CA4003 - Compiler Construction

Lexical Analysis

David Sinclair

Lexical Analysis

Lexical Analysis takes a stream of characters and generates a stream of tokens (names, keywords, punctuation, etc.).

A key task is to remove all the white spaces and comments. This will make parsing much easier.

A *lexical token* is a sequence of characters that can be treated as a single unit.

Some *tokens* have a value. This is called a *lexeme*.

A *scanner* is the piece of code that performs the *lexical analysis*. A key issue for scanners is speed. Scanners can be automatically generated using a combination of *regular expression* and *finite automata*.

Lexical Analysis [2]

```
bool compare(char *s, char *t)   /* compare 2 strings */
{
    if (strcmp(s, t)
        return (true);
    else
        return (false);
}
```

would yield:

```
BOOL ID(compare) LBRACK CHAR STAR ID(s) COMMA CHAR
STAR ID(t) RBRACK LBRACE IF LBRACK ID(strcmp) LBRACK
ID(s) COMMA ID(t) RBRACK RETURN LBRACK TRUE RBRACK
SEMI ELSE RETURN LBRACK FALSE RBRACK SEMI RBRACE EOF
```

Specifying Patterns

The scanner has to recognise the various parts of a language.

- *white space*

```
<ws>  ::=  <ws> ' '
       |   <ws> '\t'
       |   ' '
       |   '\t'
```

- *keywords and operators*, specified as literal patterns, `if, while, do`
- *comments*, opening and closing delimiters, `/* ... */`

Specifying Patterns [2]

- *identifiers*, rules vary from language to language but typically an *identifier* starts with a character and contains characters, numerics and limited symbols.
- *numbers*
  - integers: 0 or digit from 1-9 followed by digits from 0-9
  - decimals: integer . digits from 0-9
  - reals: (integer or decimal) E (+ or -) digits from 0-9

Regular Expressions provide a powerful means to specify these patterns.

Regular Expressions

Each *regular expression* represents a set of strings.

- **Symbol:** For each symbol a in the language, the regular expression a denotes the string a.
- **Alternation:** If M and N are 2 regular expressions, then $M|N$ denotes a string in M or a string in N.
- **Concatenation:** If M and N are 2 regular expressions, then MN denotes a string $\alpha\beta$ where α is in M and β is in N.
- **Epsilon:** The regular expression ϵ denotes the empty string.
- **Repetition:** The *Kleene closure* of M, denoted M^* is the set of zero or more concatenations of M.

Kleene closure binds tighter than concatenation, and concatenation binds tighter than alternation.

Regular Expression Examples

$(0\|1)^*$	0, 1, 10, 11, 100, 101, 110, ...
$(0\|1)^*0$	0, 10, 100, 110, 1000, 1010, ...
$(a\|b)^*$	ϵ, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, ...
$(a\|b)^*aa(a\|b)^*$	a string of a's and b's that contains at least one pair of consecutive a's.
$b^*(abb^*)^*(a\|\epsilon)$	a string of a's and b's with no consecutive a's.
$ab^*(c\|\epsilon)$	a string starting with an a, followed by zero or more b's and ending in an optional c. a, ac, ab, abc, abb, abbc, ...

Regular Expression Shorthands

The following abbreviations are generally used:

- $[axby]$ means $(a|x|b|y)$
- $[a-e]$ means $[abcde]$
- $M?$ means $(M|\epsilon)$
- M^+ means (MM^*)
- . means any single character except a newline character
- "a^{+*}" is a quotation and the string in quotes literally stands for itself.

Regular Expressions for Tokens

`do`	DO
$[a-zA-Z][a-zA-Z0-9]^*$	ID
$[0-9]^+$	NUM
$([0-9]^+"."[0-9]^*)\|([0-9]^*"."[0-9]^+)$	REAL
$"//"[a-zA-Z0-9]^*"\backslash n"\|(""\|"\backslash n"\|"\backslash t")^+$	comment or white space

Disambiguation Rules for Scanners

Is `do99` and identifier or a keyword (`do`) followed by a number (`99`)?

Most modern lexical-analyser generators follow 2 rules to disambiguate situations like above.

- **Longest match:** The longest initial substring that can match any regular expression is taken as the next token.
- **Rule priority:** In the case where the longest initial substring is matched by multiple regular expressions, the first regular expression that matches determines the token type.

So `do99` is an identifier.

Finite Automata

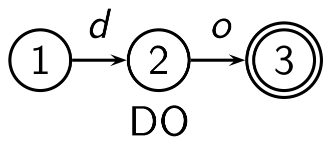
Regular expressions are good for specifying lexical tokens. *Finite automata* are good for recognising regular expressions.
A *finite automata* consists of a set of *nodes* and *edges*. *Edges* go from one node to another node and are labelled with a *symbol*. Nodes represent states. One of the nodes represents the *start node* and some of the node are *final states*.

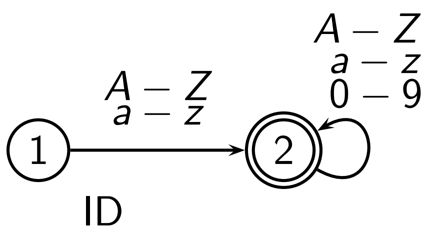
A *deterministic finite automaton* (DFA) is a finite automaton in which no pairs of edges leading away from a node are labelled with the same symbol.

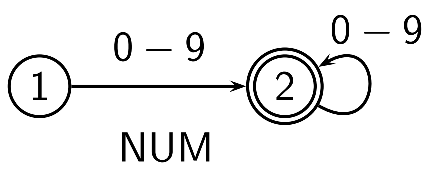
A *nondeterministic finite automaton* (NFA) is a finite automaton in which two or more edges leading away from a node are labelled with the same symbol.

Finite Automata [2]

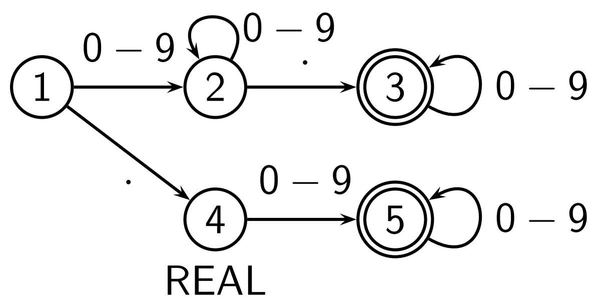
Using our regular expressions from before we would have:

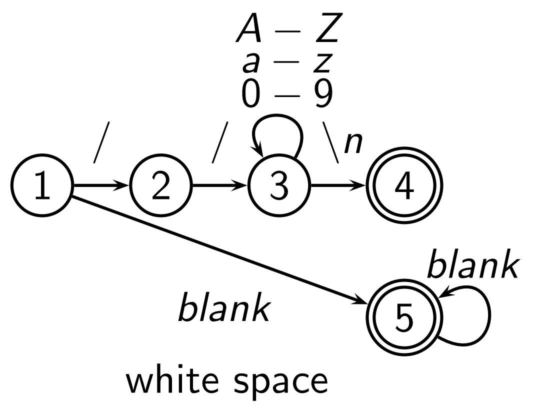


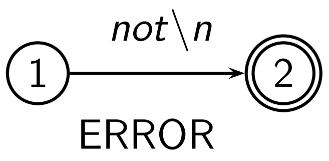




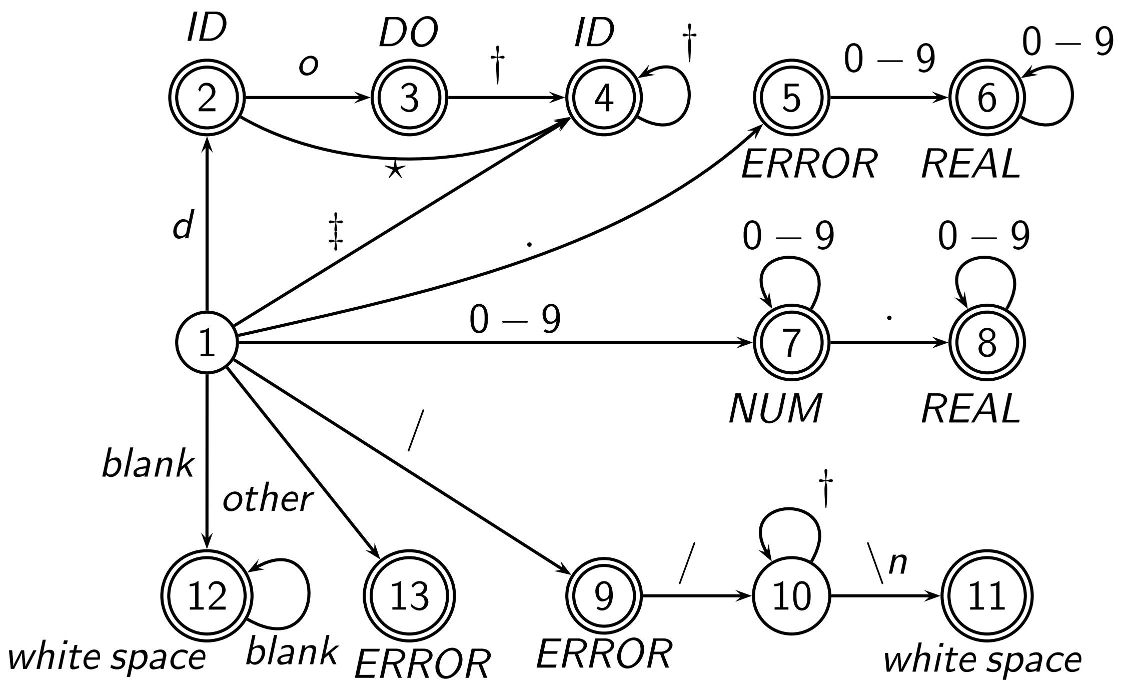
Finite Automata [3]







Combined Finite Automaton



$† = [a - zA - Z0 - 9]$

$‡ = [a - ce - zA - Z]$

$\star = [a - np - zA - Z0 - 9]$

Encoding a Finite Automaton

A finite automaton can be encoded by:

- **Transition matrix:** a 2-dimensional array, indexed by input character and state number, that contains the next state.

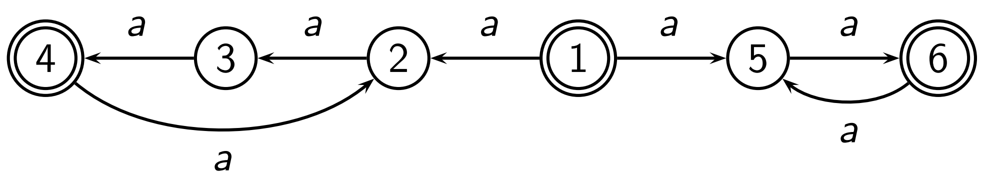
```
int edges[][] = {/* ws,..., 0, 1, 2, ...  d, e, f, ...  o, ...  */
    /* state 0 */ { 0,..., 0, 0, 0, ..., 0, 0, 0, ..., 0, ...  },
    /* state 1 */ { 0,..., 7, 7, 7, ..., 2, 4, 4, ..., 4, ...  },
    /* state 2 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 3, ...  },
    /* state 3 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ...  },
    /* state 4 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ...  },
    /* state 5 */ { 0,..., 6, 6, 6, ..., 0, 0, 0, ..., 0, ...  },
...
}
```

- **action array:** an array, indexed by final state number, that contains the resulting action, e.g. if the final state is 2 then return `ID`, if the final state is 3 then return `DO`, etc.

How do you recognize the longest match?

Nondeterministic Finite Automata

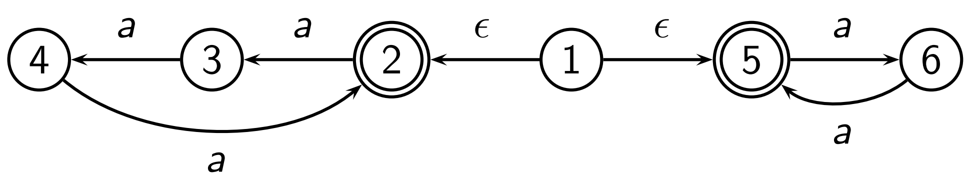
In general we need a nondeterministic finite automaton (NFA) to recognise a regular expression.



This NFA recognises strings of a's whose length is a multiple of 2 or 3.
From the start state, state 1, the automaton can transition to state 2 or 5 on receiving an a. NFA always "guesses" correctly which edge to take when faced with a nondeterministic choice.

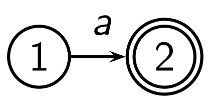
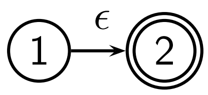
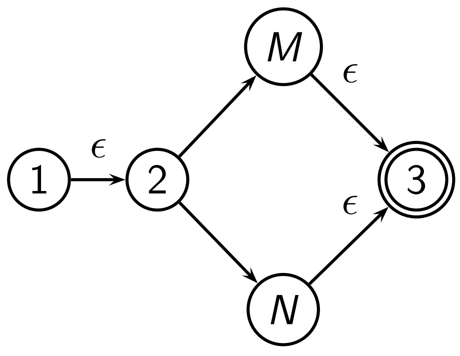
Nondeterministic Finite Automata [2]

If an edge is labelled with ϵ it can be taken without consuming a character.
An equivalent automaton to the last one is:

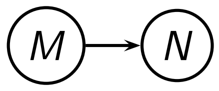
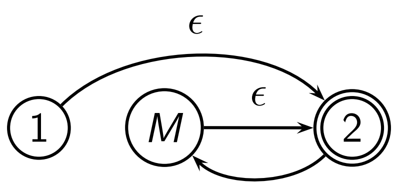
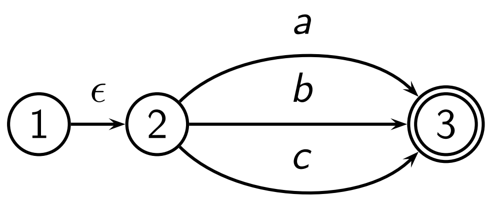


Regular Expressions to NFA

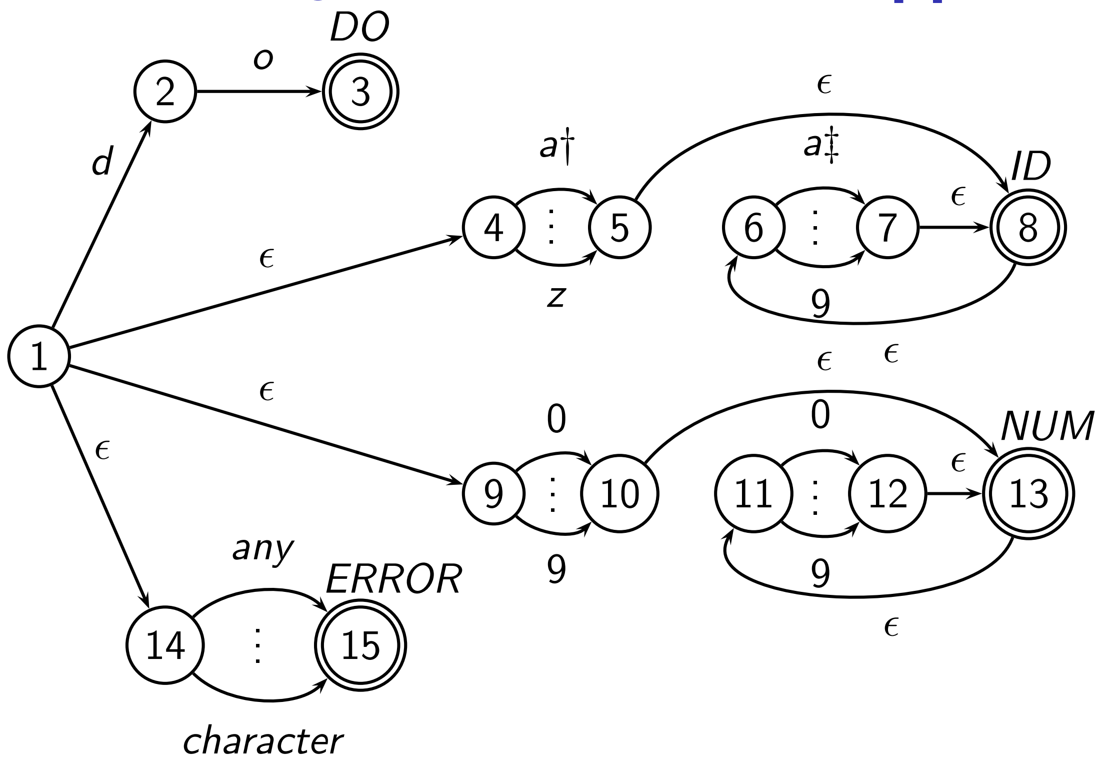
It is relatively easy to convert a regular expression into an NFA.

Regular Expression	NFA
a	
ϵ	
$M\|N$	

Regular Expressions to NFA [2]

Regular Expression	NFA
MN	
M^*	
M^+	constructed as MM^*
$M?$	constructed as $M\|\epsilon$
$[abc]$	

Regular Expressions to NFA [3]



$\dagger = [a-zA-Z]$ and $\ddagger = [a-zA-Z0-9]$

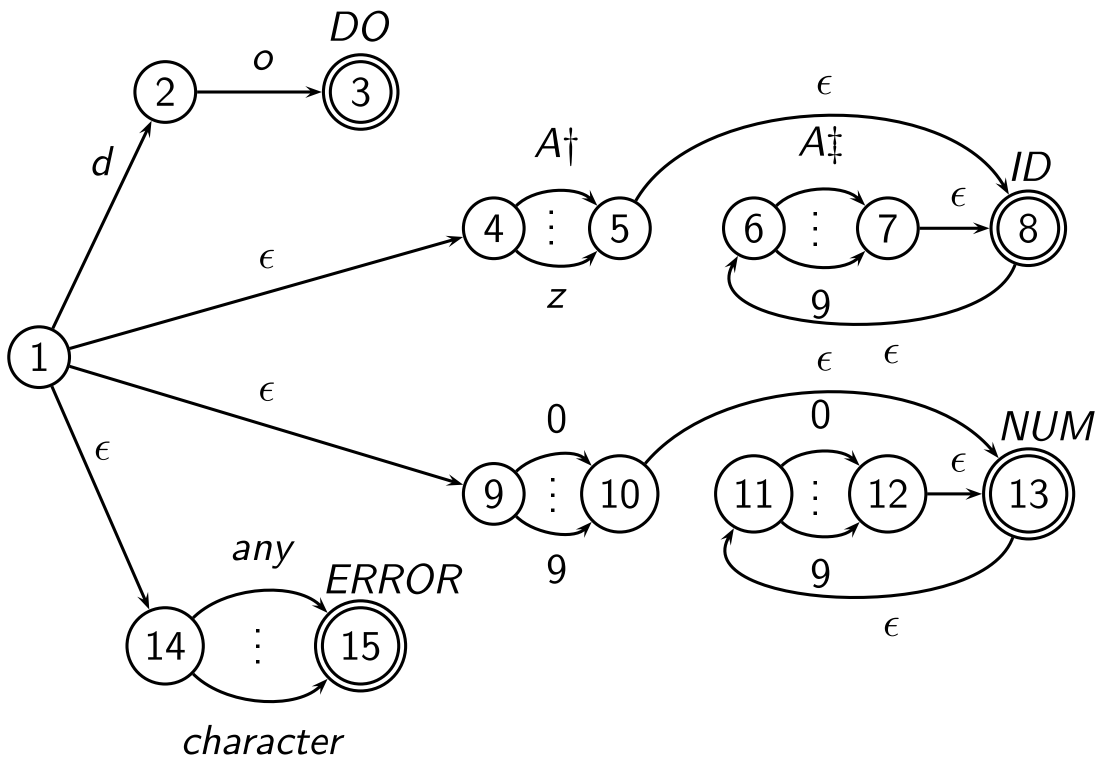
NFA to DFA

We can't execute a nondeterministic finite automaton. Rather than viewing the NFA as "guessing" the correct path it is more accurate to view the NFA as executing all nondeterministic paths concurrently and if any path terminates successfully then the NFA terminates successfully.
The *subset construction algorithm* converts a NFA to a DFA using 2 concepts:

- $\epsilon - closure(S)$: all the states that can be reached from state S without consuming any symbols, i.e. using ϵ edges only.
- $DFAedge(d, c)$: all the states that can be reached from the states in d by consuming only edges labelled with c and ϵ.

NFA to DFA [2]

Input: dog



NFA to DFA [3]

Let Σ be the alphabet of symbols and s_1 the start state.

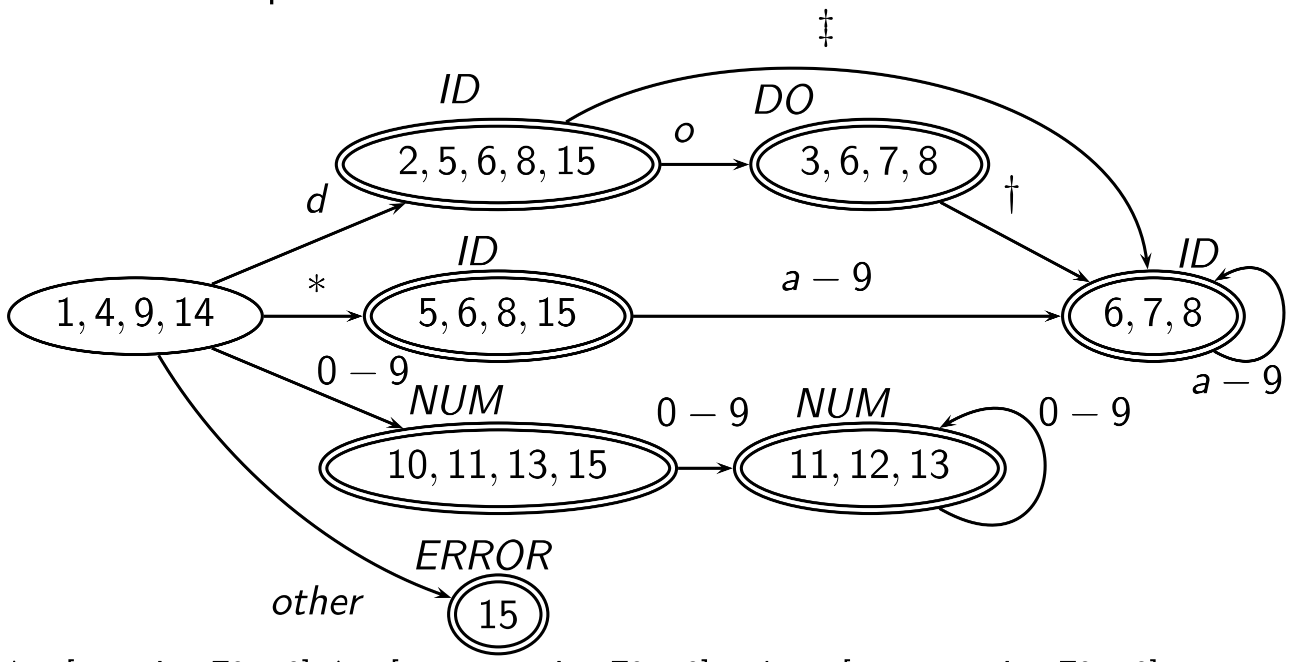
```
states[0] $\leftarrow$ ; states[1] $\leftarrow$ $\epsilon$-$closure(s_1)$
p $\leftarrow$ 1; j $\leftarrow$ 0
while $j \leq p$
        foreach $c \in \Sigma$
              $e \leftarrow$ DFAedge(states[$j$], $c$)
              if $e =$ states[$i$] for some $i \leq p$
              then trans[$j$,$c$] $\leftarrow i$
              else $p \leftarrow p + 1$
                     states[$p$] $\leftarrow e$
                     trans[$j$,$c$] $\leftarrow p$
              endif
        endfor
       $j \leftarrow j + 1$
endwhile
```

NFA to DFA [4]

- When the algorithm terminates we can discard the `states` array and keep the `trans` array to recognise the tokens.
- A state i in the DFA is *final* if one of the states in `states [i]` is a *final* state.
    - We also need to record the type of token the *final* DFA state recognises.
    - If more that one state in `states[i]` is *final* then the token type from `states[i]` that occurs first in the regular expression rules specifies the token type of this DFA state (*Rule Priority*).
- The generated DFA is suboptimal. Equivalent states can be merged together.
    - s_i is equivalent to s_j if a machine starting is s_1 accepts a string σ if and only if a machine starting in s_2 accepts σ.

NFA to DFA [5]

The subset construction algorithm would generate the following DFA from our previous NFA.



$\dagger = [a-zA-Z0-9]$, $\ddagger = [a-np-zA-Z0-9]$ and $* = [a-ce-zA-Z0-9]$

How would this simplify?

Lexical Analyser Generators

The conversion from regular expression to NFA to DFA is automatic and can be built into a *lexical-analyser generator.*

JavaCC is a program that can generate *lexical analysers* and *parsers*.

A **JavaCC** program consists of two sections:

- A *lexical specification* section that defines the tokens and white spaces.
- A *production* section (more of which later in the course).

A JavaCC Program

```
PARSER_BEGIN(MyParser)
    class MyParser
PARSER_END(MyParser)

/* For the reg exp on the right generates */
/* the token on the left */
TOKEN : {
    < IF: "if" >
  | <#DIGIT: ["0"-"9"] >
  | <#CHAR: ["a"-"z"]|["A"-"Z"]
  | <ID: (<CHAR>)(<CHAR> | <DIGIT>)* >
  | <NUM: (<DIGIT>)+ >
  | <REAL: ( (<DIGIT>)+ "." (<DIGIT>)* ) |
           ( (<DIGIT>)* "." (<DIGIT>)* ) >
}
```

A JavaCC Program [2]

```
/* The reg exp here will be skipped */
SKIP : {
    <"//" (<CHAR> | <DIGIT>)* ("\n" | "\r" | "\r\n") >
  | " "
  | "\t"
  | "\n
}

void Start ( ) :
{ }
{  ( <IF> | <ID> | <NUM> | <REAL> )*  }
```

JavaCC Grammar File

The JavaCC grammar file has the following format:

```
javacc_input ::= javacc_options
                 "PARSER_BEGIN" "(" <IDENTIFIER> ")"
                 java_compilation_unit
                 "PARSER_END" "(" <IDENTIFIER> ")"
                 (production)*
                 <EOF>
```

The name following `PARSER_BEGIN` and `PARSER_END` must be the same. If this name is `MyParser` the following files will be generated:

MyParser.java	The generated parser
MyParserTokenManager.java	The generated token manager (scanner)
MyParserConstants.java	A bunch of useful constants

A Java program is placed between the `PARSER_BEGIN` and `PARSER_END`. This must contain a class declaration with the same name as the generated parser.

Reading Tokens

The token manager produced by JavaCC provides one public method:

Token getNextToken() throws ParseException

This method returns the next available token in the input stream and moves the pointer one step in the input stream.
This method is not normally called directly, with all access being made through the parser interface.

Two very important attributes of Token objects are as follows:

kind	the kind of the token (as specified in the token manager)
image	the lexeme associated with the token

Another JavaCC Example

The following straight-line programming language from Appel's book (Chapter 1) will be used to illustrate some examples of using JavaCC.

Stm	$\rightarrow$	*Stm* ; *Stm*	(CompoundStm)
Stm	$\rightarrow$	id := *Exp*	(AssignStm)
Stm	$\rightarrow$	print (*ExpList*)	(PrintStm)
Exp	$\rightarrow$	id	(IdExp)
Exp	$\rightarrow$	num	(NumExp)
Exp	$\rightarrow$	*Exp Binop Exp*	(OpExp)
Exp	$\rightarrow$	(*Stm* , *Exp*)	(EseqExp)
ExpList	$\rightarrow$	*Exp* , *ExpList*	(PairExpList)
ExpList	$\rightarrow$	*Exp*	(LastExpList)
Binop	$\rightarrow$	$+$	(Plus)
Binop	$\rightarrow$	-	(Minus)
Binop	$\rightarrow$	$\times$	(Times)
Binop	$\rightarrow$	/	(Div)

Another JavaCC Example [2]

```
/*******************************
 ***** SECTION 1 - OPTIONS *****
 *******************************/

options { JAVA_UNICODE_ESCAPE = true; }

/*********************************
 ***** SECTION 2 - USER CODE *****
 *********************************/

PARSER_BEGIN(SLPTokeniser)

public class SLPTokeniser {

  public static void main(String args[]) {

    SLPTokeniser tokeniser;
    if (args.length == 0) {
      System.out.println("Reading from standard input . . .");
      tokeniser = new SLPTokeniser(System.in);
    } else if (args.length == 1) {
      try {
        tokeniser = new SLPTokeniser(new java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
        System.err.println("File " + args[0] + " not found.");
        return;
      }
    }
```

Another JavaCC Example [3]

```
    else {
      System.out.println("SLP Tokeniser:  Usage is one of:");
      System.out.println("         java SLPTokeniser < inputfile");
      System.out.println("OR");
      System.out.println("         java SLPTokeniser inputfile");
      return;
    }

    /*
     * We've now initialised the tokeniser to read from the appropriate place,
     * so just keep reading tokens and printing them until we hit EOF
     */

    for (Token t = getNextToken(); t.kind!=EOF; t = getNextToken()) {
      // Print out the actual text for the constants, identifiers etc.
      if (t.kind==NUM)
      {
         System.out.print("Number");
         System.out.print("("+t.image+") ");
      }
      else if (t.kind==ID)
      {
         System.out.print("Identifier");
         System.out.print("("+t.image+") ");
      }
      else
         System.out.print(t.image+" ");
    }
  }
}
PARSER_END(SLPTokeniser)
```

Another JavaCC Example [4]

```
/*******************************************
 ***** SECTION 3 - TOKEN DEFINITIONS *****
 *******************************************/

TOKEN_MGR_DECLS :
{
   static int commentNesting = 0;
}

SKIP : /*** Ignoring spaces/tabs/newlines ***/
{
    " "
  | "\t"
  | "\n"
  | "\r"
  | "\f"
}

SKIP : /* COMMENTS */
{
    "/*" { commentNesting++; } : IN_COMMENT
}

<IN_COMMENT> SKIP :
{
    "/*" { commentNesting++; }
  | "*/" { commentNesting--;
           if (commentNesting == 0)
              SwitchTo(DEFAULT);
         }
  | <~[]>
}
```

Another JavaCC Example [5]

```
TOKEN :  /* Keywords and punctuation */
{
  < SEMIC : ";" >
| < ASSIGN : ":=" >
| < PRINT : "print" >
| < LBR : "(" >
| < RBR : ")" >
| < COMMA : "," >
| < PLUS_SIGN : "+" >
| < MINUS_SIGN : "-" >
| < MULT_SIGN : "*" >
| < DIV_SIGN : "/" >
}

TOKEN :  /* Numbers and identifiers */
{
  < NUM : (<DIGIT>)+ >
| < #DIGIT : ["0" - "9"] >
| < ID : (<LETTER>)+ >
| < #LETTER : ["a" - "z", "A" - "Z"] >
}


TOKEN : /* Anything not recognised so far */
{
  < OTHER : ~[] >
}

/***************************************************************************
 * SECTION 4 - THE GRAMMAR & PRODUCTION RULES - WOULD NORMALLY START HERE *
 ***************************************************************************/
```