编译原理课程实践1：LEX词法分析器

实践报告

516021910528

申佳旻

john980118@sjtu.edu.cn

实践环境

编写环境

- 系统
  - Surface Book
  - 处理器：Intel Core i7-6600U
  - Windows：
    - Windows 1803 (OS内部版本 17134.345)
    - 64位操作系统，基于x64的处理器
  - 适用于 Linux 的 Windows 子系统：Ubuntu 16.04.5
- 编译器：
  - gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10)
    - Target: x86_64-linux-gnu
  - flex 2.6.0

测试环境

- 系统
  - 阿里云轻量应用服务器
  - Ubuntu 16.04.5 LTS
- 编译器
  - gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.10)
    - Target: x86_64-linux-gnu
  - flex 2.6.0

功能要求

基本功能

1. 指数转换 将代码中所有指数形式的常量转换为小数形式。
2. 加减法计算 在上述基础上，计算出所有常量加减法表达式的结果。这里假定表达式中不含空格和括号。

扩展功能

1. 识别二进制、八进制、十六进制整型字面量，并转换成十进制参与运算
2. 识别不超过$2^{64}-1$的整形数并参与运算，无法转换的整型字面量不做处理

实现思路

> 内容参考了Python2.7的说明文档
>
> https://docs.python.org/2.7/reference/lexical_analysis.html#numeric-literals

整数字面量的词法分析

Python文档中定义了整型和长整形字面量的词法规则如下：

```
longinteger    ::=  integer ("l" | "L")
integer        ::=  decimalinteger | octinteger | hexinteger | bininteger
decimalinteger ::=  nonzerodigit digit* | "0"
octinteger     ::=  "0" ("o" | "O") octdigit+ | "0" octdigit+
hexinteger     ::=  "0" ("x" | "X") hexdigit+
bininteger     ::=  "0" ("b" | "B") bindigit+
nonzerodigit   ::=  "1"..."9"
octdigit       ::=  "0"..."7"
bindigit       ::=  "0" | "1"
hexdigit       ::=  digit | "a"..."f" | "A"..."F"
```

- 整数可分为整型数和长整形数
- 整型数可分为十进制整数、八进制整数、十六进制整数和二进制整数，其中：
  - 十进制整数可以为 `0` ，非零的十进制整数首位不能为 `0` ，数字包括 `0-9`
  - 八进制整数以前导 `0o` 、 `0O` 或 `0` 开头，数字包括 `0-7`
  - 十六进制整数以 `0x` 或 `0X` 开头，数字包括 `0-9` 和 `a-f` 、 `A-F`
  - 二进制整数以 `0b` 开头，数字包括 `0` 和 `1`
- 长整形数在在整型数后添加 `l` 或 `L` 作为类型标识

由此，我们得到整型数涉及到的终结符包括作为数字的 `0123456789abcdefABCDEF` 以及用于标识类型的 `lLoOxXbB` ，可归纳其正规式如下：

```
digit           [0-9]
nonzerodigit    [1-9]
octdigit        [0-7]
bindigit        [01]
hexdigit        [0-9|a-f|A-f]

decimalinteger  ({nonzerodigit}{digit}*)|0
octinteger      0[oO]?{octdigit}+
hexinteger      0[xX]{hexdigit}+
bininteger      0[bB]{bindigit}+

integer         {decimalinteger}|{octinteger}|{hexinteger}|{bininteger}
longinteger     {integer}[lL]
```

下面是一些整数的例子

```
7     2147483647                        0177
3L    79228162514264337593543950336L    0377L   0x100000000L
      79228162514264337593543950336             0xdeadbeef
```

浮点数字面量的词法分析

Python文档中规定了浮点数字面量的词法规则如下：

```
floatnumber   ::=  pointfloat | exponentfloat
pointfloat    ::=  [intpart] fraction | intpart "."
exponentfloat ::=  (intpart | pointfloat) exponent
intpart       ::=  digit+
fraction      ::=  "." digit+
exponent      ::=  ("e" | "E") ["+" | "-"] digit+
```

- 浮点数可分为有小数点的浮点数和由指数构成的浮点数
- 有小数点的浮点数分为整数部分、小数点和小数部分
  - 整数部分和小数部分均由至少一位的数字组成
  - 整数部分和小数部分可以缺失一个，但小数点必须存在
- 由指数构成的浮点数分为尾数部分和幂数部分
  - 尾数部分可以是整数也可以是带有小数点的浮点数
  - 幂数部分只能是整数，可以带有正负号
  - 尾数和幂数之间可以用 `E` 或 `e` 连接

可以归纳其正规式如下：

```
intpart         {digit}+

pointfloat      ({intpart}?[.]{intpart})|({intpart}[.])
exponentfloat   ({intpart}|{pointfloat})[eE][-+]?{intpart}

floatnumber     {pointfloat}|{exponentfloat}
```

在Python中，小数和浮点数的一部分可能由 `0` 开头，构成一个八进制数的格式，但会被按照十进制解析。例如，`077e010` 会被解析成 `77e10` 。

下面是一些浮点数的例子：

```
3.14    10.    .001    1e100    3.14e-10    0e0
```

字面量的识别和转换

在实现的过程中需要对此做出一些调整。

首先，对于整数和浮点数的解析应当尽量使用C标准库中的 `strtoll` 和 `strtof` 两个函数进行解析，以减少代码量。但由于C语言和Python语言在一些细节上的差异，部分特殊的情况需要单独进行处理

整型数的解析

> 内容参考了cppreference上对于 `strtol` ， `strtoll` 的解释
>
> https://en.cppreference.com/w/c/string/byte/strtol

- 在Python文档中说明，在x86-32体系中，超过2147483647（$2^{31}-1$）的整数将被视为长整形数。我们这里做了部分简化。即识别末尾的 `l` 和 `L` 标记，但全部使用64位有符号数（ `int64_t` ）进行计算。
- `strtoll` 函数声明如下

  ```
  long long strtoll( const char *restrict str, char **restrict str_end, int base );
  ```

  根据文档说明，当 `base` 为 `0` 时，函数可以自动检测数值进制。但只能检测八进制、十进制和十六进制。且对于八进制字面量只能识别以 `0` 作为前缀的情况，以 `0o` 和 `0O` 作为前缀的情况需要单独处理。因此二进制整型字面量和八进制整型字面量需分别单独处理。

由此，整数部分的正规式可修改如下：

```
digit           [0-9]
nonzerodigit    [1-9]
octdigit        [0-7]
bindigit        [01]
hexdigit        [0-9|a-f|A-f]

decimalinteger  ({nonzerodigit}{digit}*)|0
octinteger      0[oO]?{octdigit}+
hexinteger      0[xX]{hexdigit}+
bininteger      0[bB]{bindigit}+

autointeger     ({decimalinteger}|{hexinteger})[lL]?
handoint        {octinteger}[lL]?
handbint        {bininteger}[lL]?
```

浮点数的输出问题

> 内容参考了cppreference上对于 `printf` ， `fprintf` ， `sprintf` 的解释
>
> https://en.cppreference.com/w/c/io/fprintf

注：后经咨询助教，只需保留到小数点后5-6位，遂将原有代码注释掉，改为输出小数点后6位。

对于十进制小数的输出，C语言提供了 `f` 、 `e` 、 `a` 和 `g` 几种格式。其中 `a` 格式用于输出浮点数的十六进制表示形式，不符合我们的应用要求。 `e` 格式和 `f` 格式需要完全手动地指定输出浮点数格式的精度，如精度设置过大则会多处许多后缀0，实现起来比较棘手。 `g` 格式在指定一个较大的精度之后可以自动消除后缀0，使用起来相对方便。

但使用 `g` 格式输出仍然面临几个问题。

首先，我们仍然要指定一个略大的精度值，以保证所有的小数都被输出出来。这里我选择使用匹配串的长度 `yyleng` 直接作为浮点数的精度。因为源串一定不可能使用比串长度少的字符表示比串长度更高的精度。

其次，使用 `g` 格式输出将导致某些情况下浮点数被以指数的形式输出出来。具体来讲，当使用小数形式输出的字符串长度超过使用指数形式输出所需字符串长度时，就会以指数的形式输出浮点数。这时我们应当找到一个适当的精度，强制使用 `f` 格式输出浮点数。

- 对于一个大于1的浮点数，若小数表示长于指数表示，则这个数的小数部分必为0。否则一定是使用小数形式表示更短。因此当浮点数大于1时，应当使用 `.1f` 格式输出浮点数。
- 对于一个小于1的浮点数，其幂数一定是负数。很容易发现这个浮点数作为小数输出时，将精度设置为尾数的精度于幂数的绝对值之和，恰好可以输出全部的有效位。

计算问题

要实现表达式的运算，可以先识别出每一个带符号的字面量，然后通过判断两字面量是否连续判断其是否构成一个表达式进行运算。也可以先识别出整个表达式，再逐一识别里面带符号的字面量进行合并计算。更一般的做法应当是将字面量和运算符视作两类单词，然后在文法分析的阶段进行合并计算。

在这里我找到了flex支持的 `REJECT` 操作，这一操作可以实现拒绝当前已匹配的最优规则，要求分析器寻找次优规则来匹配这一字符串。利用这一操作和适当的状态管理，我们可以比较容易地实现先识别出一个表达式，然后对这一表达式重新进行词法分析，以识别出里面包含的字面量，再进行合并计算。

因此我们还应补充带符号字面量的正规式定义和表达式的正规式定义。其中带符号的字面量只需在原有整数和浮点数前添加符号，而只含加减法的表达式可以由带符号字面量直接拼接而成。正规式修改如下：

```
digit           [0-9]
nonzerodigit    [1-9]
octdigit        [0-7]
bindigit        [01]
hexdigit        [0-9|a-f|A-f]

decimalinteger  ({nonzerodigit}{digit}*)|0
octinteger      0[oO]?{octdigit}+
hexinteger      0[xX]{hexdigit}+
bininteger      0[bB]{bindigit}+

autointeger     [-+]?({decimalinteger}|{hexinteger})[lL]?
handoint        [-+]?{octinteger}[lL]?
handbint        [-+]?{bininteger}[lL]?

intpart         {digit}+

pointfloat      ({intpart}?[.]{intpart})|({intpart}[.])
exponentfloat   ({intpart}|{pointfloat})[eE][-+]?{intpart}

floatnumber     [-+]?({pointfloat}|{exponentfloat})

```

```
23    expressions     ({autointeger}|{handoint}|{handbint}|{floatnumber})+
```

实践感想

在拿到这份作业的时候并不觉得难度很大，但是实际动手的时候才发现有很多细节的方面需要考虑。

在完成这份文档之前，其实我先写了另外一个版本。但实现得很蹩脚，许多细节的地方（比如浮点数的输出格式、不同进制的不同形式的表达）都没有考虑到。直到开始写这份报告的时候才想起来，我应该去查一下Python官方对于这门语言词法规则的说明。同时这一次查阅文档也让我对Python的文档有了新的理解。

最后，我还是认为表达式计算应当属于语义分析的内容。