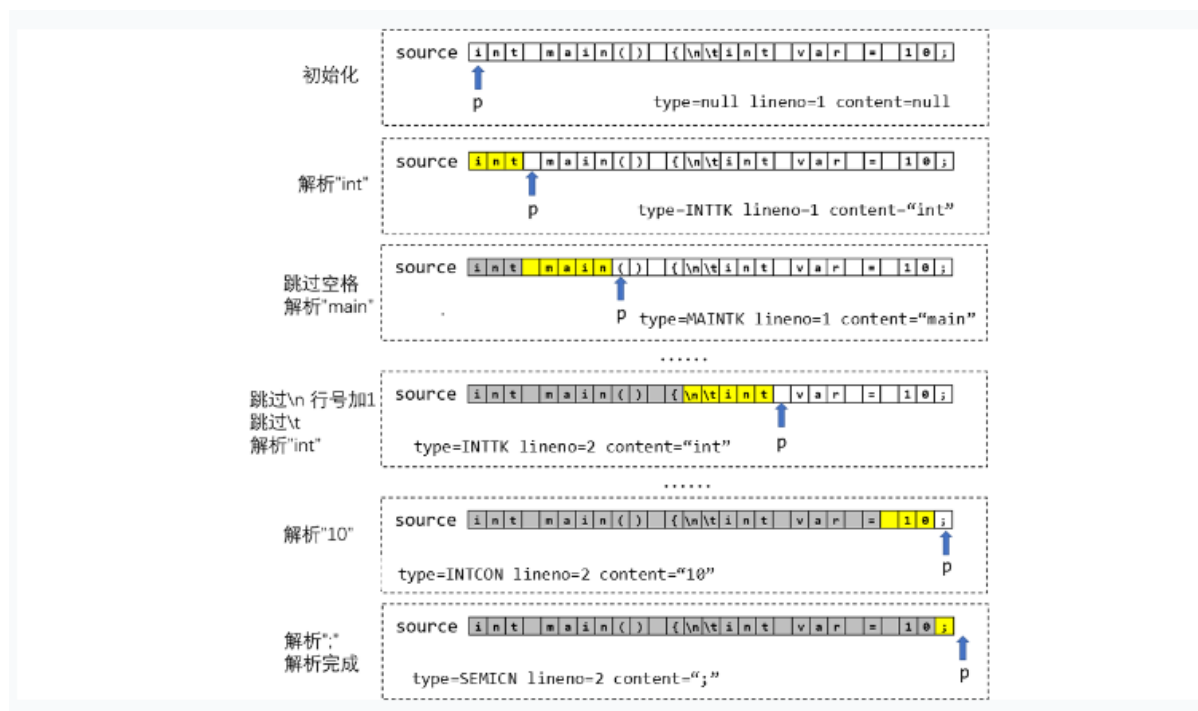
```
1 public class Compiler {
2     // 引入必要的自定义类，如 EnvInitializer 等
3     public static void main(String[] args) throws Exception {
4         /*
5          * Description: 初始化
6          * @author TonyZhan
7          */
8         envInitializer.initialize();    // 初始化各输入输出文件
9         String testfileCode =
10         FileProcess.readFile(envInitializer.testfilePath);    // 读取文件
11
12         /*
13          * Description: 词法分析
14          * @author TonyZhan
15          */
16         lexerAnalyzer.initLexerAnalyzer(testfileCode);    // 将源文件的字符串形式
17         输入到 lexerAnalyzer
18         lexerAnalyzer.lexerAnalyze();
19     }
20 }
```

三、词法分析器的工作过程

(1) 基本流程

在 Java 中，我们实现一个较为简单的词法分析功能。类似于指针移动的思想，我们可以对下标进行遍历，读取每一个下标对应的字符，并对字符进行分类处理。如遇到了 `\n` 也就意味着需要换行，`lineNum` 需要自增 1。但是需要注意，我们可能需要使用**双指针**来具体判别每一个 `token`，并在识别完 `token` 后再将**主指针**（也就是对下标进行遍历的前一个指针）更新到最新的**副指针**的位置。在这个过程中我们又需要使用**贪婪匹配**来实现正确的词法解析。这意味着在**保证匹配某一单词类型的情况下，匹配尽可能多的字符**。在这种情况下，我们需要不断读入新的符号，直到新读入的符号无法让自动机进行合法的状态转移。这时，最后读入的符号便必然不属于当前单词，需要重新推入流中。贪婪匹配十分重要，因为文法中很可能存在两个不同的单词类型，其中一个是另一个的前缀。例如 `<` 和 `<=`。若不采用贪婪匹配，则 `<=` 将总会被识别为 `<`。

接下来我们举例演示词法分析器的工作过程。为方便解释，我们用 `source` 表示输入编译器的源程序字符流，用主指针 `p` 指示尚处于流中的第一个字符。此时主指针遍历对应的操作为**取出 `p` 所指向的字符，并将 `p` 向右移动一格**；副指针的行为则是从主指针开始往后一直遍历，直到取出一个合法的 `token` 为止



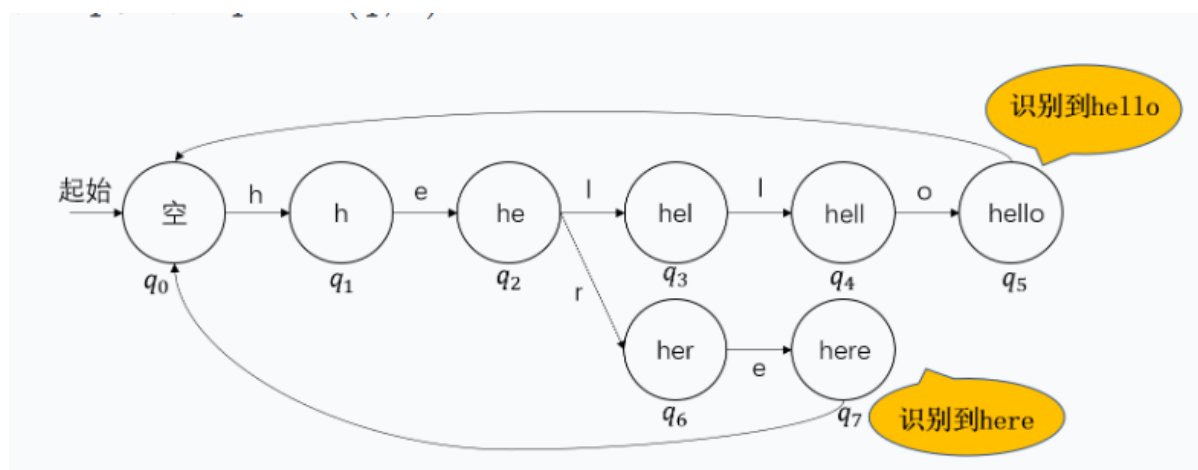
对于如图所示的输入流，解析的主要流程如下：

1. 初始化时，`p` 指向源文件 `code` 的开头，并设置当前行号 `lineNum = 1`。其余部分为空。
2. 遍历主指针 `p`，然后副指针从 `p` 开始不断读入符号，直到**识别**单词 `int`，此时设置对应的单词类型和行号，并将本次解析读入的所有符号作为单词的内容。
3. 更新主指针 `p` 的位置到副指针结束的地方，继续遍历，首先**识别**空格字符并**跳过**，随后按照与步骤 2 相同的方式解析第二个单词 `main`。
4. 经过若干步后，识别到 `\n`，此时行号自增 `lineNum++`。**跳过**该字符，随后按与步骤 3 相同的方式解析得到单词 `int`。
5. 按此方法继续解析后续的 `var`、`=`、`10`。

上面的步骤十分概要，忽略了许多细节。比如说，我们要如何识别不同类型的单词？当识别到空白字符或注释时我们又要如何跳过？接下来我们解释这些问题。

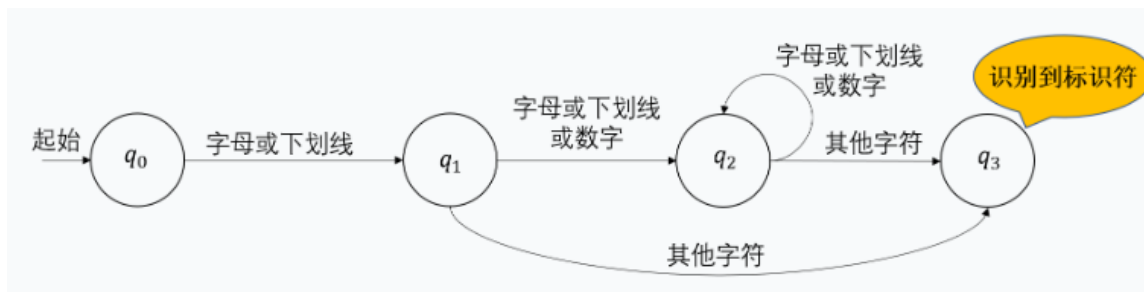
(2) 手搓有穷自动机

在理论课上，我们会学习正则文法与有穷自动机的相关知识。有穷自动机是处理形式化语言的重要工具，常常用于**识别**特定模式的字符串。有穷自动机内部包含一个当前状态 `qq` 和转移函数 σ ，对于一个输入的字符 `aa`，有穷自动机将根据当前状态 `qq` 和字符 `aa` 转移到下一个状态 `pp`，即有 $p = \sigma(q, a)$ 。



上图是一个用于识别 "hello" 和 "here" 两个单词的有穷自动机，初始时处于起始状态 q_0q_0 ，后续每读入一个字符 aa ，都将根据当前状态和 aa 进行一次状态转移（即沿着图中的对应边进入下一个状态），在到达 q_5q_5 或 q_7q_7 后，FSA就成功识别了一个 "hello" 或 "here" 单词。需要注意的是，在读入 "he" 时，我们不能判断当前这个单词是 "hello" 还是 "here"，需要根据读入的下一个字符进行判断并转移到不同的状态分支（对应图中的 q_2q_2 ）。需要额外说明的是，这个例子并没有考虑异常情况，如读入字符 "z"。一般来说，我们可以设定一死状态，并设定当读入异常字符时，转移到该状态。

对于任意单词类型，我们都可以根据其正则表达式给出对应的有穷自动机。例如对于标识符（`[a-zA-Z_][0-9a-zA-Z_]*`），其所对应的有穷自动机如下所示：



我们当然可以通过模拟状态及状态转移的方式实现有穷自动机，例如使用 C++ 和 Java 的正则库，然而在实验中**我们并不希望同学们使用这种方法**。一方面，对于词法分析这种规则相对固定的情况，此方法并不高效；另一方面，亲自使用代码实现（手搓）有穷自动机的逻辑更能够加深各位对有穷自动机和词法分析相关知识的理解。

当使用代码模拟有穷自动机时，一段顺序代码段便对应了有穷自动机的某一状态，而分支、循环则意味着状态的转移。我们还以对标识符为例。标识符的有穷自动机等价于如下代码：

```
1 public class LexerAnalyzer {
2     private static final LexerAnalyzer lexerAnalyzer = new LexerAnalyzer();
3     private int lineNum = 1;    // 初始化行号
4     private int readIndex;     // 初始化主指针
5     private String code;       // 源代码
6     public void initLexerAnalyzer(String code){
7         this.code = code;
8     }
9     public void lexerAnalyze() {
10        int codeLength = code.length();
11        code += "\0\0";        // 避免指针移动到文件末尾的时候越界
12
13        // 主指针不断遍历 code，对不同的类型进行分类判断
14        for (readIndex = 0; readIndex < codeLength; readIndex++) {
15            char first = code.charAt(readIndex);
16            char second = code.charAt(readIndex + 1);
17            String doubleChars = "" + first + second;
18            String singleChar = "" + first;
19            if (first == '\n') {
20                lineNum++;
21            }
22            // 标识符
23            else if (Character.isLetter(first) || first == '_') {
24                handleIdentifier(code, codeLength, singleChar);
25            }
26            // int, char, ... 均同理
27        }
28    }
```

```

29     private void handleIdentifier(String code, int codeLength, String
startCode) {
30         StringBuilder identifier = new StringBuilder(startCode);
31         for (int j = readIndex + 1; j < codeLength; j++) {
32             char id = code.charAt(j);
33             if (!(Character.isLetterOrDigit(id) || id == '_')) {
34                 readIndex = j - 1;
35                 break;
36             }
37             identifier.append(id);
38         }
39         String identifierContent = identifier.toString();
40         TokenType identifierType;
41         if (reservedWords.containsKey(identifierContent)) {
42             identifierType = reservedWords.get(identifierContent);
43         } else {
44             identifierType = TokenType.IDENFR;
45         }
46         tokenArrayListManager.addToken(new Token(identifierContent,
identifierType, lineNum));
47     }
48 }

```

在标识符的识别过程中，词法分析器首先在主循环中发现当前字符 `first` 是一个字母或下划线，这表明可以进入标识符的处理函数 `handleIdentifier`。此时，使用 `StringBuilder identifier` 来构建标识符内容，并将起始字符（由 `first` 构成的 `startCode`）作为初始部分加入。

随后进入 `for` 循环，从 `readIndex + 1` 开始，逐个读取后续字符 `id`。只要 `id` 满足 `Character.isLetterOrDigit(id) || id == '_'` 的条件，就将其添加到 `identifier` 中，并保持当前状态，继续读取下一个字符。这个过程模拟了一个有限状态自动机处于“标识符识别中”状态下的持续状态转移。

一旦遇到第一个不满足上述条件的字符（即 `id` 不是字母、数字或下划线），说明该字符不能作为标识符的一部分，于是跳出循环，并将 `readIndex` 设置为当前字符前一位（即 `j - 1`），以保证主循环在下一轮可以重新处理这个非标识符字符。

接着，将 `identifier.toString()` 得到的字符串赋值为 `identifierContent`，用于判断该单词是否为保留字。通过查找 `reservedWords.containsKey(identifierContent)`，决定当前 `token` 的类型是保留字对应的 `TokenType`，还是普通标识符 `TokenType.IDENFR`。最后，将识别出的单词封装为一个 `Token` 对象，并调用 `tokenArrayListManager.addToken(...)` 将其加入 `token` 序列中。

整个识别过程以变量 `readIndex` 控制当前字符位置，以 `identifier` 累积字符构建识别单元，通过字符分类判断和逐步推进模拟有限自动机的状态转移行为，最终在状态无法继续转移时结束当前 `token` 的识别并生成结果。这个过程体现了典型的**基于状态控制和字符驱动的 DFA（确定有限自动机）思想**。