CS 314 Principles of Programming Languages

Lecture 2: Syntax Analysis (Scanning)

Prof. Zheng Zhang

Rutgers University

September 6, 2018

Announcement

- **First recitation starts this coming Wednesday**
- **Homework 1 will be released after lecture 3.**
- **My office hour:**

  Thursday 2pm - 3pm at CoRE 315
- **TA office hours will be announced soon.**

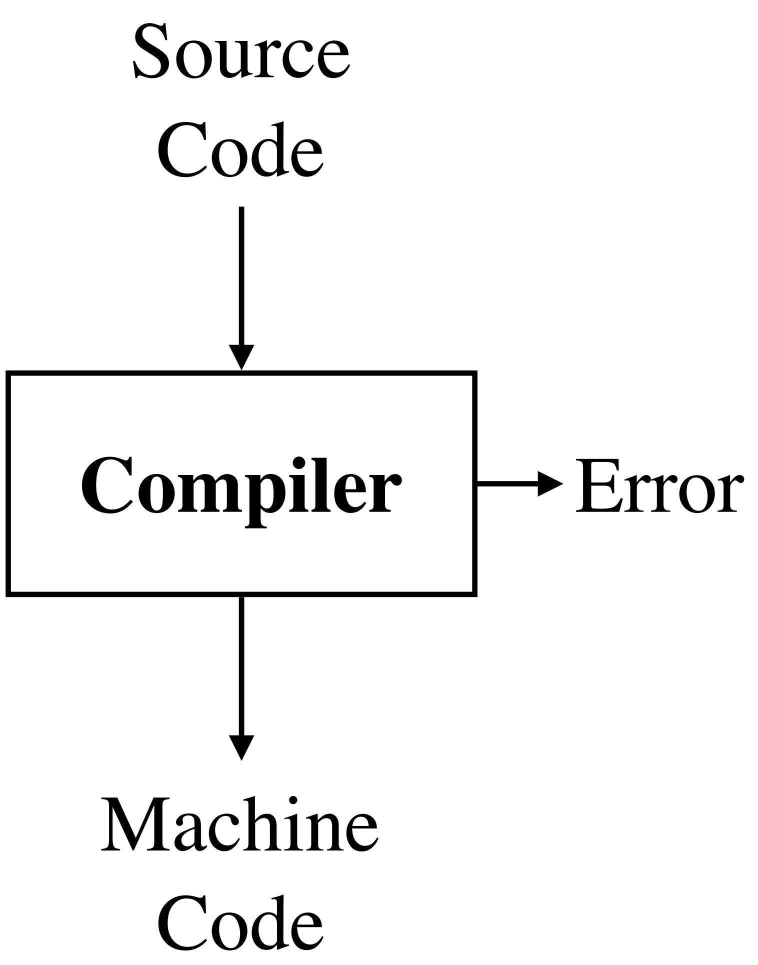
Last Class

- Overview of compilation
- Syntax and semantics
- Formal language definition
- A rule-based rewriting system
- Introduction to regular expression

Compilation and Interpretation

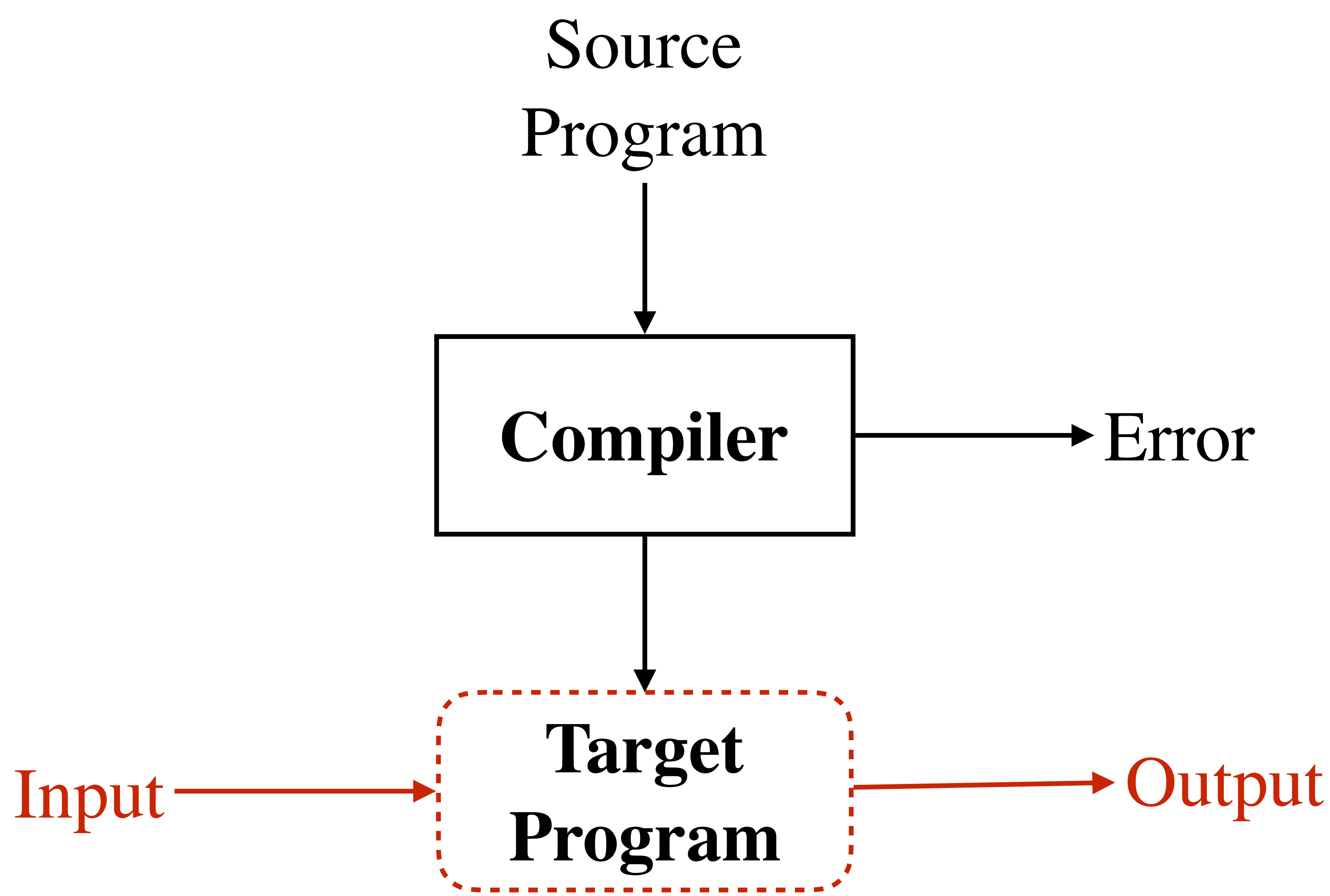
Compiler

- 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 to higher level notations

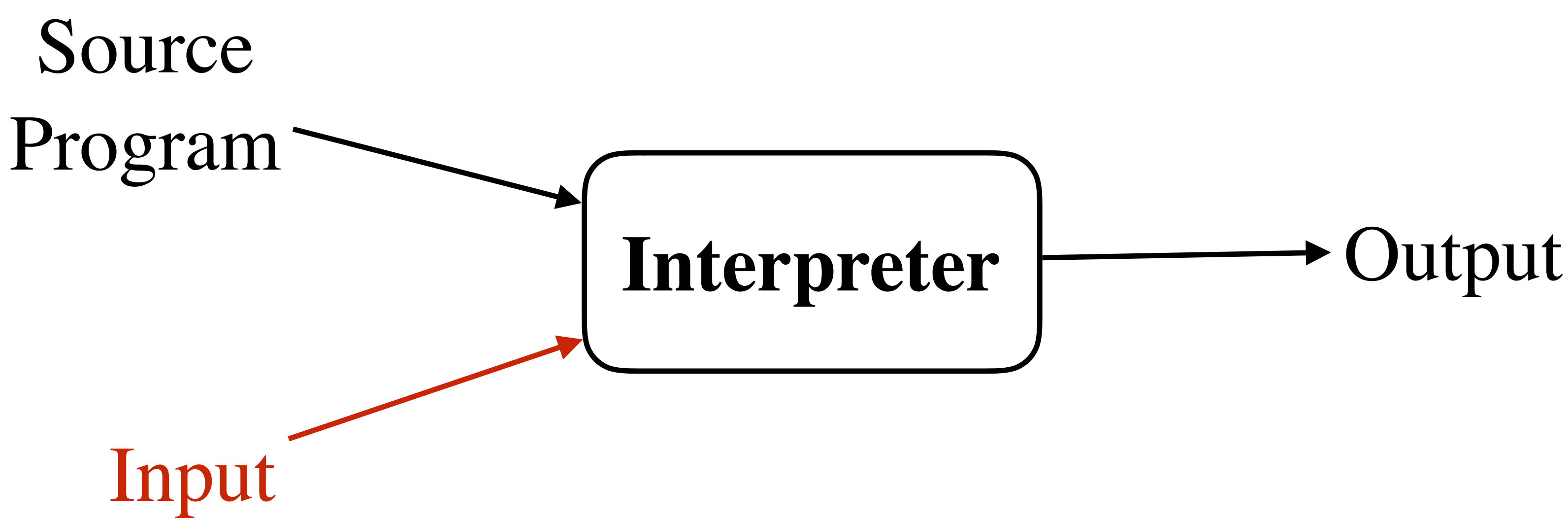
Compilation and Interpretation



Pure Compilation

- Mainly refers to translation
- Take a program in source language, output a program in target language (usually machine code)

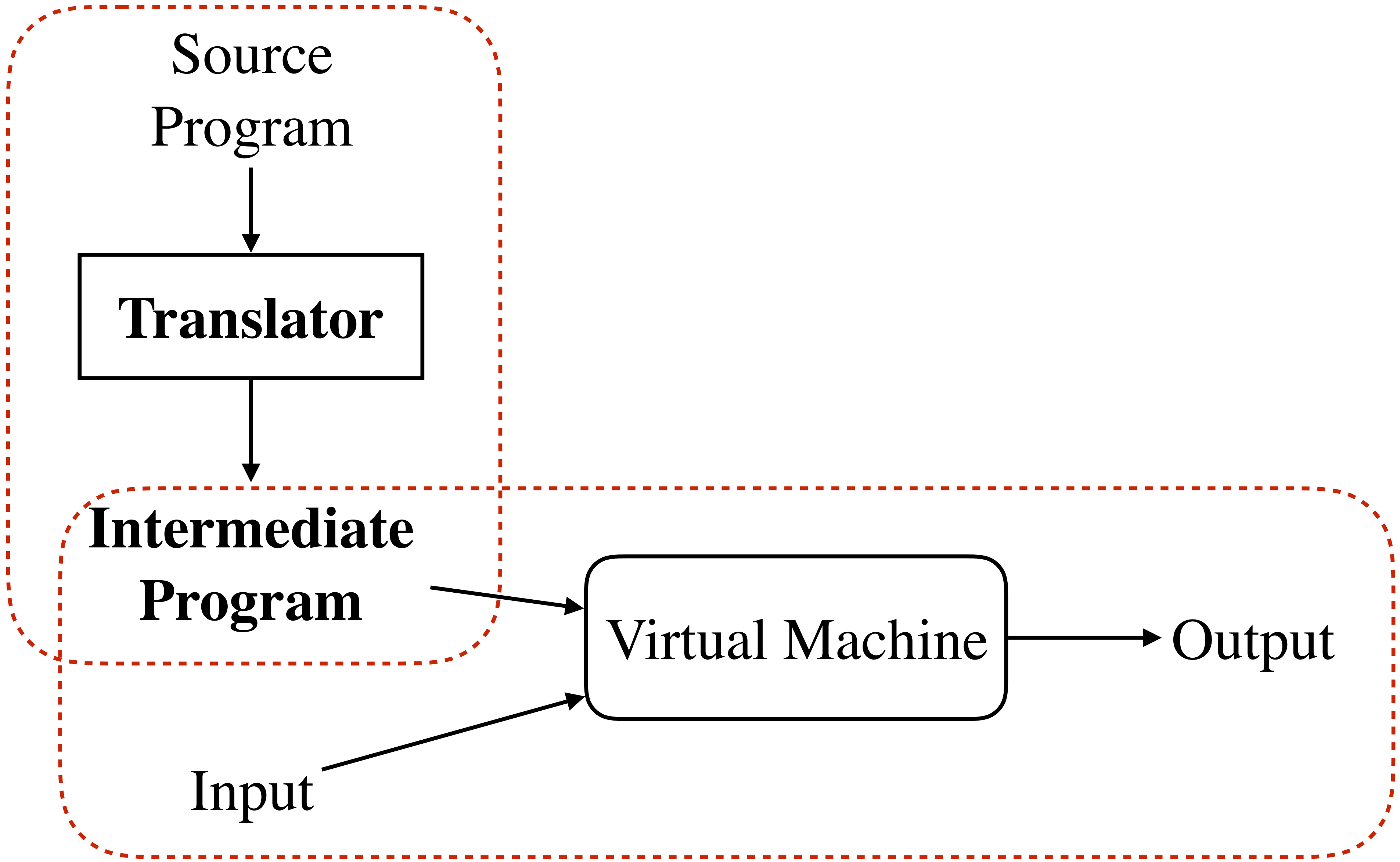
Compilation and Interpretation



Interpretation

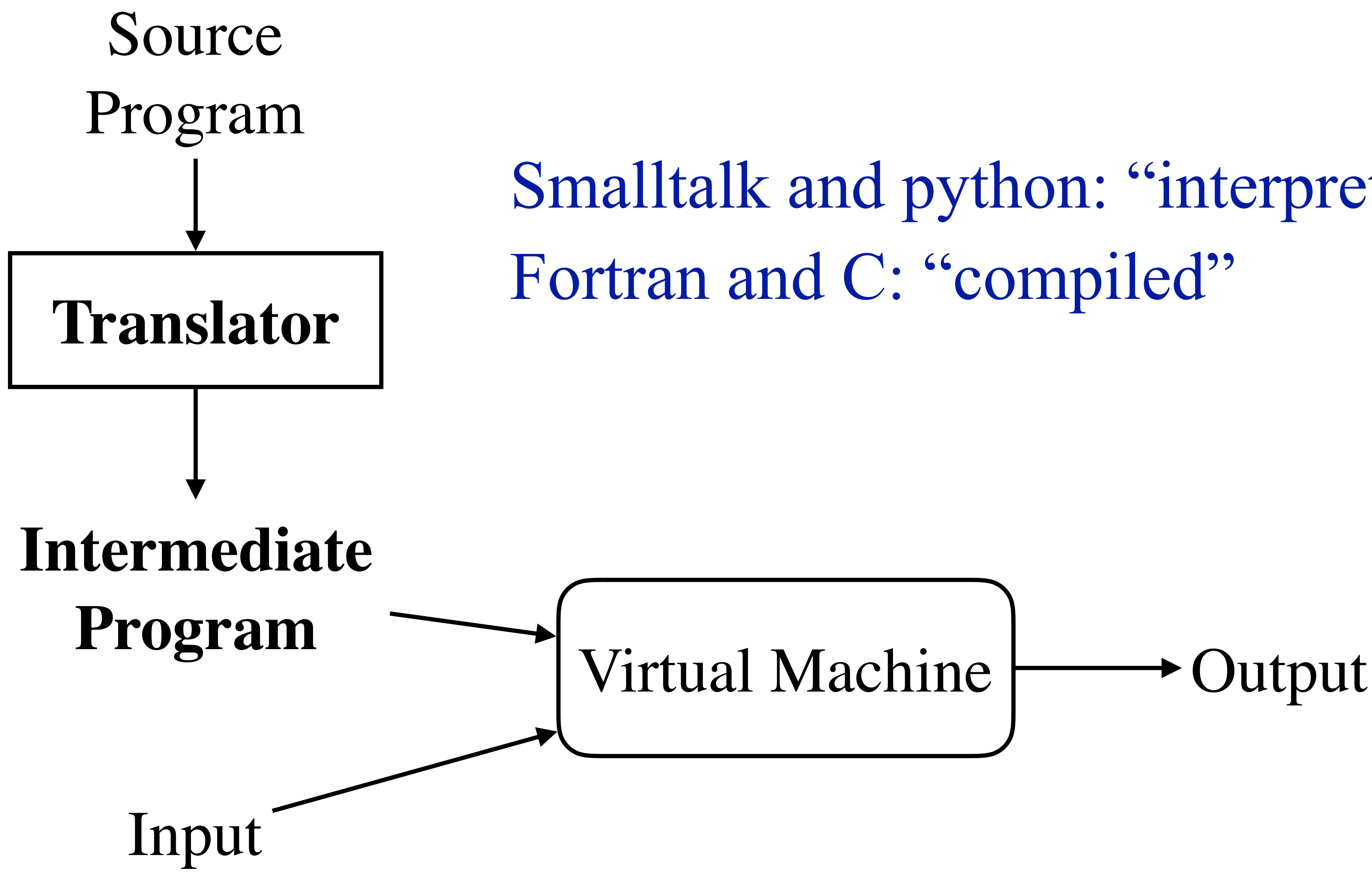
- Interpreter stays around for the execution of the program
- Interpreter is the locus of control during execution

Compilation and Interpretation



- Most language implementations include a mixture of both compilation and interpretation.
- Common case is compilation or simple pre-processing, followed by interpretation.

Compiled V.S. Interpreted Languages



Smalltalk and python: "interpreted"
Fortran and C: "compiled"

- We generally say that:
  A language is "interpreted" if the initial translator is "simple",
  or "compiled" if the initial translator is "complicated"
- Very subjective, but a language is still "compiled" if the translator has thorough analysis and non-trivial transformation.

Syntax and Semantics of Programming Languages

Syntax:

Describes what a legal program looks like

Semantics:

Describes what a correct (legal) program means

Syntax of Programming Languages

The syntax of programming languages is often defined in two layers:

tokens and ***sentences***.

- *tokens* - *legal combination of characters in the language*

Question: How to spell a token (word)?

Answer: Regular expressions

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

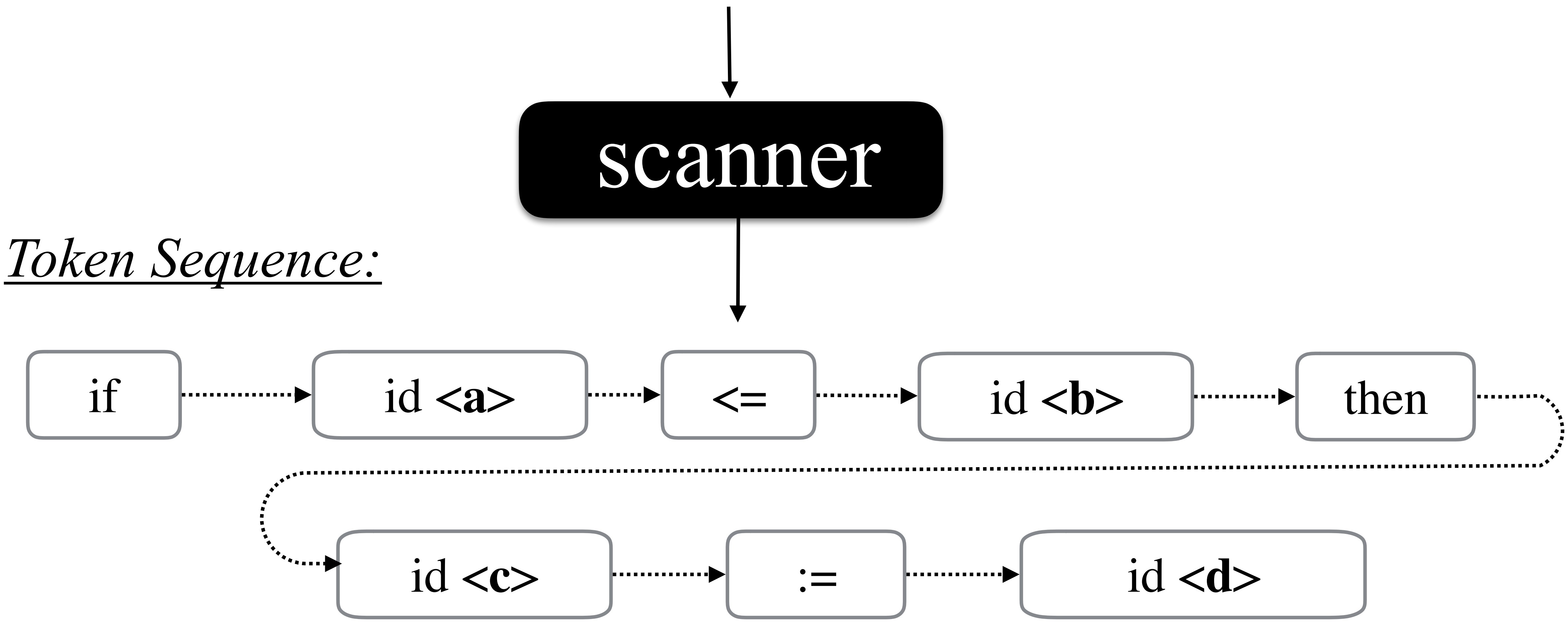
Question: How to build correct sentences with tokens?

Answer: (Context - free) grammars (CFG)

Lexical Analysis (Scott 2.1, 2.2)

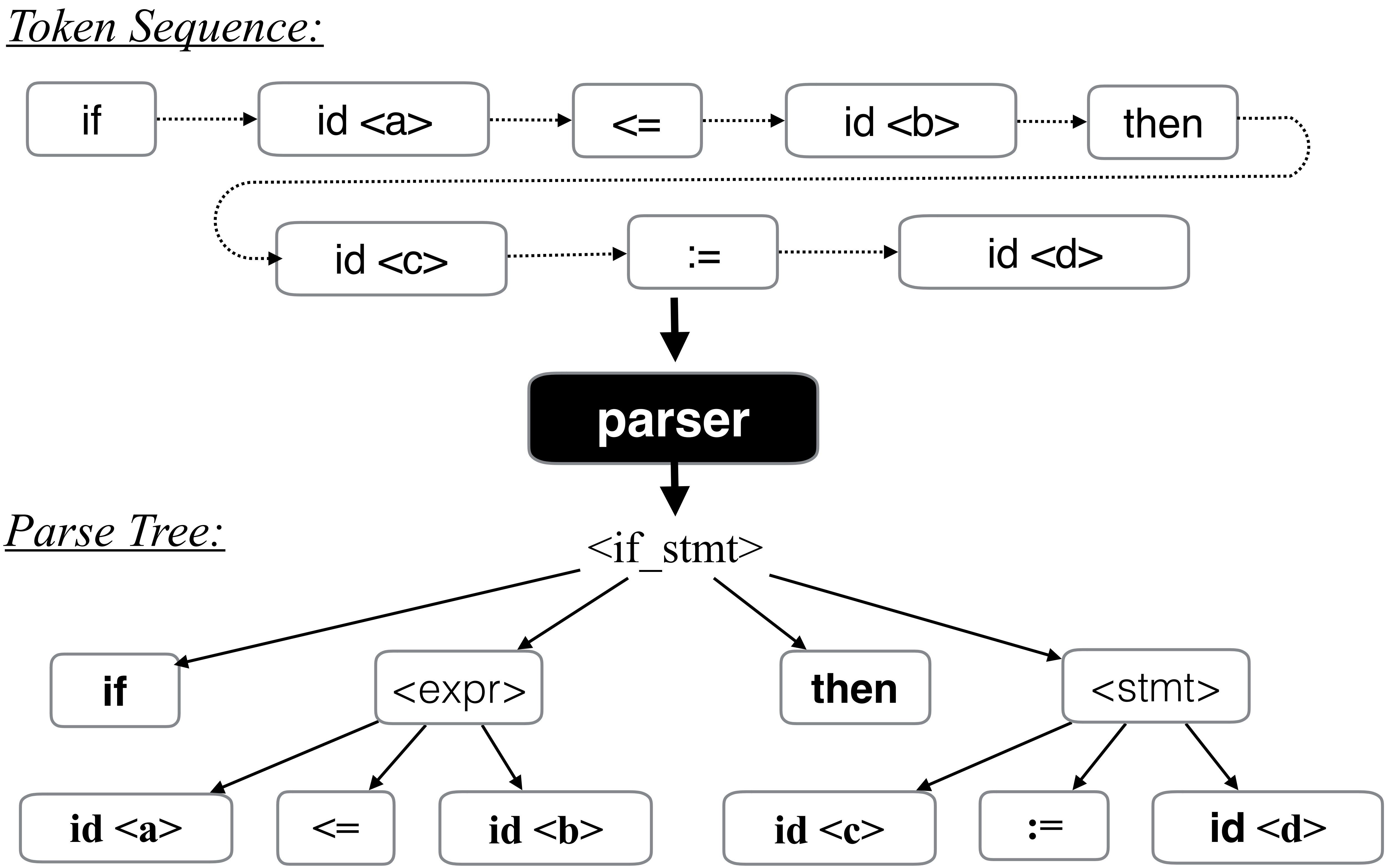
Character Sequence:

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



Token Sequence:

Syntax Analysis (Scott, Chapter 2.3)



Tokens (Scott 2.1, 2.2)

Tokens (Analogous to *Words of Language*)

- Smallest “atomic” units of further syntax analysis
- Used to build all the other constructs
- Example, in **C**:

  Keywords: **for if goto volatile**…

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

  Number: (Example: 3.14 28 …)

  Identifier: (Example: b square addEntry …)

Formalisms for Lexical and Syntactic Analysis

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

- *Language Specification* → formalism to describe what a valid program (word/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 (word/sentence) looks like.
- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

We use **regular expression** to specify tokens (words)

Regular Expressions

A syntax (notation) to specify regular languages.

RE p	**Language L(*p*)**

L(*p*): the set of strings that can be represented using the expression ***p***

Regular Expressions

A syntax (notation) to specify regular languages.

RE p	**Language L(*p*)**
$r \mid s$	$L(r) \cup L(s)$

Either r or s is a regular expression, i.e. ***0|11***

L(*p*): the set of strings that can be represented using the expression ***p***

Regular Expressions

A syntax (notation) to specify regular languages.

RE p	**Language L(*p*)**
$r \mid s$	$L(r) \cup L(s)$
rs	$\{RS \mid R \in L(r), S \in L(s)\}$

L(*p*): the set of strings that can be represented using the expression ***p***

Regular Expressions

A syntax (notation) to specify regular languages.

RE p	**Language L(*p*)**
$r \mid s$	$L(r) \cup L(s)$
rs	$\{RS \mid R \in L(r), S \in L(s)\}$
$r+$	$L(r) \cup L(rr) \cup L(rrr) \cup \ldots$

L(*p*): the set of strings that can be represented using the expression ***p***

Regular Expressions

A syntax (notation) to specify regular languages.

RE p	**Language L(*p*)**
$r \mid s$	$L(r) \cup L(s)$
rs	$\{RS \mid R \in L(r), S \in L(s)\}$
$r+$	$L(r) \cup L(rr) \cup L(rrr) \cup \ldots$
r^* $(r^* = r^+ \mid \epsilon)$	$\{\epsilon\} \cup L(r) \cup L(rr) \cup \ldots$
(s)	$L(s)$

Any number of r's concatenated.

Regular Expressions

A syntax (notation) to specify regular languages.

RE p	**Language L(*p*)**
$r \mid s$	$L(r) \cup L(s)$
rs	$\{RS \mid R \in L(r), S \in L(s)\}$
$r+$	$L(r) \cup L(rr) \cup L(rrr) \cup \ldots$
r^* $(r^* = r^+ \mid \epsilon)$	$\{\epsilon\} \cup L(r) \cup L(rr) \cup \ldots$
(s)	L(s)
a	{**a**}
ϵ	$\{\epsilon\}$

Any number of r's concatenated.

A RE can simply be a letter from the alphabet Σ or an empty string ϵ

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, …}$\cup${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, …}$\cup${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, …}$\cup${b}**
ab*	**{a, ab, abb, abbb, abbbb, . . .}**
ab*|c^{+}	**{a, ab, abb, abbb, abbbb, . . . } $\cup$ {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

Concatenation has higher precedence over alternation |.

Regular Expressions for Programming Languages

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

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

Regular Expressions for Programming Languages

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

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

integer constant: digit^{+}

identifier: $\text{letter}(\text{letter} \mid \text{digit})^{*}$

real constant: $\text{digit}^{*}.\text{digit}^{+}$

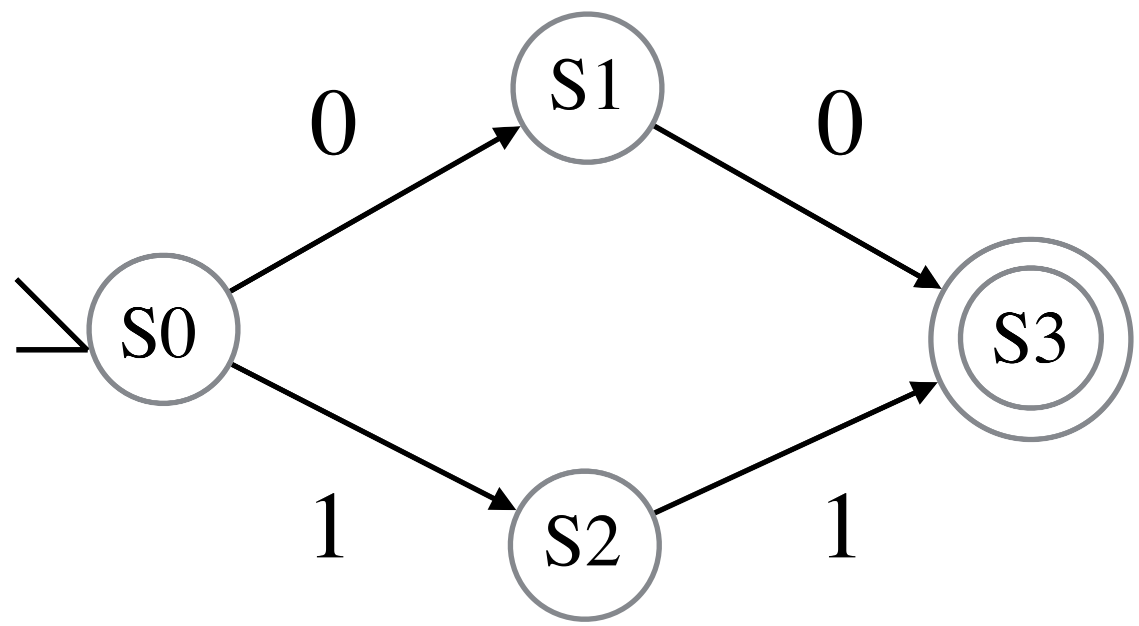
Formalisms for Lexical and Syntactic Analysis

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

- *Language Specification* → formalism to describe what a valid program (word/sentence) looks like.
- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

We use **finite state automata** to recognize regular language

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

  $$(\text{state, input}) \rightarrow \text{state}$$

  formally,

  $$S \times \Sigma \rightarrow S$$

Regular Expressions for Programming Languages

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

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

integer constant: digit^+

identifier: $\text{letter}(\text{letter} \mid \text{digit})^*$

real constant: $\text{digit}^*.\text{digit}^+$

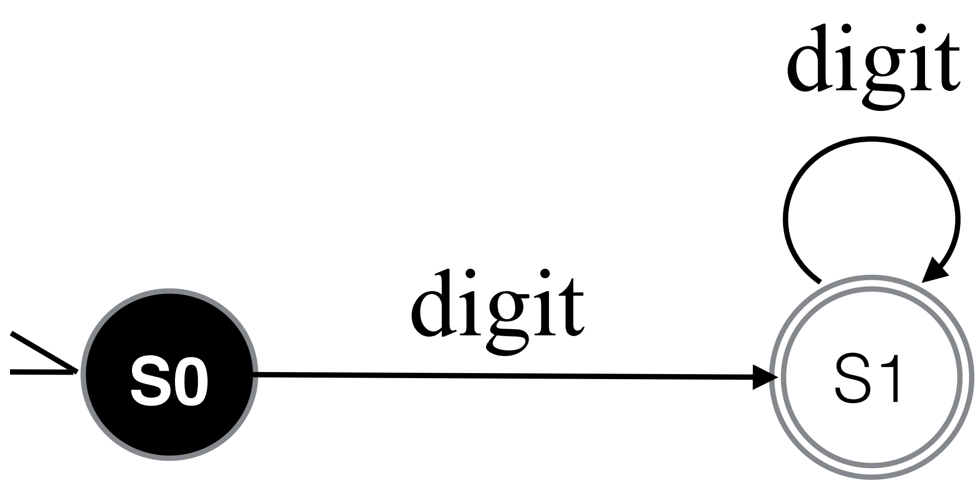
Recognizers for Regular Expressions

<u>*Example 1:*</u>

Integer Constant

RE: digit^+

FSA:



Recognizers for Regular Expressions(Cont.)

Example 2:

Identifier

RE: letter(letter | digit)*

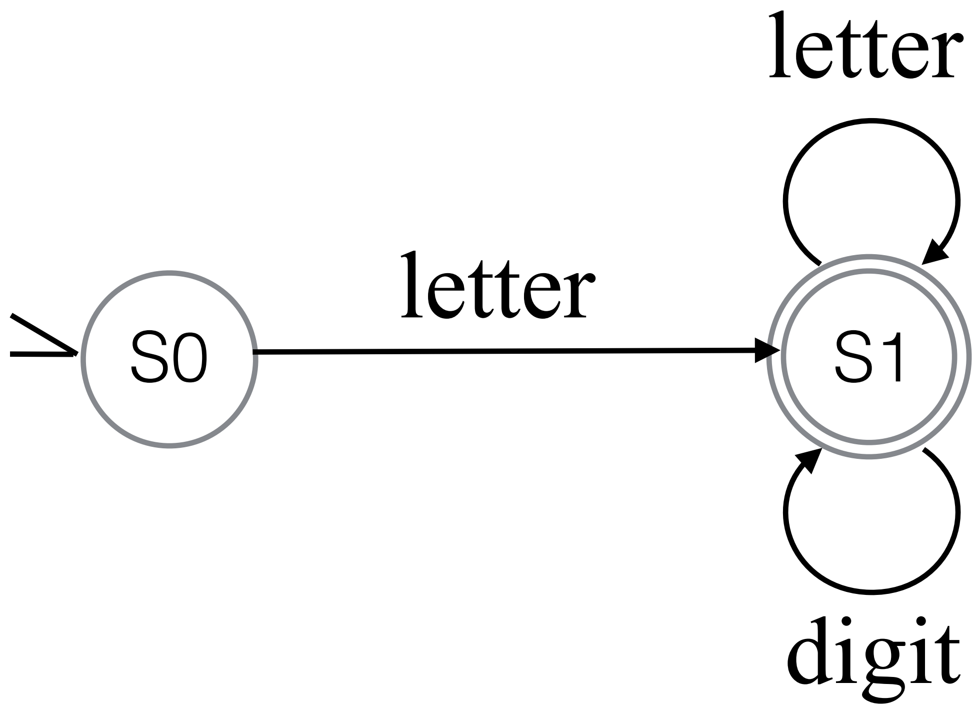
Recognizers for Regular Expressions(Cont.)

Example 2:

Identifier

RE: letter(letter | digit)*

FSA:



Recognizers for Regular Expressions(Cont.)

Example 3:

Real constant

RE: $digit^{*}.digit^{+}$

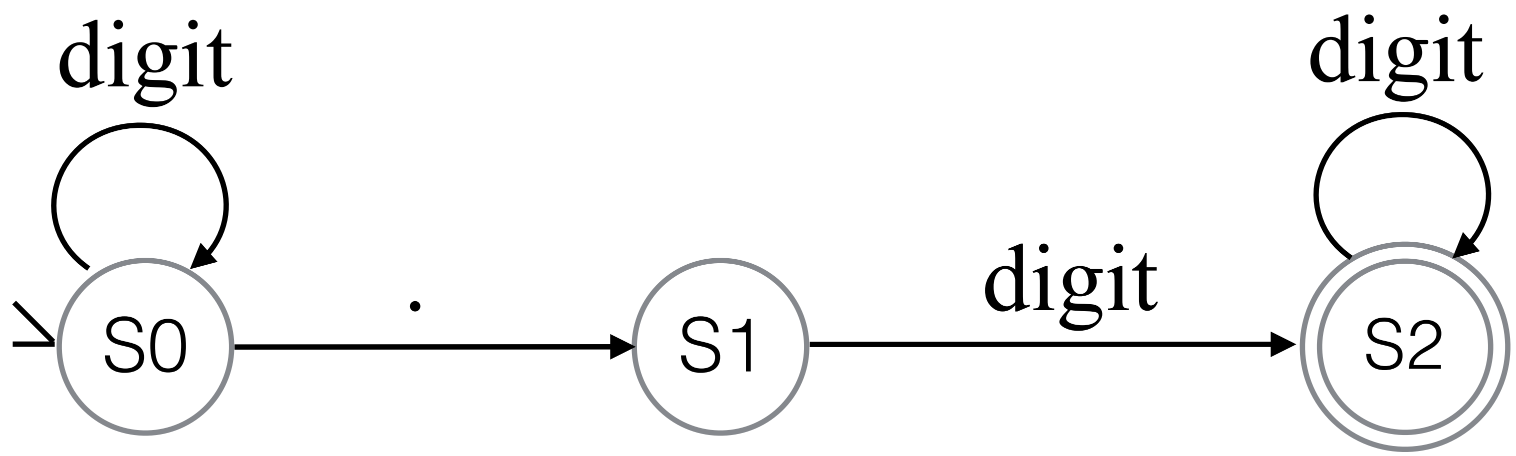
Recognizers for Regular Expressions(Cont.)

Example 3:

Real constant

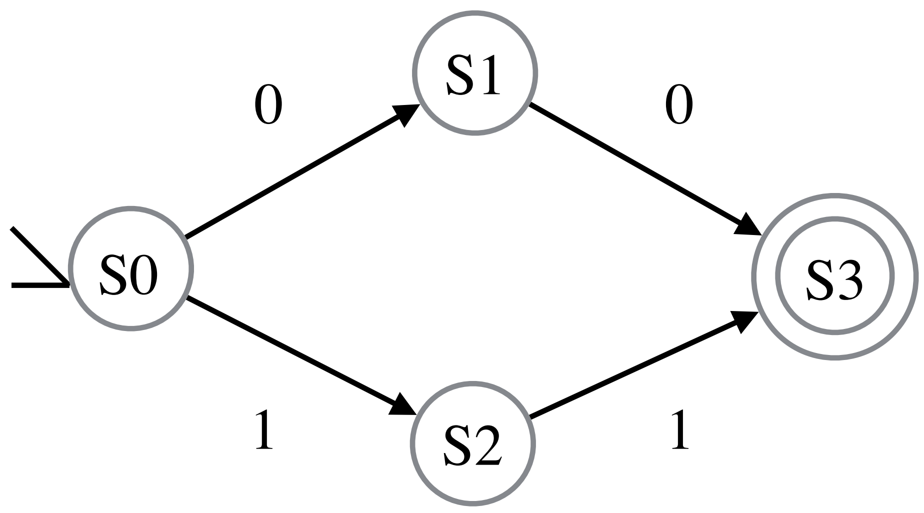
RE: digit*.digit$^+$

FSA:



Finite State Automata

Transitions can be represented using a transition table:



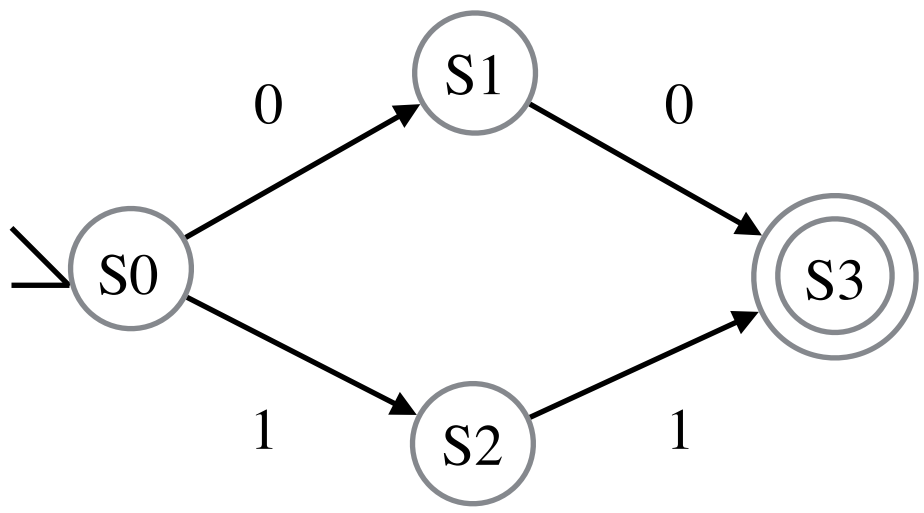
Input

State	*0*	*1*
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 \ Input	*0*	*1*
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)

Practical Recognizers

“identifier” regular expression:

$letter \rightarrow$ (a | b | c | ... | z | A | B | C | ... | Z)

$digit \rightarrow$ (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

$id \rightarrow$ letter (letter | digit)*

Practical Recognizers

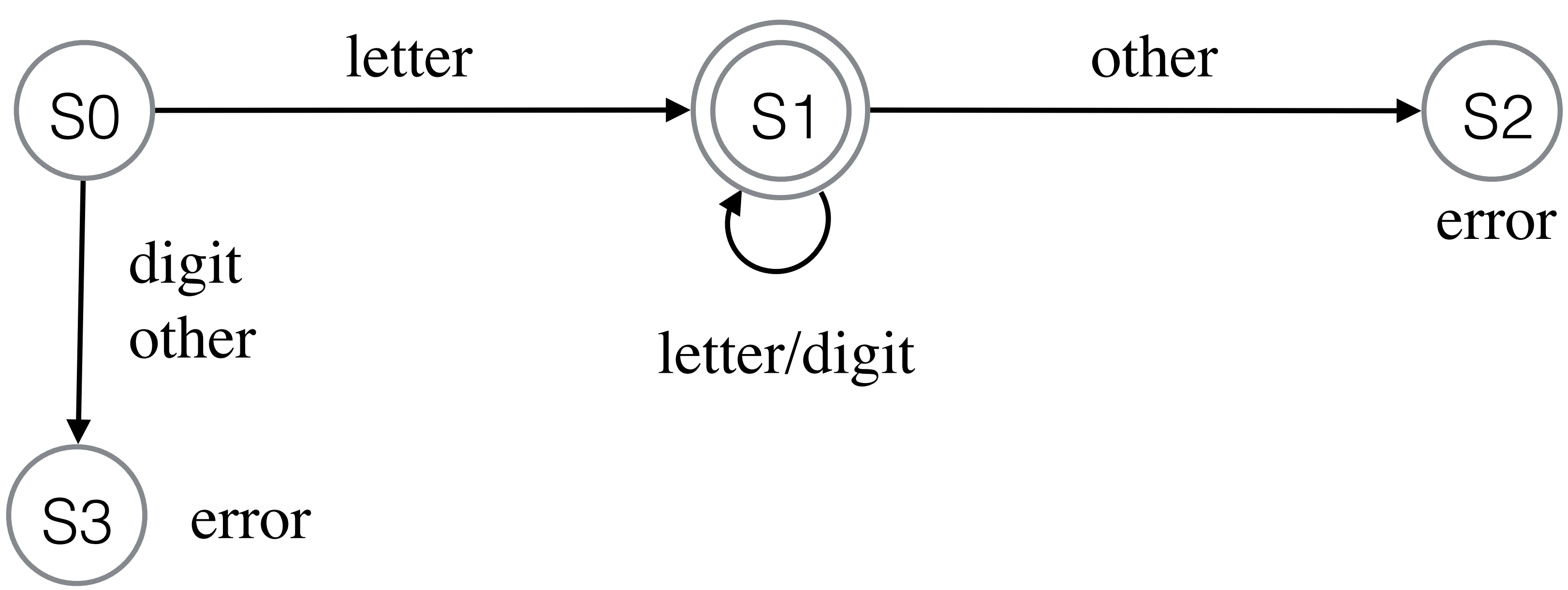
“identifier” regular expression:

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

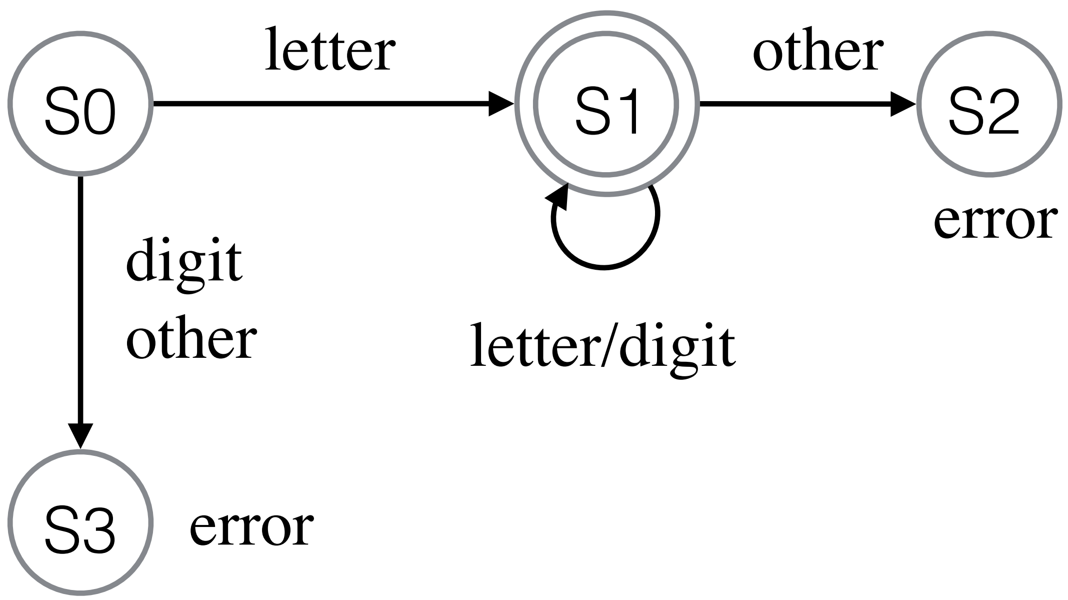
digit → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

id → letter (letter | digit)*

Recognizer for “identifier”:



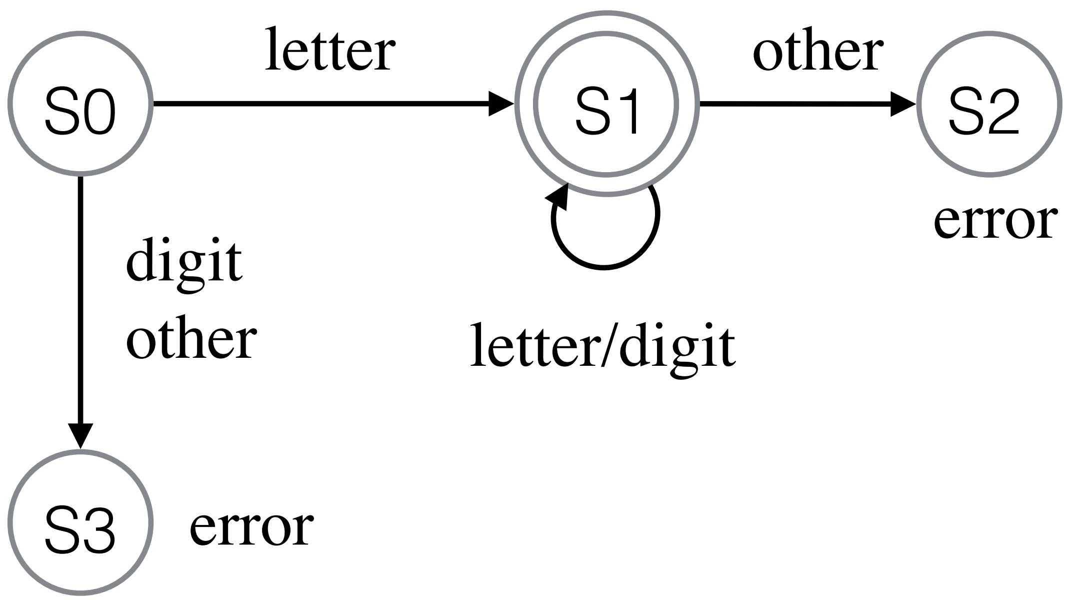
Implementation: Code for the recognizer



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 == DELIMITER)
                       done = true;
                  break;
             case S2: /* error state */
             case S3: /* error state */
                  token type = error;
                  done = true;
                  break;
      }
}
return token_type;
```

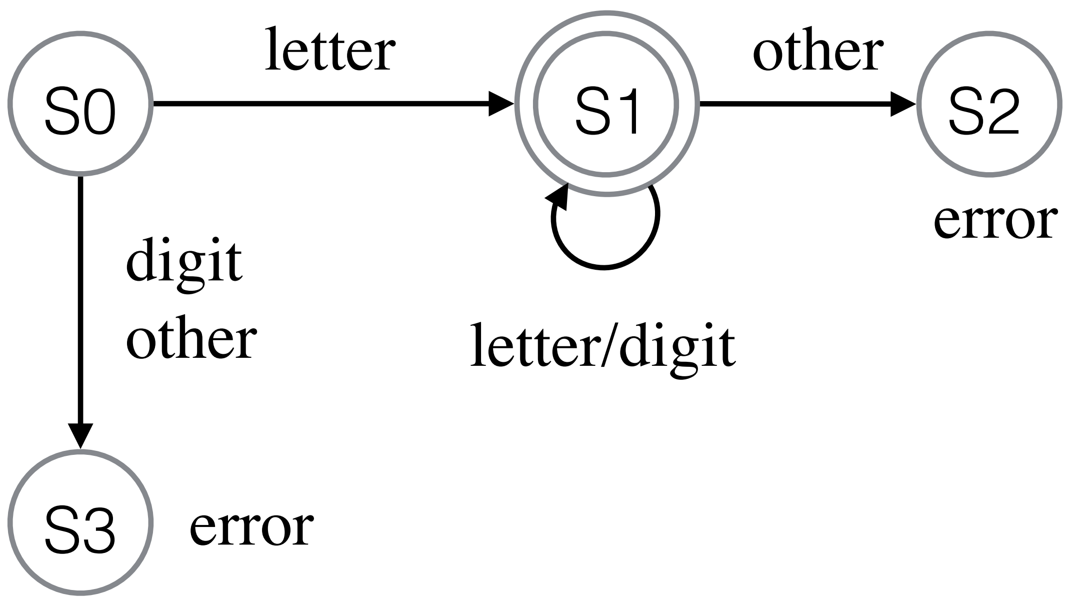
Implementation: Code for the recognizer



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 == DELIMITER)
                done = true;
            break;
        case S2: /* error state */
        case S3: /* error state */
            token type = error;
            done = true;
            break;
    }
}
return token_type;
```

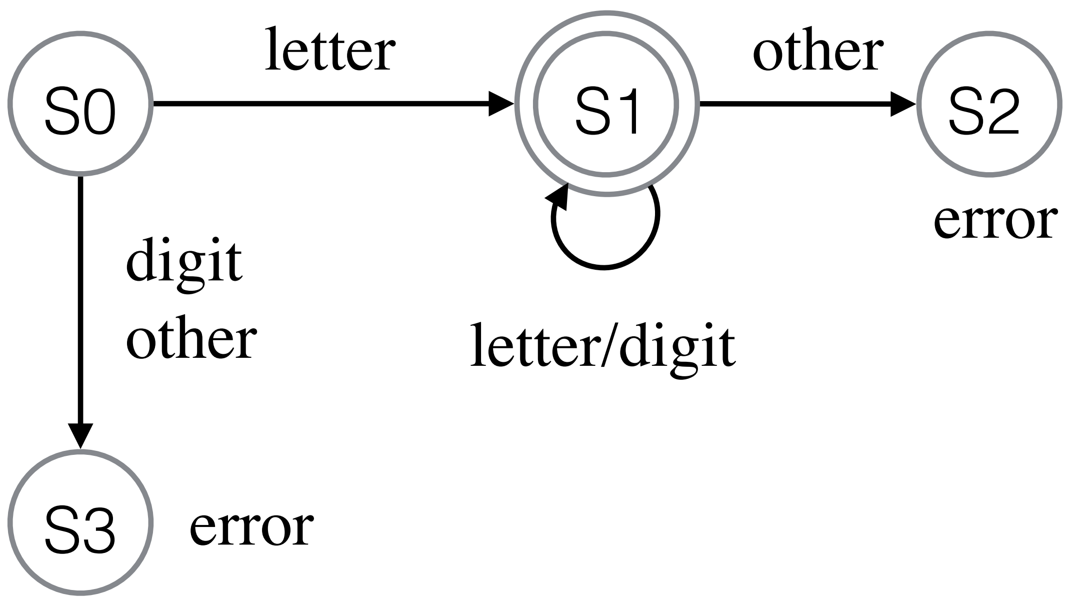
Implementation: Code for the recognizer



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 == DELIMITER)
                done = true;
            break;
        case S2: /* error state */
        case S3: /* error state */
            token type = error;
            done = true;
            break;
    }
}
return token_type;
```

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 == DELIMITER)
                       done = true;
                  break;
             case S2: /* error state */
             case S3: /* error state */
                  token type = error;
                  done = true;
                  break;
      }
}
return token_type;
```

Next Lecture

Things to do:

- Read Scott, Chapters 2.3 - 2.5