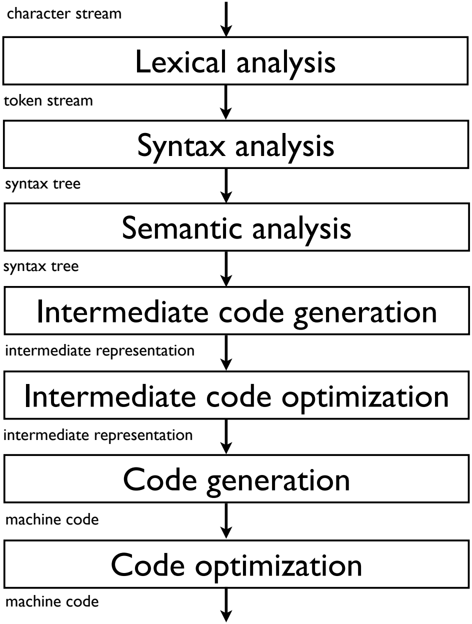
Part 2

Lexical analysis

Outline

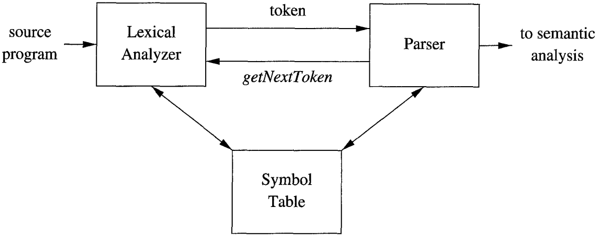
1. Principle
2. Regular expressions
3. Analysis with non-deterministic finite automata
4. Analysis with deterministic finite automata
5. Implementing a lexical analyzer

Structure of a compiler



Lexical analysis or scanning

- Goals of the lexical analysis
  - Divide the character stream into meaningful sequences called lexemes.
  - Label each lexeme with a token that is passed to the parser (syntax analysis)
  - Remove non-significant blanks and comments
  - Optional: update the symbol tables with all identifiers (and numbers)
- Provide the interface between the source program and the parser



(Dragonbook)

Example

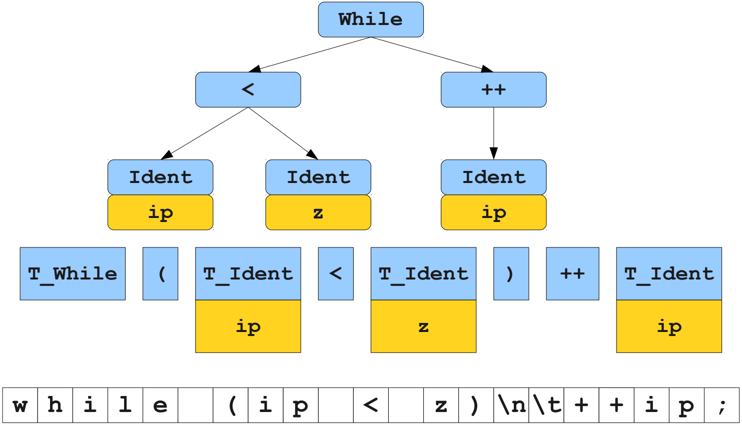
T_While	(	T_Ident	<	T_Ident	)	++	T_Ident
		ip		z			ip

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```

(Keith Schwarz)

Example




```
while (ip < z)
  ++ip;
```

(Keith Schwarz)

Lexical versus syntax analysis

Why separate lexical analysis from parsing?

- Simplicity of design: simplify both the lexical analysis and the syntax analysis.
- Efficiency: specialized techniques can be applied to improve lexical analysis.
- Portability: only the scanner needs to communicate with the outside

Tokens, patterns, and lexemes

- A token is a $\langle name, attribute \rangle$ pair. Attribute might be multi-valued.
  - Example: $\langle Ident, ip \rangle$, $\langle Operator, < \rangle$, $\langle$ ")", $NIL \rangle$
- A pattern describes the character strings for the lexemes of the token.
  - Example: a string of letters and digits starting with a letter, $\{<, >, \leq, \geq, ==\}$, ")".
- A lexeme for a token is a sequence of characters that matches the pattern for the token
  - Example: **ip**, "$<$", ")" in the following program

```
while (ip < z)
   ++ip
```

Defining a lexical analysis

1. Define the set of tokens
2. Define a pattern for each token (ie., the set of lexemes associated with each token)
3. Define an algorithm for cutting the source program into lexemes and outputting the tokens

Choosing the tokens

- Very much dependent on the source language
- Typical token classes for programming languages:
  - One token for each keyword
  - One token for each "punctuation" symbol (left and right parentheses, comma, semicolon...)
  - One token for identifiers
  - Several tokens for the operators
  - One or more tokens for the constants (numbers or literal strings)
- Attributes
  - Allows to encode the lexeme corresponding to the token when necessary. Example: pointer to the symbol table for identifiers, constant value for constants.
  - Not always necessary. Example: keyword, punctuation...

Describing the patterns

- A pattern defines the set of lexemes corresponding to a token.
- A lexeme being a string, a pattern is actually a language.
- Patterns are typically defined through regular expressions (that define regular languages).
  - Sufficient for most tokens
  - Lead to efficient scanner

Reminder: languages

- An alphabet Σ is a set of characters

  Example: $\Sigma = \{a, b\}$

- A string over Σ is a finite sequence of elements from Σ

  Example: $aabba$

- A language is a set of strings

  Example: $L = \{a, b, abab, babbba\}$

- Regular languages: a subset of all languages that can be defined by regular expressions

Reminder: regular expressions

- Any character $a \in \Sigma$ is a regular expression $L = \{a\}$
- ϵ is a regular expression $L = \{\epsilon\}$
- If R_1 and R_2 are regular expressions, then
  - $R_1 R_2$ is a regular expression
    $L(R_1 R_2)$ is the concatenation of $L(R1)$ and $L(R2)$
  - $R_1|R_2$ $(= R_1 \bigcup R_2)$ is a regular expression
    $L(R_1|R_2) = L(R_1) \bigcup L(R_2)$
  - R_1^* is a regular expression
    $L(R_1^*)$ is the Kleene closure of $L(R_1)$
  - (R_1) is a regular expression
    $L((R_1)) = L(R_1)$

- Example: a regular expression for even numbers:

$$(+|-|\epsilon)(0|1|2|3|4|5|6|7|8|9)^*(0|2|4|6|8)$$

Notational conveniences

- Regular definitions:

$$\begin{aligned} \mathit{letter} &\rightarrow \mathrm{A}|\mathrm{B}|\ldots|\mathrm{Z}|\mathrm{a}|\mathrm{b}|\ldots|\mathrm{z} \\ \mathit{digit} &\rightarrow 0|1|\ldots|9 \\ \mathit{id} &\rightarrow \mathit{letter}(\mathit{letter}|\mathit{digit})^* \end{aligned}$$

- One or more instances: $r^+ = rr^*$
- Zero or one instance: $r? = r|\epsilon$
- Character classes:

$$[\mathrm{abc}]{=}\mathrm{a}|\mathrm{b}|\mathrm{c}$$
$$[\mathrm{a\text{-}z}]{=}\mathrm{a}|\mathrm{b}|\ldots|\mathrm{z}$$
$$[0\text{-}9]{=}0|1|\ldots|9$$

Examples

- Keywords:

  if, while, for, ...

- Identifiers:

  $$[a\text{-}zA\text{-}Z\_][a\text{-}zA\text{-}Z\_0\text{-}9]^*$$

- Integers:

  $$[+-]?[0\text{-}9]^+$$

- Floats:

  $$[+-]?(([0\text{-}9]^+\ (.[0\text{-}9]^*)?|.[0\text{-}9]^+)([eE][+-]?[0\text{-}9]^+)?)$$

- String constants:

  $$``([a\text{-}zA\text{-}Z0\text{-}9]|\backslash[a\text{-}zA\text{-}Z])^*"$$

Algorithms for lexical analysis

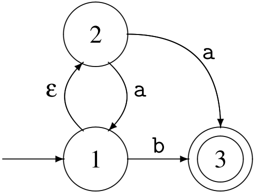
- How to perform lexical analysis from token definitions through regular expressions?
- Regular expressions are equivalent to finite automata, deterministic (DFA) or non-deterministic (NFA).
- Finite automata are easily turned into computer programs
- Two methods:
  1. Convert the regular expressions to an NFA and simulate the NFA
  2. Convert the regular expression to an NFA, convert the NFA to a DFA, and simulate the DFA.

Reminder: non-deterministic automata (NFA)

A non-deterministic automaton is a five-tuple $M = (Q, \Sigma, \Delta, s_0, F)$ where:

- Q is a finite set of states,
- Σ is an alphabet,
- $\Delta \subset (Q \times (\Sigma \bigcup \{\epsilon\}) \times Q)$ is the transition relation,
- $s \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states

Example:



(Mogensen)

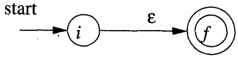
Transition table

State	a	b	ϵ
1	$\emptyset$	$\{3\}$	$\{2\}$
2	$\{1,3\}$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

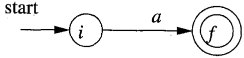
Reminder: from regular expression to NFA

A regular expression can be transformed into an equivalent NFA

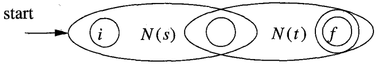
$\boxed{\epsilon}$



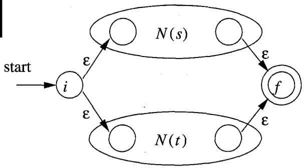
$\boxed{a}$



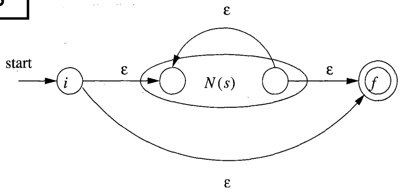
$\boxed{st}$



$\boxed{s|t}$



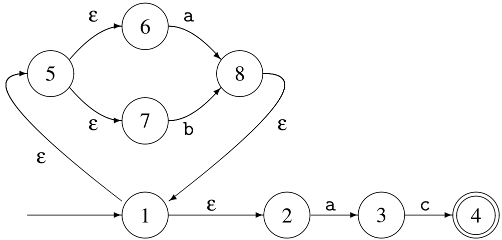
$\boxed{s^*}$



(Dragonbook)

Reminder: from regular expression to NFA

Example: (a|b)*ac (Mogensen)



The NFA $N(r)$ for an expression r is such that:

- $N(r)$ has at most twice as many states as there are operators and operands in R.
- $N(r)$ has one initial state and one accepting state (with no outgoing transition from the accepting state and no incoming transition to the initial state).
- Each (non accepting) state in $N(r)$ has either one outgoing transition or two outgoing transitions, both on ϵ.

Simulating an NFA

Algorithm to check whether an input string is accepted by the NFA:

```
S = ϵ-closure(s0);
c = nextChar();
while ( c != eof ) {
        S = ϵ-closure(move(S, c));
        c = nextChar();
}
if ( S ∩ F != ∅ ) return "yes";
else return "no";
```

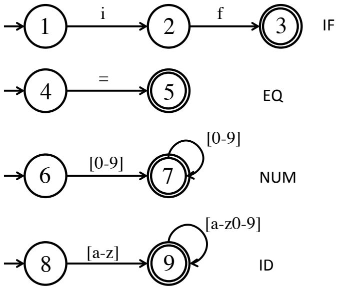
(Dragonbook)

- *nextChar*(): returns the next character on the input stream
- *move*(S, c): returns the set of states that can be reached from states in S when observing c.
- ϵ-*closure*(S): returns all states that can be reached with ϵ transitions from states in S.

Lexical analysis

- What we have so far:
  - Regular expressions for each token
  - NFAs for each token that can recognize the corresponding lexemes
  - A way to simulate an NFA
- How to combine these to cut apart the input text and recognize tokens?
- Two ways:
  - Simulate all NFAs in turn (or in parallel) from the current position and output the token of the first one to get to an accepting state
  - Merge all NFAs into a single one with labels of the tokens on the accepting states

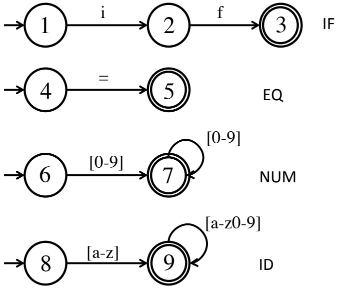
Illustration



- Four tokens: IF=if, ID=[a-z][a-z0-9]*, EQ='=', NUM=[0-9]$^+$
- Lexical analysis of $x = 6$ yields:

$$\langle ID, x\rangle, \langle EQ\rangle, \langle NUM, 6\rangle$$

Illustration: ambiguities



- Lexical analysis of $ifu26 = 60$
- Many splits are possible:

$$\langle IF \rangle, \langle ID, u26 \rangle, \langle EQ \rangle, \langle NUM, 60 \rangle$$

$$\langle ID, ifu26 \rangle, \langle EQ \rangle, \langle NUM, 60 \rangle$$

$$\langle ID, ifu \rangle, \langle NUM, 26 \rangle, \langle EQ \rangle, \langle NUM, 6 \rangle, \langle NUM, 0 \rangle$$

$$\ldots.$$

Conflict resolutions

- Principle of the longest matching prefix: we choose the longest prefix of the input that matches any token
- Following this principle, $ifu26 = 60$ will be split into:

$$\langle ID, ifu26\rangle, \langle EQ\rangle, \langle NUM, 60\rangle$$

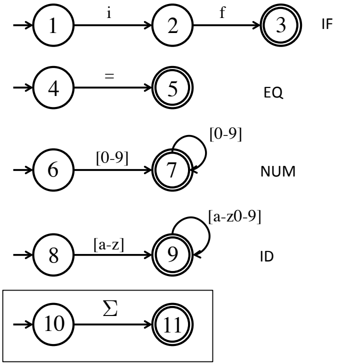
- How to implement?
  - Run all NFAs in parallel, keeping track of the last accepting state reached by any of the NFAs
  - When all automata get stuck, report the last match and restart the search at that point
- Requires to retain the characters read since the last match to re-insert them on the input
  - In our example, '$=$' would be read and then re-inserted in the buffer.

Other source of ambiguity

- A lexeme can be accepted by two NFAs
  - Example: keywords are often also identifiers (*if* in the example)
- Two solutions:
  - Report an error (such conflict is not allowed in the language)
  - Let the user decide on a priority order on the tokens (eg., keywords have priority over identifiers)

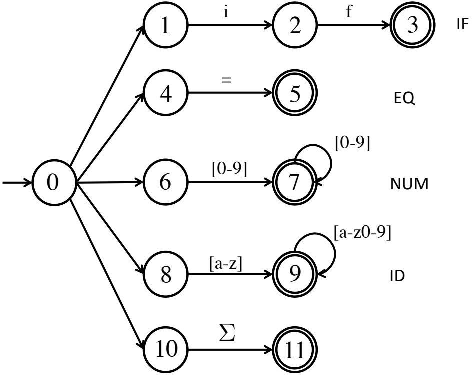
What if nothing matches

- What if we can not reach any accepting states given the current input?
- Add a "catch-all" rule that matches any character and reports an error



Merging all automata into a single NFA

- In practice, all NFAs are merged and simulated as a single NFA
- Accepting states are labeled with the token name



Lexical analysis with an NFA: summary

- Construct NFAs for all regular expression
- Merge them into one automaton by adding a new start state
- Scan the input, keeping track of the last known match
- Break ties by choosing higher-precedence matches
- Have a catch-all rule to handle errors

Computational efficiency

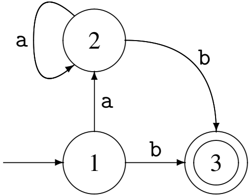
```
S = ε-closure(s0);
c = nextChar();
while ( c != eof ) {
        S = ε-closure(move(S, c));
        c = nextChar();
}
if ( S ∩ F != ∅ ) return "yes";
else return "no";
```

(Dragonbook)

- In the worst case, an NFA with $|Q|$ states takes $O(|S||Q|^2)$ time to match a string of length $|S|$
- Complexity thus depends on the number of states
- It is possible to reduce complexity of matching to $O(|S|)$ by transforming the NFA into an equivalent deterministic finite automaton (DFA)

Reminder: deterministic finite automaton

- Like an NFA but the transition relation $\Delta \subset (Q \times (\Sigma \bigcup \{\epsilon\}) \times Q)$ is such that:
  - Transitions based on ϵ are not allowed
  - Each state has at most one outgoing transition defined for every letter
- Transition relation is replaced by a transition function $\delta : Q \times \Sigma \rightarrow Q$
- Example of a DFA



(Mogensen)

Reminder: from NFA to DFA

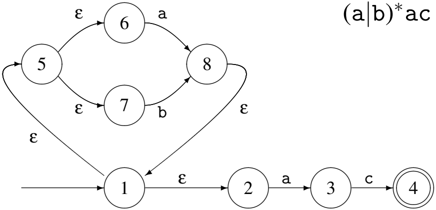
- DFA and NFA (and regular expressions) have the same expressive power
- An NFA can be converted into a DFA by the subset construction method
- Main idea: mimic the simulation of the NFA with a DFA
  - Every state of the resulting DFA corresponds to a set of states of the NFA. First state is ϵ-*closure*(s_0).
  - Transitions between states of DFA correspond to transitions between set of states in the NFA:

$$\delta(S, c) = \epsilon\text{-}closure(move(S, c))$$

  - A set of the DFA is accepting if any of the NFA states that it contains is accepting
- See INFO0016 or the reference book for more details

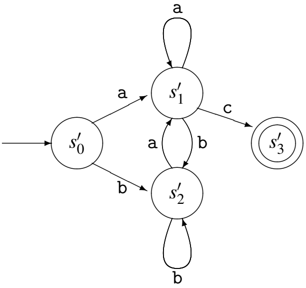
Reminder: from NFA to DFA

NFA



$(\mathtt{a}|\mathtt{b})^*\mathtt{ac}$

DFA



s'_0	$\{1,2,5,6,7\}$
s'_1	$\{3,8,1,2,5,6,7\}$
s'_2	$\{8,1,2,5,6,7\}$
s'_3	$\{4\}$

(Mogensen)

Simulating a DFA

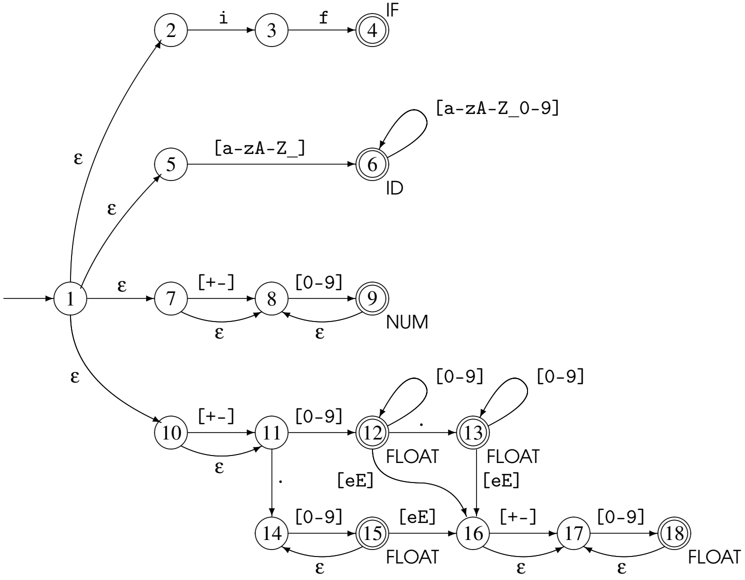
$s = s_0;$
$c = nextChar();$
while (c != **eof**) {
$\quad s = move(s, c);$
$\quad c = nextChar();$
}
if (s is in F) **return** "yes";
else return "no";

- Time complexity is $O(|S|)$ for a string of length $|S|$
- Now independent of the number of states

Lexical analysis with a DFA: summary

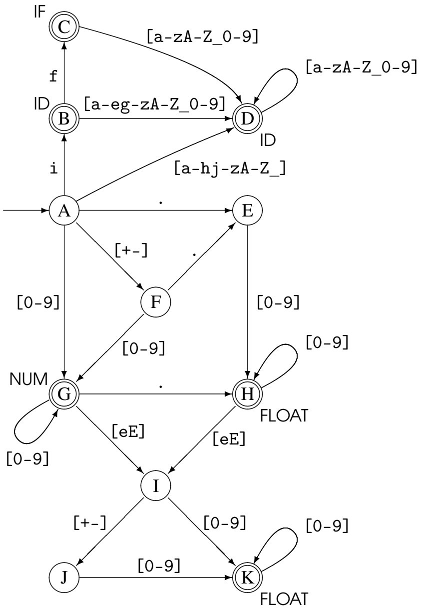
- Construct NFAs for all regular expressions
- Mark the accepting states of the NFAs by the name of the tokens they accept
- Merge them into one automaton by adding a new start state
- Convert the combined NFA to a DFA
- Convey the accepting state labeling of the NFAs to the DFA (by taking into account precedence rules)
- Scanning is done like with an NFA

Example: combined NFA for several tokens



(Mogensen)

Example: combined DFA for several tokens



Try lexing on the strings:

- *if* 17
- 3*e*-*y*

Speed versus memory

- The number of states of a DFA can grow exponentially with respect to the size of the corresponding regular expression (or NFA)
- We have to choose between low-memory and slow NFAs and high-memory and fast DFAs.

Note:

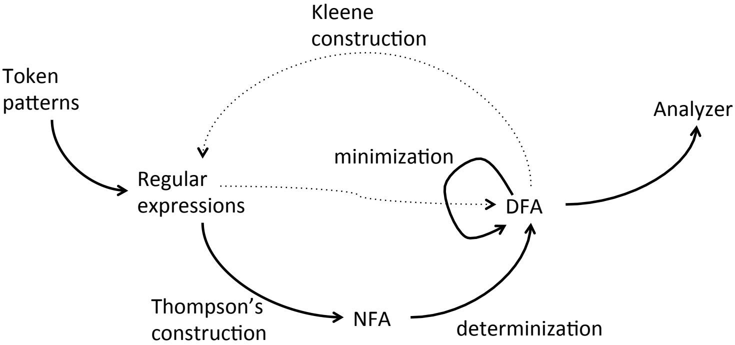
- It is possible to minimise the number of states of a DFA in $O(n \log n)$ (Hopcroft's algorithm[1])
  - Theory says that any regular language has a unique minimal DFA
  - However, the number of states may remain exponential in the size of the regular expression after minimization

[1] `http://en.wikipedia.org/wiki/DFA_minimization`

Keywords and identifiers

- Having a separate regular expression for each keyword is not very efficient.
- In practice:
  - We define only one regular expression for both keywords and identifiers
  - All keywords are stored in a (hash) table
  - Once an identifier/keyword is read, a table lookup is performed to see whether this is an identifier or a keyword
- Reduces drastically the size of the DFA
- Adding a keyword requires only to add one entry in the hash table.

Summary


Kleene
construction
Token
patterns
Analyzer
minimization
Regular
expressions
DFA
Thompson's
construction
NFA
determinization

Some langage specificities

Language specificities that make lexical analysis hard:

- Whitespaces are irrelevant in Fortran.

```
DO 5 I = 1,25
DO5I = 1.25
```

- PL/1: keywords can be used as identifiers:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

- Python block defined by indentation:

```
if w == z:
     a = b
else:
     e = f
g = h
```

(the lexical analyser needs to record current identation and output a token for each increase/decrease in indentation)

(Keith Schwarz)

Some langage specificities

- Sometimes, nested lexical analyzers are needed
- For example, to deal with nested comments:

  ```
  /* /* where do my comments end? here? */ or here? */
  ```

  - As soon as /* is read, switch to another lexical analyzer that
    - only reads /* and */,
    - counts the level of nested comments at current position (starting at 0),
    - get back to the original analyzer when it reads */ and the level is 0
- Other example: Javadoc (needs to interpret the comments)

NB: How could you test if your compiler accepts nested comments without generating a compilation error?

```
int nest = /*/*/0*/**/1;
```

Implementing a lexical analyzer

- In practice (and for your project), two ways:
  - Write an ad-hoc analyser
  - Use automatic tools like (F)LEX.
- First approach is more tedious. It is only useful to address specific needs.
- Second approach is more portable

Example of an ad-hoc lexical analyser

(source: http://dragonbook.stanford.edu/lecture-notes.html)

Definition of the token classes (through constants)

```
#define T_SEMICOLON  ';'     // use ASCII values for single char tokens
#define T_LPAREN  '('
#define T_RPAREN  ')'
#define T_ASSIGN  '='
#define T_DIVIDE  '/'
 ...

#define T_WHILE  257         // reserved words
#define T_IF  258
#define T_RETURN  259
 ...

#define T_IDENTIFIER  268    // identifiers, constants, etc.
#define T_INTEGER  269
#define T_DOUBLE   270
#define T_STRING  271

#define T_END  349           // code used when at end of file
#define T_UNKNOWN  350       // token was unrecognized by scanner
```

Example of an ad-hoc lexical analyser

Structure for tokens

```
struct token_t {
  int type;                   // one of the token codes from above
  union {
    char stringValue[256];    // holds lexeme value if string/identifier
    int intValue;             // holds lexeme value if integer
    double doubleValue;       // holds lexeme value if double
  } val;
};
```

Main function

```
int main(int argc, char *argv[])
{
  struct token_t token;

  InitScanner();
   while (ScanOneToken(stdin, &token) != T_END)
    ; // this is where you would process each token
  return 0;
}
```

Example of an ad-hoc lexical analyser

Initialization

```
static void InitScanner()
{
  create_reserved_table();  // table maps reserved words to token type
  insert_reserved("WHILE", T_WHILE)
  insert_reserved("IF", T_IF)
  insert_reserved("RETURN", T_RETURN)
  ....
}
```

Example of an ad-hoc lexical analyser

Scanning (single-char tokens)

```
static int ScanOneToken(FILE *fp, struct token_t *token)
{
  int i, ch, nextch;

  ch = getc(fp);     // read next char from input stream
  while (isspace(ch))     // if necessary, keep reading til non-space char
    ch = getc(fp);     // (discard any white space)

  switch(ch) {
    case '/':    // could either begin comment or T_DIVIDE op
      nextch = getc(fp);
      if (nextch == '/' || nextch == '*')
        ; // here you would skip over the comment
      else
        ungetc(nextch, fp); // fall-through to single-char token case

    case ';': case ',': case '=':  // ... and other single char tokens
      token->type = ch; // ASCII value is used as token type
      return ch;     // ASCII value used as token type
```

Example of an ad-hoc lexical analyser

Scanning: keywords

```
case 'A': case 'B': case 'C':   // ... and other upper letters
  token->val.stringValue[0] = ch;
  for (i = 1; isupper(ch = getc(fp)); i++) // gather uppercase
    token->val.stringValue[i] = ch;
  ungetc(ch, fp);
  token->val.stringValue[i] = '\0';  // lookup reserved word
  token->type = lookup_reserved(token->val.stringValue);
  return token->type;
```

Scanning: identifier

```
case 'a': case 'b': case 'c':  // ... and other lower letters
  token->type = T_IDENTIFIER;
  token->val.stringValue[0] = ch;
  for (i = 1; islower(ch = getc(fp)); i++)
    token->val.stringValue[i] = ch; // gather lowercase
  ungetc(ch, fp);
  token->val.stringValue[i] = '\0';
  if (lookup_symtab(token->val.stringValue) == NULL)
    add_symtab(token->val.stringValue); // get symbol for ident
  return T_IDENTIFIER;
```

Example of an ad-hoc lexical analyser

Scanning: number

```
case '0': case '1': case '2': case '3':  //....  and other digits
  token->type = T_INTEGER;
  token->val.intValue = ch - '0';
  while (isdigit(ch = getc(fp))) // convert digit char to number
    token->val.intValue = token->val.intValue * 10 + ch - '0';
  ungetc(ch, fp);
  return T_INTEGER;
```

Scanning: EOF and default

```
case EOF:
  return T_END;

default:    // anything else is not recognized
  token->val.intValue = ch;
  token->type = T_UNKNOWN;
  return T_UNKNOWN;
```

Flex

- `flex` is a free implementation of the Unix `lex` program
- `flex` implements what we have seen:
  - It takes regular expressions as input
  - It generates a combined NFA
  - It converts it to an equivalent DFA
  - It minimizes the automaton as much as possible
  - It generates C code that implements it
  - It handles conflicts with the longest matching prefix principle and a preference order on the tokens.
- More information
  - `http://flex.sourceforge.net/manual/`

Input file

- Input files are structured as follows:

```
%{
Declarations
%}
Definitions
%%
Rules
%%
User subroutines
```

- Declarations and User subroutines are copied without modifications to the generated C file.
- Definitions specify options and name definitions (to simplify the rules)
- Rules: specify the patterns for the tokens to be recognized

Rules

- In the form:

```
pattern1 action1
pattern2 action2
...
```

- Patterns are defined as regular expressions. Actions are blocks of C code.
- When a sequence is read that matches the pattern, the C code of the action is executed
- Examples:

```
[0-9]+ {printf("This is a number");}
[a-z]+ {printf("This is symbol");}
```

Regular expressions

- Many shortcut notations are permitted in regular expressions:
  - `[]`, `-`, `+`, `*`, `?`: as defined previously
  - `.`: a dot matches any character (except newline)
  - `[^x]`: matches the complement of the set of characters in `x` (ex: all non-digit characters `[^0-9]`).
  - `x{n,m}`: `x` repeated between `n` and `m` times
  - `"x"`: matches `x` even if `x` contains special characters (ex: `"x*"` matches `x` followed by a star).
  - `{name}`: replace with the pattern defined earlier in the definition section of the input file

Interacting with the scanner

- User subroutines and action may interact with the generated scanner through global variables:
  - `yylex`: scan tokens from the global input file `yyin` (defaults to stdin). Continues until it reaches the end of the file or one of its actions executes a return statement.
  - `yytext`: a null-terminated string (of length `yyleng`) containing the text of the lexeme just recognized.
  - `yylval`: store the attributes of the token
  - `yylloc`: location of the tokens in the input file (line and column)
  - ...

Example 1: hiding numbers

- hide-digits.l:

```
%%
[0-9]+ printf("?");
. ECHO;
```

- To build and run the program:

```
% flex hide-digits.l
% gcc -o hide-digits lex.yy.c ll
% ./hide-digits
```

Example 2: wc

- count.l:

```
%{
   int numChars = 0, numWords = 0, numLines = 0;
%}
%%
\n          {numLines++; numChars++;}
[^ \t\n]+   {numWords++; numChars += yyleng;}
.           {numChars++;}
%%

int main() {
    yylex();
    printf("%d\t%d\t%d\n", numChars, numWords, numLines);
}
```

- To build and run the program:

```
% flex count.l
% gcc -o count lex.yy.c ll
% ./count < count.l
```

Example 3: typical compiler

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}
```

Example 3: typical compiler

User defined subroutines

```
%%

int installID() {/* function to install the lexeme, whose
                     first character is pointed to by yytext,
                     and whose length is yyleng, into the
                     symbol table and return a pointer
                     thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```