CS 314 Principles of Programming Languages

Lecture 2: Syntax Analysis

Zheng (Eddy) Zhang

Rutgers University

January 22, 2018

Announcement

- **First recitation starts this Wednesday**
- **Homework 1 will be release tomorrow**
- **My office hour:**

  Wednesday 5pm to 6pm, CoRE 310.

Syntax and Semantics of Prog. Languages

Syntax:

Describes what a legal program looks like

Semantics:

Describes what a correct (legal) program means

A formal language is a (possibly infinite) set of sentences (finite sequences of symbols) over a finite alphabet Σ of (terminal) symbols:

$L \subseteq \Sigma^*$

Examples:

- L = { identifiers of length 2} with Σ = {a, b, c}
- L = {strings of only 1s and only 0s}
- L = {strings starting with **$** and ending with #, and any combination of 0s and 1s in between }
- L = {all syntactically correct Java programs}

Syntax and Semantics: How does it work?

Syntactic representation of “values”

What do the following syntactic expressions have in common?

X

1010

A

$ ||||||||| #

3+ 19 − (2×6)

Syntax and Semantics: How does it work?

Syntactic representation of "values"

What do the following syntactic expressions have in common?

X
1010
A
$ ||||||||| #
3+ 19 − (2×6)

Answer: They are possible representations of the integer value "10" (written as a decimal number)

What is computation?

Possible answer: A (finite) sequence of syntactic manipulations of value representations ending in a "*normal form*" which is called the result. *Normal forms* cannot be manipulated any further.

Syntax and Semantics: How does it work?

Here is a "game" (**rewrite system**):

Input: Sequence of characters starting with **$** and ending with #, and any combination of **0**s and **1**s in between.

Rules: You may replace a character pattern **X** at any position within the character sequence on the left-hand-side by the pattern **Y** on the right-hand-side: **X⇒Y**:

Rule 1 **$1 ⇒ 1&**

Rule 2 **$0 ⇒ 0$**

Rule 3 **&1 ⇒ 1$**

Rule 4 **&0 ⇒ 0&**

Rule 5 **$# ⇒ →A**

Rule 6 **&# ⇒ →B**

Replace patterns using the rules as often as you can.
When you cannot replace a pattern any more, stop.

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

0 0 **$ #** is rewritten as **0 0** **→A** by rule 5

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

0 0 **$ #** is rewritten as **0 0** **→A** by rule 5

no more rules can be applied (**STOP**)

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

0 0 **$ #** is rewritten as **0 0** **→A** by rule 5

no more rules can be applied (**STOP**)

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

More examples:

$ 0 1 1 0 1 #

$ 1 0 1 0 0 #

$ 1 1 0 0 1 #

Questions:

- Can we get different "results" for the same input string?
- Does all this have a meaning (**semantics**), or are we just pushing symbols?

Syntax Without Semantics?

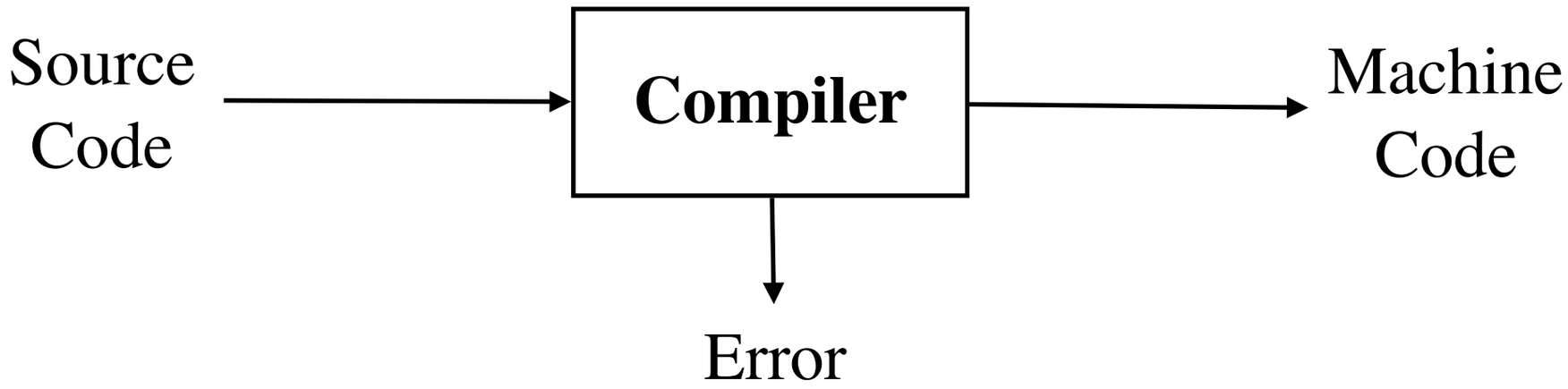
The green apple is colorless.

Syntax Without Semantics?

The green apple is colorless.

Syntactically correct, but semantically incorrect!

Review: Compilers



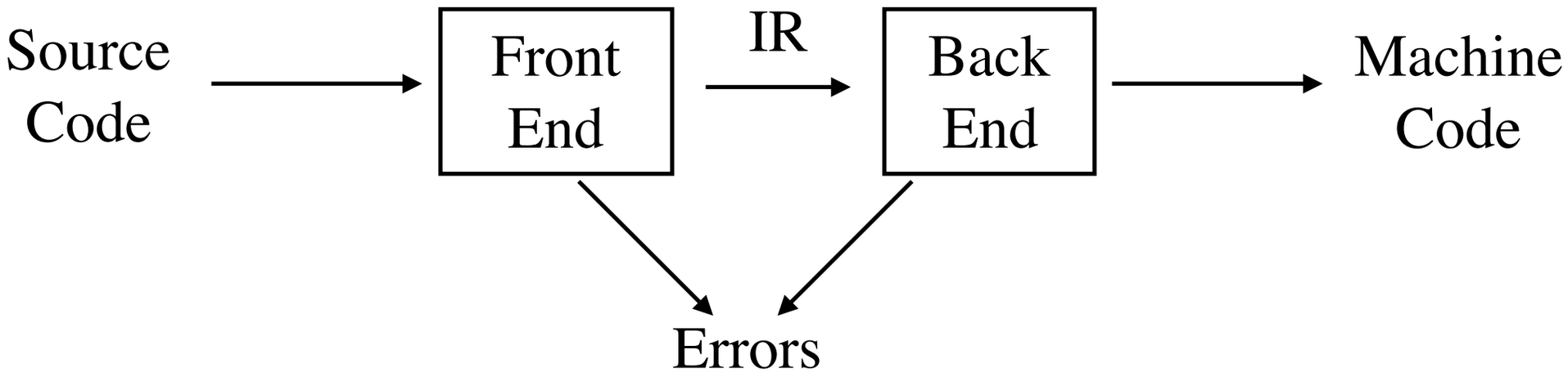
Implications:

- Recognize legal (and illegal) programs
- Generate correct code
- Manage storage of all variables and code
- Need format for object (or assembly) code

Big step up from assembler – higher level notations

Traditional Two-Pass Compilers

Pass: reading and writing entire program

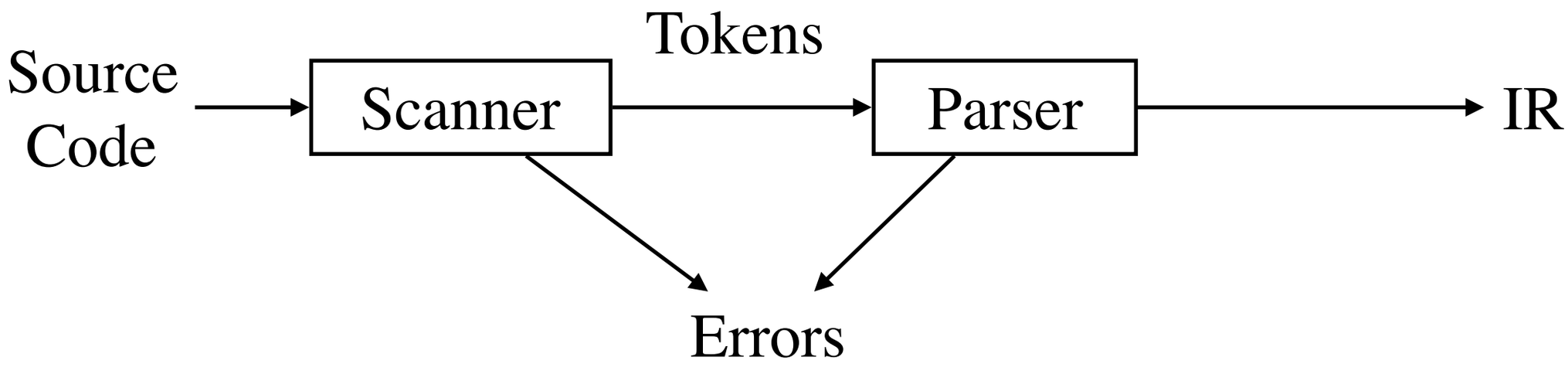


Implications:

- 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

Front end is O(n)
Back end is NP-Complete

Front End



Parser: syntax & semantic analyzer, IR code generator
(syntax-directed translator)

Front End Responsibilities:

- Recognize legal programs
- Report errors
- Produce IR
- Preliminary storage map
- Shape the code for the back end

Much of front end construction can be automated

Syntax and Semantics of Prog. Languages

The syntax of programming languages is often defined in two layers:
tokens and *sentences*

- *tokens - basic units of the language*

Question: How to spell a token (word)?

Answer: Regular expressions

- *sentences - legal combinations of tokens in the language*

Question: How to build correct sentences with tokens?

Answer: (context - free) grammars (CFG)

- E.g., Backus-Naur form (BNF) is a formalism used to express the syntax of programming languages.

Formalisms for Lexical and Syntactic Analysis

Two issues in ***Formal Languages:***

- *Language Specification* → formalism to describe what a valid program (sentence) looks like.

- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

Formalisms for Lexical and Syntactic Analysis

Two issues in ***Formal Languages:***

- *Language Specification* → formalism to describe what a valid program (sentence) looks like.

- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

1. Lexical Analysis: Converts source code into sequence of tokens.
   Regular grammar/expression to specify.
   Finite automata to recognize.

2. Syntax Analysis: Structures tokens into parse tree.
   Context-free grammars to specify.
   Push-down automata to recognize (will be covered in CS 415)

Lexical Analysis (Scott 2.1, 2.2)

Character sequence

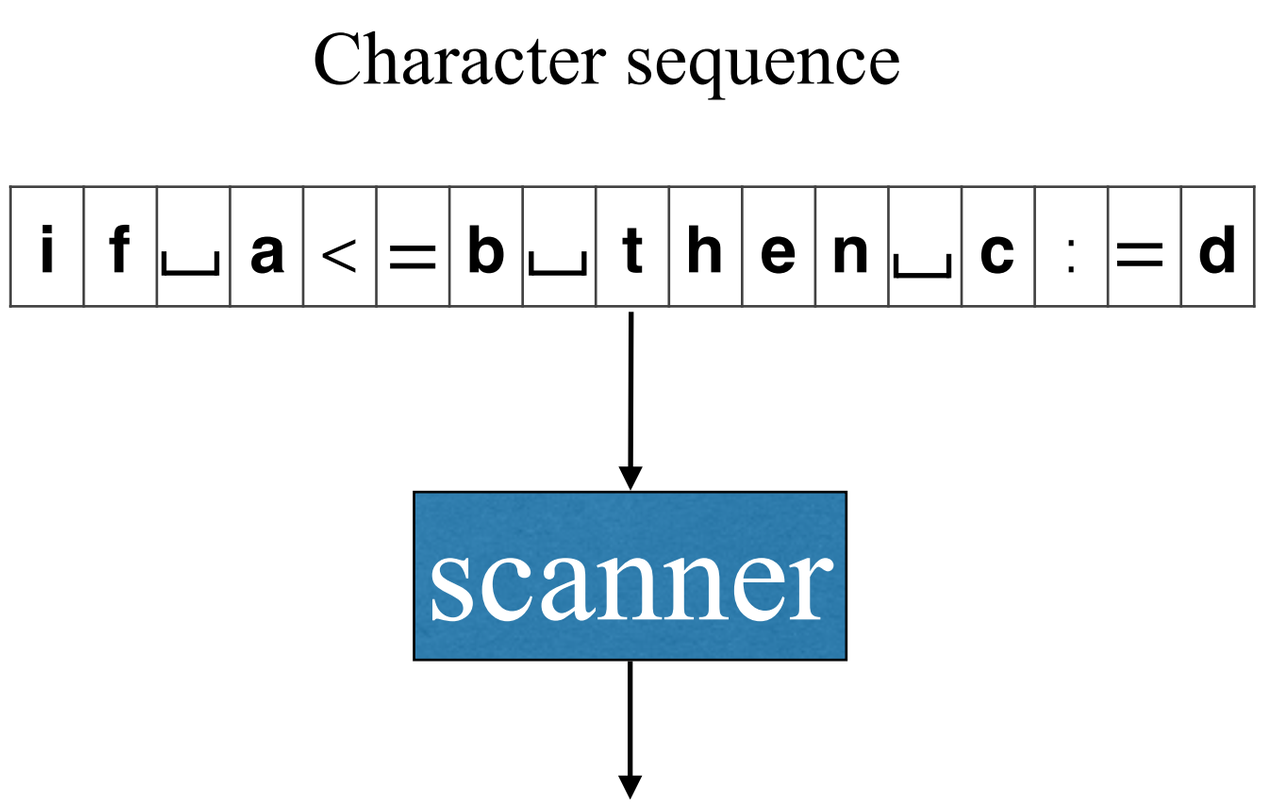
i f a < = b t h e n c : = d

Lexical Analysis (Scott 2.1, 2.2)

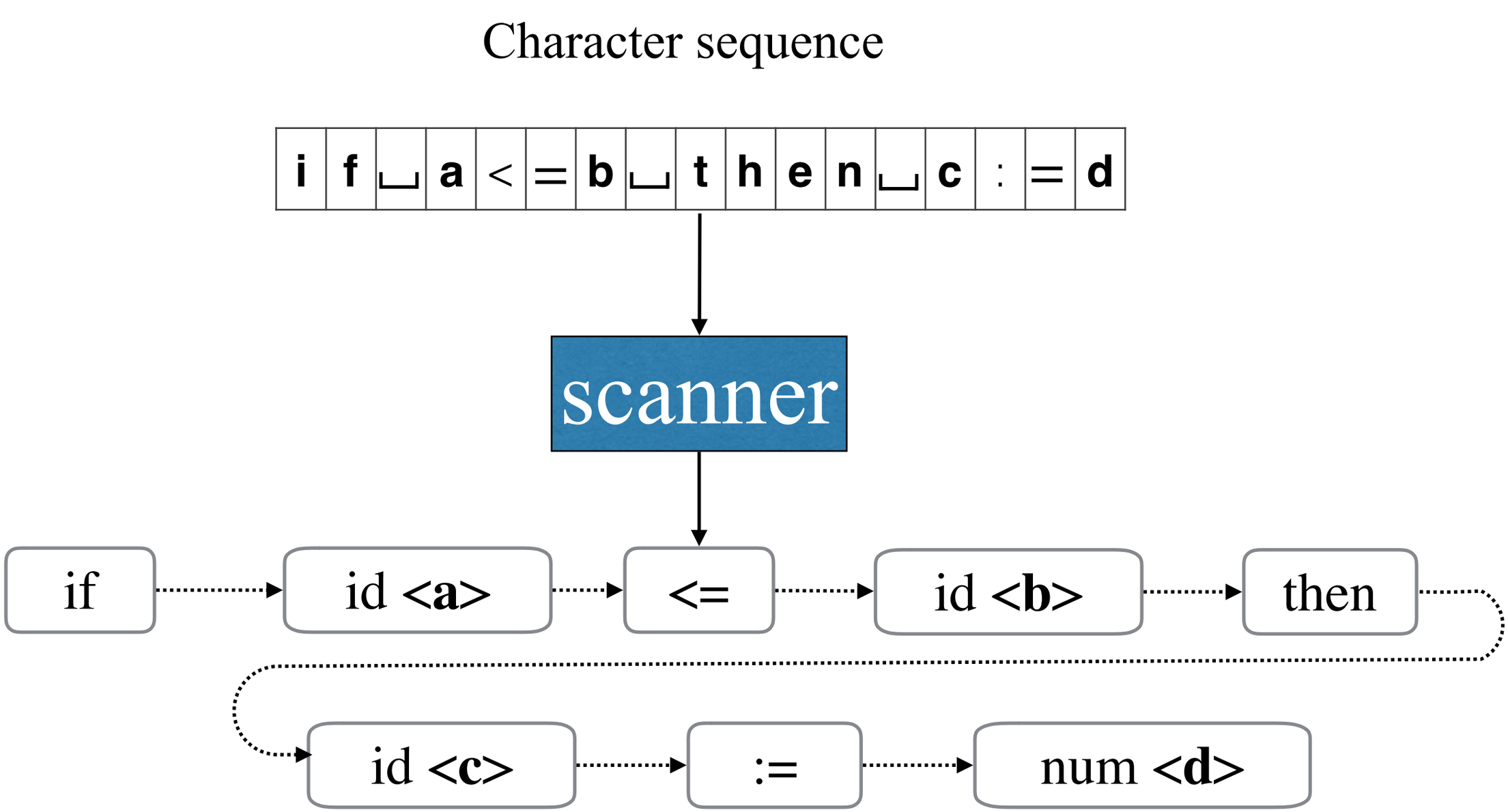
Character sequence

i	f	␣	a	<	=	b	␣	t	h	e	n	␣	c	:	=	d

Lexical Analysis (Scott 2.1, 2.2)



Lexical Analysis (Scott 2.1, 2.2)



Lexical Analysis (Scott 2.1, 2.2)

Tokens (*Terminal Symbols of CFG, Words of Lang*.)

- Smallest "atomic" units of syntax
- Used to build all the other constructs
- Example, **C**:

  keywords: **for if goto volatile**…

  = * / - < > == <= >= <> ( ) [ ] ; := . , …

  number: (Example: 3.14 28 …)

  identifier: (Example: b square addEntry …)

Lexical Analysis (cont.)

Identifiers

- Names of variables, etc.
- Sequence of terminals of restricted form;
  Example, in C language: **A31,** but not **1A3**
- Upper/lower case sensitive?

Keywords

- Special identifiers which represent tokens in the language
- May be reserved (reserved words) or not
  - E.g., C: "**if**" is reserved.

Delimiters – When does character string for token end?

- Example: identifiers are longest possible character sequence that does not include a delimiter.
- Most languages have more delimiters such as '␣', new line, keywords, ...

Regular Expressions

A syntax (notation) to specify regular languages.

RE r	**Language L(r)**
a	**{a}**
ϵ	{**ϵ**}

Regular Expressions

A syntax (notation) to specify regular languages.

RE r	**Language L(r)**
r | s	L(r) ∪ L(s)
rs	{rs | r ∈ L(r), s ∈ L(s)}
r_+	L(r) ∪ L(rr) ∪ L(rrr) ∪…
r* (r* = r^+ | ϵ)	{ϵ} ∪ L(r) ∪ L(rr) ∪ L(rrr) ∪... (any number of r's concatenated)
(s)	L(s)

Examples of Expressions— Solution

RE	Language
a|bc	
(b|c)a	
a ϵ	
a*|b	
ab*	
ab*|c^+	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	
a ϵ	
a*|b	
ab*	
ab*|c^+	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	
a*|b	
ab*	
ab*|c^{+}	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	**{a}**
a*|b	
ab*	
ab*|c^{+}	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	**{a}**
a*|b	**{ϵ, a, aa, aaa, aaaa, …}∪{b}**
ab*	
ab*|c^{+}	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	**{a}**
a*|b	**{ϵ, a, aa, aaa, aaaa, …}∪{b}**
ab*	**{a, ab, abb, abbb, abbbb, . . .}**
ab*|c^{+}	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	**{a}**
a*|b	**{ϵ, a, aa, aaa, aaaa, …}∪{b}**
ab*	**{a, ab, abb, abbb, abbbb, . . .}**
ab*|c^{+}	**{a, ab, abb, abbb, abbbb, . . . } ∪ {c,cc,ccc,…}**
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	**{a}**
a*|b	**{ϵ, a, aa, aaa, aaaa, …}∪{b}**
ab*	**{a, ab, abb, abbb, abbbb, . . .}**
ab*|c^+	**{a, ab, abb, abbb, abbbb, . . . } ∪ {c,cc,ccc,…}**
(a|b)*	**{ϵ, a, b, aa, ab, ba, bb, aaa, aab, . . .}**
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	**{a}**
a*|b	**{ϵ, a, aa, aaa, aaaa, …}∪{b}**
ab*	**{a, ab, abb, abbb, abbbb, . . .}**
ab*|c^{+}	**{a, ab, abb, abbb, abbbb, . . . } ∪ {c,cc,ccc,…}**
(a|b)*	**{ϵ, a, b, aa, ab, ba, bb, aaa, aab, . . .}**
(0|1)*0	binary numbers ending in 0

Regular Expressions for Programming Languages

Let *letter* stand for A | B | C | . . . | Z

Let *digit* stand for 0 | 1 | 2 | . . . | 9

integer constant:

identifier:

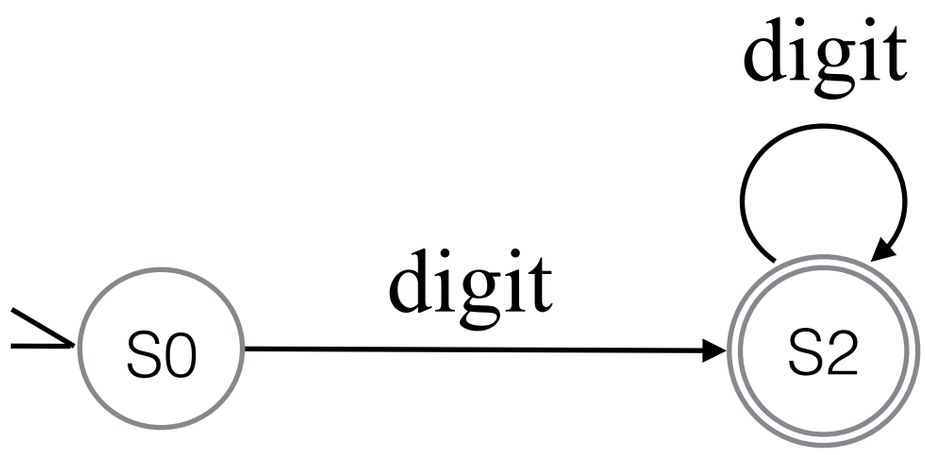
real constant:

Recognizers for Regular Expressions

Example 1: integer constant

RE: digit^{+}

FSA:

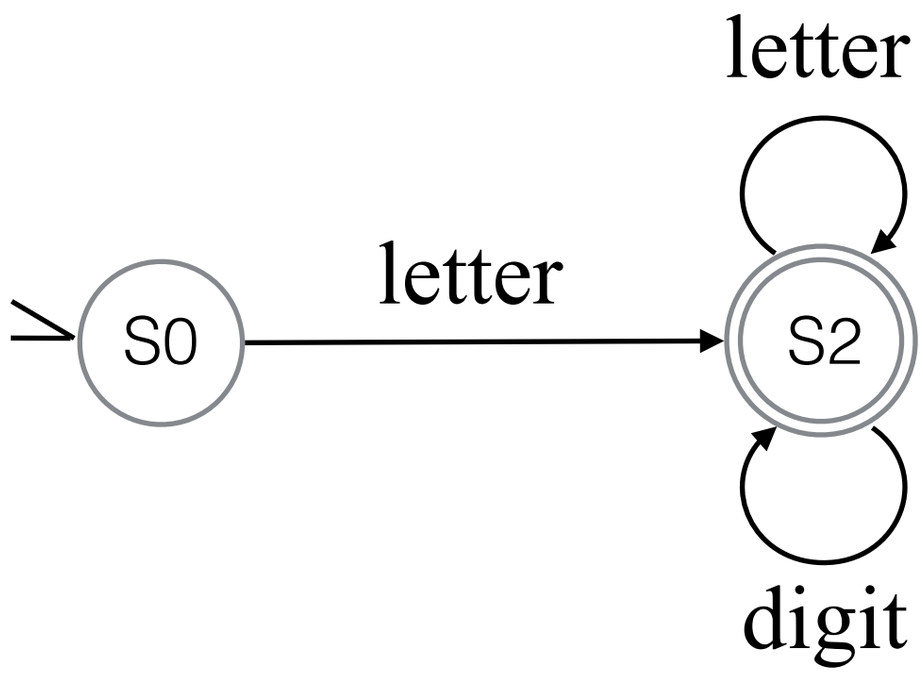


Recognizers for Regular Expressions(Cont.)

Example 2: identifier

RE: letter(letter | digit)*

FSA:

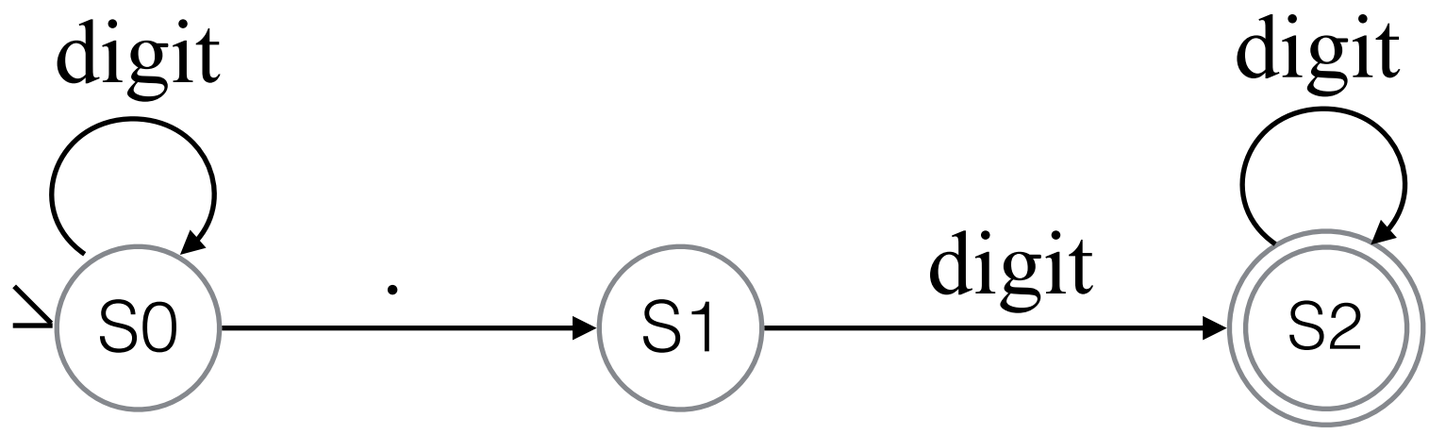


Recognizers for Regular Expressions(Cont.)

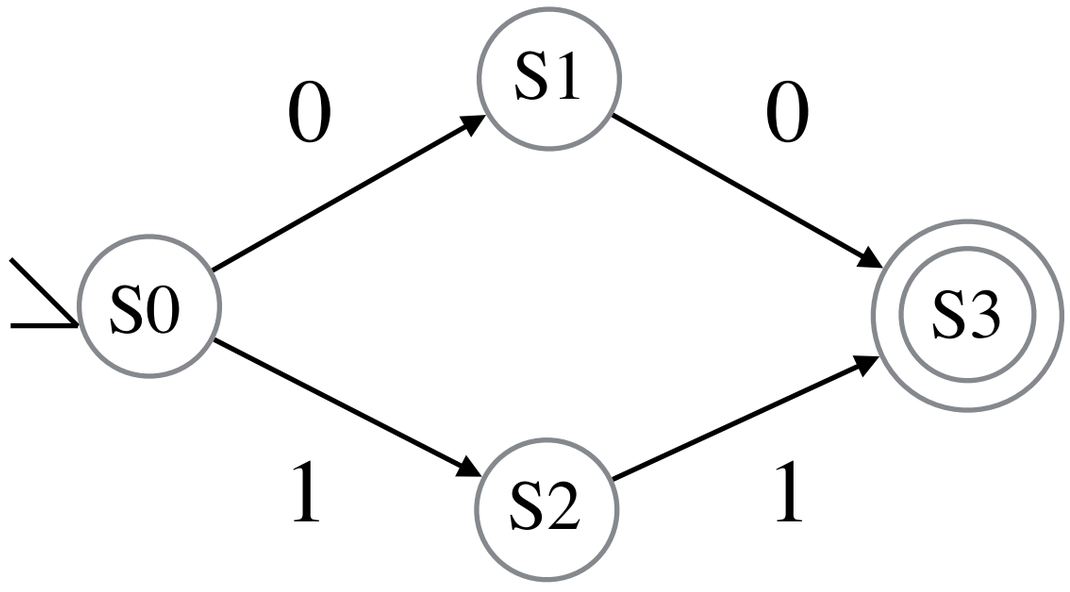
Example 3: Real constant

RE: digit*.digit^{+}

FSA:



Finite State Automata

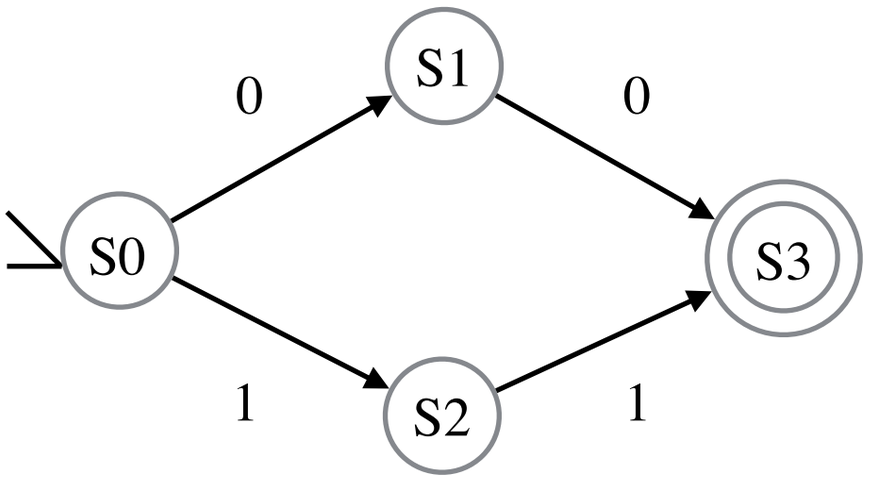


A Finite-State Automaton is a quadruple: < S, s, F, T >

- S is a set of states, e.g., {S0, S1, S2, S3}
- s is the start state, e.g., S0
- F is a set of final states, e.g., {S3}
- T is a set of labeled transitions, of the form (state, input) → state [i.e., $S \times \Sigma \rightarrow S$]

Finite State Automata

Transitions can be represented using a transition table:

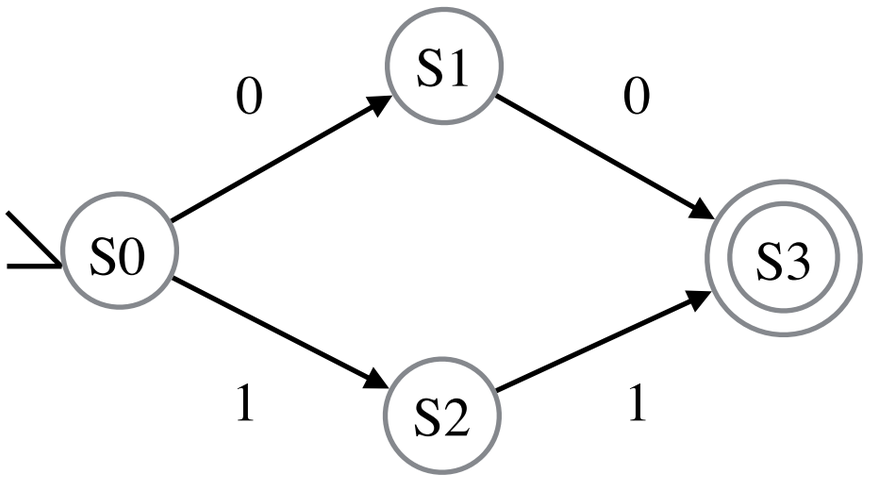


State	*0*	*1*	Input
S0	*S1*	*S2*	
S1	*S3*	-	
S2	-	*S3*	

An FSA *accepts* or *recognizes* an input string N **iff** there is some path from start state to a final state such that the labels on the path spell N.

Finite State Automata

Transitions can be represented using a transition table:



State	*0*	*1*	Input
S0	*S1*	*S2*	
S1	*S3*	-	
S2	-	*S3*	

Error

An FSA *accepts* or *recognizes* an input string N **iff** there is some path from start state to a final state such that the labels on the path spell N.

Lack of entry in the table (or no arc for a given character) indicates an *error—reject*.

Practical Recognizers

- Recognizer should be a deterministic finite automaton (DFA)
- Read until the end of a token
- Report errors (error recovery)

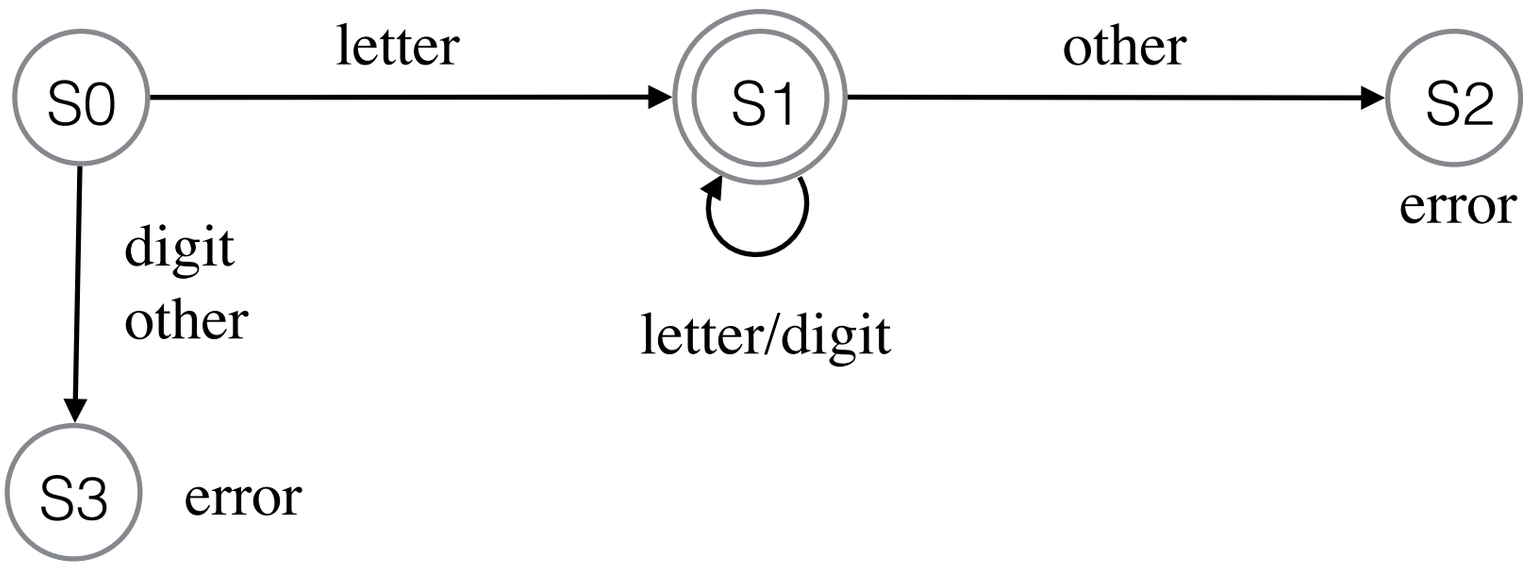
identifier

letter → (a | b | c | ... | z | A | B | C | ... | Z)

digit → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

id → letter (letter | digit)*

Recognizer for identifier: (**transition diagram**)



Implementation: Table for the recognizer

Two tables control the recognizer.

char_class:

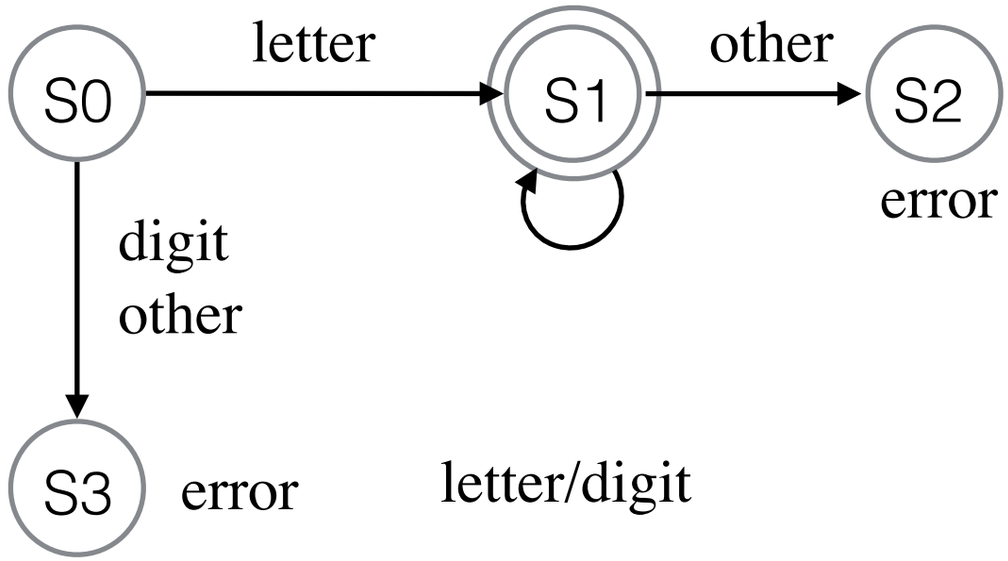
	a - z	*A - Z*	*0 - 9*	*other*
value	*letter*	*letter*	*digit*	*other*

next_state:

class	*S0*	*S1*	*S2*	*S3*
letter	*S1*	*S1*	—	—
digit	*S3*	*S1*	—	—
other	*S3*	*S2*	—	—

To change languages, we can just change tables.

Implementation: Code for the recognizer



class	*S0*	*S1*	*S2*	*S3*
letter	*S1*	*S1*	—	—
digit	*S3*	*S1*	—	—
other	*S3*	*S2*	—	—

```
char ← next_char();
state ← S0;
done ← false;
while( not done ) {
      class ← char_class[char];
      state ← next_state[class,state];
      switch(state) {
            case S1:
                  /* building an id */
                  token_value ← token_value + char;
                  char ← next_char();
                  if (char == EOF)
                        done = true;
                  break;
            case S2: /* error state */
            case S3: /* error state */
                  token type = error;
                  done = true;
                  break;
      }
}
return token_type;
```

Improved efficiency

Table driven implementation is slow relative to direct code.
Each state transition involves:

1. Classifying the input character
2. Finding the next state
3. An assignment to the state variable
4. A trip through the case statement logic
5. A branch (while loop)

We can do better by “encoding” the state table in the scanner code

1. Classify the input character
2. Test character class locally
3. Branch directly to next state

This takes many fewer instructions.

Implementation: Faster scanning

```
S0:     char ← next_char();
        token value ← “”;
        class ← char_class[char];
        if (class != letter)
              goto S3;

S1:     token value ← token_value + char;
        char ← next_char();
        class ← char_class[char];
        if (class != other)
              goto S1;
        if (char == DELIMITER )
              token type = identifier;
              return token_type;
        goto S2;

S2:
S3:    token type ← error;
        return token type;
```

class	*S0*	*S1*	*S2*	*S3*
letter	*S1*	*S1*	—	—
digit	*S3*	*S1*	—	—
other	*S3*	*S2*	—	—

Implementation: Faster scanning

```
S0:     char ← next_char();
        token value ← “”;
        class ← char_class[char];
        if (class != letter)
              goto S3;

S1:     token value ← token_value + char;
        char ← next_char();
        class ← char_class[char];
        if (class != other)
              goto S1;
        if (char == DELIMITER )
              token type = identifier;
              return token_type;
         goto S2;

S2:
S3:    token type ← error;
        return token type;
```

Faster code

```
char ← next_char();
state ← S0;
done ← false;
while( not done ) {
      class ← char_class[char];
      state ← next_state[class,state];
      switch(state) {
            case S1:
                  /* building an id */
                  token_value ← token_value + char;
                  char ← next_char();
                  if (char == EOF)
                       done = true;
                  break;
            case S2: /* error state */
            case S3: /* error state */
                  token type = error;
                  done = true;
                  break;
      }
}
return token_type;
```

Original code

Next Lecture

Things to do:

- Read Scott, Chapters 2.3 - 2.5