Compiler Design – 1 – Languages and Grammars

Languages

- A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols.
- A grammar can be regarded as a device that enumerates the sentences of a language."
- Two broad categories of formal languages: generative and analytic
- A generative grammar formalizes an algorithm that generates valid strings in a language
- An analytic grammar is a set of rules to reduce an input string to a boolean result that indicates the validity of the string in the given language.

A generative grammar describes how to write a language, and an analytic grammar describes how to read it (a parser).

Formal Grammar

A formal grammar of this type consists of:

- a finite set of terminal symbols
- a finite set of nonterminal symbols
- a finite set of production rules with a left and a right-hand side consisting of a sequence of these symbols
- a start symbol

Formal Grammar

A formal grammar is a quad-tuple $G = (N, \Sigma, P, S)$ where

N is a finite set of non-terminals

Σ is a finite set of terminals and is disjoint from N

P is a finite set of production rules of the form

$$w \in (N \cup \Sigma)^* \rightarrow w \in (N \cup \Sigma)^*$$

$S \in N$ is the start symbol

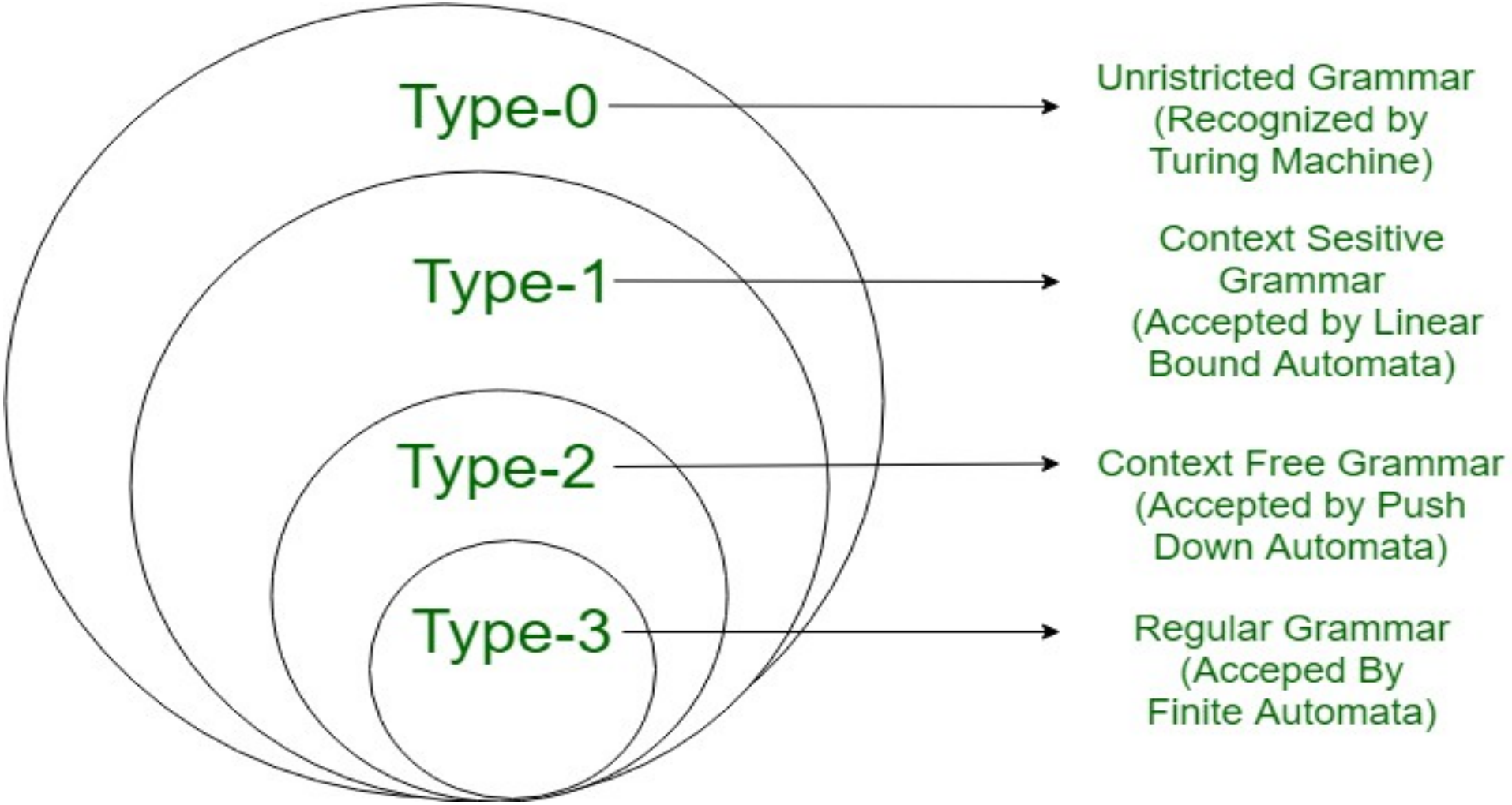
Chomsky argues that each sentence in a language has two levels of representation: deep structure and surface structure

Deep structure is a direct representation of the semantics underlying the sentence

Surface structure is the syntactical representation

Deep structures are mapped onto surface structures via transformations

Chomsky hierarchy of languages



Classes of Grammar: Chomsky Hierarchy

- **Type-0 grammars (unrestricted grammars) include all formal grammars.**

They generate exactly all languages that can be recognized by a Turing machine

- **Type-1 grammars (context-sensitive grammars)**

generate the context-sensitive languages.

These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$

*(*A *a nonterminal and* α, β *and* γ *strings of terminals and nonterminals)*

The strings α *and* β *may be empty, but* γ *must be nonempty*

The rule $S \rightarrow \varepsilon$ *is allowed if* S *does not appear on the right side of any rule*

The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automaton (a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.)

Classes of Grammar: Chomsky Hierarchy

- **Type-2 grammars (context-free grammars) generate the context-free languages.**

  These are defined by rules of the form $A \to \gamma$

  A *is a nonterminal and* γ *a string of terminals and nonterminals*

  These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton

  Context-free languages are the theoretical basis for the syntax of most programming languages

Classes of Grammar: Chomsky Hierarchy

- **Type-3 grammars (regular grammars) generate the regular languages.**

  Such a grammar restricts its rules to a single non terminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single non terminal

  A grammar is regular if it has rules of the form → A -> a or A -> aB or A -> ε

  The rule S → ε is also allowed here if S does not appear on the right side of any rule

  These languages are exactly all languages that can be decided by a finite state automaton

  Additionally, this family of formal languages can be obtained by regular expressions

  Regular languages are commonly used to define search patterns and the lexical structure of programming languages

Compiler

a program that converts high level language instructions into a machine-code or lower-level form so that they can be read and executed by a computer.

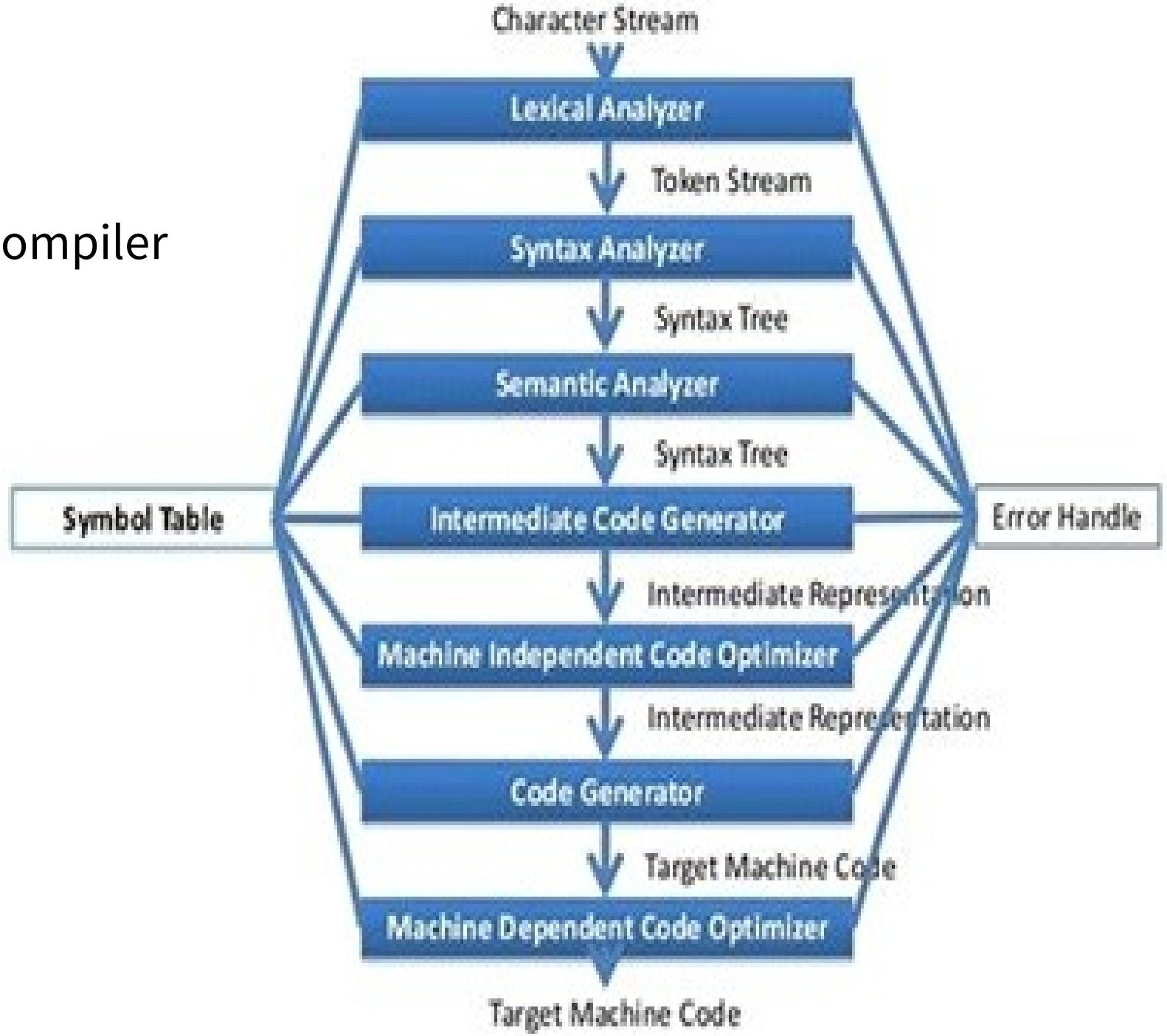
Why learn Compiler Design?

a) Compilers are important tools for programmers and computer scientists.

b) The techniques used for constructing a compiler are useful for other purposes as well.

c) There is a good chance that a programmer or computer scientist will need to write a compiler or interpreter for a domain-specific language.

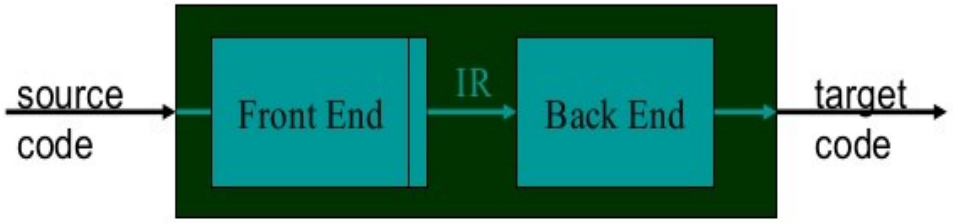
Phases of Compiler



Single Pass Compiler

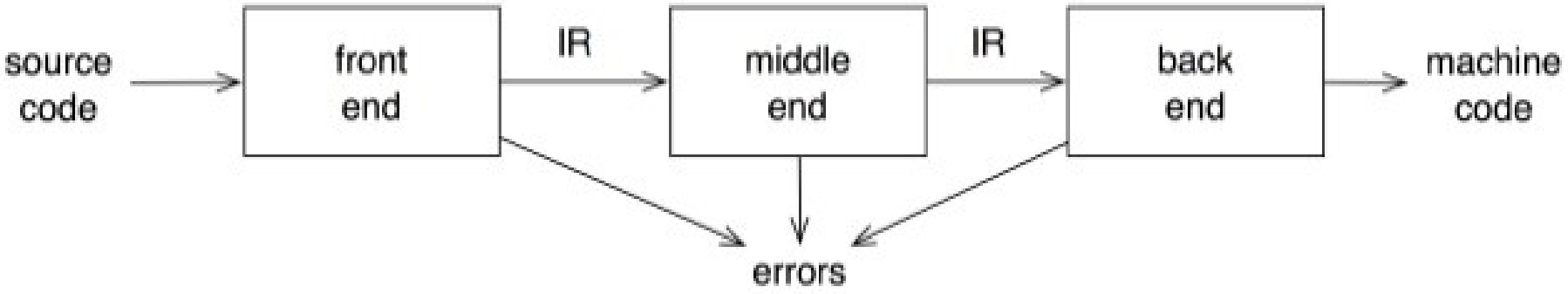
- Source code directly transforms into machine code.
  - For example Pascal

Two Pass Compiler



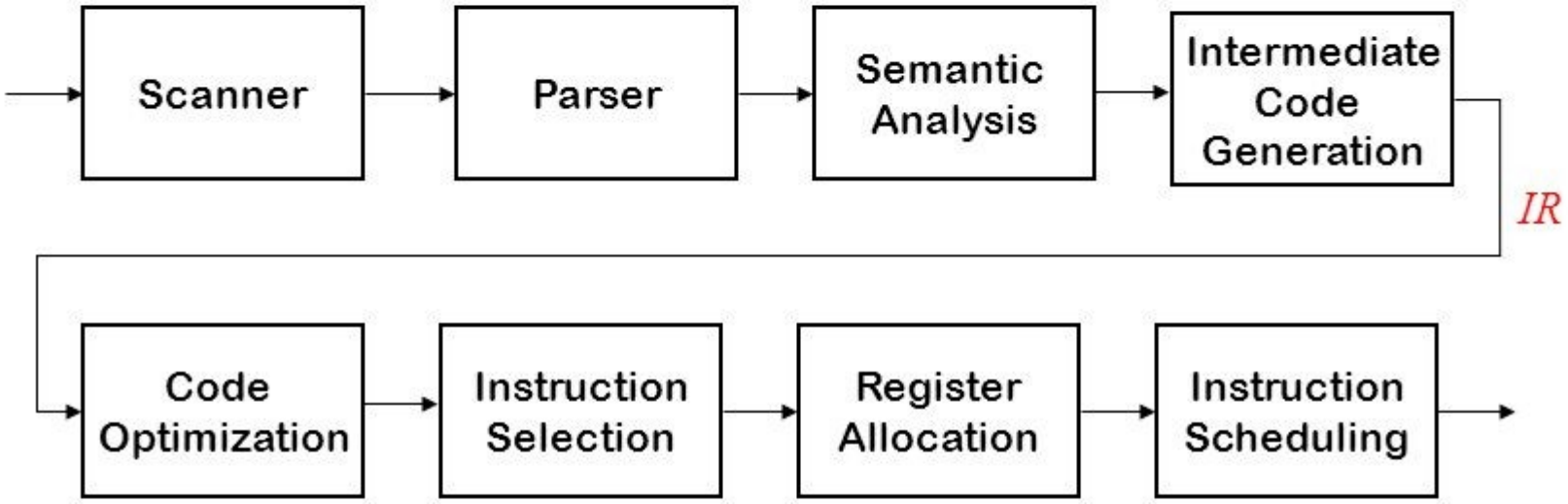
- intermediate representation (IR)
- front end maps legal code into IR
- back end maps IR onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes ⇒ better code

Multipass compiler



- analyzes and changes IR
- goal is to reduce runtime
- must preserve values

Structure of a Compiler



- Front end of a compiler is efficient and can be automated
- Back end is generally hard to automate and finding the optimum solution requires exponential time
- Intermediate code generation can effect the performance of the back end

Lexical Analyzer

Responsibilities

Scans the input program

Removes white spaces

Removes comments

Extracts and identifies tokens

Generates lexical errors

Passes tokens to parser

Terminologies

Token: classification for a common set of strings

Examples: Identifier, Integer, Float, Operator,....

Pattern: The rules that characterize the set of strings for a token

Examples: [0-9]+

Lexeme: Actual sequence of characters that matches a pattern and has a given Token class

Examples:

Identifier: sum, data, x

Integer: 345, 2, 0, 629,....

Lexical Errors

Error Handling is localized with respect to the input source code

For example: fi (a == f(x)) … generates no lexical error in C

In what situations do errors occur?

Prefix of remaining input does not match any defined token

Possible error recovery actions:

1. Deleting or Inserting Input Characters
2. Replacing or Transposing Characters
3. Or, skip over to next separator to *ignore* problem

Overall strategy

Use one character of *look-ahead*

Perform a case analysis

1)Based on lookahead char

2)Based on current lexeme

Outcome

1)If char can extend lexeme, continue.

2)If char cannot extend lexeme, find what the complete lexeme is (upto the previous character) and return its token. Put the lookahead back into the symbol stream.

Regular expressions

ε is a regular expression, L(ε) = {ε}

If a is a symbol in ∑, then a is a regular expression, L(a) = {a}

(r) | (s) is a regular expression denoting the language L(r) ∪ L(s)

(Strings from both languages)

(r)(s) is a regular expression denoting the language L(r)L(s)

(Strings constructed by concatenating a string from the first language with a string from the second language)

(r)* is a regular expression denoting (L(r))*

(Each string in the language is a concatenation of any number of strings in the language of s)

(r) is a regular expression denoting L(r)

Regular expressions

Examples:

letter → A | B | … | Z | a | b | … | Z | _

digit → 0 | 1 | … | 9

id → letter (letter | digit)*

Regular expressions

Extensions

One or more instances: (r)+

Zero of one instances: r?

Character classes: [abc]

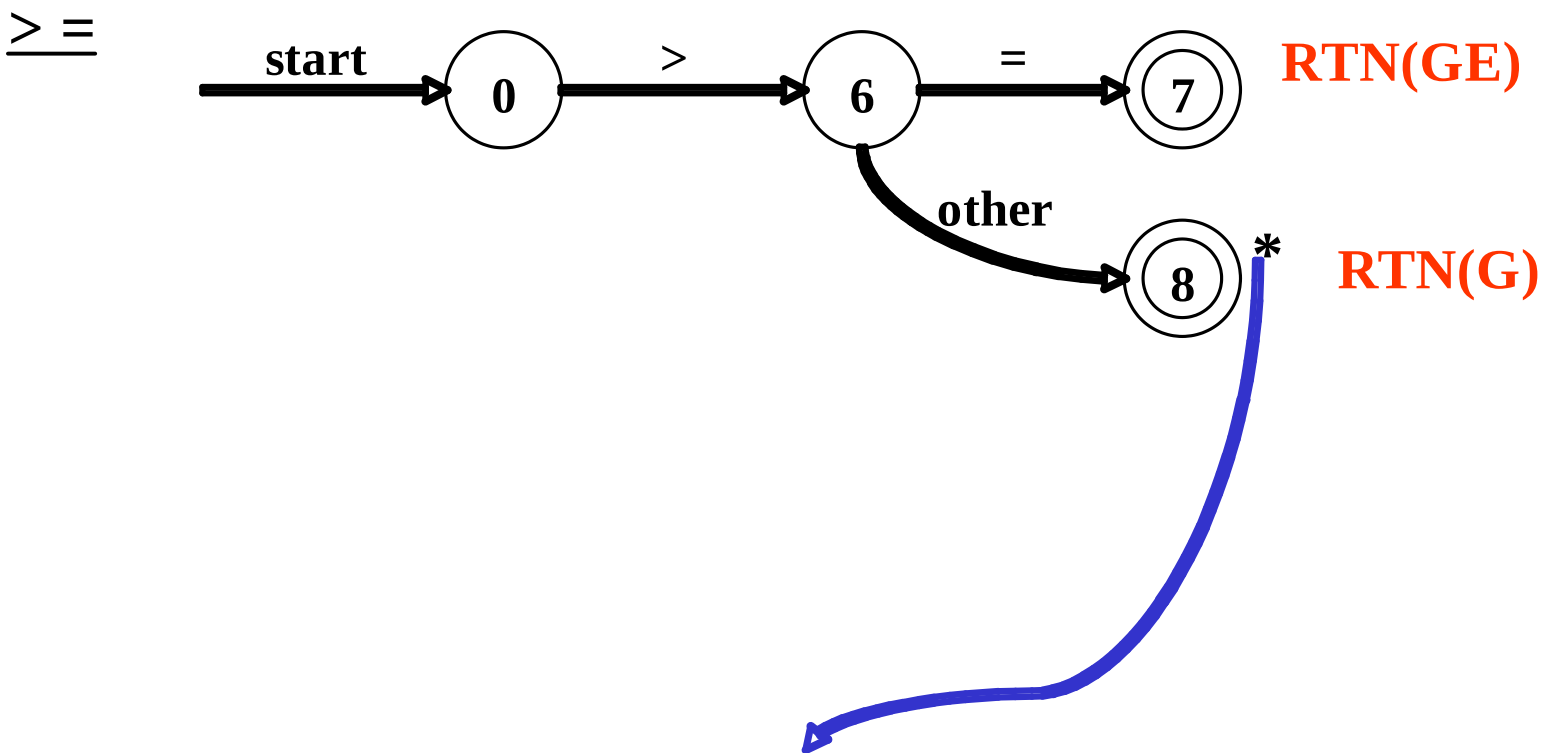
Example:

letter → [A-Za-z]

digit → [0-9]

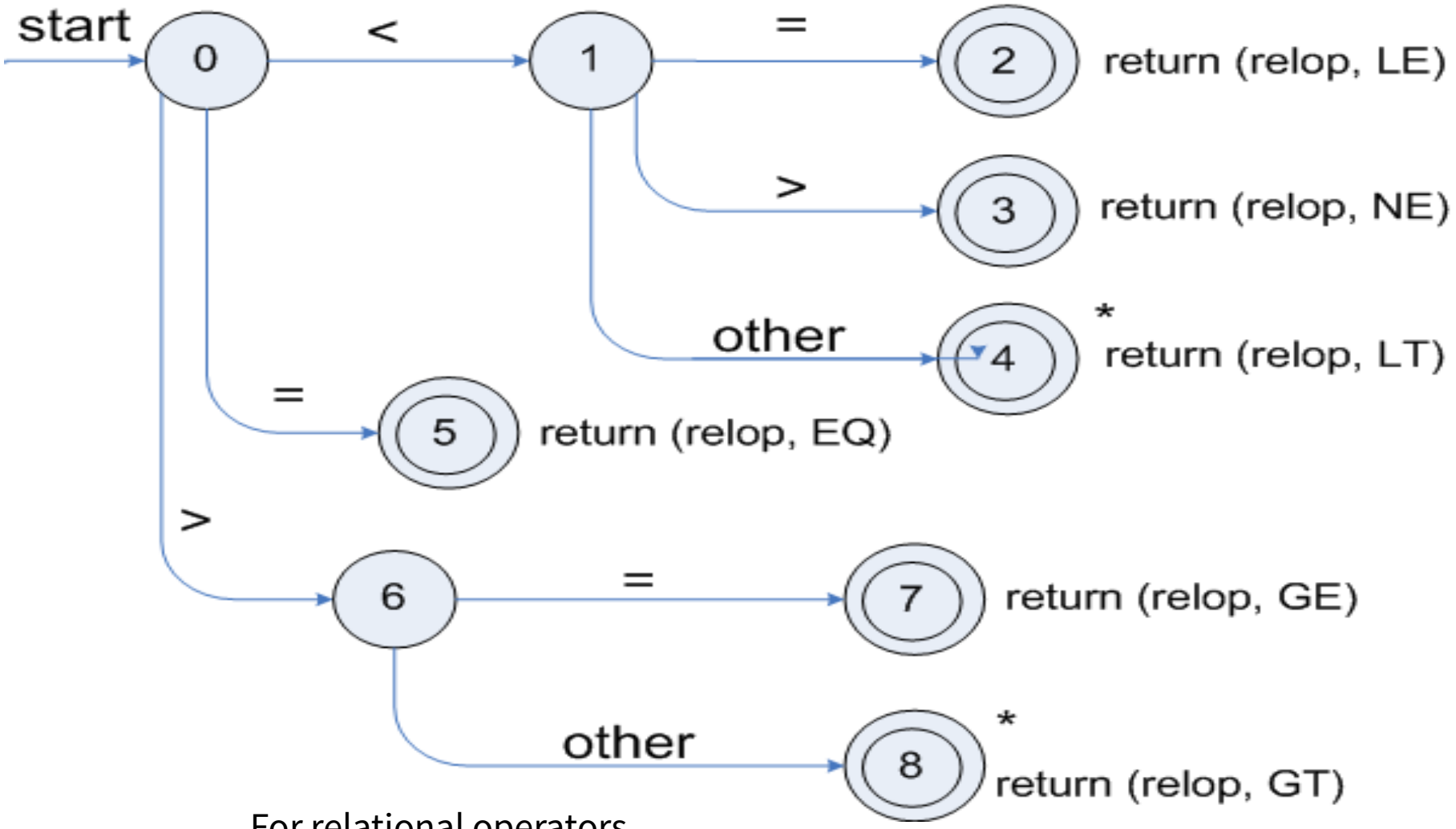
id → letter(letter|digit)*

Transition diagrams



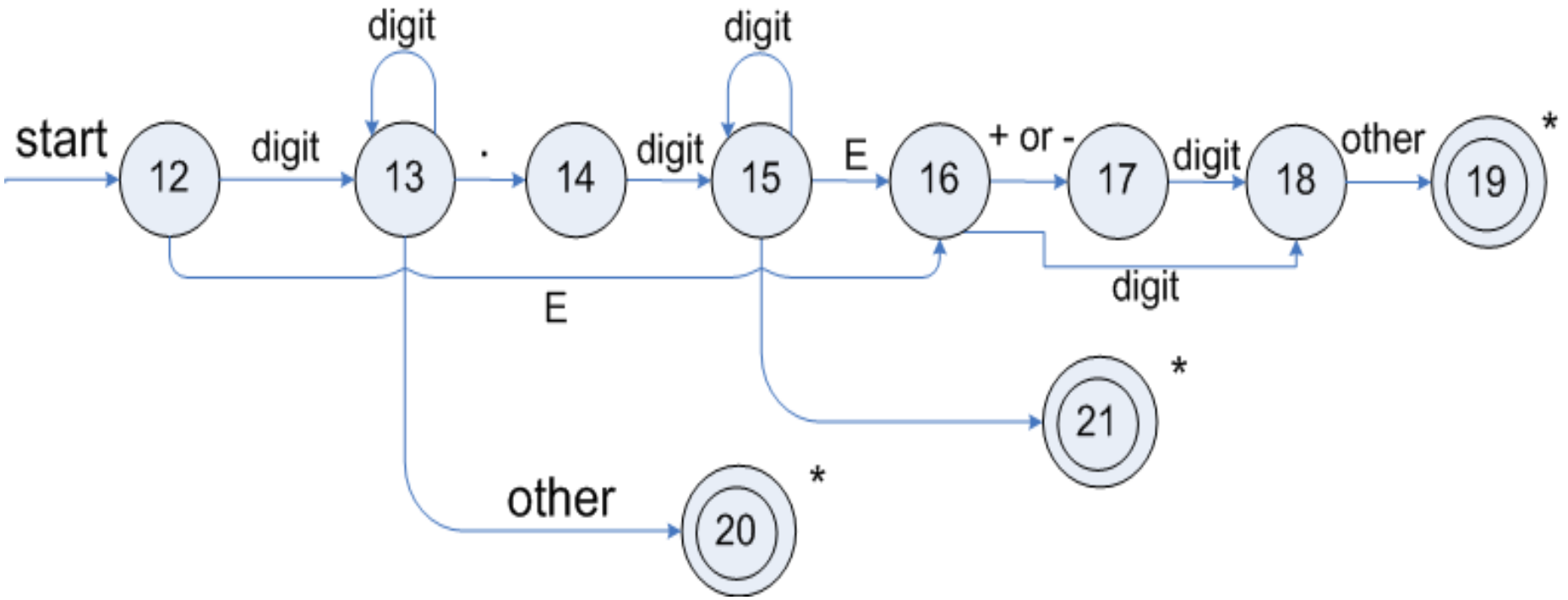
An additional character is read, that needs to be unread(return to input buffer

Transition diagrams



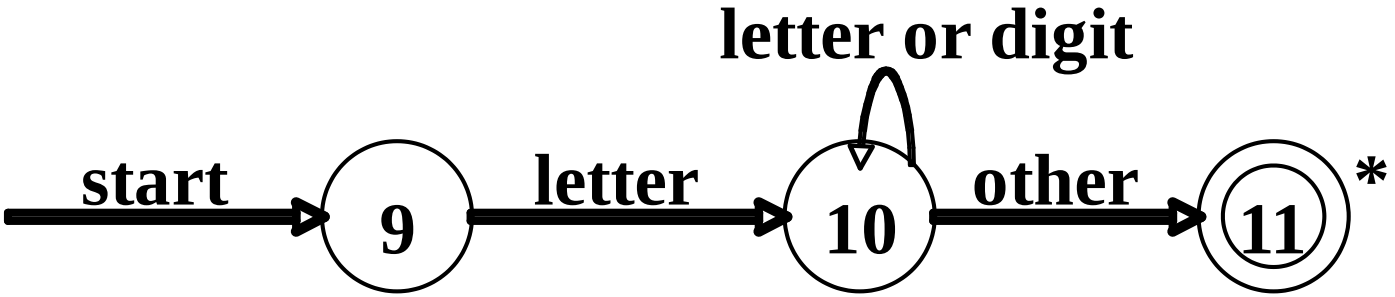
For relational operators

Transition diagrams



For unsigned numbers

Transition diagrams



For identifiers

What to do for keywords?

- Use the “Identifier” token
- After a match, lookup the keyword table
- If found, return a token for the matched keyword
- If not, return a token for the *true* identifier

Recognising Tokens

Regular expressions provide specifications for the tokens in a language

Finite automaton is used to recognise a token – implementation

A finite automaton consists of

An input alphabet Σ

A set of states S

A start state n

A set of accepting states $F \subseteq S$

A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automaton

Deterministic Finite Automata (DFA)

One transition per input per state

No ε-moves

Nondeterministic Finite Automata (NFA)

Can have multiple transitions for one input in a given state

Can have ε-moves

***Finite* automata have *finite* memory**

Need only to encode the current state

Conversion can be automated

NFAs and DFAs recognize the same set of languages (regular languages)

DFAs are easier to implement