CSL302: Compiler Design

Lexical Analysis

Vishwesh Jatala

Assistant Professor

Department of CSE

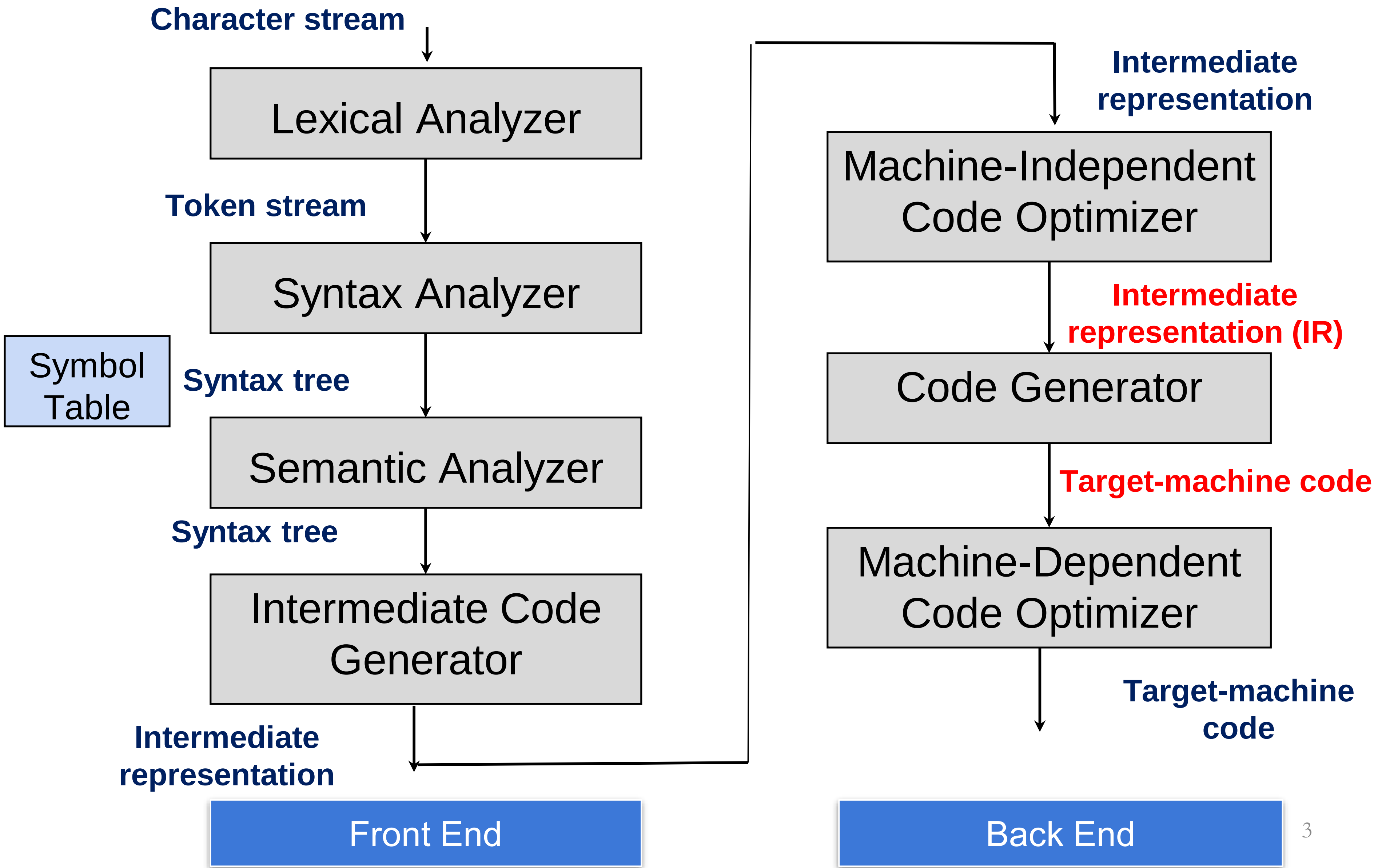
Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

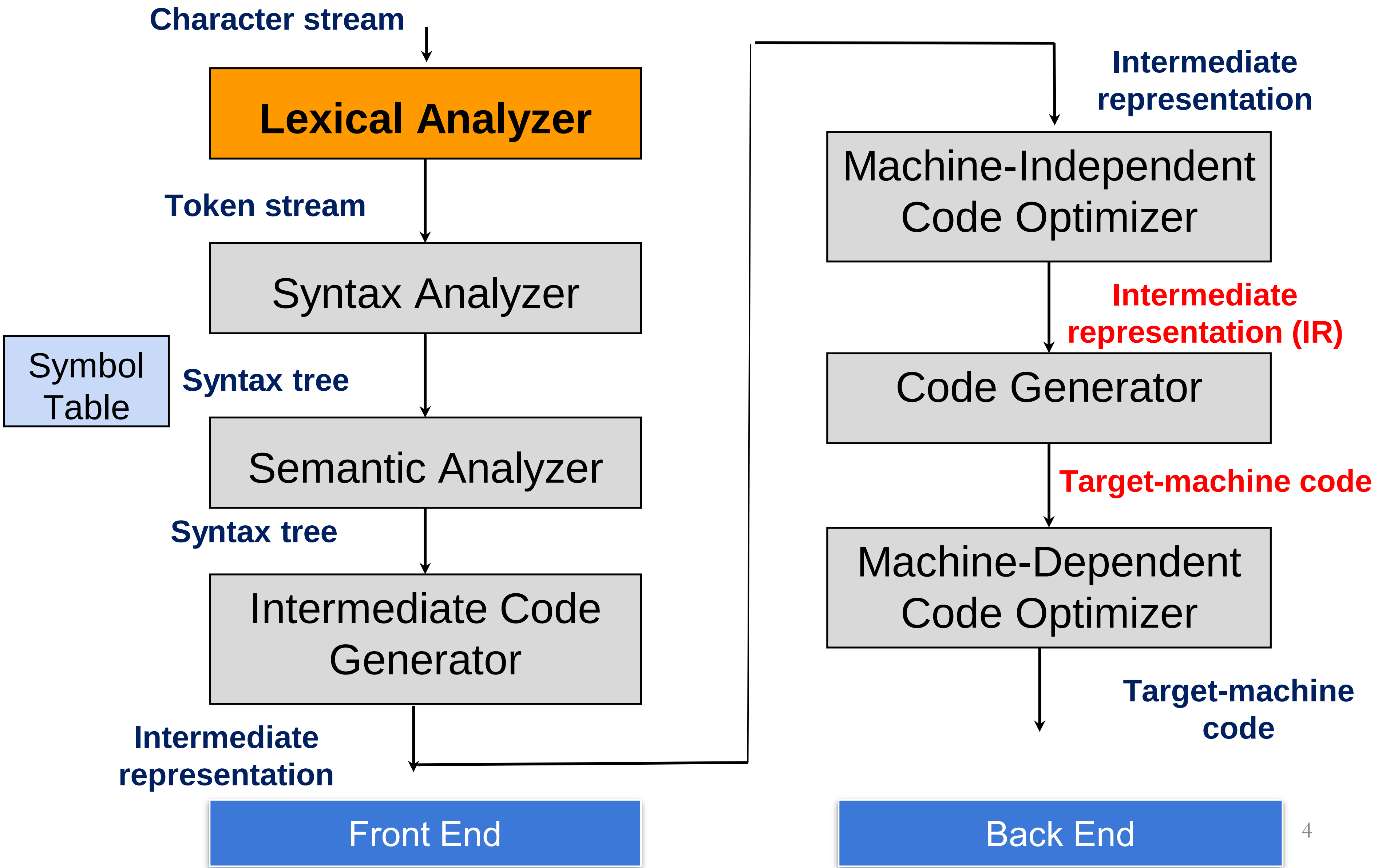
Acknowledgement

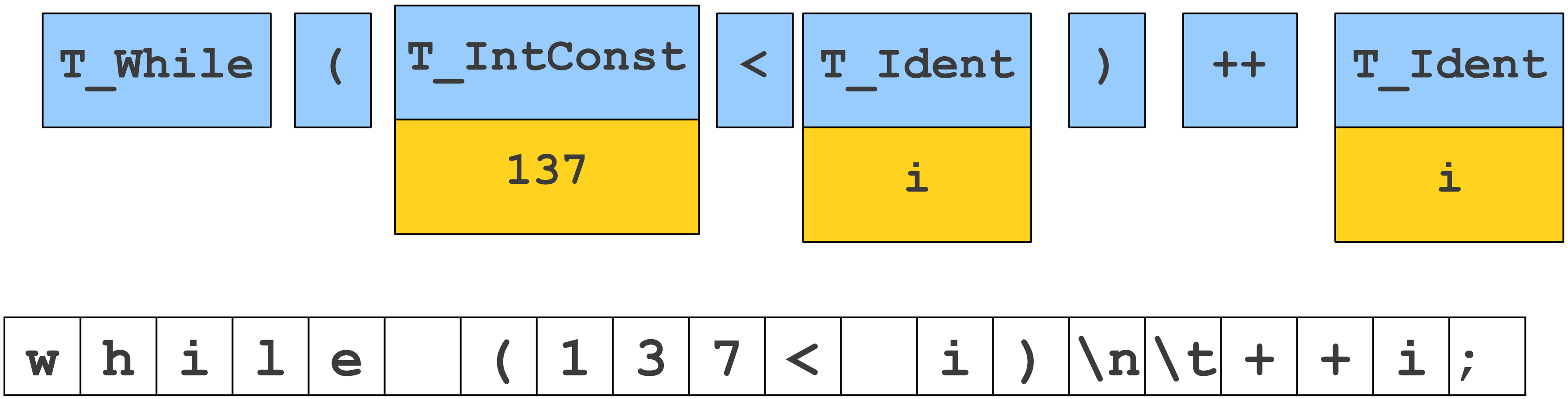
- References for today's slides
  - *Stanford University:*
    - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*
  - *Lecture notes of Prof. Amey Karkare (IIT Kanpur) and Late Prof. Sanjeev K Aggarwal (IIT Kanpur)*
  - *Suggested textbook for the course*

Compiler Design



Compiler Design






```
while (137< i)
    ++i;
```

Goals of Lexical Analysis

- Convert from physical description of a program into sequence of of **tokens**.
  - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
- Each token is associated with a **lexeme**.
  - The actual text of the token: “137,” “int,” etc.
- Each token may have optional **attributes**.
  - Extra information derived from the text – perhaps a numeric value.
- The token sequence will be used in the parser to recover the program structure.

Choosing Tokens

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k)
    {  myArray[k]++;
}
```

What Tokens are Useful Here?

```
for (int k = 0; k < myArray[5]; ++k)
    {  myArray[k]++;
}
```

```
for          {
int          }
=            ;
(            <
)            [
++           ]

Identifier
IntegerConstant
```

Choosing Good Tokens

- Very much dependent on the language.
- Typically:
  - Give keywords their own tokens.
  - Give different punctuation symbols their own tokens.
  - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
  - Discard irrelevant information (whitespace, comments)

Tokens

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Associating Lexemes with Tokens

Lexemes and Tokens

- Tokens give a way to categorize lexemes by what information they provide.
- Some tokens might be associated with only a single lexeme:
  - Tokens for keywords like **`if`** and **`while`** probably only match those lexemes exactly.
- Some tokens might be associated with lots of different lexemes:
  - All variable names, all possible numbers, all possible strings, etc.

Sets of Lexemes

- Idea: Associate a set of lexemes with each token.
- We might associate the "number" token with the set { **`0`**, **`1`**, **`2`**, …, **`10`**, **`11`**, **`12`**, … }
- We might associate the "string" token with the set { **`""`**, **`"a"`**, **`"b"`**, **`"c"`**, … }
- We might associate the token for the keyword **`while`** with the set { **`while`** }.

How to describe tokens?

- Potentially infinite lexemes
- Programming language tokens can be described by regular languages
- Regular languages
  - Are easy to understand
  - There is a well understood and useful theory
  - They have efficient implementation
- Regular languages have been discussed in great detail in the “Theory of Computation” course

How to specify tokens

- Regular definitions
  - Let r_i be a regular expression and d_i be a distinct name
  - Regular definition is a sequence of definitions of the form

    $d_1 \rightarrow r_1$

    $d_2 \rightarrow r_2$

    …..

    $d_n \rightarrow r_n$
  - Where each r_i is a regular expression

Examples

- Identifier

  letter → a| b| …|z| A| B| …| Z

  digit → 0| 1| …| 9

  identifier → letter(letter|digit)*

Examples

- Email address
  cse@iitbhilai.ac.in

Write regular expression!

Examples

- Email address
  cse@iitbhilai.ac.in
- Σ = letter U {@, . }
- letter $\rightarrow$ a| b| ...| z| A| B| ...| Z
- name $\rightarrow$ letter$^+$
- address $\rightarrow$ name '@' name '.' name '.' name

Regular expressions in specifications

- Regular expressions describe many useful languages
- Regular expressions are only specifications; implementation is still required
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers
- Goal: Partition the input into tokens

1. Write a regular expression for lexemes of each token
   - number -> $digit^+$
   - identifier -> $letter(letter|digit)^+$
2. Construct R matching all lexemes of all tokens
   - R = R1 + R2 + R3 + ...
3. Let input be $x_1 \ldots x_n$
   - for $1 \leq i \leq n$ check $x_1 \ldots x_i \in L(R)$
4. $x_1 \ldots x_i \in L(R)$ $x_1 \ldots x_i \in L(Rj)$ for some j
   - smallest such j is token class of $x_1 \ldots x_i$
5. Remove $x_1 \ldots x_i$ from input; go to (3)

- The algorithm gives priority to tokens listed earlier
  - Treats "if" as keyword and not identifier
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$
  - $x_1 \ldots x_j \in L(R)$
  - Pick up the longest possible string in L(R)
  - The principle of "maximal munch"
  - do/double
- Regular expressions provide a concise and useful notation for string patterns

Examples

IF → if

ELSE → else

DO → do

DOUBLE → double

identifier → letter(letter|digit)*

letter → a| b| …|z| A| B| …| Z

digit → 0| 1| …| 9

COMP_OP → >|<|>=|<=|==|!=

ARITH_OP → +|-|*|/

Recognizing Regular Expressions

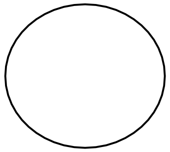
Transition Diagrams

- Regular expressions are declarative specifications
- Transition diagram is an implementation
- A transition diagram consists of
  - An input alphabet belonging to Σ
  - A set of states S
  - A set of transitions $\text{state}_i \rightarrow^{input} \text{state}_j$
  - A set of final states F
  - A start state n
- Transition $s1 \rightarrow^{a} s2$ is read:

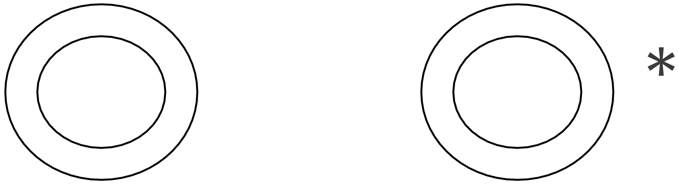
  in state s1 on input a go to state s2
- If end of input is reached in a final state then accept
- Otherwise, reject

Pictorial Notation

- A state

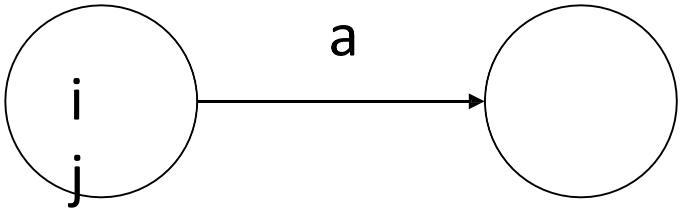


- A final state



- Transition

- Transition from state i to state j on an input a



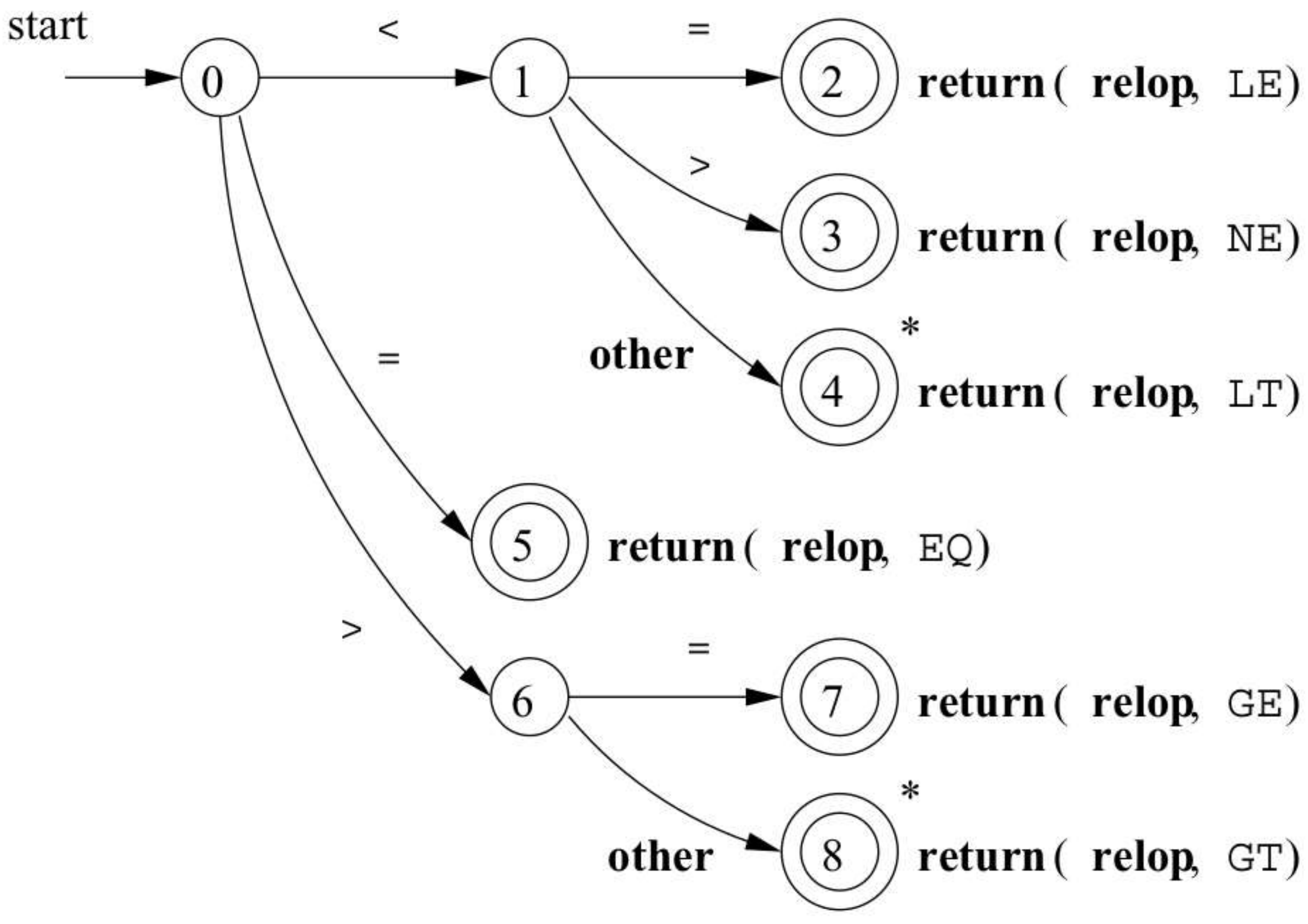
Transition Diagram for relop

- Draw the transition diagram for recognizing the relation operators (<=, >=, <>)

Transition Diagram for *relop*

- Draw the transition diagram for recognizing the relation operators (<, <=, <>, =, >, >=)

Transition Diagram for *relop*

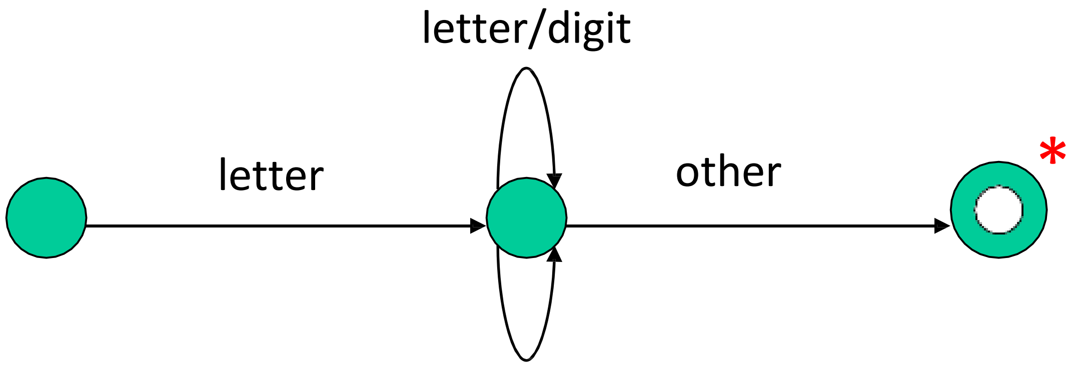


How to Recognize Tokens

- Consider

  id -> letter(letter|digit)*

Transition diagram for identifier



Questions?