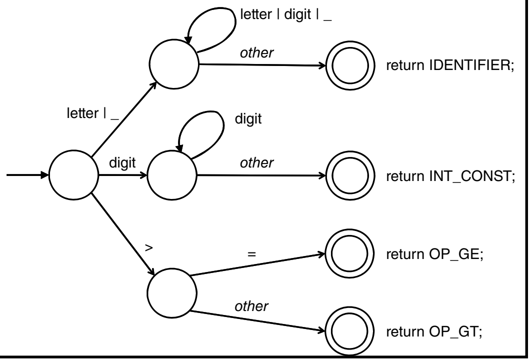
Scanner Automaton



Ambiguity Resolution

Imagine a rule for C identifiers:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

And the rule for a keyword such as if:

```
"if"
```

How do we resolve the fact that `if` is a keyword and `if8` is an identifier?

Two rules:

1. **Longest match** – The match with the longest string will be chosen.
2. **Rule priority** – for two matches of the same length, the first regex will be chosen. I.e., Rule order matters.

Generating Lexical Analyzers

Lexer Generators

Lex and flex – Tools to generate table-driven lexical analyzers from regular expressions.

Lex and flex work primarily with C.

Java versions exist. We will use JFlex as it's input format is close to flex for C.

The general format of these tools is to take an input file in the following format:

```
REGEX1 { Java code to do on match }
REGEX2 { Java code to do on match }
…
```

JFlex

```
import java.util.*;
%%
%class SimpleLexer
%{
private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
}
private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
}
%}
WhiteSpace = [ \t\r\n]
Identifier = [a-zA-Z_][a-zA-Z0-9_]*
Integer = 0 | [1-9][0-9]*
%%
"+"     { return symbol(PLUS); }
{Integer} { return symbol(INTEGER, Integer.parseInt(yytext())); }
```

JFlex

```
import java.util.*;
%%
%class SimpleLexer
%{
private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
}
private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
}
%}
WhiteSpace = [ \t\r\n]
Identifier = [a-zA-Z_][a-zA-Z0-9_]*
Integer = 0 | [1-9][0-9]*
%%
"+"     { return symbol(PLUS); }
{Integer} { return symbol(INTEGER, Integer.parseInt(yytext())); }
```

Divides sections

JFlex

1. User Code – The text up to the first line starting with %% is copied verbatim to the top of the generated lexer class (before the actual class declaration).

```
import java.util.*;
%%
%class SimpleLexer
%{
private Symbol symbol(int type) {
       return new Symbol(type, yyline, yycolumn);
}
private Symbol symbol(int type, Object value) {
       return new Symbol(type, yyline, yycolumn, value);
}
%}
WhiteSpace = [ \t\r\n]
Identifier = [a-zA-Z_][a-zA-Z0-9_]*
Integer = 0 | [1-9][0-9]*
%%
"+"     { return symbol(PLUS); }
{Integer} { return symbol(INTEGER, Integer.parseInt(yytext())); }
```

JFlex

2. Options and Declarations – It consists of a set of options, code that is included inside the generated scanner class, lexical states and macro declarations.

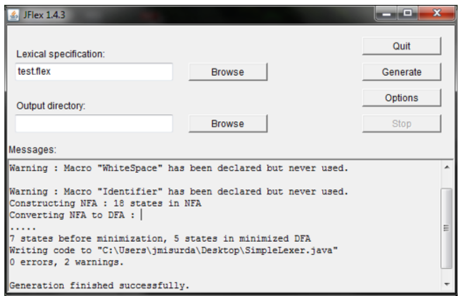
```
import java.util.*;
%%
%class SimpleLexer
%{
private Symbol symbol(int type) {
       return new Symbol(type, yyline, yycolumn);
}
private Symbol symbol(int type, Object value) {
       return new Symbol(type, yyline, yycolumn, value);
}
%}
WhiteSpace = [ \t\r\n]
Identifier = [a-zA-Z_][a-zA-Z0-9_]*
Integer = 0 | [1-9][0-9]*
%%
"+"     { return symbol(PLUS); }
{Integer} { return symbol(INTEGER, Integer.parseInt(yytext())); }
```

JFlex

3. Lexical Rules – Regular expressions and actions (Java code) that are executed when the scanner matches the associated regular expression.

```
import java.util.*;
%%
%class SimpleLexer
%{
private Symbol symbol(int type) {
       return new Symbol(type, yyline, yycolumn);
}
private Symbol symbol(int type, Object value) {
       return new Symbol(type, yyline, yycolumn, value);
}
%}
WhiteSpace = [ \t\r\n]
Identifier = [a-zA-Z_][a-zA-Z0-9_]*
Integer = 0 | [1-9][0-9]*
%%
"+"     { return symbol(PLUS); }
{Integer} { return symbol(INTEGER, Integer.parseInt(yytext())); }
```

JFlex



Lexical specification: test.flex

Output directory:

Messages:

```
Warning : Macro "WhiteSpace" has been declared but never used.

Warning : Macro "Identifier" has been declared but never used.
Constructing NFA : 18 states in NFA
Converting NFA to DFA : |
.....
7 states before minimization, 5 states in minimized DFA
Writing code to "C:\Users\jmisurda\Desktop\SimpleLexer.java"
0 errors, 2 warnings.

Generation finished successfully.
```

Generated Code

```
/** The transition table of the DFA*/
private static final String ZZ_TRANS_PACKED_0 =
    "\1\0\1\2\1\3\1\4\2\0\2\2\12\0\1\4\1\5";
                                ⋮
switch (zzAction < 0 ? zzAction : ZZ_ACTION[zzAction]) {
    case 2: { return symbol(PLUS);}
    case 3: break;
    case 1: { return symbol(INTEGER,Integer.parseInt(yytext()));}
    case 4: break;
    default:
        if (zzInput == YYEOF && zzStartRead == zzCurrentPos) {
            zzAtEOF = true;
            return null;
        } else {
            zzScanError(ZZ_NO_MATCH);
        }
}
```

Conclusions

Regular languages are powerful and convenient ways to accomplish lexical analysis

- Regular language can describe keywords, numbers, strings, comments, identifiers, email addresses, phone numbers, etc.
- However, it is the weakest formal language

Many languages are not regular

- C programming language is not
- "(((...)))" is also not

Finite automata cannot remember # of times

- We need more powerful languages for describing these structures

In the next part of the class, we introduce **context-free languages** which can express more complex things.