# Lexical analysis

Prof Rajeev Barua
Program Analysis -- Slide Set 4

# Lexical analysis: definition and tokens

Lexical analysis is the first task of the compiler. It has the following inputs and outputs:

- **Input**: The text of a high level program.
- **Output**: Individual words or "tokens" in the program representing keywords, identifiers, numbers, operators, syntax elements, etc.

**Examples of tokens**:

| Type | Examples |
|---|---|
| ID | `foo  n14  last` |
| NUM | `73  0  00  515  082` |
| REAL | `66.1  .5  10.  1e67  5.5e-10` |
| IF | `if` |
| COMMA | `,` |
| NOTEQ | `!=` |
| LPAREN | `(` |
| RPAREN | `)` |

**The following are not tokens because the preprocessor removes them before lexical analysis**:

| | |
|---|---|
| *comment* | `/* try again */` |
| *preprocessor directive* | `#include<stdio.h>` |
| *preprocessor directive* | `#define NUMS 5 , 6` |
| *macro* | `NUMS` |
| *blanks, tabs, and newlines* | |

# Lexical analysis: example input & output

Given a program such as

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

the lexical analyzer will return the stream

```
FLOAT    ID(match0)    LPAREN    CHAR    STAR    ID(s)    RPAREN
LBRACE    IF    LPAREN    BANG    ID(strncmp)    LPAREN    ID(s)
COMMA    STRING(0.0)    COMMA    NUM(3)    RPAREN    RPAREN
RETURN    REAL(0.0)    SEMI    RBRACE    EOF
```

*From page 17 of the text book*

# Regular expressions

We need a way to recognize tokens automatically.

- We use a mathematical formulation called regular expressions for that.
- Regular expressions are a way of specifying legal expressions for tokens

An example of a regular expression: $(a|b)*aa(a|\epsilon)*$

Regular expressions are specified as follows:

- **Symbols** (eg a)
- **Alternation**. (eg M|N means M or N)
- **Concatenation** (eg M.N or MN)
- **Epsilon** (ϵ) implies empty string "".
- **Repetition** M* means concatenation of zero or more instances of M.
  - M+ means one or more instances.

A regular expression is a way of representing a set of strings that are members of the language specified by the regular expression.

# Examples of regular expressions

**In general**:

| | |
|---|---|
| $(0 \mid 1)^* \cdot 0$ | Binary numbers that are multiples of two. |
| $b^*(abb^*)^*(a\mid\epsilon)$ | Strings of a's and b's with no consecutive a's. |
| $(a\mid b)^*aa(a\mid b)^*$ | Strings of a's and b's containing consecutive a's. |

**For programming languages**:

| | |
|---|---|
| `if` | `{return IF;}` |
| `[a-z][a-z0-9]*` | `{return ID;}` |
| `[0-9]+` | `{return NUM;}` |
| `([0-9]+"."[0-9]*)\|([0-9]*"."[0-9]+)` | `{return REAL;}` |
| `("--"[a-z]*"\n")\|(" "\|"\n"\|"\t")+` | `{ /* do nothing */ }` |
| `.` | `{error();}` |

FIGURE 2.2. Regular expressions for some tokens.

Rule 5: -- is for starting comments in this language.

Rule 6: . matches any single character except newline, and is present to match any single character not matched by any other rule.

# Ambiguity and disambiguation of regular expressions

Above rules are ambiguous. Examples:

- `"if"` could be a keyword or an identifier.
- `"if8"` could be a single identifier, or a keyword`"if"` followed by token `"8"`.

**Disambiguation rules**:

- **Longest match**: the longest initial substring of the input that can match any regular expression is taken as the next token.
  - Eg of use: `"if8"` matches identifier since that is the token with the longest match.

- **Rule priority**: For a particular longest initial substring, the first regular expression that matches is used.
  - Thus the order of writing rules is important!
  - Eg of use: `"if"` matches a reserved word, not identifier, by rule priority.

# Finite Automaton

A **finite automaton (FA)** is a graph-based formalism for recognizing regular expressions.

- It contain states and edges.
- Examples of FAs are on the right.
- Plural form: Finite Automata.

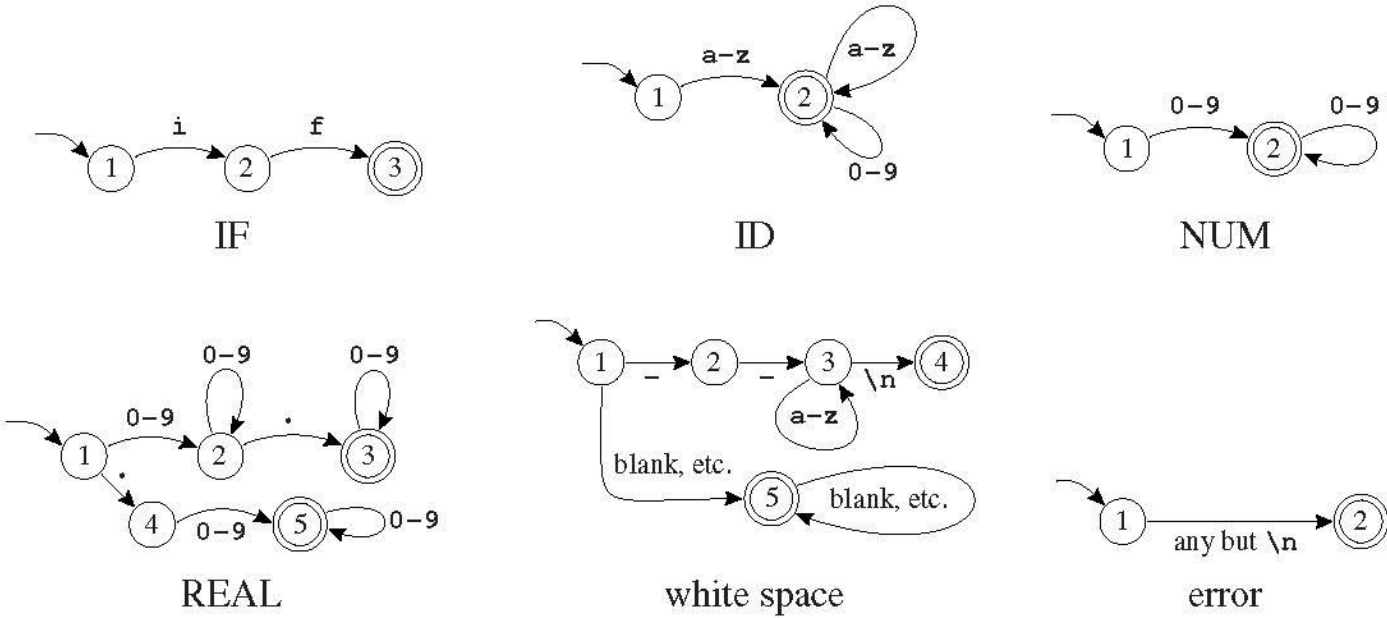


**FIGURE 2.3.** Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

# The need to combine finite automata

Next we shall try to combine multiple FAs into one.

To do so, we need concept of DFA (deterministic FA):

- In a DFA, there cannot be more than one edge out of any state with the same input character.
  - So it knows where to go next!
- A combined FA must be a DFA to be useful as a automated recognizer in a computer program!

Combining the FAs on the previous page in the obvious way does not work.

- For example, combining the FAs for IF and ID results in two outgoing edges from state 1 for the letter `"i"`.
  - This violates the condition for a DFA above!

# Combining finite automata

A formal method for combining FAs into a single DFA is used in compilers.

- However it is out of syllabus.
- In this course we will look at an informal (intuitive) method, which you should know how to do.

An example of combining the multiple prior FAs into a single DFA is on the next slide.

# Combining finite automata: Example

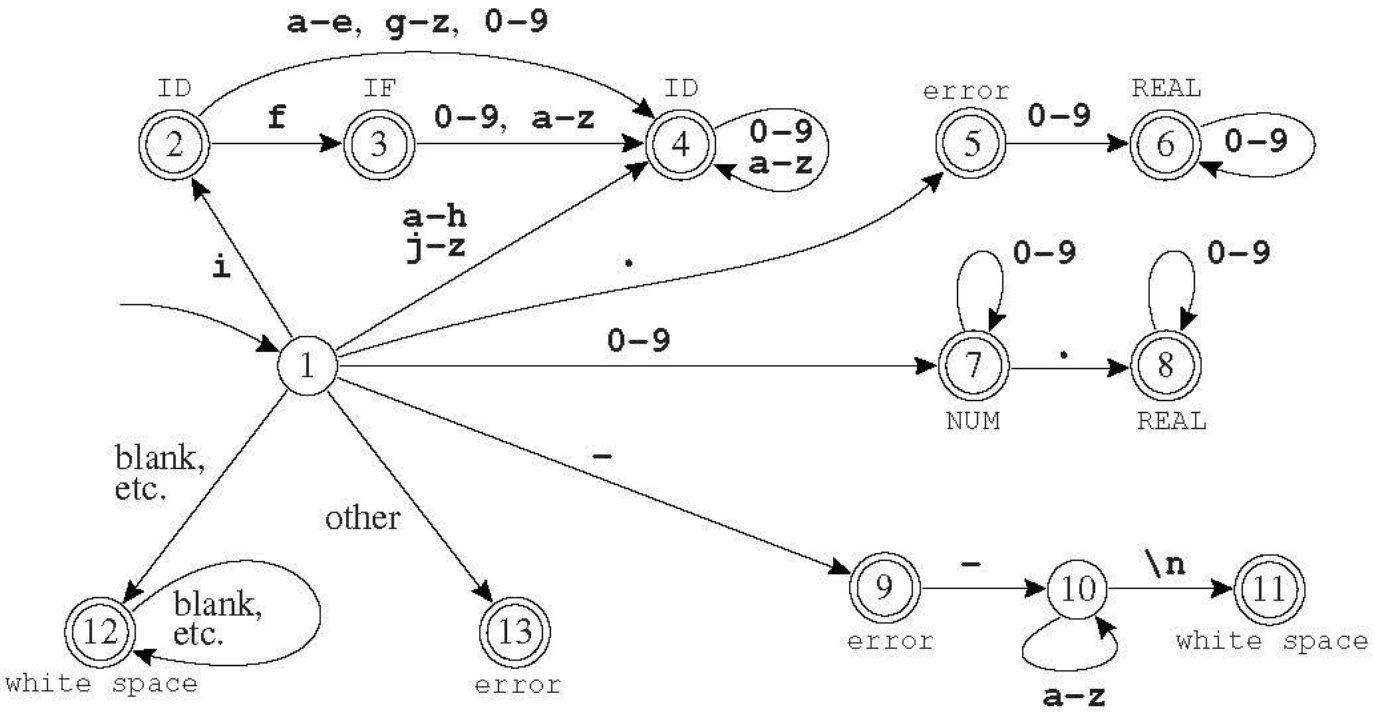


**FIGURE 2.4.** Combined finite automaton.

In this kind of DFA, if a character is missing from the out-edges of a state, and it is a final state, then we end the token and move to start state.

If it is NOT a final state, then generate an error, and move to the start state.

Rest of chapter 2 is out of syllabus. It deals with formal methods of combining regular expressions into DFAs.

# There are existing softwares for Lexical analysis

E.g., **`lex`** on Unix-based system