Implementation of Lexical Analyzer through Transition Diagrams

BITS Pilani
Pilani Campus

Dr. Shashank Gupta
Assistant Professor
Department of Computer Science and Information Systems

Challenges in Regular Definitions

Although, regular expressions are sufficient enough to describe the programming language tokens.

- However, they get failed most of the time to identify the appropriate token.
- They may also pass the invalid tokens to the subsequent translation phases of compiler.

Hence, regular expressions are just language specifications.

- However, implementation challenges needs to be handled using some different principles.

Challenges in Regular Definitions

Consider any string str and the corresponding regular expression r

- Does str belongs to language produced by L (r)?

Response in the form of yes/no is not sufficient here, since, the goal is to partition the input stream into precise tokens.

- **Hence, tokenization is an implementation issue.**

Development Steps for Tokenization

1. Construct regular expressions for lexemes of each token

For example

$$Digit \rightarrow 0|1|2|----|9$$

$$Digits \rightarrow Digit^{+}$$

$$Fraction \rightarrow '.'Digits|\in$$

$$Exponent \rightarrow (E\,(+|-|\in)\,Digits)|\in$$

$$Number \rightarrow Digits\;Fraction\;Exponent$$

Steps for Tokenization

2. Construct R matching all lexemes of tokens

$R = R_1 + R_2 + R_3 + R_4 + R_5 +$ ------ (in some well-defined precedence order)

3. Consider input stream be $s_1 s_2$------s_n

for $1 \leq i \leq n$, verify whether s_1---$s_i \in L(R)$.

4. s_1---$s_i \in L(R) \Rightarrow s_1$---$s_i \in L(R_x)$ for some x.

smallest such x is a token of class of s_1---s_i

5. Discard the tokenized input s_1---s_i from input stream and goto step 3.

Tokenization

The procedure must give preference to the tokens specified earlier using regular expressions.

If $s_1 \text{---} s_i \in L(R)$ and $s_1 \text{---} s_j \in L(R)$

- Select the longest string in L(R) according to the principle of longest match (also known as **Maximal Munch**)

Tokenization

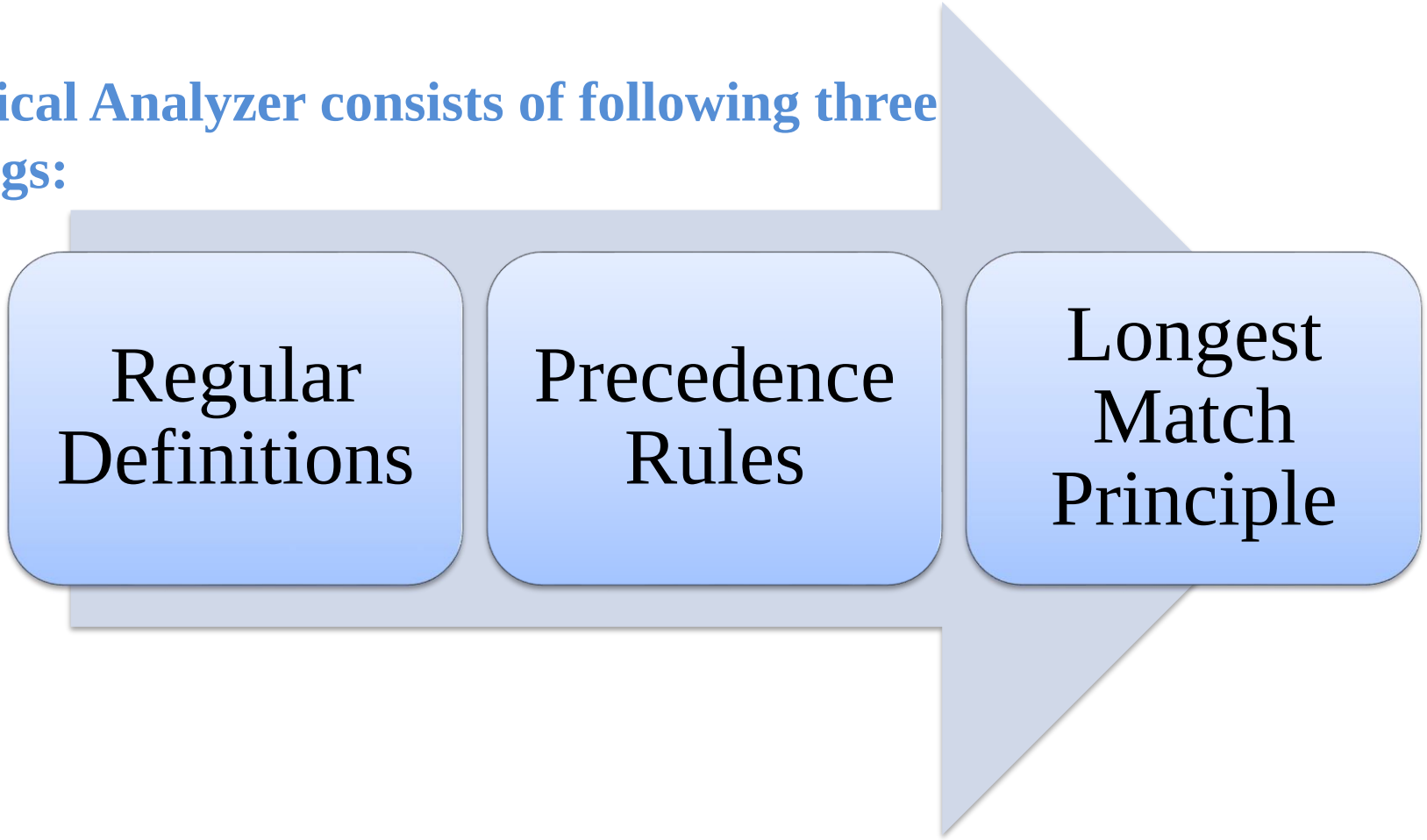
`iff=0` would be tokenized either as

- `if  f = 0` or `iff = 0`

On the other hand, tokens needs to be prioritized in certain order for resolving the conflicts.

Lexical Analyzer

Lexical Analyzer consists of following three things:



Specification and Recognition of Tokens

Regular expressions are very popular for specifications of tokens.

Transition diagrams are used to implement regular definitions and to recognize tokens.

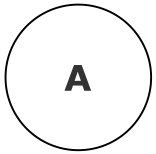
Transition Diagrams

It is a way of manually implementation of tokenization.

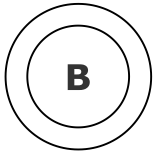
Hence, regular definitions are just declarative specifications. Finite automata is systematic and efficient way of implementation.

Pictorial Notations

State

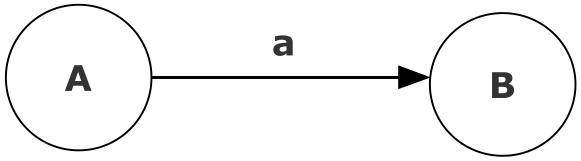


Final State



Transition

Transition from State A to B on an I/P a



Implementation of LA through Transition Diagrams

- Consider the following language specification in the form of regular definition as follows:

$$Letter \rightarrow A|B|C|----|Z|a|b|c|----|z$$

$$Digit \rightarrow 0|1|2|----|9$$

$$Identifier \rightarrow Letter\ (Letter|Digit)^*$$

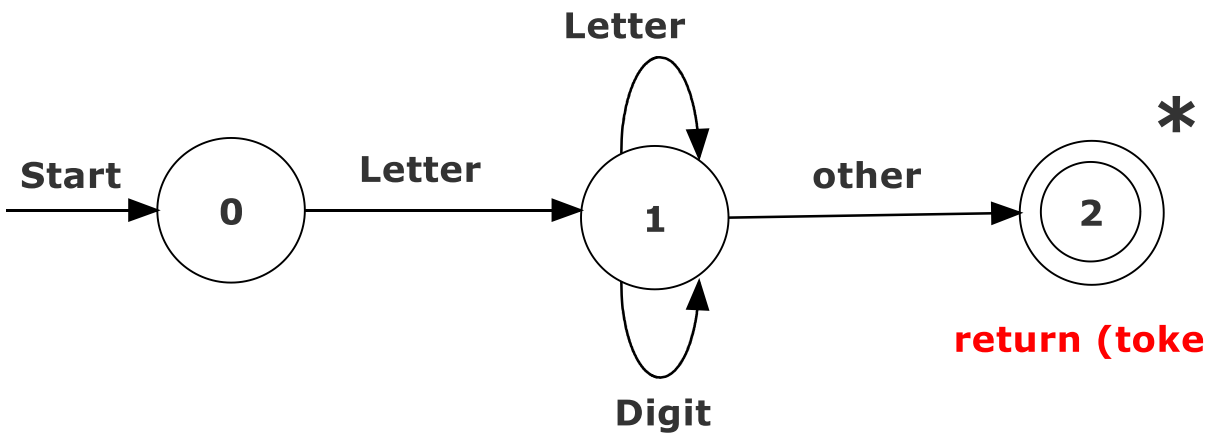
Design a lexical analyzer that will return pair <token, lexeme> to the syntax analyzer

Transition Diagram for Identifier

$$Letter \rightarrow A|B|C|-----|Z|a|b|c|-----|z$$

$$Digit \rightarrow 0|1|2|-----|9$$

$$Identifier \rightarrow Letter\ (Letter|Digit)^*$$



return (token_id, pointer to symbol table)

* Indicates Retraction State

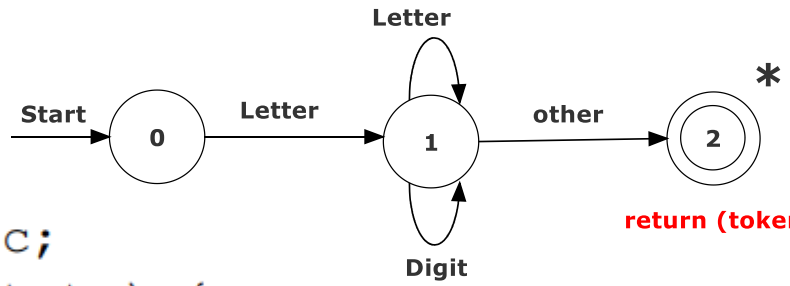
Transition Diagrams

Transitions may be labelled with a symbol, group of symbols or regular definitions.

Few states may be treated as **Retracting States** that indicates that the lexeme does not include the symbol that brought us to the accepting state.

All states has an action attached to it, which is executed when the state is reached. Usually, such actions returns a token and its attribute value.

Implementation of Transition Diagram for Identifiers



return (token_id, pointer to symbol table)

```
TOKEN gettoken() {
   TOKEN mytoken; char c;
   while(1) { switch (state) {
     /* recognize reserved words and identifiers */
       case 0: c = nextchar(); if (letter(c))
               state = 1; else state = failure();
               break;
       case 1: c = nextchar();
               if (letter(c) || digit(c))
               state = 1; else state = 2; break;
       case 2: retract(1);
               mytoken.token = search_token();
               if (mytoken.token == IDENTIFIER)
               mytoken.value = get_id_string();
               return(mytoken);
```

Transition Diagram for Relational Operators

$$relop \rightarrow >> \mid >=$$