Scanning

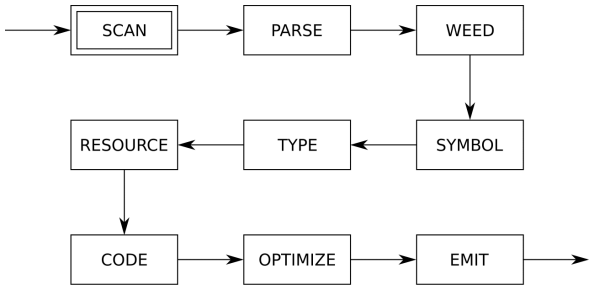
COMP 520: Compiler Design (4 credits)

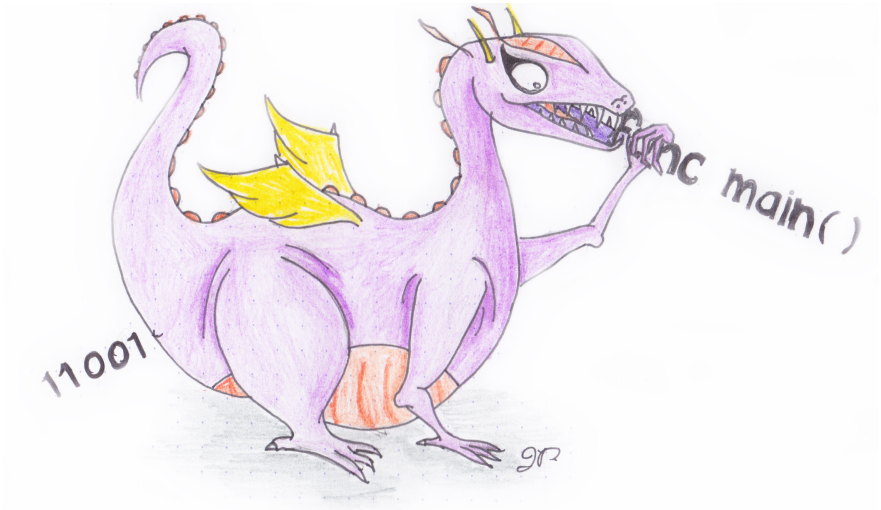
Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/





Announcements (Wednesday, January 9th)

Milestones

- Pick your group (3 recommended)
- Create a GitHub account, learn git as needed

Midterm

- **Date**: *Hopefully* Tuesday/Wednesday, February 26/27 from 18:00-19:30 (To be confirmed)
- Let me know if there are any conflicts!

Readings

Textbook, Crafting a Compiler

- Chapter 2: *A Simple Compiler*
- Chapter 3: *Scanning–Theory and Practice*

Modern Compiler Implementation in Java

- Chapter 1: *Introduction*
- Chapter 2: *Lexical Analysis*

Flex tool

- Manual - `https://github.com/westes/flex`
- Reference book, Flex & bison - `http://mcgill.worldcat.org/title/flex-bison/oclc/457179470`

Scanning

The scanning phase of a compiler

- Is the first phase of a compiler;
- Is also called lexical analysis (Google – “relating to the words or vocabulary of a language”);
- Takes arbitrary source files as input;
- Identifies meaningful sequences of characters; and
- Outputs tokens (one per meaningful sequence).

Overall

- A scanner transforms a string of characters into a string of tokens.
- *Note: at this point, we do not have any semantic or syntactic information*

Example

```
var a = 5
if (a == 5)
{
    print "success"
}
```

```
tVAR
tIDENTIFIER(a)
tASSIGN
tINTEGER(5)
tIF
tLPAREN
tIDENTIFIER(a)
tEQUALS
tINTEGER(5)
tRPAREN
tLBRACE
tIDENTIFIER(print)
tSTRING(success)
tRBRACE
```

Things of note

- Keywords are special sequences of characters that take precedence over any other rule (reserved) and are part of the language;
- Tokens may have associated data (identifiers, constants, etc); and
- Whitespace is ignored.

COMP 330 Review

Languages

- Σ is an *alphabet*, a (usually finite) set of symbols;
- A *word* is a finite sequence of symbols from an alphabet;
- Σ^* is a set consisting of all possible words using symbols from Σ; and
- A *language* is a subset of Σ^*.

Examples

- **Alphabet:** Σ={0,1}
- **Words:** {ϵ, 0, 1, 00, 01, 10, 11, . . . , 0001, 1000, . . .}
- **Language:**
  - {1, 10, 100, 1000, 10000, 100000, . . .}: “1” followed by any number of zeros
  - {0, 1, 1000, 0011, 11111100, . . .}: *?!*

Regular Languages

A regular language

- Is a language for which a regular expression exists; or (equivalently)
- Is a language that can be accepted by a DFA (deterministic finite automaton).

A regular expression

- Is a string that defines a language (set of strings); and
- In fact, is a string that defines a *regular* language.

Regular Expressions

In a scanner, tokens are defined by *regular expressions*

- $\emptyset$ is a regular expression [the empty set: a language with no strings]
- ε is a regular expression [the empty string]
- a, where $a \in \Sigma$ is a regular expression [Σ is our alphabet]
- if M and N are regular expressions, then $M|N$ is a regular expression
  [alternation: either M or N]
- if M and N are regular expressions, then $M \cdot N$ is a regular expression
  [concatenation: M followed by N]
- if M is a regular expression, then M* is a regular expression
  [zero or more occurences of M]

What are M? and M^+?

Examples of Regular Expressions

Given a language with alphabet Σ={a,b}, the following are regular expressions

- a* = {ϵ, a, aa, aaa, aaaa, . . . }
- (ab)* = {ϵ, ab, abab, ababab, . . . }
- (a|b)* = {ϵ, a, b, aa, bb, ab, ba, . . . }
- a*ba* = strings with exactly 1 “b”
- (a|b)*b(a|b)* = strings with at least 1 “b”

Your turn

Write regular expressions for the following languages

- {a, aa, aaa, aaaa, . . . }
- {ab, ababab, abababab, . . . }
- Strings with at most one “b”

Are these languages regular?

Given the alphabet Σ={a,b,c}, write a regular expression for each language if possible

- n “a”s, followed by any number of “b”s, followed by n “a”s
- All sentences that contain exactly 1 “a”, exactly 2 “b”s, and any number of “c”s, in *any* order
- All sentences that contain an odd number of characters
- All sentences that contain an odd number of characters, and the middle character must be an “a”
- All sentences that contain an even number of “a”s, an even number of “b”s and an even number of “c”s in *any* order

Regular Expressions for Programming Languages

We can write regular expressions for the tokens in a source language with standard POSIX notation

- Simple operators: `"*"`, `"/"`, `"+"`, `"-"`
- Parentheses: `"("`, `")"`
- Integer constants: `0|([1-9][0-9]*)`
- Identifiers: `[a-zA-Z_][a-zA-Z0-9_]*`
- Keywords: `if`, `while`
- Whitespace: `[␣\t\n\r]+`

[. . .] defines a *character class*

- Matches a single character from a set (allows characters to be “alternated”); and
- Can be negated using “^” (i.e. `[^\n]`).

The wildcard character

- Is represented as “.” (dot); and
- Matches all characters except newlines (default in most implementations).

Finite State Machines

Internally, scanners use *finite state machines* (FSMs) to perform lexical analysis.

A finite state machine

- Represents a set of possible states for a system; and
- Uses transitions to link related states.

Intuitively, scanners use states to represent how much of each token they have seen so far. Transitions are executed for each input character, moving from one state to another.

A *deterministic finite automaton* (DFA)

- Is a machine which recognizes regular languages;
- For an input sequence of symbols, the automaton either *accepts* or *rejects* the string; and
- It works *deterministically* - that is given some input, there is only one sequence of steps.

DFAs – "Crafting a Compiler"

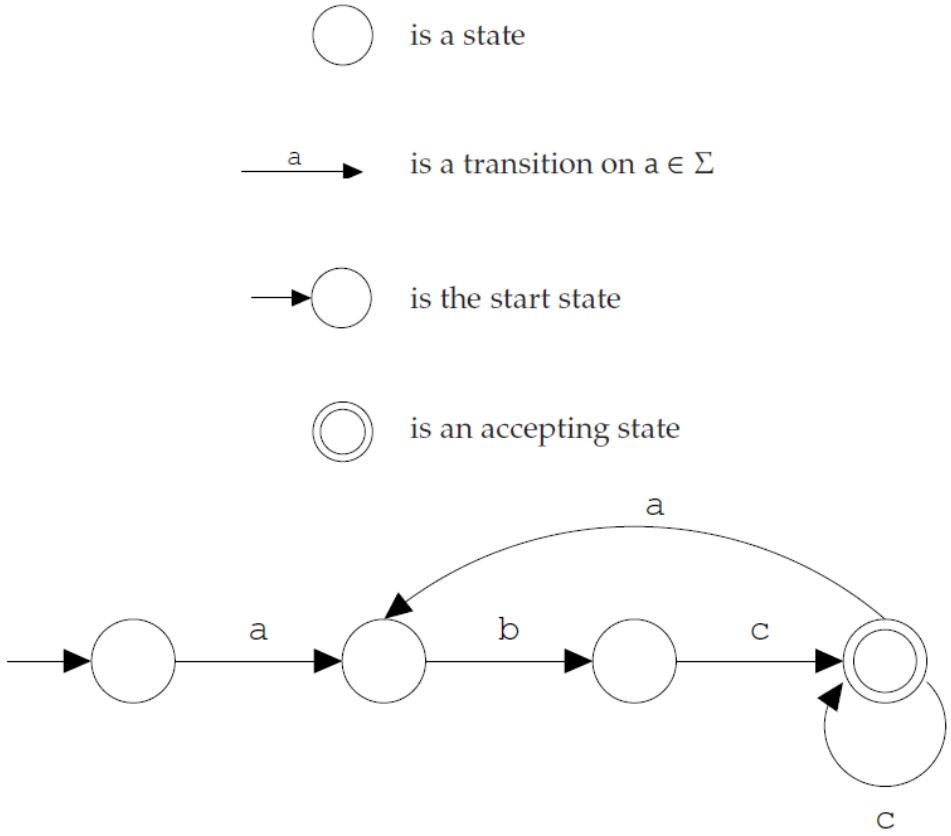
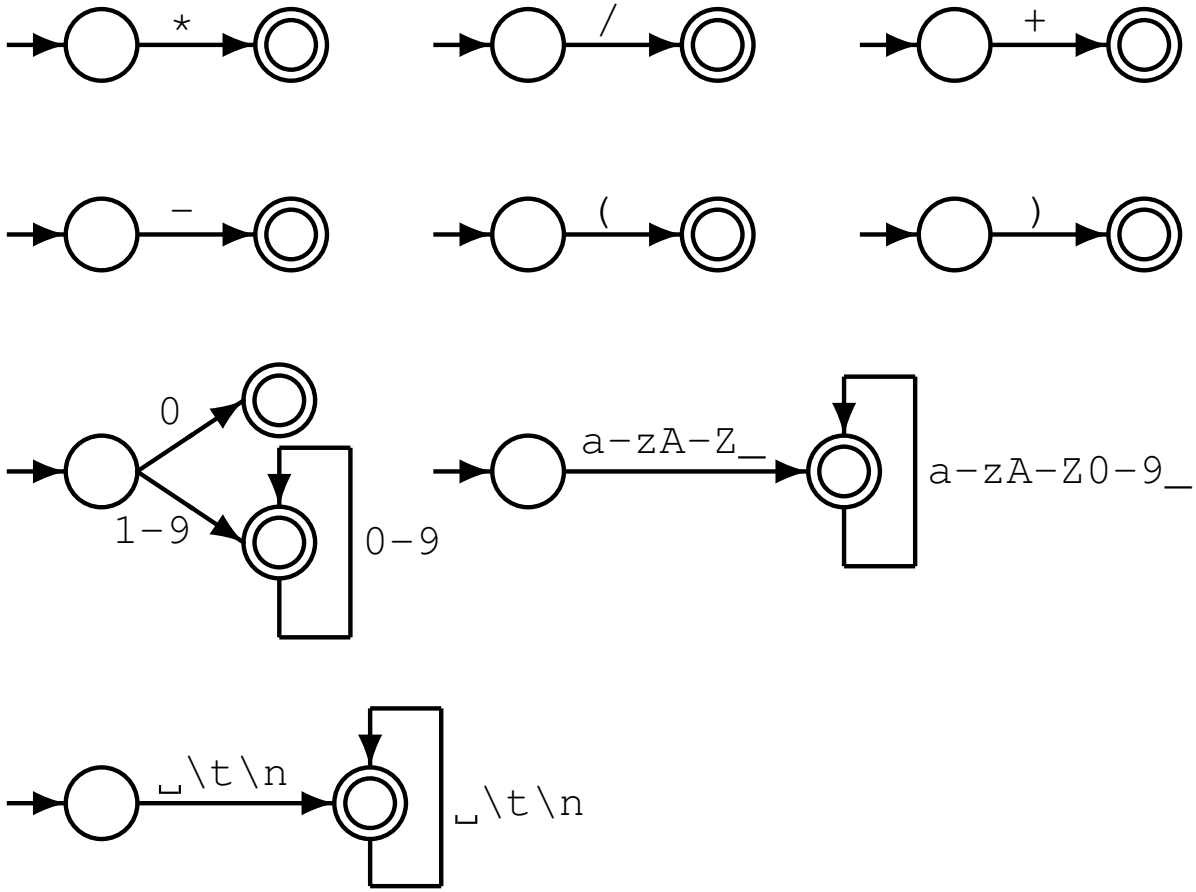


Figure 3.1: Components of a finite automaton drawing and their use to construct an automaton that recognizes $(a\,b\,c^+)^+$.

DFAs (for the previous example regexes)



Your Turn!

Design DFAs for the following languages

- Canonical example: binary strings divisible by 3 using only 3 states
- Recall the regex example: All sentences that contain an even number of “a”s, an even number of “b”s and an even number of “c”s in *any* order. Design a DFA using 8 states
- Floating point numbers of form: {1., 1.1, .1} (a digit on at least one side of the decimal)

The regular expression for the last example is easy, but (much) more complex for the other two

Nondeterministic finite automaton

Constructing a DFA directly from a regular expression is hard. A more popular construction involves an intermediate step with *nondeterministric finite automata*.

A nondeterministric finite automaton

- Is a machine which recognizes regular languages;
- For an input sequence of symbols, the automaton either *accepts* or *rejects* the string;
- It works *nondeterministically* - that is given some input, there is potentially more than one path; and
- An NFA accepts a string if at least one path leads to an accept.

Since they both recognize regular languages, DFAs and NFAs are equally powerful!

Regular Expressions to NFA (1) – "Crafting a Compiler"

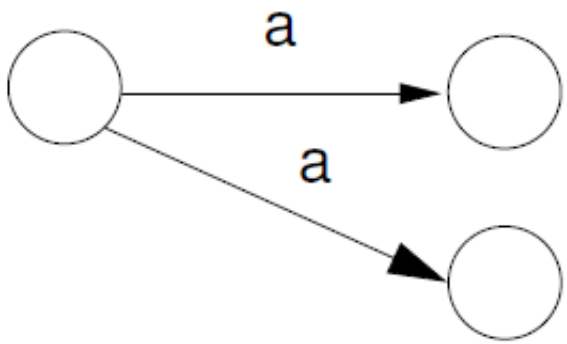


Figure 3.17: An NFA with two a transitions.

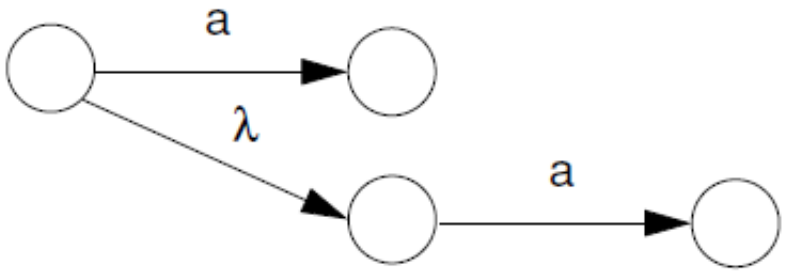


Figure 3.18: An NFA with a λ transition.

Regular Expressions to NFA (2) – “Crafting a Compiler"

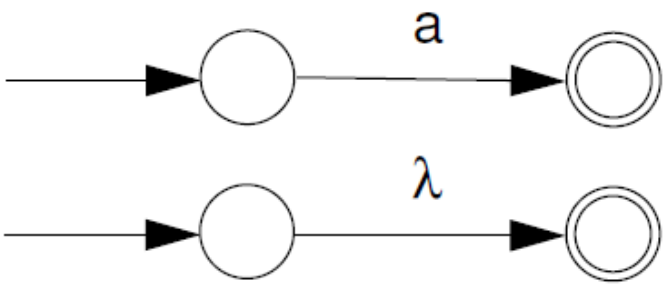


Figure 3.19: NFAs for a and λ.

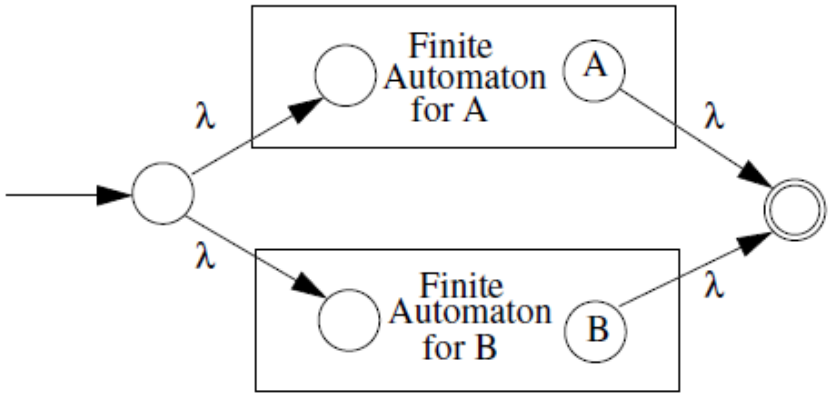


Figure 3.20: An NFA for $A \mid B$.

Regular Expressions to NFA (3) – “Crafting a Compiler"

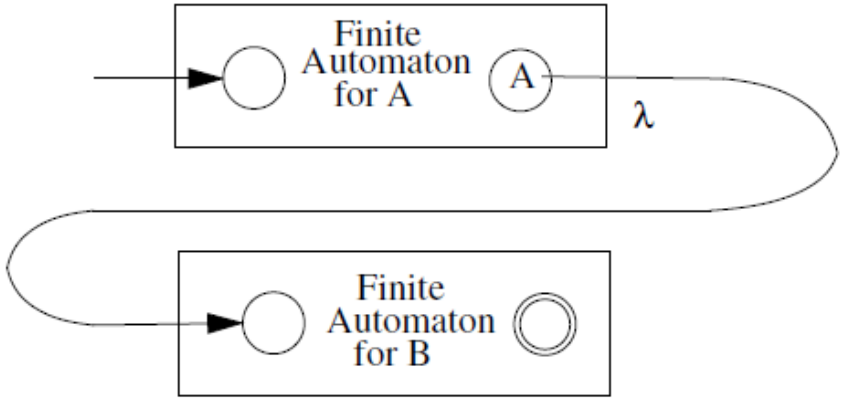


Figure 3.21: An NFA for AB.

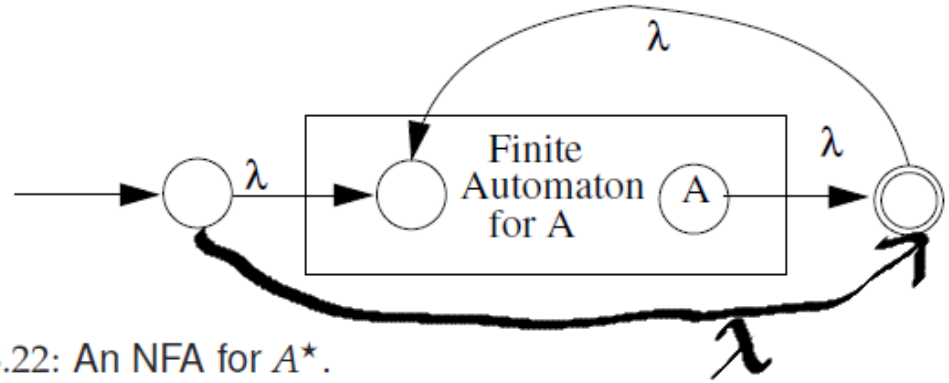


Figure 3.22: An NFA for $A^\star$.

Converting from Regular Expressions to DFAs

Internally, scanners use DFAs to recognize tokens - not regular expressions. Therefore, they must first perform a conversion. `flex` (your scanning tool) follows a well defined algorithm that

1. Accepts a list of regular expressions (regex);
2. Converts each regex internally to an NFA (Thompson construction);
3. Converts each NFA to a DFA (subset construction); and
4. May minimize DFA.

See “Crafting a Compiler", Chapter 3; or “Modern Compiler Implementation in Java", Chapter 2

Takeaways

You should know

1. Understand the definition of a regular language, whether that be: prose, regular expression, DFA, or NFA; and
2. Given the definition of a regular language, construct either a regular expression or an automaton.

You do not need to know

1. Specific algorithms for converting between regular language definitions; and
2. DFA minimization.

Announcements (Friday, January 11th)

Milestones

- Pick your group (3 recommended)
- Create a GitHub account, learn git as needed
- Learn `flex`/`bison` or `SableCC`

Midterm

- **Date**: To be determined, getting a room reserved is hard!

Office Hours

- **Monday**/**Wednesday**: 9:30-10:30
- If this does not work for you then please do send a message via email, Facebook group, etc.

Scanners

From your perspective, a scanner (or *lexer*)

- Can be generated using tools like `flex` (or `lex`), `JFlex`, `...`; and
- Is list of *regular expressions* (i.e. regular languages), one for each token type.

Internally, a *scanner*

- Transforms your regular expressions to *deterministic finite automata* (DFAs); and
- Adds some glue code to make it work.

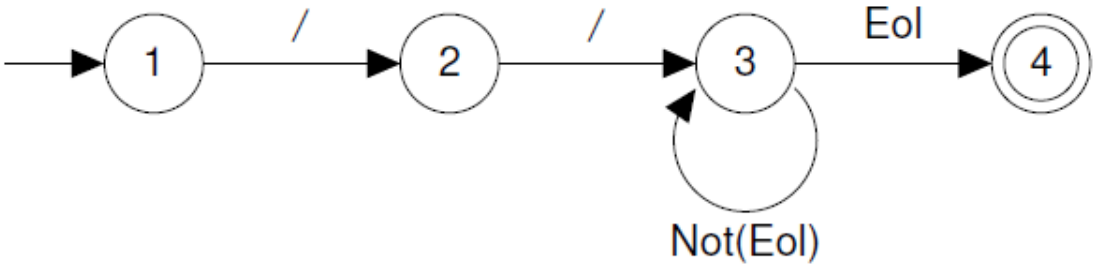
The technology behind scanning tools is well defined theoretically, and can (relatively) easily be implemented for the constructs in this class. But we have tools for efficiency!

The Go scanner is implemented by hand and shows a general strategy:

`https://github.com/golang/go/blob/master/src/go/scanner/scanner.go`

Scanner Tables – "Crafting a Compiler"

(a)



(b)

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

Scanner Algorithm – "Crafting a Compiler"

```
/* Assume CurrentChar contains the first character to be scanned */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then /* Return or process the valid token */
else /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.

Matching Rules

Assume the scanning tool has constructed a collection of DFAs, one for each lexical rule

```
reg_expr1     ->     DFA1
reg_expr2     ->     DFA2
...
reg_rexpn     ->     DFAn
```

How do we decide which regular expression should match the next characters to be scanned?

`flex` matches on *all* regular expressions, and follows a set of arbitrary rules to select which token is the successful match (“first longest match”).

Matching Rules – Algorithm

Given DFAs $D_1, \ldots, D_n$, ordered by the input rule order, a `flex`-generated scanner executes

```
while input is not empty do
    s_i := the longest prefix that D_i accepts
    l := max{|s_i|}
    if l > 0 then
        j := min{i : |s_i| = l}
        remove s_j from input
        perform the j^th action
    else (error case)
        move one character from input to output
    end
end
```

- The *longest* initial substring match forms the next token, and it is subject to some action;
- The *first* rule to match breaks any ties; and
- Non-matching characters are echoed back.

Why the “longest match” principle?

Example: keywords

```
...
import                  return tIMPORT;
[a-zA-Z_][a-zA-Z0-9_]*  return tIDENTIFIER;
...
```

Given a string “`importedFiles`”, we want the token output of the scanner to be

```
tIDENTIFIER(importedFiles)
```

and not

```
tIMPORT tIDENTIFIER(edFiles)
```

Since we prefer longer matches, we get the right result.

Why the “first match” principle?

Example: keywords

```
...
continue                 return tCONTINUE;
[a-zA-Z_][a-zA-Z0-9_]*   return tIDENTIFIER;
...
```

Given a string “`continue foo`”, we want the token output of the scanner to be

```
tCONTINUE tIDENTIFIER(foo)
```

and not

```
tIDENTIFIER(continue) tIDENTIFIER(foo)
```

Since both `tCONTINUE` and `tIDENTIFIER` match with the same length, there is a tie. Using the “first match” rule, we break the tie by looking at the rule order and get the correct result.

Problem Cases (of course)

In some languages, the “first longest match” (flm) rules are not enough.

FORTRAN equals

FORTRAN allows for the following tokens:

```
.EQ., 363, 363., .363
```

flm analysis of `363.EQ.363` gives us:

```
tFLOAT(363) E Q tFLOAT(0.363)
```

What we actually want is:

```
tINTEGER(363) tEQ tINTEGER(363)
```

Solution

To distinguish between a `tFLOAT` and a `tINTEGER` followed by a “.”, `flex` allows us to use look-ahead, using `'/'`:

```
363/.EQ. return tINTEGER;
```

A look-ahead matches on the full pattern, but only processes the characters before the `'/'`. All subsequent characters are returned to the input stream for further matches.

Problem Cases (of course)

FORTRAN ignores whitespace

1. `DO5I = 1.25` $\rightsquigarrow$ `DO5I=1.25`

   in C, these are equivalent to an assignment:

   ```
   do5i = 1.25;
   ```

2. `DO 5 I = 1,25` $\rightsquigarrow$ `DO5I=1,25`

   in C, these are equivalent to looping:

   `for (i=1; i<25; ++i) {...}` (`5` is interpreted as a line number)

Solution

1. First case, flm analysis is correct

   ```
   tIDENTIFIER(DO5I) tASSIGN tFLOAT(1.25)
   ```

2. Second case, flm analysis gives the incorrect result. What we want is:

   ```
   tDO tINTEGER(5) tIDENT(I) tASSIGN tINTEGER(1) tCOMMA tINTEGER(25)
   ```

   But we cannot make decision on `tDO` until we see the comma, look-ahead comes to the rescue:

   ```
   DO/(letter|digit)*=(letter|digit)*, return tDO;
   ```

Context-Sensitive Grammars

In some languages, the correct token type for the sequence of characters may depend on its context

C language

Given the following snippet of a C program, is this a either cast to type `a` or a multiplication expression? How do we return the correct token?

```
(a) * b
```

There are two main options used in practice to resolve this ambiguity

- Feed semantic information into the scanner (yikes!); or
- Scan a more general language and resolve the ambiguity in a later phase.

See `https://en.wikipedia.org/wiki/The_lexer_hack` for more details

Context-Sensitive Grammars

Golang

Go (in a looser way) also suffers from context sensitivity in its grammar. (For some reason) both function calls and casts share the same syntax.

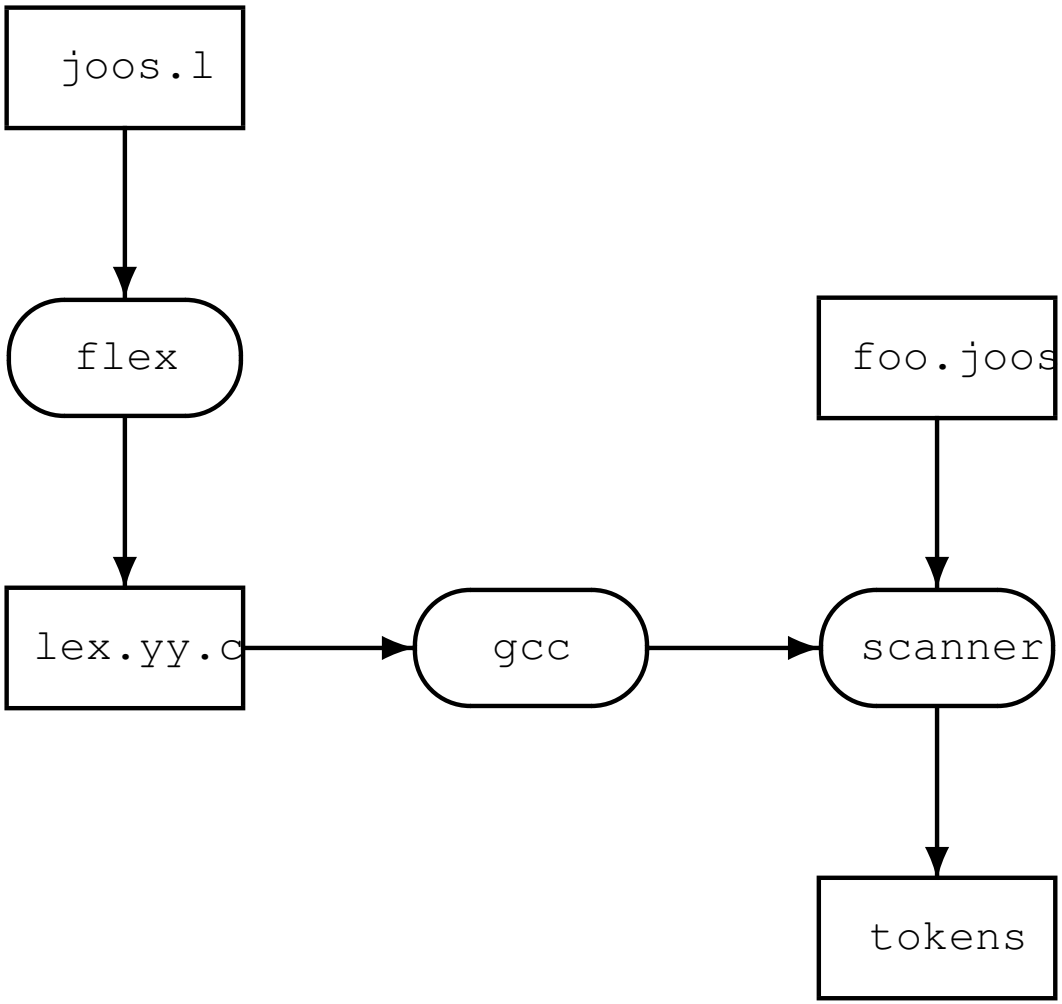
```
int(a)
```

Is this a call to a function `int`, or a cast to type `int`? It all depends if `int` is a type or an identifier. How do we return the correct token?

Russ Cox might disagree that this is an “ambiguity at the syntactic level” (`http://grokbase.com/t/gg/golang-nuts/142pkyzh7r/go-nuts-parsing-go-code-without-context`), but the issue still remains

Onto the Practice!

In practice, we use tools to generate scanners instead of writing them by hand (although some production compilers still use hand written scanners for `C`)



`flex`

`flex` uses a single `.l` file to define the scanner. The `.l` file

- Has 3 main sections divided by %%
  1. Declarations, helper code;
  2. Regular expression rules and associated actions;
  3. User code; and
- Saves much effort in compiler design.

`flex` supports (amongst other things)

- Line numbers; and
- Interoperability with the `bison` parser tool.

Example scanner: `https://github.com/comp520/Examples/tree/master/flex%2Bbison/scanner`

Skeleton `flex` File

```
/* The first section of a flex file contains:
 *   1. A code section for includes and other arbitrary C code
 *   2. Helper definitions for regexes
 *   3. Scanner options
 */
%{ /* Code section */ %}

/* Helper definitions */
DIGIT [0-9]

/* Scanner options, line number generation */
%option yylineno

/* The second section contains regular expressions, one per line, followed by
 * the scanner action. Actions are executed when a token is matched. An empty
 * action is treated as a NOP.
 */
%%
RULE ACTION

%%
/* User code comes in the last section */
```

Example `flex` File - TinyLang

```
%{
#include <stdio.h>
%}
DIGIT [0-9]
%option yylineno
%%
[\r\n]+
[ \t]+  printf("Whitespace, length %lu\n", yyleng);

"+"     printf("Plus\n");
"-"     printf("Minus\n");
"*"     printf("Times\n");
"/"     printf("Divide\n");
"("     printf("Left parenthesis\n");
")"     printf("Right parenthesis\n");

0|([1-9]{DIGIT}*) { printf("Integer constant: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* { printf("Identifier: %s\n", yytext); }

.       { fprintf(stderr, "Error: (line %d) unexpected character `%s'\n", yylineno, yytext);
          exit(1); }
%%

int main() { yylex(); return 0; }
```

Running a `flex` Scanner

After the scanner file is complete, using `flex` to create a scanner is really simple

```
$ vim tiny.l

$ flex tiny.l

## flex has generated a file ‘lex.yy.c’

$ gcc -o tiny lex.yy.c -lfl
```

Running a `flex` Scanner

```
$ echo "a*(b-17) + 5/c" | ./tiny
```

Output

```
Identifier: a
Times
Left parenthesis
Identifier: b
Minus
Integer constant: 17
Right parenthesis
Whitespace, length 1
Plus
Whitespace, length 1
Integer constant: 5
Div
Identifier: c
```

Line Numbers

Having line information handy is essential for producing detailed error messages. There are two different implementations: manual, and automatic

Examples on GitHub:

`https://github.com/comp520/Examples/tree/master/flex%2Bbison/linenumbers`

Manual line and character counting

```
%{
    int lines = 0, chars = 0;
%}

%%
\n  lines++; chars++;
.   chars++;

%%
int main() {
    yylex();
    printf("#lines = %d, #chars = %d\n", lines, chars);
    return 0;
}
```

Line Numbers

Getting automated position information in `flex`

- Is easy for line numbers: option and variable `yylineno`; but
- Is more involved for character positions.

If position information is useful for further compilation phases

- It can be stored in a structure `yylloc` provided by the parser (`bison`); but
- **Must** be updated by a user action.

```
typedef struct yyltype {
    int first_line, first_column, last_line, last_column;
} yyltype;

%{
    #define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;
%}
%option yylineno

%%
. { fprintf(stderr, "Error: (line %d) unexpected char `%s'\n", yylineno, yytext);
    exit(1); }
```

Scanner Actions

Actions in a `flex` file can either

- Do nothing – ignore the characters;
- Perform some arbitrary computation, call a function, etc.; and/or
- Return a token (token definitions provided by the parser).

```
%{
#include <stdlib.h>    /* atoi   */
#include <stdio.h>     /* printf */
#include "y.tab.h"     /* Token types */
%}

%%
[aeiouy]
[0-9]+         printf("%d", atoi(yytext) + 1);
func           return tFUNCTION;
%%

int main() { yylex(); return 0; }
```

Extended Scanner Actions

The basic functionality of `bison` expects a token type to be returned. In some cases though, a token is not enough

- Need to capture the value of an identifier; or
- Need the value of a string, integer, or float literal.

In these cases, `flex` **provides**

- `yytext`: the scanned sequence of characters;
- `yyleng`: the length of the scanned sequence;
- `yylval`: a user-defined variable from the parser (`bison`) to be returned with the token; and
- `yylloc`: a `bison` defined variable for storing token location.

```
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.string_const = strdup(yytext);
    return tIDENTIFIER;
}
```

Scanner Efficiency

Compiler efficiency is extremely important, but scanners operate on a character by character basis. In reality, scanning is one of the more time consuming elements of a (simple) compiler.

Recall: to produce a string of tokens, we match on *every* regular expression in the scanner.

Something quite simple we can do is

- Reduce the number of regular expressions;
- By observing that keywords are valid identifiers; and
- Use a (fast) lookup mechanism to determine if it is a reserved word.

Scanner Error Handling

Say in our language, integers do not have a leading zero. The following assignment is thus invalid

```
var a : int
a = 011
```

Using the standard `0|([1-9][0-9]*)` regular expression for integers and the flm rules, the scanner produces the following *valid* token stream

```
tVAR
tIDENTIFIER(a)
tCOLON
tINT
tIDENTIFIER(a)
tASSIGN
tINTVAL(0)
tINTVAL(11)
```

The scanner does not identify an error(?!), so the detection is left to a later phase of the compiler. In our case, this will be identified by the parser.

However, we should ask: Is this a syntactic error, or should the scanner throw a lexical error?

Scanner Error Handling - Syntactic Error

It's tempting to assume this is a lexical error since integers cannot be defined with leading zeros. However, what if the user intended to write the following addition

```
var a : int
a = 0 + 11
```

It may not be a useful computation, but it is *valid*. The corrected token stream is therefore

```
tVAR
tIDENTIFIER(a)
tCOLON
tINT
tIDENTIFIER(a)
tASSIGN
tINTVAL(0)
tPLUS             // this is new
tINTVAL(11)
```

If we assume this is a syntactic error, the original program was simply missing the addition operator and an informative error message can be displayed to the user

Scanner Error Handling - Lexical Error

On the other hand, we may decide a lexical error is more appropriate.

Solution: Define 2 regular expressions

1. Valid integers: `0|([1-9][0-9]*)`
2. All integers: `([0-9]*)`

For an invalid integer

1. Valid integers regular expression matches on the leading zero only - this is of length 1
2. All integers regular expression matches on the entire input number (length > 1)

Using the longest match principle we choose the all integers regular expression and throw an error.

For a valid integer

1. Valid integers regular expression matches on the entire input $\boldsymbol{n}$
2. All integers regular expression matches on the entire input $\boldsymbol{n}$

Using the first match principle we choose the valid regex and produce a `tINTVAL(n)` token.

Summary

- A scanner transforms a string of characters into a string of tokens;
- Scanner generating tools like `flex` allow you to define a regular expression for each type of token;
- Internally, the regular expressions are transformed to a deterministic finite automata (DFAs) for matching;
- The head of the input string is matched on all machines to determine the next token;
- To break ties (more than one matching token type), scanners use 2 principles: “longest match” and “first match”.