词法分析

1 While 语言

- 常数
  - `N ::= 0 | 1 | ...`
  - While 语言的常数是以非 0 数字开头的一串数字或者 0。
- 保留字
  - While 语言的保留字有：var, if, then, else, while, do
- 变量名
  - `V ::= ...`
  - While 语言变量名的第一个字符为字母或下划线，while 语言的变量名可以包含字母、下划线与数字。
  - 保留字不是变量名。
  - 例如：`a0`, `__x`, `leaf_counter`等等都可以是 while 语言的变量名。
- 表达式
  - `E :: = N | V | -E | E+E | E-E | E*E | E/E | E%E |`
    `E<E | E<=E | E==E | E!=E | E>=E | E>E |`
    `E&&E | E||E | !E`
  - 优先级：`|| < && < ! < Comparisons < +,- < *,/,%`
  - 同优先级运算符之间左结合，可以使用小括号改变优先级。
  - 例如：`a0+1`, `x<=y&&x>0`等等都是 while 语言的表达式。
- 语句

分号是语句的分隔符，而不是结束符号。

```
C :: = var V |
       V = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C }
```

大括号约定要有

到这里为止我们就定义了一个最简单的程序语言 while 语言。我们已经可以用这个程序语言写一些比较复杂的程序了。

2 While 语言 + Dereference + Built-in functions

表达式

```
E ::= N | V | -E | E+E | E-E | E*E | E/E | E%E |
      E<E | E<=E | E==E | E!=E | E>=E | E>E |
      E&&E | E||E | !E | *E |
      malloc(E) | read_int() | read_char()
```

语句

```
C ::= var V |
      write_int(E) |
      write_char(E) |
      E = E |
      C; C |
      if (E) then { C } else { C } |
      while (E) do { C }
```

While 程序示例 1

```
var x;
var n;
var flag;
x = read_int();
n = 2;
flag = 1;
while (n * n <= x && flag) do {
  if (x % n == 0)
  then { flag = 0 }
  else { n = n + 1 }
};
write_int(flag)
```

While 程序示例 2

```
var x;
var l;
var h;
var mid;
x = read_int();
l = 0;
h = x + 1;
while (l + 1 < h) do {
  mid = (l + h) / 2;
  if (mid * mid <= x)
  then { l = mid }
  else { h = mid }
};
write_int(l)
```

While 程序示例 3

```
var n;
var s;
s = 0;
n = 1;
while (n != 0) do {
  n = read_int();
  s = s + n
};
write_int(s)
```

3 词法分析的基本知识

词法分析是编译器前端的第一步，它的主要任务是将源代码的字符串切分为一个一个的标记（token）。例如在我们的 While+DB 语言中有以下几类标记：

- 运算符：`+ - * / % < <= == != >= > ! && ||`
- 赋值符号：`=`
- 间隔符：`( ) { } ;`
- 自然数：`0 1 2 ...`
- 变量名：`a0 __x ...`
- 保留字：`var if then else while do`
- 内置函数名：`malloc read_char read_int write_char write_int`

怎么切分？

根据不同标记承担的语法功能是否相同，可以将标记分为若干类。事实上，在上述列举的标记除了所有自然数分为一类，所有变量名分为一类之外，其他每个标记应当单独分为一类。下面是标记分类在 C 语言中的定义（lang.h）。

```
enum token_class {
  // 运算符
    TOK_OR = 1, TOK_AND, TOK_NOT,
    TOK_LT, TOK_LE, TOK_GT, TOK_GE, TOK_EQ, TOK_NE,
    TOK_PLUS, TOK_MINUS, TOK_MUL, TOK_DIV, TOK_MOD,
  // 赋值符号
    TOK_ASGNOP,
  // 间隔符号
    TOK_LEFT_BRACE, TOK_RIGHT_BRACE,
    TOK_LEFT_PAREN, TOK_RIGHT_PAREN,
    TOK_SEMICOL,
  // 自然数
    TOK_NAT,
  // 变量名
    TOK_IDENT,
  // 保留字
    TOK_VAR, TOK_IF, TOK_THEN, TOK_ELSE, TOK_WHILE, TOK_DO,
  // 内置函数名
    TOK_MALLOC, TOK_RI, TOK_RC, TOK_WI, TOK_WC
};
```

其中，`TOK_NAT` 与 `TOK_IDENT` 两类标记需要额外存储他们的值。这个 C 语言的 `union` 说的是，如果标记属于 `TOK_NAT`，那么就在 `n` 这个域中存储这个标记表示的无符号整数值，如果标记属于 `TOK_IDENT`，那么就在 `i` 这个域中存储这个标识符对应的字符串的地址，其他情况下，不需要存储额外的信息。

```
union token_value {
  unsigned int n;
  char * i;
  void * none;
};
```

4 正则表达式

正则表达式是用于描述字符串集合的一种语言。正则表达式本身的语法结构定义如下：

单个字符（c）| 表示空字符串（e）| 字符串并集（r1 | r2）| 连接（rr）| r*

- $r ::= c \mid \epsilon \mid r|r \mid rr \mid r^*$
- 优先级：* > 连接 > |

具体而言，正则表达式可以表达五种意思。第一，单个字符；第二，空字符串；第三，两个字符串的连接；第四：两类字符串的并集；第五：重复出现一类字符串 0 次 1 次或多次。例如：

- `(a|b)c` 表达的字符串有 `ac`, `bc`。
- `(a|b)*` 表达的字符串有空串, `a`, `b`, `ab`, `aaaaa`, `babbb` 等。
- `ab*` 表达的字符串有 `a`, `ab`, `abb`, `abbb`, `abbbb` 等。
- `(ab)*` 表达的字符串有空串, `ab`, `abab` 等。

下面是正则表达式中的一些常见简写：

- 字符的集合：例如 `[a-cA-C]` 表示 `a|b|c|A|B|C`。
- 可选的字符串：$r?$ 表示 $r|\epsilon$。例如 `a?b` 表达的字符串有 `b`, `ab`。
- 字符串：例如 `"abc"` 表达的字符串是 `abc`。
- 重复至少一次：例如 `a+` 表达的字符串是 `a`, `aa`, `aaa`, 等等。
- 自然数常量：`"0"|[1-9][0-9]*`。
- 标识符（不排除保留字、内置函数名）：`[_a-zA-Z][_a-zA-Z0-9]*`。

5 Flex 词法分析工具

安装 Flex

- 请预先在电脑上安装 flex，windows 系统上的安装方式请参考：
- `blog.csdn.net/m944256098a/article/details/104992880`
- 输入（.l 文件）：基于正则表达式的词法分析规则
- 输出（.c 文件）：用 C 语言实现的词法分析器

输入文件头

- 说明 Flex 生成词法分析器的基本参数
- 指定所生成的词法分析器的文件名
- 词法分析器所需头文件 (.h 文件) 与辅助函数

- 例如（lang.l）：

```
%option noyywrap yylineno
%option outfile="lexer.c" header-file="lexer.h"
%{
#include "lang.h"
%}
```

第一行照抄

词法分析用到的辅助函数和全局变量 lang.h

```
extern union token_value val;
unsigned int build_nat(char * c, int len);
char * new_str(char * str, int len);
void print_token(enum token_class c);
```

词法分析规则

- 所有词法分析规则都放在一组 `%%` 中。
- 每一条词法分析规则由匹配规则和处理程序两部分构成。
- 匹配规则用正则表达式表示。
- 处理程序是一段 C 代码。
- 例如（lang.l）：

```
%%
0|[1-9][0-9]* {
    val.n = build_nat(yytext, yyleng);
    return TOK_NAT;
}
"var" { return TOK_VAR; }
...
%%
```

Flex 专有 C 程序变量

- `yytext`：经过词法分析切分得到的字符串。
- `yyleng`：经过词法分析切分得到的字符串的长度。

Flex 词法分析的附加规则

- 如果有多种可行的切分方案，则选择长度较长的方案。
- 如果同一种切分方案符合多个词法分析规则（正则表达式），则选择先出现的规则。
- `==` 会匹配 `==` 而不是 `=`。
- 标识符的词法分析规则不需要排除保留字，只需在 Flex 中把保留字放在标识符之前即可。

lang.l

```
%%
...
"write_int" { return TOK_WI; }
"write_char" { return TOK_WC;}
[_A-Za-z][_A-Za-z0-9]* {   变量名
    val.i = new_str(yytext, yyleng);
    return TOK_IDENT;
}
";" { return TOK_SEMICOL; }
"(" { return TOK_LEFT_PAREN; }
...
%%
```

main.c （只打印词法分析结果）

```
#include "lang.h"
#include "lexer.h" 有哪些函数是词法分析器提供调用的
int main() {
  enum token_class c;
  while (1) {
    c = yylex();
    if (c != 0) {
      print_token(c);
    }
    else {
      break;
    }
  }
}
```

编译 Makefile

```
lang.o: lang.c lang.h
 tab      gcc -c lang.c
lexer.c: lang.l
        flex lang.l
lexer.o: lexer.c lang.h
        gcc -c lexer.c
main.o: main.c lang.h lexer.c
编译目标  gcc -c main.c
main: main.o lang.o lexer.o       依赖文件
        gcc main.o lang.o lexer.o -o main
%.c: %.l 编译器                    生成可执行文件
```

测试 sample00.jtl

```
  var n;
  var m;
  n = read_int();
  m = n + 1;
  write_int(m + 2)
```

```
Qinxiang$ ./main < sample00.jtl
VAR
IDENT(n)
SEMICOL
VAR
IDENT(m)
...
```

Flex 生成的词法分析器其算法流程如下：先根据输入的字符串找到正则表达式的匹配结果（若有多种结果则选最长、最靠前的），再更新字符串扫描的初始地址，并执行匹配结果对应的处理程序。

如果处理程序中有`return`语句中断了词法分析器`yylex()`的执行，下一次重新启动`yylex()`时会从新的字符串扫描初始地址开始继续进行词法分析。如果处理程序中没有`return`语句中断，那么词法分析器就会自动继续执行后续的词法分析。

因此，处理程序中可以不用每次 return，相应的也不用反复重新启动`yylex()`。下面为代码概要：

lang.l

```
%%
0|[1-9][0-9]* {
    val.n = build_nat(yytext, yyleng);
    print_token(TOK_NAT);
}
"var" { print_token(TOK_VAR); }
...
%%
```

main.c

```
int main() {
  yylex();
}
```

```
Qinxiang$ ./main < sample00.jtl
VAR
IDENT(n)
SEMICOL
VAR
IDENT(m)
...
```

Flex 生成的词法分析器其算法流程如下：先根据输入的字符串找到正则表达式的匹配结果（若有多种结果则选最长、最靠前的），再更新字符串扫描的初始地址，并执行匹配结果对应的处理程序。

如果处理程序中有`return`语句中断了词法分析器`yylex()`的执行，下一次重新启动`yylex()`时会从新的字符串扫描初始地址开始继续进行词法分析。如果处理程序中没有`return`语句中断，那么词法分析器就会自动继续执行后续的词法分析。

因此，处理程序中可以不用每次 return，相应的也不用反复重新启动`yylex()`。下面为代码概要：

lang.l

```
%%
0|[1-9][0-9]* {
    val.n = build_nat(yytext, yyleng);
    print_token(TOK_NAT);
}
"var" { print_token(TOK_VAR); }
...
%%
```

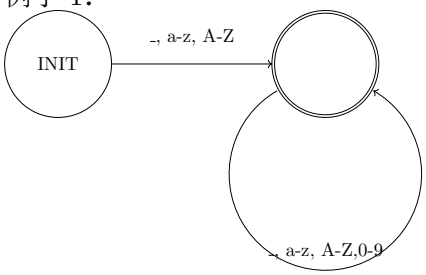
main.c

```
int main() {
  yylex();
}
```

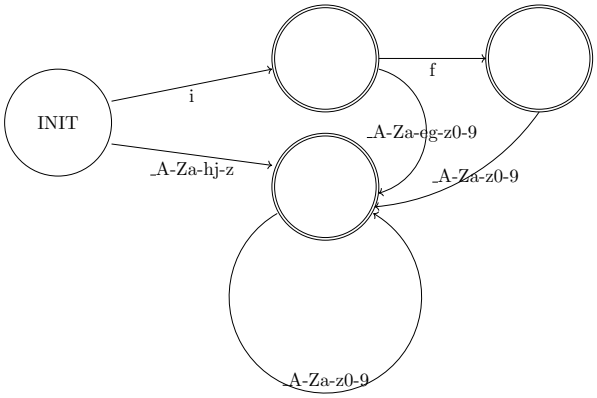
6 有限状态自动机

- 有限状态自动机包含一个状态集（图中的节点）与一组状态转移规则（图中的边）。
- 每一条状态转移规则上有一个符号
- 状态集中包含一个起始状态，和一组终止状态；起始状态也可能是一个终止状态。

例子 1：



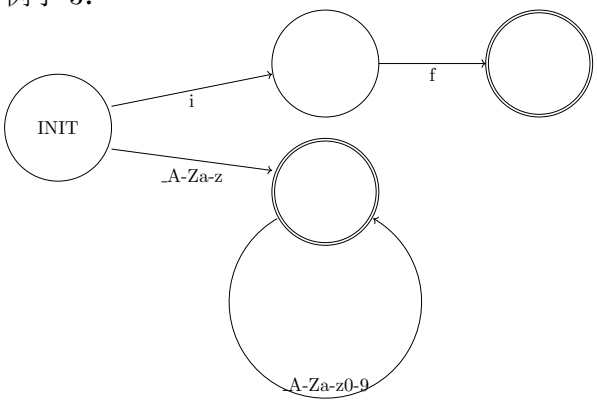
例子 2：



有多个终止状态，且在之间可以转移

下面这个自动机的例子与前面不同，从起点出发，经过字符 `i` 能到达的节点有两个，这也是自动机。

例子 3：



NFA

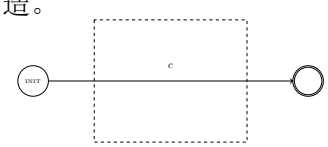
- 确定性有限状态自动机（DFA，Deterministic Finite Automata）
  - 每一条状态转移规则上的符号都是 ascii 码字符。
  - 从任何一个状态出发，每个字符至多对应一条状态转移规则。
- 非确定性有限状态自动机（NFA，Nondeterministic Finite Automata）
  - 每一条状态转移规则上的符号要么是 ascii 码字符，要么是 ϵ。
  - 从一个状态出发，每个符号可能对应多条状态转移规则。
  - 通过 ϵ 转移规则时，不消耗字符串中的字符。
- 定义：一个自动机能接受一个字符串，当且仅当存在一种状态转移的方法，可以从自动机的起始状态出发，依次经过这个字符串的所有字符，到达一个终止状态。

7 将正则表达式转化为 NFA

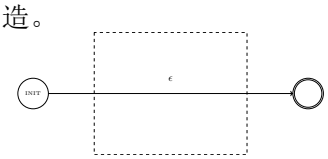
每个正则表达式构造一个 NFA，使得一个字符串能被这个 NFA 接受当且仅当这个字符串是正则表达式表示的字符串集合的元素。在下面定义的构造中，我们构造出的 NFA 总之只有一个起点、一个终点。但是请特别注意：起点也可能在图中有入度，终点也可能在图中有出度。

?

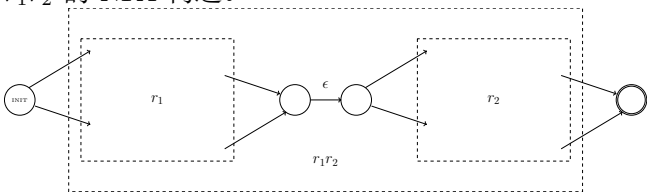
c 的 NFA 构造。



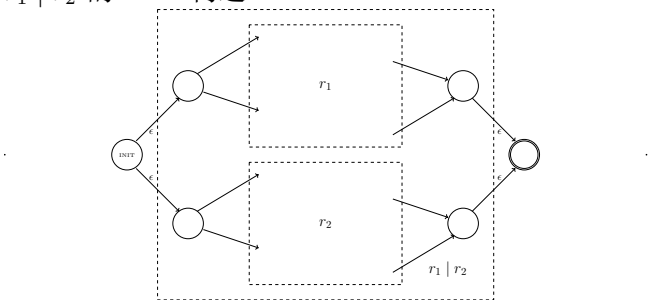
ϵ 的 NFA 构造。



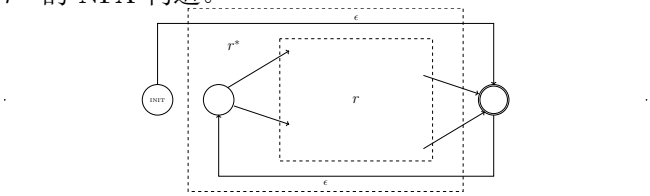
$r_1 r_2$ 的 NFA 构造。



$r_1 \mid r_2$ 的 NFA 构造。



r^* 的 NFA 构造。

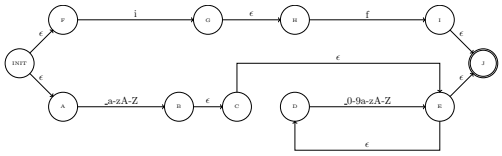


8 将 NFA 转化为 DFA

关键方法：考虑在 NFA 中所有当前可能处于的状态的集合。

例子：

- 正则表达式：`if|[_a-zA-Z][_0-9a-zA-Z]*`
- NFA：



- 起始状态：{ INIT, A, F }　空串，可能在INIT，A，F三个节点

  i: { INIT, A, F } → { B, C, D, E, G, H, J }

  f: { B, C, D, E, G, H, J } → { D, E, I, J }

  _0-9a-zA-Z: { D, E, I, J } → { D, E, J }

  _0-9a-zA-Z: { D, E, J } → { D, E, J }

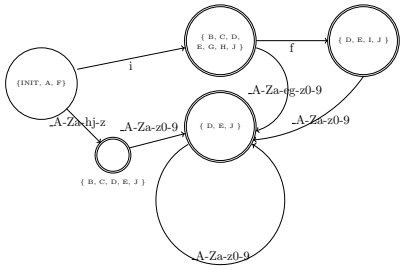
  _0-9a-eg-zA-Z: { B, C, D, E, G, H, J } → { D, E, J }

  ?

  _0-9a-hj-zA-Z: { INIT, A, F } → { B, C, D, E, J }

  _0-9a-zA-Z: { B, C, D, E, J } → { D, E, J }

- 构造生成的 DFA：　理论上来说是2的N次方，但是实际情况（语法分析）比较简单


{ B, C, D,
E, G, H, J }
{ D, E, I, J }
{INIT, A, F}
i
f
_A-Za-eg-z0-9
_A-Za-hj-z
_A-Za-z0-9
{ D, E, J }
_A-Za-z0-9
{ B, C, D, E, J }
_A-Za-z0-9