NJIT

New Jersey's Science & Technology University

THE EDGE IN KNOWLEDGE

SYNTAX

A Cautionary Tale

- If you think you have complaints about complexity of a programming language…
- Watch this:
- https://youtu.be/a9xAKttWgP4

Implementing a Programming Language

- All language implementations must analyze source code, regardless of the specific implementation approach
- Nearly all syntax analysis is based on a formal description of the syntax of the source language

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, represented in BNF)

Using BNF Notation to Describe Syntax

- Provides a clear and concise syntax description
- The parser can often be based directly on the BNF
- Parsers based on BNF are easy to maintain

Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* - less complex approaches can be used for lexical analysis; separating them simplifies the parser
- *Efficiency* - separation allows optimization of the lexical analyzer
- *Portability* - parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a "front-end" for the parser
- Identifies substrings of the source program that belong together - *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*
  - `sum` is a lexeme; its token might be `IDENT`

Lexical Analysis (continued)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs a table-driven lexical analyzer from such a description
  - Design a state diagram that describes the tokens and write a program that implements the state diagram
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram

State Diagram Design

– A naïve state diagram would have a transition from every state on every character in the source language - such a diagram would be very large!

Lexical Analysis (continued)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent - use a digit class

- Note for C++ programmers: <cctype> provides character classes

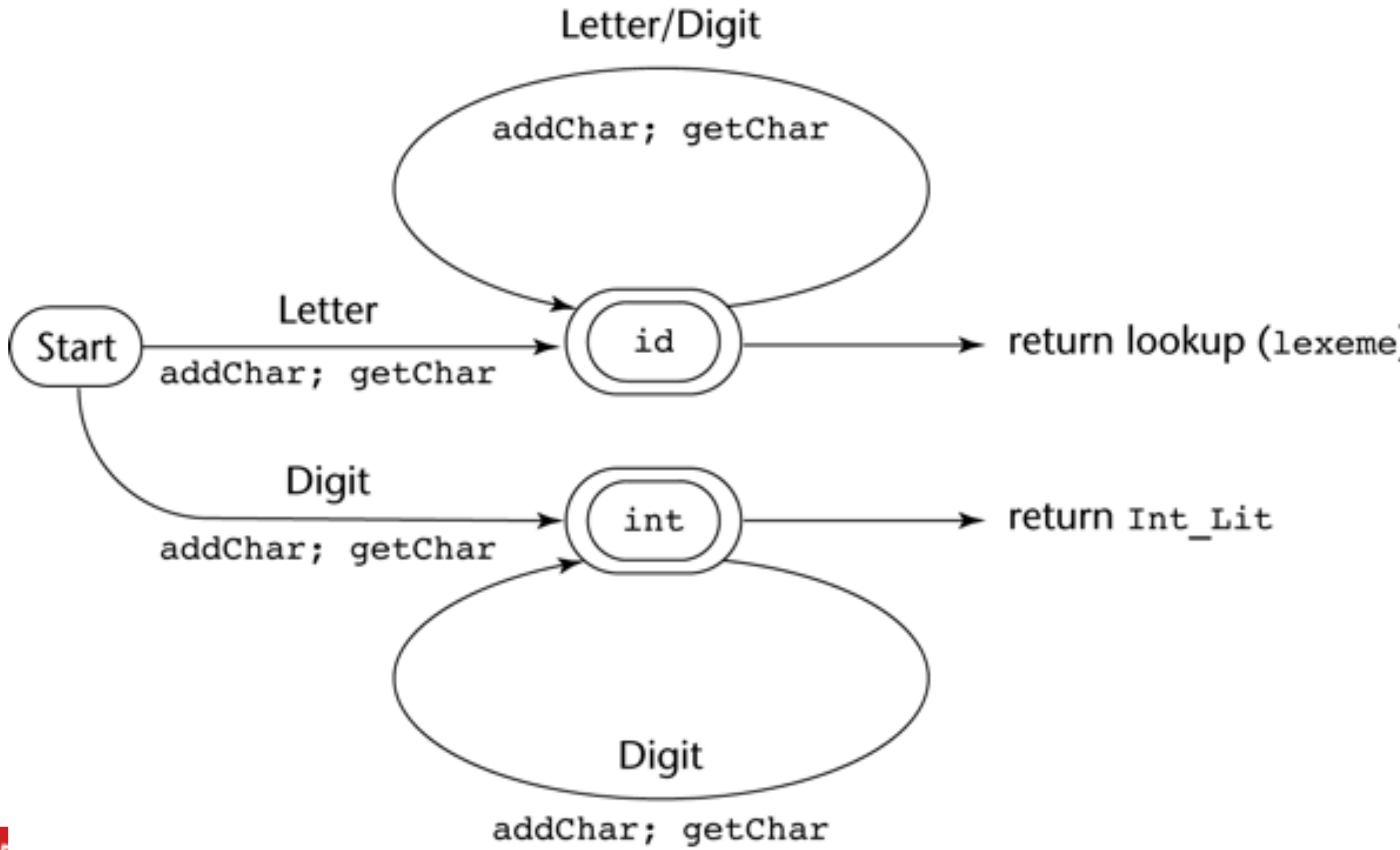
Lexical Analysis (continued)

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis (continued)

- For this example, assume these utility subprograms:
  - **`getChar`** - gets the next character of input, puts it in **`nextChar`**, determines its class and puts the class in **`charClass`**
  - **`addChar`** - puts the character from **`nextChar`** into the place the lexeme is being accumulated, **`lexeme`**
  - **`lookup`** - determines whether the string in **`lexeme`** is a reserved word (returns a code)

State Diagram



Regular Grammars

- A regular grammar is a simple scheme for using rules to represent strings to recognize
- Regular grammars are the simplest and least powerful of the grammars
- Regular grammars are useful in expressing and recognizing tokens

What Makes A Regular Grammar?

Set of

productions: *P*

terminal symbols: *T*

nonterminal symbols: *N*

A *production* has the form

$A \rightarrow \omega B$

$A \rightarrow \omega$

$\omega \in T^*, B \in N$

where $A, B \in N$ and $\omega \in T^*$

That is, there's only one nonterminal on the right hand side of the rule. T means "zero or more" terminals*

Example Regular Grammar for Integers

Integer → 0 Integer
Integer → 1 Integer
Integer → 2 Integer
Integer → 3 Integer
Integer → 4 Integer
Integer → 5 Integer
Integer → 6 Integer
Integer → 7 Integer
Integer → 8 Integer
Integer → 9 Integer
Integer → 0
Integer → 1
Integer → 2
Integer → 3
Integer → 4
Integer → 5
Integer → 6
Integer → 7
Integer → 8
Integer → 9

Simplify it

A more compact expression:

Integer → 0 Integer | 1 Integer | 2 Integer | 3 Integer | 4 Integer | 5 Integer | 6 Integer | 7 Integer | 8 Integer | 9 Integer | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Or

Integer → 0 Integer | 1 Integer | ... | 9 Integer |
0 | 1 | ... | 9

- This is a “Right Regular Grammar” because all nonterminal symbols on the right side of the production are the rightmost symbols on the right hand side
- It follows that there’s also left regular grammars

Patterns in strings

- Regular grammars can be used to recognize strings that match a particular pattern
- They can't recognize all patterns in general
- This: $\{ a^n b^n \mid n \geq 1 \}$
  - Is not a regular language and so can't be recognized with a regular grammar
- In other words, a regular grammar cannot balance paired items: ( ), { }, begin end

Regular Expressions

- A regular expression is a notation for expressing patterns of characters
- Regular expressions are all over CS
  - String finding in editors
  - Pattern expansion is built into command interpreters (saying "ls *.c" in UNIX is a form of this)
- Many languages have a regex library of some form
- Some languages with string types may have regular expression matching built in

Making Regular Expressions

- The sequence of characters in a regular expression are matched against a string
- A character in a regular expression is either a regular character, which must exactly match the character in the string, or a metacharacter, which stands for something else
  - Example: a dot ('.') in a regular expression is a metacharacter that means "matches any single character in the string"
- Escaping a metacharacter with a backslash changes the metacharacter to its non-metacharacter meaning
  - Example, \. (backslash dot) matches the character dot, NOT the metacharacter meaning of "any character"

<u>RegExpr</u>	<u>Meaning</u>
x	a character x
\x	an escaped character, e.g., \n
M | N	M or N
M N	M followed by N
M*	zero or more occurrences of M
M+	One or more occurrences of M
M?	Zero or one occurrence of M
[*characters*]	choose from the characters in []
[aeiou]	the set of vowels
[0-9]	the set of digits
. (that's a dot)	Any single character

You can parenthesize items for clarity

Example Regular Expressions for Tokens

[a-z_A-Z][0-9a-z_A-Z]*

an identifier

0[0-7]+

an octal constant

0x[0-9a-fA-F]+

a hex constant

[+-]?([0-9]*\.[0-9]+|[0-9]+\.[0-9]*)e[+-][0-9]+

a floating point number

Matching strings to regular expressions

- Matching can be automated
  - There are libraries that compile regular expressions and use them to match strings
- A tool known as lex (or flex) is designed to use regular expressions to automatically generate a lexical analyzer for a compiler/interpreter
- The matcher is a simple machine called a finite state machine or finite state automata

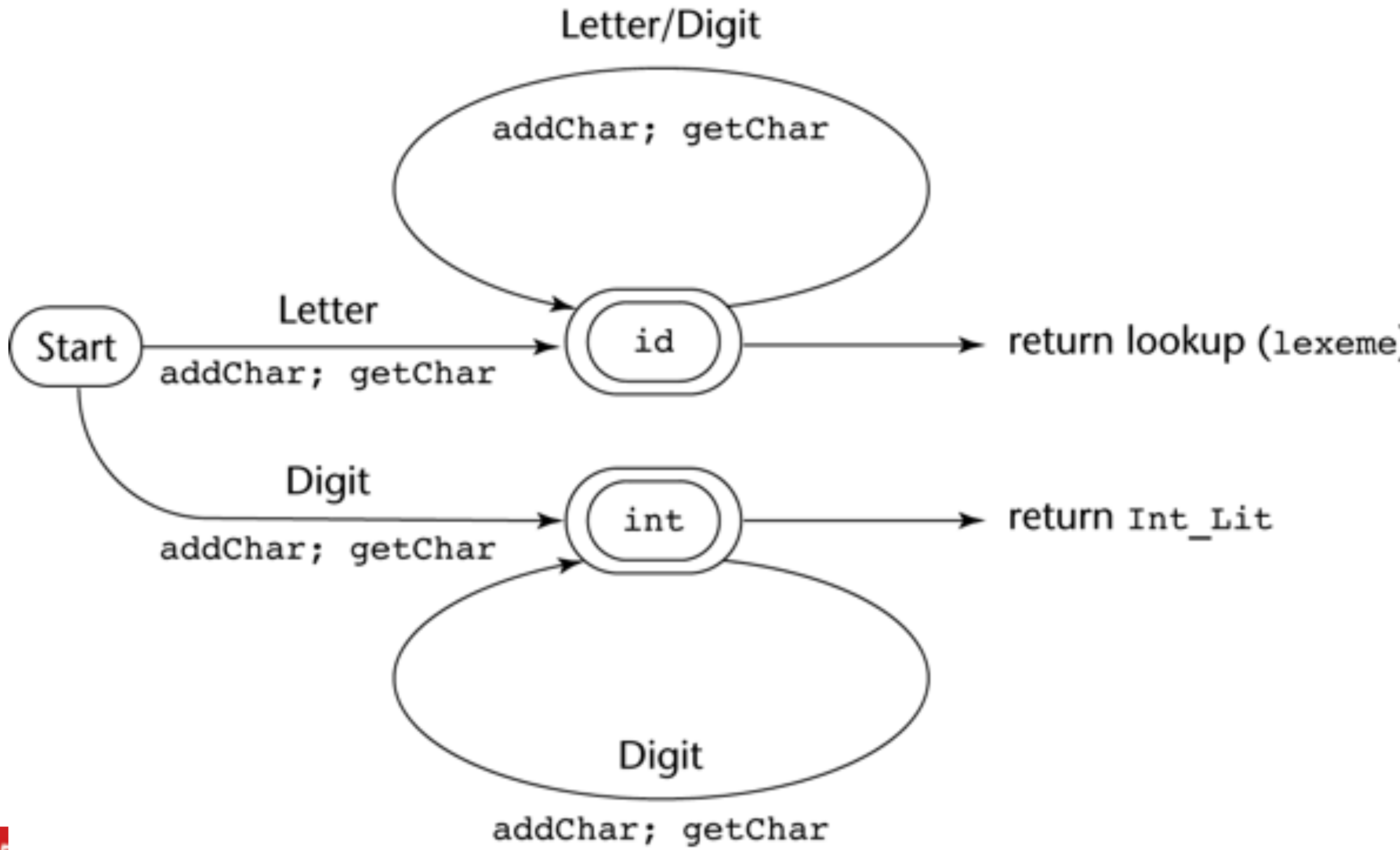
Finite State Automata

- Set of states:
  - A useful representation is as graph nodes
- Input alphabet + unique end symbol
- State transition function
  - Labelled (using alphabet) arcs in graph
- Unique start state
- A final state or an “accepting” state

Deterministic FSA

- A finite state automaton is *deterministic* if for each state and each input symbol, there is at most one outgoing arc from the state labeled with the input symbol.

State Diagram



NJIT

THE EDGE IN KNOWLEDGE