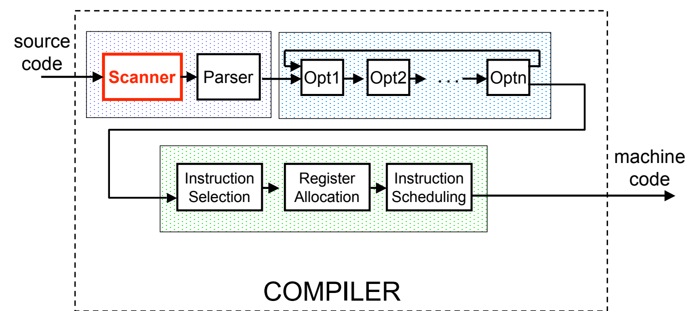
Lexical Analysis

ICS312 - Spring 2009
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

The Big Picture Again



Lexical Analysis

- Lexical Analysis, also called “scanning” or “lexing”
- It does two things:
  - Transforms the input source string into a sequence of substrings
  - Classifies them according to their “role”
- The input is the source code
- The output is a list of tokens
- Example input:

```
if (x == y)
        z = 12;
else
        z = 7;
```

- This is really a single string:

i	f		(	x	=	=	y	)	\n	\t	z		=		1	2	;	\n	e	l	s	e	\n	\t	z		=		7	;	\n

Tokens

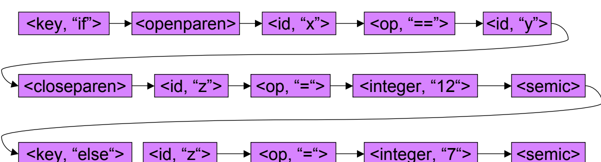
- A token is a syntactic category
- Example tokens:
  - Identifier
  - Integer
  - Floating-point number
  - Keyword
  - etc.
- In English we’d talk about
  - Noun
  - Verb
  - Adjective
  - etc.

Lexeme

- A lexeme is the string that represents an instance of a token
- The set of all possible lexemes that can represent a token instance is described by a pattern
- For instance, we can decide that the pattern for an identifier is
  - A string of letters, numbers, or underscores, that starts with a capital letter

Lexing output

i	f		(	x	=	=	y	)	\n	\t	z		=		1	2	;	\n	e	l	s	e	\n	\t	z		=		7	;	\n



- Note that the lexer removes non-essential characters
  - Spaces, tabs, linefeeds
  - And comments!
  - Typically a good idea for the lexer to allow arbitrary numbers of white spaces, tabs, and linefeeds

The Lookahead Problem

- Characters are read in from left to right, one at a time from the input string
- The problem is that it is not always possible to determine whether a token is finished or not without looking at the next character
- Example:
  - Is character 'f' the full name of a variable, or the first letter of keyword "for"?
  - Is character '=' an assignment operator or the first character of the "==" operator?
- In some languages, a lot of lookahead is needed
- Example: FORTRAN
  - Fortran removes ALL white spaces before processing the input string
  - DO 5 I = 1.25 is valid code that sets variable DO5I to 1.25
  - But "DO 5 I = 1.25" could also be the beginning of a for loop!

The Lookahead Problem

- It is typically a good idea to design languages that require "little" lookahead
  - For each language, it should be possible to determine how many lookahead characters are needed
- Example with 1-character lookahead:
  - Say that I get an"if" so far
  - I can *look* at the next character
  - If it's a ' ', '(','\t', then I don't read it; I stop here and emit a TOKEN_IF
  - Otherwise I read the next character and will most likely emit a TOKEN_ID
- In practice one implements lookhead/pushback
  - When in need to look at next characters, read them in and push them onto a data structure (stack/fifo)
  - When in need of a character get it from the data structure, and if empty from the file

A Lexer by Hand?

- Example: Say we want to write the code to recognizes the keyword "if"

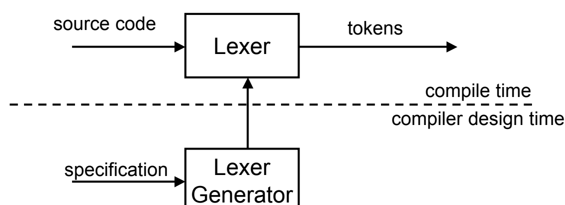
```
c = readchar();
if (c == 'i') {
        c = readchar();
        if (c == 'f') {
                c = readchar();
                if (c not alphanumeric) {
                        pushback(c);
                        emit(TOKEN_IF)
                } else {
                        // build a TOKEN_ID
                }
        } else {
                // something else
        }
} else {
        // something else
}
```

A Lexer by Hand?

- There are many difficulties for writing a lexer by hand as in the previous slide
  - Many types of tokens
    - fixed string
    - special character sequences (operators)
    - numbers defined by specific/complex rules
  - Many possibilities of token overlaps
    - Hences many nested if-then-else in the code of the lexer
- Coding all this by hand is very painful
- And it's difficult to get it right
- But note that some compilers have a hand-implemented-lexer to achieve higher speed

Regular Expressions

- To avoid the endless nesting of if-then-else to capture all types of possible tokens one needs a formalization of the lexing process
- If we have a good formalization, we could even generate the lexing code automatically!



Lexer Specification

- Question: How do we formalize the job a lexer has to do to recognize the tokens of a pecific language?
- Answer: We need a language!
  - More specifically, we're going to talk about the language of tokens!
- What's a language?
  - An alphabet (typically called ∑)
    - e.g., the ASCII characters
  - A subset of all the possible strings over ∑
- We just need to provide a formal definition of a the language of the tokens over ∑
  - Which strings are tokens
  - Which strings are not tokens
- It turns out that for all (reasonable) programming languages, the tokens can be described by a regular language
  - I.e., a language that can be recognized by a finite automaton
  - See ICS 241 and later slides
  - A lot of theory here that I'm not going to get into

Describing Tokens

- The most popular way to describe tokens is to use regular expressions
- Regular expressions are just notations, which happen to be able to represent regular languages
  - A regular expression is a string (in a meta-language) that describes a pattern (in the token language)
- If A is a regular expression, then L(A) is the language represented by A
  - Remember that a language is just a set of valid strings
- Basic: L("c") = {"c"}
- Concatenation: L(AB) = {ab | a in L(A) and b in L(B)}
  - L("i" "f") = {"if"}
- Union: L(A|B) = {x | x in L(A) or x in L(B)}
  - L("if"|"then"|"else") = {"if", "then", "else"}
  - L(("0"|"1") ("1"|"0")) = {"00", "01", "10", "11"}

Regular Expression Overview

Expression	Meaning
ε	empty pattern
a	Any pattern represented by 'a'
ab	Strings with pattern 'a' followed by pattern 'b'
a|b	Strings with pattern 'a' or pattern 'b'
a^*	Zero or more occurrences of pattern 'a'
a^+	One or more occurrences of pattern 'a'
a^3	Exactly 3 occurrences of pattern 'a'
a?	(a | ε)
.	Any single character (not very standard)

- Let's look at how REs are used to describe tokens

REs for Keywords

- It is easy to define a RE that describes all keywords

  Key = "if" | "else" | "for" | "while" | "int" | ..

- These can be split in groups if needed

  Keyword = "if" | "else" | "for" | …
  Type = "int" | "double" | "long" | …

- The choice depends on what the next component (i.e., the parser) would like to see

RE for Numbers

- Straightforward representation for integers
  - digits = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  - integer = digits$^+$
- Typically, regular expression systems allow the use of '-' for ranges, sometimes with '[' and ']'
  - digits = 0-9
- Floating point numbers are much more complicated
  - 2.00
  - .12e-12
  - 312.00001E+12
  - 4
- Here is one attempt
  - (digit$^+$ "."? | digits* ("." digit$^+$)) (("E"|"e")("+"|"-"|ε) digit+)))?
- Note the difference between meta-character and language-characters
  - "+" versus +, "-" versus -, "(" versus (, etc.
- Often books/documentations use different fonts for each level of language

RE for Identifiers

- Here is a typical description
  - letter = a-z | A-Z
  - ident = letter (letter | digit | "_")*
    - Starts with a letter
    - Has any number of letter or digit or "_" afterwards
- In C: ident = (letter | "_") (letter | digit | "_")*

RE for Phone Numbers

- Simple RE
  - digit = 0-9
  - area = digit^3
  - exchange = digit^3
  - local = digit^4
  - phonenumber = "(" area ")" " "? exchange ("-"|" ") local
- The above describes 10^{3+3+4} strings in the L(phonenumber) language

REs in Practice

- The Linux grep utility allows the use of regular expressions
  - Example with phone numbers
    - grep “([0-9]\{3\}) \{0,1\}[0-9]\{3\}[-|][0-9]\{4\}” file
  - The syntax is different from that we’ve seen, but it’s equivalent
- Perl implements regular expressions
- Text editors implement regular expressions
  - .e.g., vi for string replacements
- At the end of the day, we often have built for ourselves tons of regular expressions

In-class Exercise

- Write regular expressions for
  - All strings over alphabet {a,b,c}
  - All strings over alphabet {a,b,c} that contain substring “abc”
  - All strings over alphabet {a,b,c} that consists of one of more a’s, followed by two b’s, followed by whatever sequence of a’s and c’s
  - All strings over alphabet {a,b,c} such that they contain at least one of substrings “abc” or “cba”

In-class Exercise

- Write regular expressions for
  - All strings over alphabet {a,b,c}
    - (a|b|c)*
  - All strings over alphabet {a,b,c} that contain substring “abc”
    - (a|b|c)*abc(a|b|c)*
  - All strings over alphabet {a,b,c} that consists of one of more a’s, followed by two b’s, followed by whatever sequence of a’s and c’s
    - a^{+}bb(a|c)*
  - All strings over alphabet {a,b,c} such that they contain at least one of substrings “abc” or “cba”
    - ((a|b|c)*abc(a|b|c)* | (a|b|c)*cba(a|b|c)*)

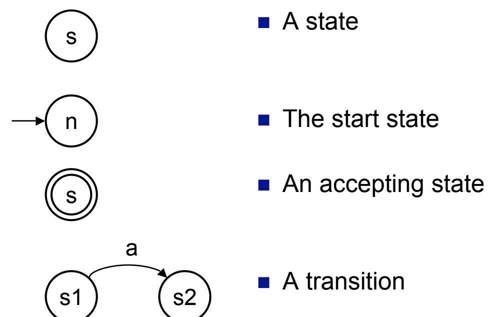
Now What?

- Now we have a nice way to formalize each token (which is a set of possible strings)
- Each token is described by a RE
  - And hopefully we have made sure that our REs are correct
  - Easier than writing the lexer from scratch
  - But still requires that one be careful
- Question: How do we use these REs to parse the input source code and generate the token stream?
- A little bit of “theory”
  - REs characterize Regular Languages
  - Regular Languages are recognized by Finita Automata
  - Therefore we can implement REs as automata

Finite Automata

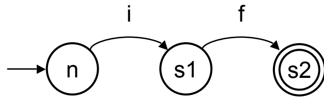
- A finite automaton is defined by
  - An input alphabet: ∑
  - A set of states: S
  - A start state: n
  - A set of accepting states: F (a subset of S)
  - A set of transitions between states: subset of SxS
- Transition Example
  - s1 -- a --> s2
  - If the automaton is in state s1, reading a character “a” in the input takes the automaton in state s2
  - Whenever reaching the “end of the input”, if the state the automaton is in in a accept state, then we accept the input
  - Otherwise we reject the input

Finite Automata as Graphs



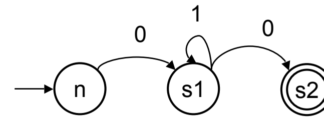
- A state
- The start state
- An accepting state
- A transition

Automaton Examples



- This automaton accepts input "if"

Automaton Examples



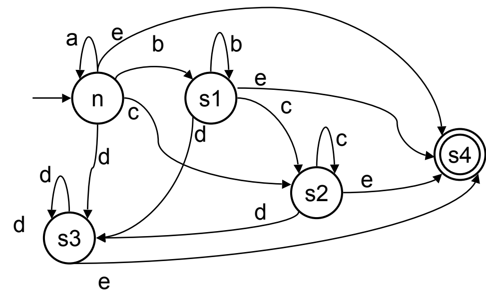
- This automaton accepts input that start with a 0, then have any number of 1's, and end with a 0
- Note the natural correspondence between automata and REs: 01^*0
- Question: can we represent all REs with simple automata?
- Answer: yes
- Therefore, if we write a piece of code that implements arbitrary automata, we have a piece of code that implements arbitrary REs, and we have a lexer!
  - Not _this_ simple, but close

Non-deterministic Automata

- The automata we have seen so far are called Deterministic Finite Automata (DFA)
  - At each state, there is at most one edge for a given symbol
  - At each state, transition can happen only if in input symbol is read
    - Or the string is rejected
- It turns out that it's easier to translate REs to Non-deterministic Finite Automata (NFA)
  - There can be "ε-transitions"!
  - There can be multiple possible transitions for a given input symbol at a state!

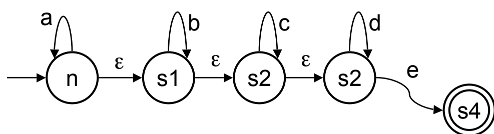
Example REs and DFA

- Say we want to represent RE "a*b*c*d*e" with aDFA



Example REs and NFA

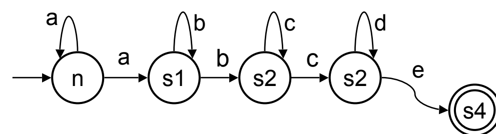
- "a*b*c*d*e": much simpler with a NFA



- With ε-transitions, the automaton can "choose" to skip ahead, non-deterministically

Example REs and NFA

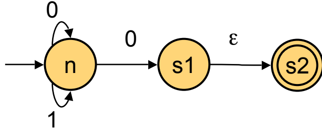
- "a$^+$b$^+$c$^+$d$^+$e": easy modification



- But now we have multiple choices for a given character at each state!
  - e.g., two "a" arrows leaving n

NFA Acceptance

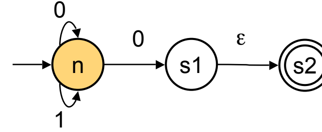
- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject



input string: 010

NFA Acceptance

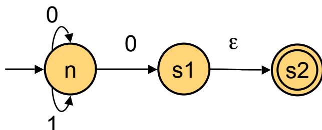
- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject



input string: 010

NFA Acceptance

- When using an NFA, one must constantly keep track of all possible states
- If at the end of the input (at least) one of these states is an accepting state, then accept, otherwise reject



input string: 010

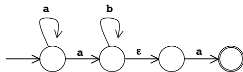
ACCEPT because of s2

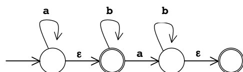
REs and NFA

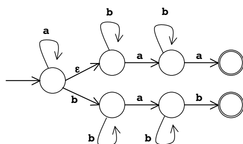
- So now we're left with two possibilities
- Possibility #1: design DFAs
  - Easy to follow transitions once implemented
  - But really cumbersome
- Possibility #2: design NFAs
  - Really trivial to implement REs as NFAs
  - But what happens on input characters?
    - Non-deterministic transitions
    - Should keep track of all possible states at a given point in the input!
- It turns out that:
  - NFAs are not more powerful than DFAs
  - There are systematic algorithms to convert NFAs into DFAs and to limit their sizes
    - See a theory course

In-class exercise

- Write REs for the following NFAs

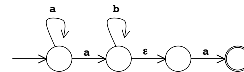




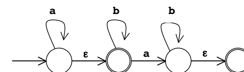


In-class exercise

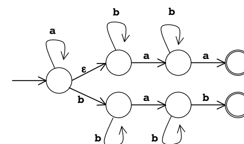
- Write REs for the following NFAs



a^+b^*a



$a^*b^*(\varepsilon|ab^*)$

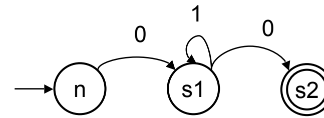


$a^*b^*(ab^*a \mid bab^+)$

Putting it All Together

- These are the steps to designing/building a lexer
  - Come up with a RE for each token category
  - Come up with an NFA for each RE
  - Convert the NFA (automatically) to a DFA
  - Write a piece of code that implements a DFA
    - Pretty easy with a decent data-structure, which is a basically a transition table
  - Implement your lexer as a "bunch of DFAs"
- Let's see an example of DFA implementation

Example DFA Implementation



state	char	next state	decision / continue
n	0	s1	REJECT / YES
n	1	-	REJECT / NO
s1	0	s2	ACCEPT / YES
s1	1	s1	REJECT / YES
s2	0	-	REJECT / NO
s2	1	-	REJECT / NO

```
state = STATE_N
while (c == getchar()) {
    transition(state,c,&next_state,
    &decision, &continue);
    if (!continue)
        return REJECT;
    state = next_state
}
return decision;
```

The "bunch of DFAs"

- How the lexer works
  - The lexer has his "bunch of NFAs/DFAs"
  - It runs them all at the same time until they have all rejected the input
  - It then rewinds to the one that accepted last
    - that is the one that accepted the longest string
    - "rewinding" uses lookahead/pushback
  - This one corresponds to the right token
- Let's look at this on an example

Example

- Say we have the following tokens (described by a RE, and thus a natural NFA, and thus a DFA):
  - TOKEN_IF: "if"
  - TOKEN_IDENT: letter (letter | "_")+
  - TOKEN_NUMBER: (digit)+
  - TOKEN_COMPARE: "=="
  - TOKEN_ASSIGN: "="
- This is a very small set of tokens for a tiny language
- The language assumes that tokens are all separated by spaces
- Let's see what happens on the following input:

i	f		i	f	0		=	=		c		x		=		2	x	3

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	accept
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Both TOKEN_IF and TOKEN_IDENT were the last ones to accept

Emit TOKEN_IF because we build our lexer with the notion of **reserved** keywords

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	ok so far
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Emit TOKEN_IDENT (with string "if0") because it accepted the latest

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	ok so far
TOKEN_ASSIGN	ok so far

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	ok so far
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	accept
TOKEN_ASSIGN	reject

Emit TOKEN_COMPARE because it accepted the latest

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Emit TOKEN_IDENT (with string “c”) because it accepted the latest

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	ok so far
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	accept
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	reject

Emit TOKEN_IDENT (with string “x”) because it accepted the latest

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

↑ (under "=")

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	ok so far
TOKEN_ASSIGN	ok so far

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

↑ (after "=")

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	accept

Emit TOKEN_ASSIGN because it was the only one accepted

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

↑ (under "2")

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	ok so far
TOKEN_COMPARE	reject
TOKEN_ASSIGN	accept

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

↑ (under "x")

DFA	DECISION
TOKEN_IF	reject
TOKEN_IDENT	reject
TOKEN_NUMBER	reject
TOKEN_COMPARE	reject
TOKEN_ASSIGN	accept

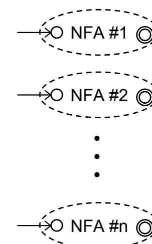
Abort and print a Syntax Error Message!!

Example

- If there had be no syntax error, the lexer would have emitted:
  - <TOKEN_IF>
  - <TOKEN_ID, “if0”>
  - <TOKEN_COMPARE>
  - <TOKEN_ID, “c”>
  - <TOKEN_ID, “x”>
  - <TOKEN_ASSIGN>
  - <TOKEN_NUMBER,”23”>

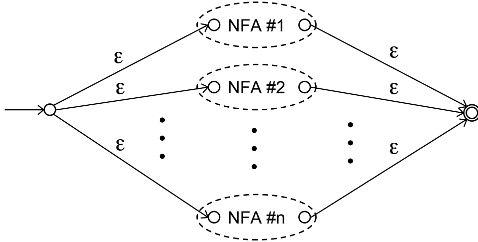
Implementing “bunch of DFAs”

- We have one NFA per token
- We can easily combine them in one single NFA



Implementing “bunch of DFAs”

- We have one NFA per token
- We can easily combine them in one single NFA



- We can then convert it to a DFA

Lexer Generation

- A lot of of the lexing process is really mechanical once on has defined the REs
  - Contrast with the horrible if-then-else nesting of the “by hand” lexer!
- And it has been understood for decades
- So there are “lexer generators” available
  - They take as input a list of token specifications
    - token name
    - regular expression
  - They produce a piece of code that is the lexer for these tokens
- Well-known examples of such generators are lex and flex
- With these tools, a complicated lexer for a full language can be developed in a few hours

Tiny flex input file

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+
   { printf( "An integer: %s (%d)\n", yytext, atoi( yytext ) ); }
{DIGIT}+"."{DIGIT}*
   { printf( "A float: %s (%g)\n", yytext, atof( yytext ) ); }
if|then|begin|end|procedure|function
   { printf( "A keyword: %s\n", yytext ); }
{ID}
   { printf( "An identifier: %s\n", yytext ); }
"+"|"-"|"*"|"/"
   { printf( "An operator: %s\n", yytext ); }
[ \t\n]+
   { /* nothing (eat up whitespace) */ }
.
   { printf( "Unrecognized character: %s\n", yytext ); }
%%
main() {
   yyin = stdin;
   yylex();
}
```

Conclusion

- 20,000 ft view
  - Lexing relies on Regular Expressions, which rely on NFAs, which rely on DFAs, which are easy to implement
  - Therefore lexing is “easy”
- Lexing has been well-understood for decades and many tools are available
- Only motivation to write a lexer by hand: speed
- In a compiler course the typical first project is to have student write a lexer using lex/flex