Instructor: **Stephen Fenner (Guest Lecturer)**

Course: CSCE 531, Compiler Construction

Date: February 5, 2013

**************Raw notes follow**

Apogee video lecture. I give a brief introduction.

Lexical analysis. This phase comes right at the beginning. He uses a tablet-type screen. Draws on it.

Defines input (source text---streams of ASCII characters) and output (stream of tokens, where a token is the smallest meaningful unit). The output of the lexical analyzer is the input to the parser. A lexical analyzer does not know about, e.g., nesting, but it can be implemented much more efficiently than a parser.

Examples of tokens: identifier. Each token has a type and an attribute. For identifiers, type is ID the attribute is a literal string. Another example: integer constant: type INTCONST, attribute is

[ASK QUESTION---what do you think it is? Correct answer is given by a student] value (of type int). Another; real constant: REALCONST, value (double). In C, every other token is uniquely identified by its type; it does not need an attribute. Examples; + PLUS, ; SEMI; = ASSIGN; “main” MAIN.

In many languages, each keyword is its own token type.

Example. int count = 0;

INTTYPE <ID “count”> ASSIGN <INTCONST 0> SEMI

Regular expression pattern matching recognizes token types.

A regular expression (regexp) is a pattern that matches certain strings (and not others)

Each token type has a corresponding regular expression that corresponds to a token type

The following are regexp’s:

\epsilon (or “”)---matches the empty string [and nothing else---this will not be repeated each time]

a matches the string “a” (and same for all the other chars

Suppose r and s are regular expressions.

1. r|s is a regexp that matches anything matched by r or s (or both): union or disjunction (OR)
2. rs (“concatenation”) May be iterated, e.g.: abc(a|b), where the parentheses are used for grouping matches “abca” and “abcb” and thing else

3. r* (“Kleene closure”) matches the concatenation of zero or more strings, each matching r.

The * has the highest precedence; | has the lowest.

Example: a* matches zero or more as, where “zero as” is the empty string

(a|b)* matches any string of as and bs (and the empty string)

(a|bc|c)* matches any string of a, b, and c, in which any b is immediately followed by a c

(a|bc|c)*a is like just above, but the string must end in a

(a|bc|c)*a* matches the same strings as (a|bc|c)* (regexps are not unique!)

Shorthands (not necessary, but convenient):

Character class, e.g., [abc] matches any single character in the square brackets, same as a|b|c same as [cab] same as [cba]

Subrange, e.g., [0-9] is any single character between 0 and 9 in the ASCII sequence; so, it matches any single decimal digit, same as [0123456789]

[^a] Complemented: matches any char except what is in the list

“)[+” matches itself. Good for matching parentheses. For “ and \, escape with backslash: \” matches “ and \\ matches \

. matches any single character except newline \n

r? ---“optional r” matches r or the empty string or both: same as r | “”

r+ ---one or more r’s, same as rr*

Recognizing some token types:

(unsigned) int constant: [0-9]+ (A sequence of one or more character digits)

identifier (Java, C, C++): [A-Za-z_][_A-Za-z0-9]* (alpha followed by alphanumeric

real constant (Pascal); int-part.int-part followed by an optional exponent. In Pascal, leading zeros are allowed: [0-9]+”.”[0-9]+([Ee][+-]?[0-9]+)? Note that [+-] is a character class, because – comes on the right, not in between

ASSIGN: “=” Note: the quotes are not necessary, but they do not hurt

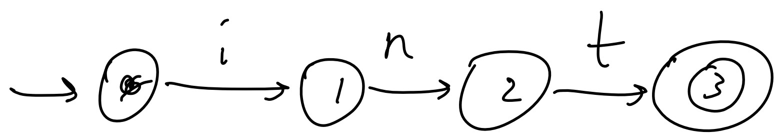
SEMI: “;”

INTTYPE: “int”

Automata for string matching (equivalent to regular expressions)

Describes them in English.

Automaton for "int"

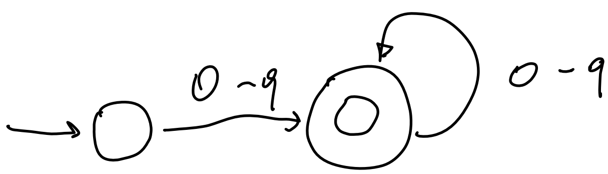


Start state has an arrow from nowhere; accepting state has a double circle. If, starting from the start state, you can read the entire input and wind up in an accepting state, then that is a match. (Otherwise not.)

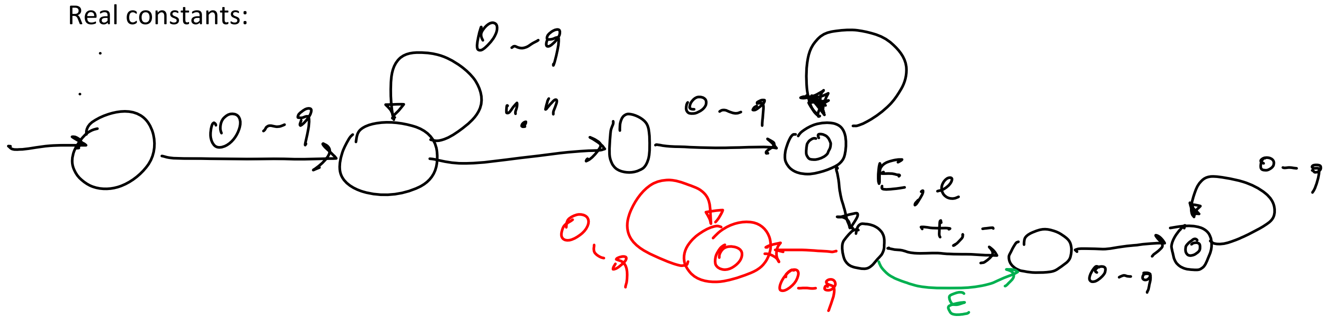
This automaton does not match "into".

Automaton for integer constants:

Note: 0-9 label on edge stands for 9 edges.

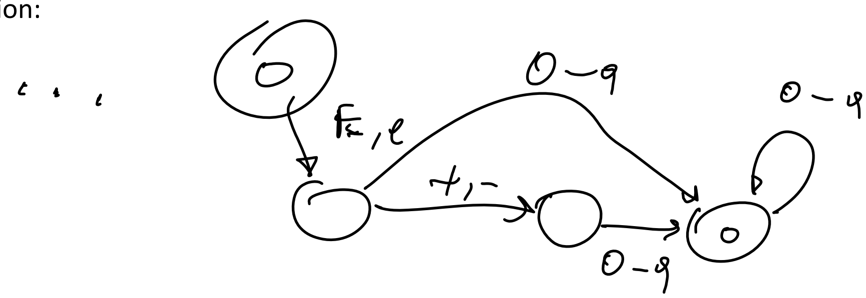


Real constants:



The epsilon-transition (green edge option) makes the automaton non-deterministic. (The automaton with the red part is deterministic.)

Another deterministic option:

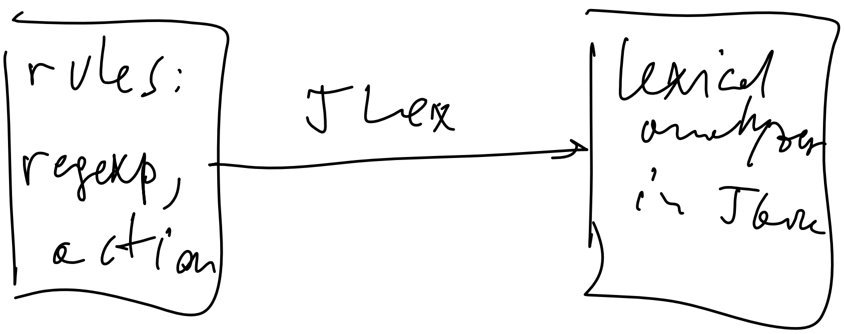


Equivalence: Every regexp is equivalent to a finite automaton.

Lexical scanner making tools:

lex, flex (fast lex), JLex (Java output)

These produce automata for each regular expression.



Steve is not familiar with JLex, but he knows lex and flex very well---he will discuss them.

Here is a typical set of rules

Before any rules and actions, declare component regular expressions. For example;

alpha [_A-Za-z]

alphanum [_A-Za-z0-9]

digit [0-9]

int_const {digit}+

%% {This separates declarations from rules; declarations can be used by placing them in braces}

{{int_const}   printf("%d", value);

{real_constant}   …..

Note: the longest match found is the official match: the lexical analyzer is greedy. Imagine that "identifier" has been declared

"main"

{identifier}

Note: by placing "main" before identifier, we accept the keyword main if the string main is encountered, even though "main" is also an identifier.

“+”

“++”

Is “a+++++b” OK?

(Answer: OK lexically, but will not produce legal code in C, because of the rules governing post-increment.)