# CS 314 Principles of Programming Languages

## Lecture 3: Syntax Analysis

Zheng (Eddy) Zhang

*Rutgers University*

January 24th, 2018

# Class Information

## Homework 1 released

- Due Tuesday 01/30/2018 11:55pm EST.
- Only accepted in **pdf** format.
- No late submission will be accepted in Sakai.
- If you have personal overriding reasons that prevent you from submitting homework, please contact me in advance.

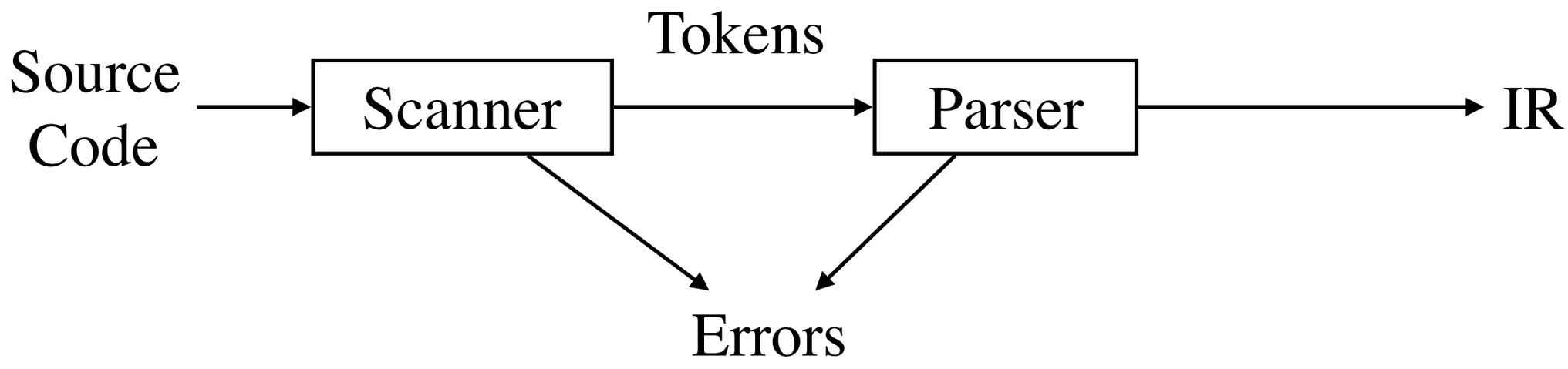
## Our course website is online now

- https://www.cs.rutgers.edu/courses/314/classes/spring_2018_zhang/
- TA office hours posted to the course website.
- All lectures will be posted to the course website in the future.

## Sakai online forum is open for homework and project questions

- Before you ask, check if a similar question is already posted.
- All the questions will be answered within **72 hours**.
- Use Sakai forum wisely: plan ahead if you want an answer before a deadline.

# Review: Front End

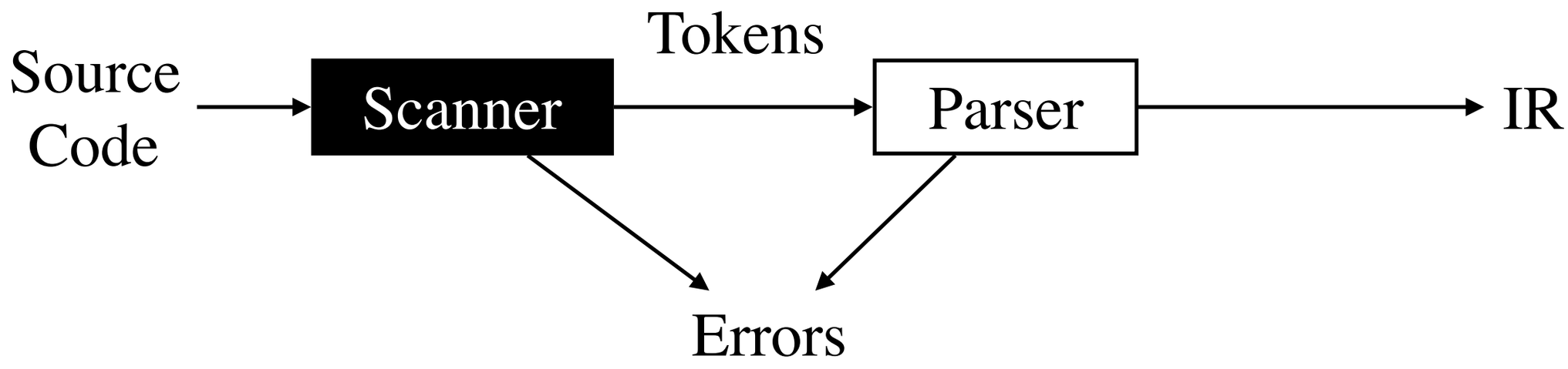



***Front End Responsibilities:***

- 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

# Review: Front End



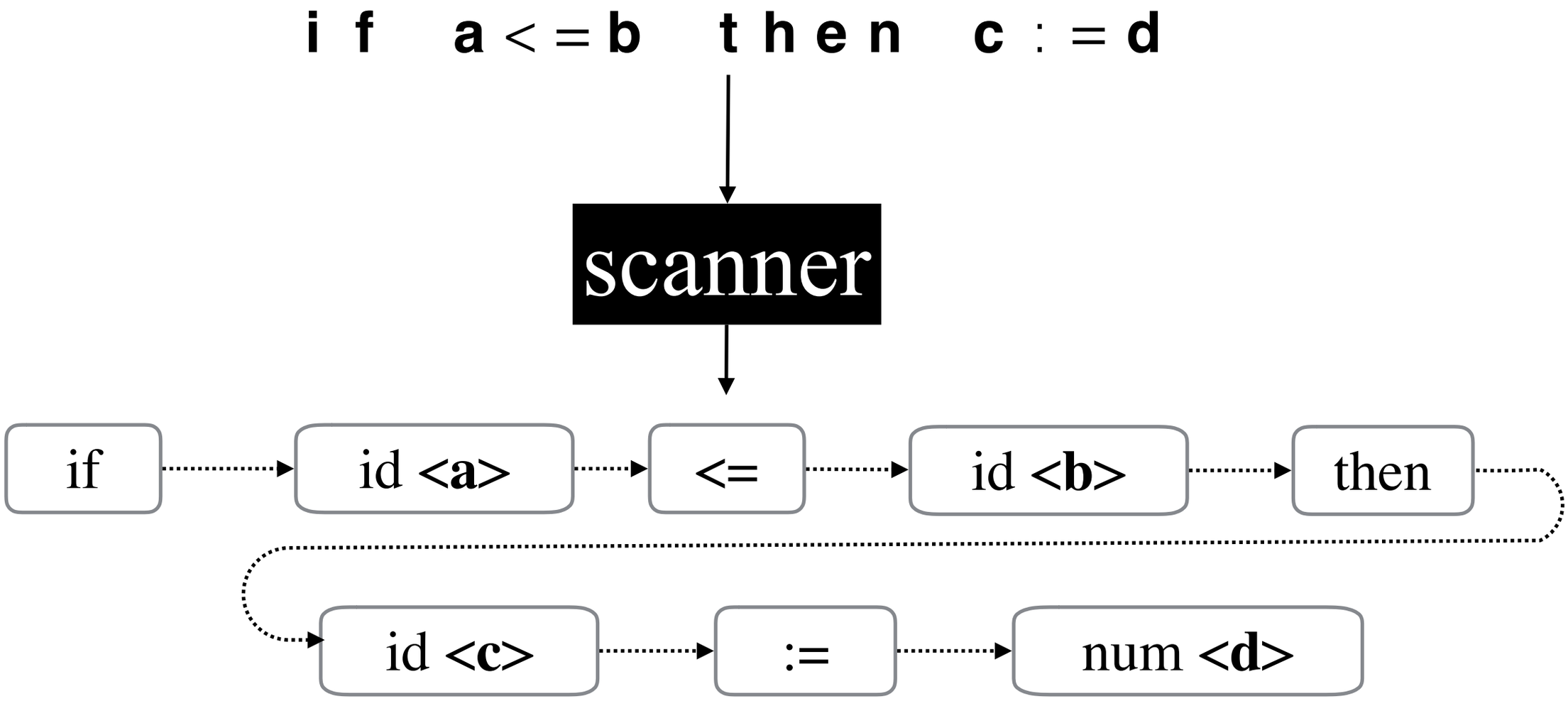

***Front End Responsibilities:***

- 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

# Review: Lexical Analysis ( Scott 2.1, 2.2)

*Character Sequence*




*Token Sequence*

# Review: Tokens ( 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 …)

# Review: 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.

# Review: 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.

# Review: Regular Expressions

A syntax (notation) to specify regular languages.

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

*Either r or s is a regular expression, i.e. **(0|1)*1***

*Any number of r's concatenated.*

# Review: 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** | **{ϵ}** |

*Either r or s is a regular expression, i.e. **(0|1)*1***

*Any number of r's concatenated.*

*A RE can simply be a symbol from the alphabet Σ or an empty symbol **ϵ***

# Review: Examples of Regular Expressions

| 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 |.*

# Review: 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:* letter(letter | digit)$^{*}$

*Real constant:* digit*.digit$^{+}$

# Review: 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.

# Review: 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.

# Review: Rewriting Game

***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?

# Review: Rewriting Game

***Example input:***

**$ 0 1 1 0 1 #**

**0 $ 1 1 0 1 #**

**0 1 & 1 0 1 #**

**0 1 1 $ 0 1 #**

**0 1 1 0 $ 1 #**

**0 1 1 0 1 & #**

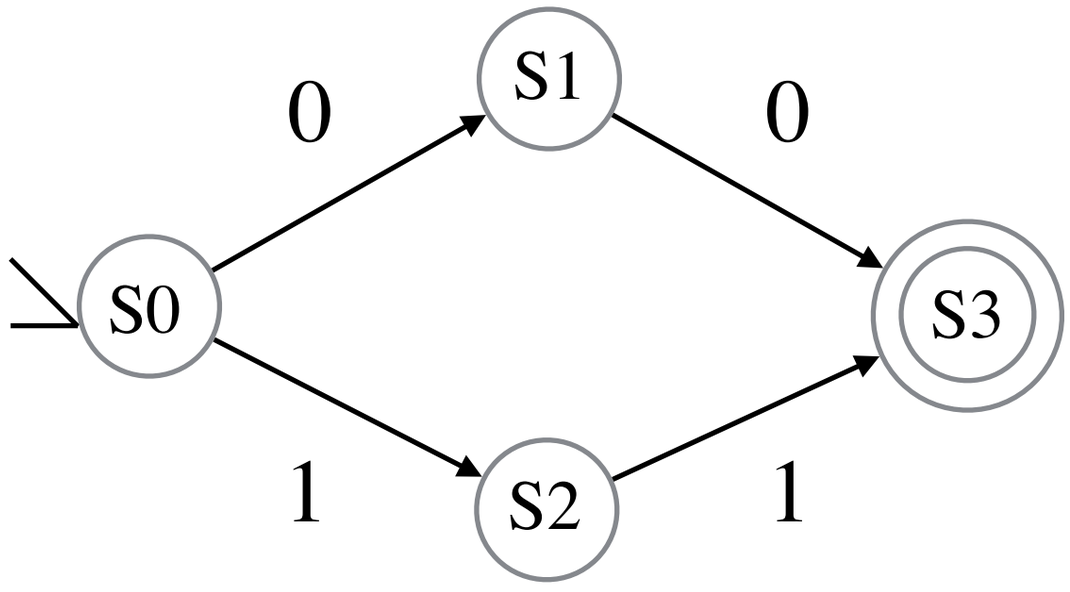
**0 1 1 0 1 →B**

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

Can you predict the outcome of rewriting the following expressions?

| | |
|---|---|
| **$ 0 1 1 1 1 #** | **0 1 1 1 1 →A** |
| **$ 1 0 1 0 0 #** | **1 0 1 0 0 →A** |
| **$ 1 1 0 0 1 #** | **1 1 0 0 1 →B** |

# Reviewer: Finite State Automata




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

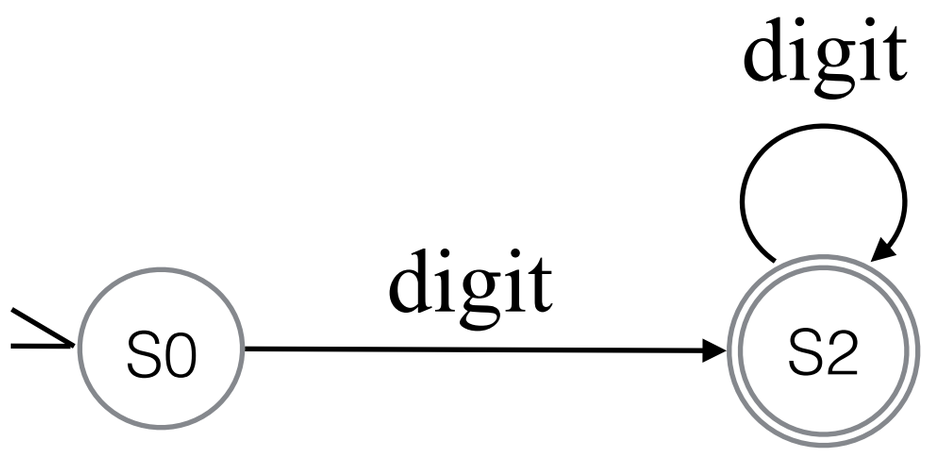
- S is a set of states, e.g., {S0, S1, S2, S3}
- s is the start state, e.g., S0
- F is a set of final states, e.g., {S3}
- T is a set of labeled transitions, of the form (state, input) → state [i.e., $S \times \Sigma \rightarrow S$]

# Review: Recognizers for Regular Expressions

*Example 1:* integer constant

RE: digit$^{+}$

FSA:

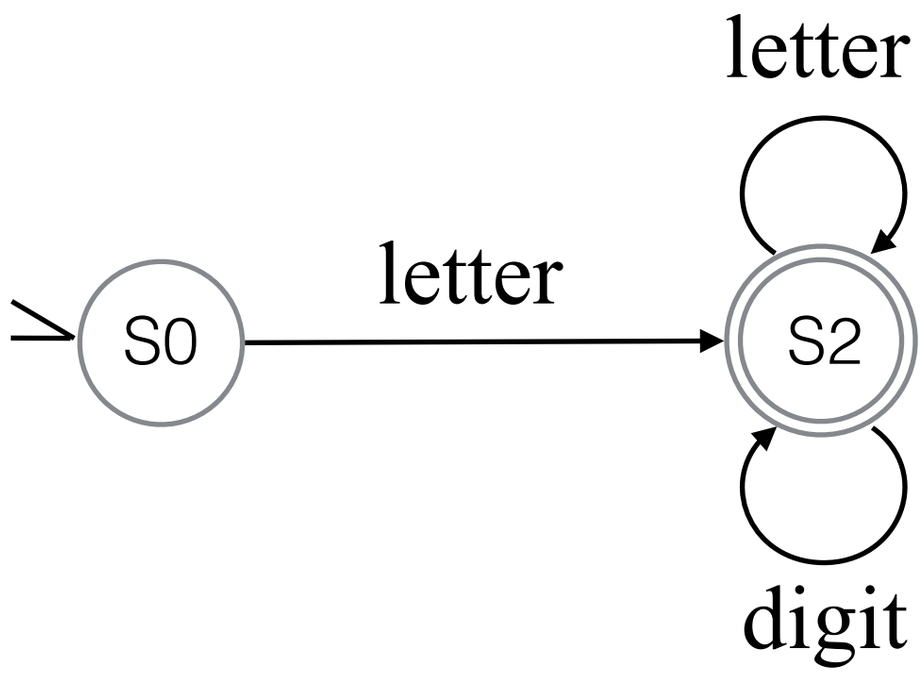


# Review: Recognizers for Regular Expressions(Cont.)

*Example 2:* identifier

RE: letter(letter | digit)*

FSA:

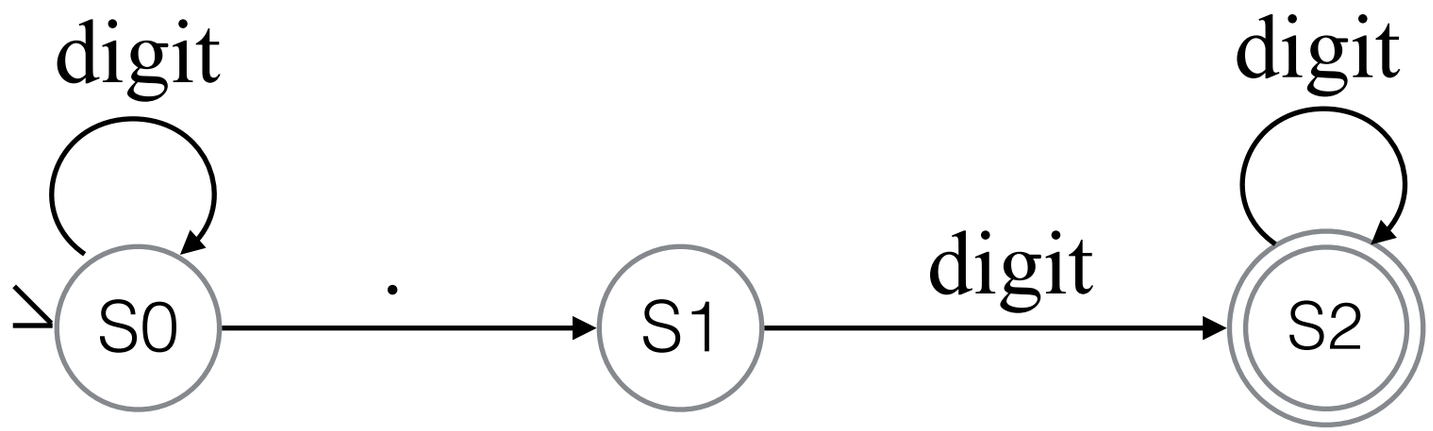


# Review: Recognizers for Regular Expressions(Cont.)

*Example 3:* Real constant

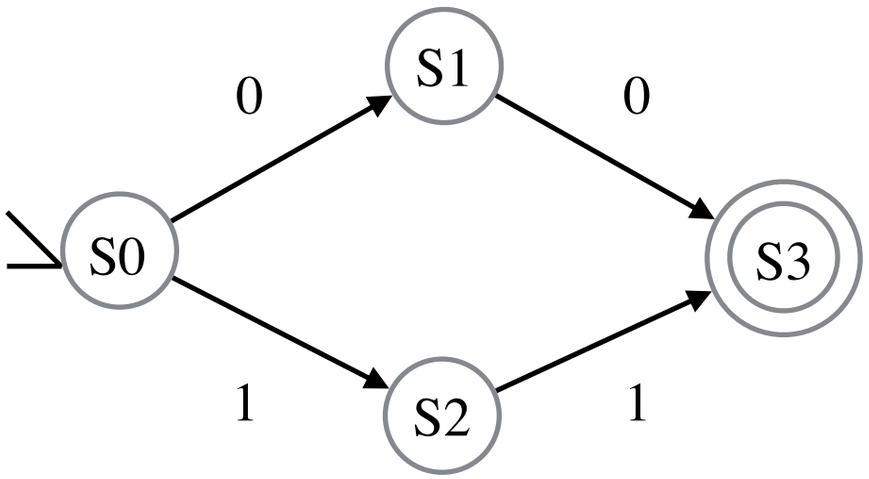
RE: digit*.digit$^{+}$

FSA:



# Review: Finite State Automata

Transitions can be represented using a transition table:

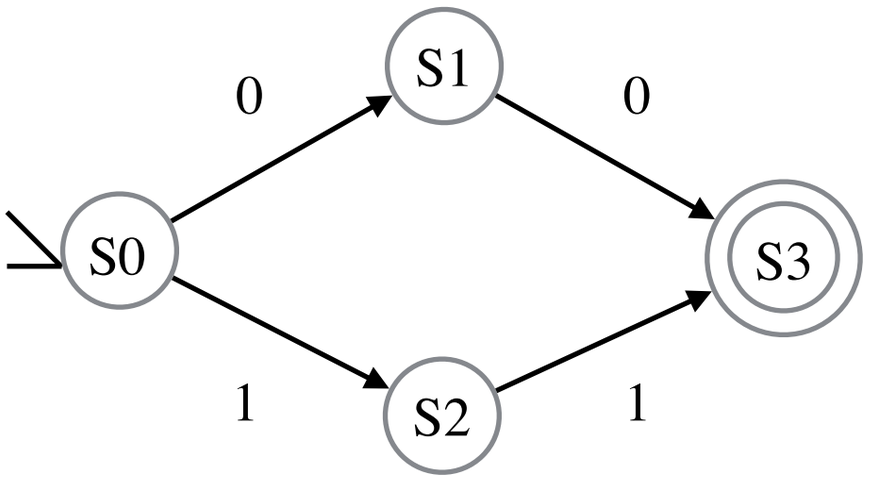



| State | *0* | *1* | Input |
|---|---|---|---|
| *S0* | *S1* | *S2* | |
| *S1* | *S3* | - | |
| *S2* | - | *S3* | |

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.

# Review: Finite State Automata

Transitions can be represented using a transition table:



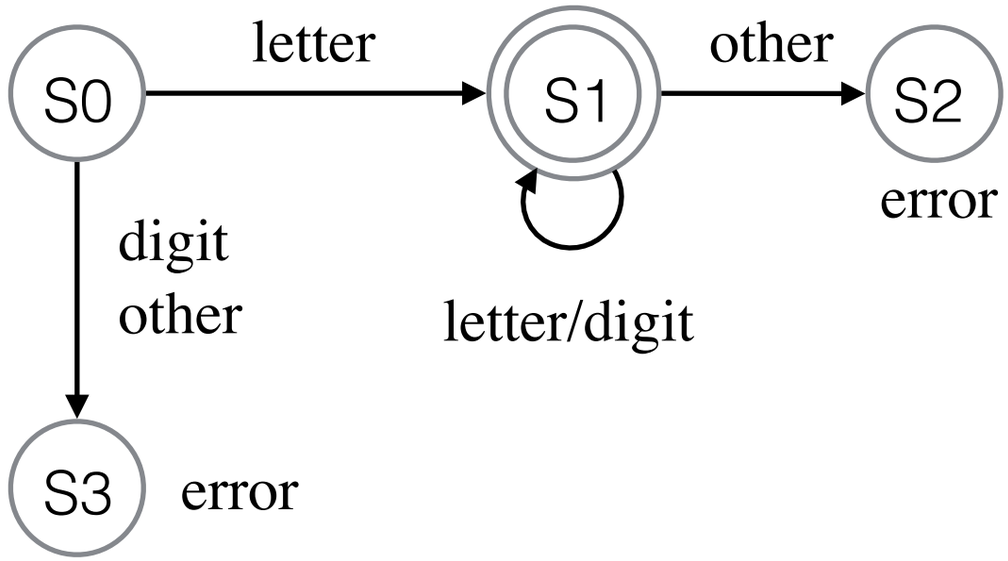

| State | *0* | *1* | Input |
|---|---|---|---|
| *S0* | *S1* | *S2* | |
| *S1* | *S3* | - | |
| *S2* | - | *S3* | |

Error

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

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

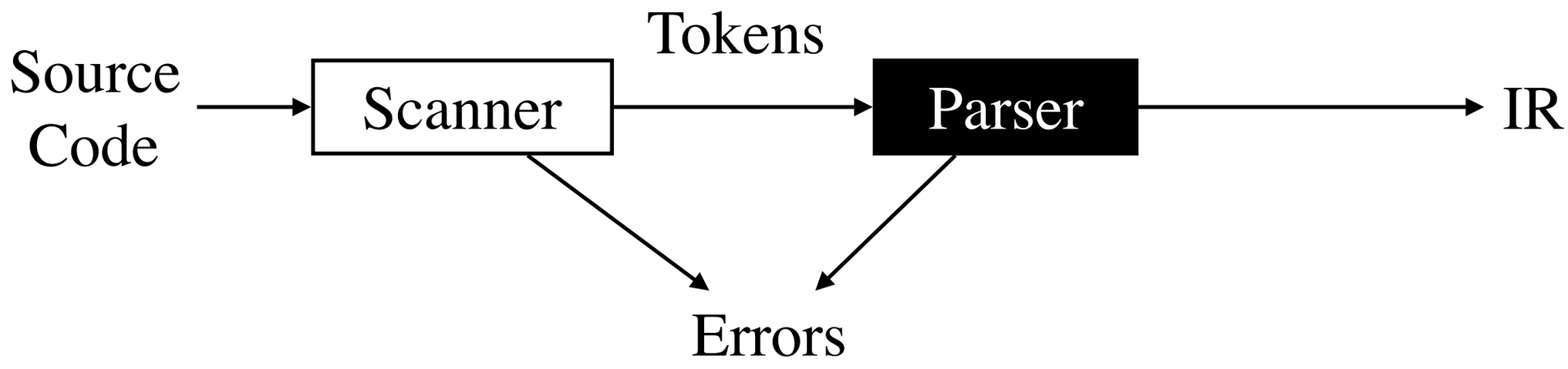
# Review: Implementation: Code for the recognizer




| *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;
```

# Review: Front End

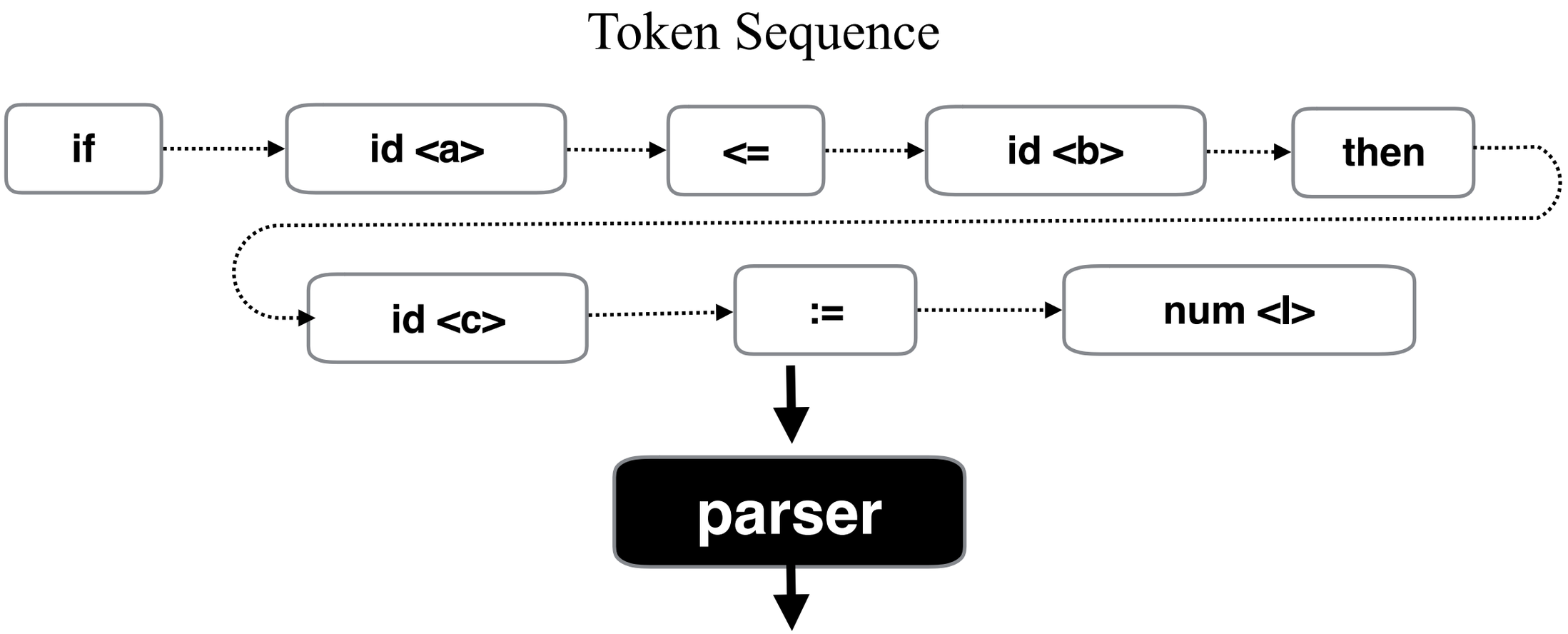



***Front End Responsibilities:***

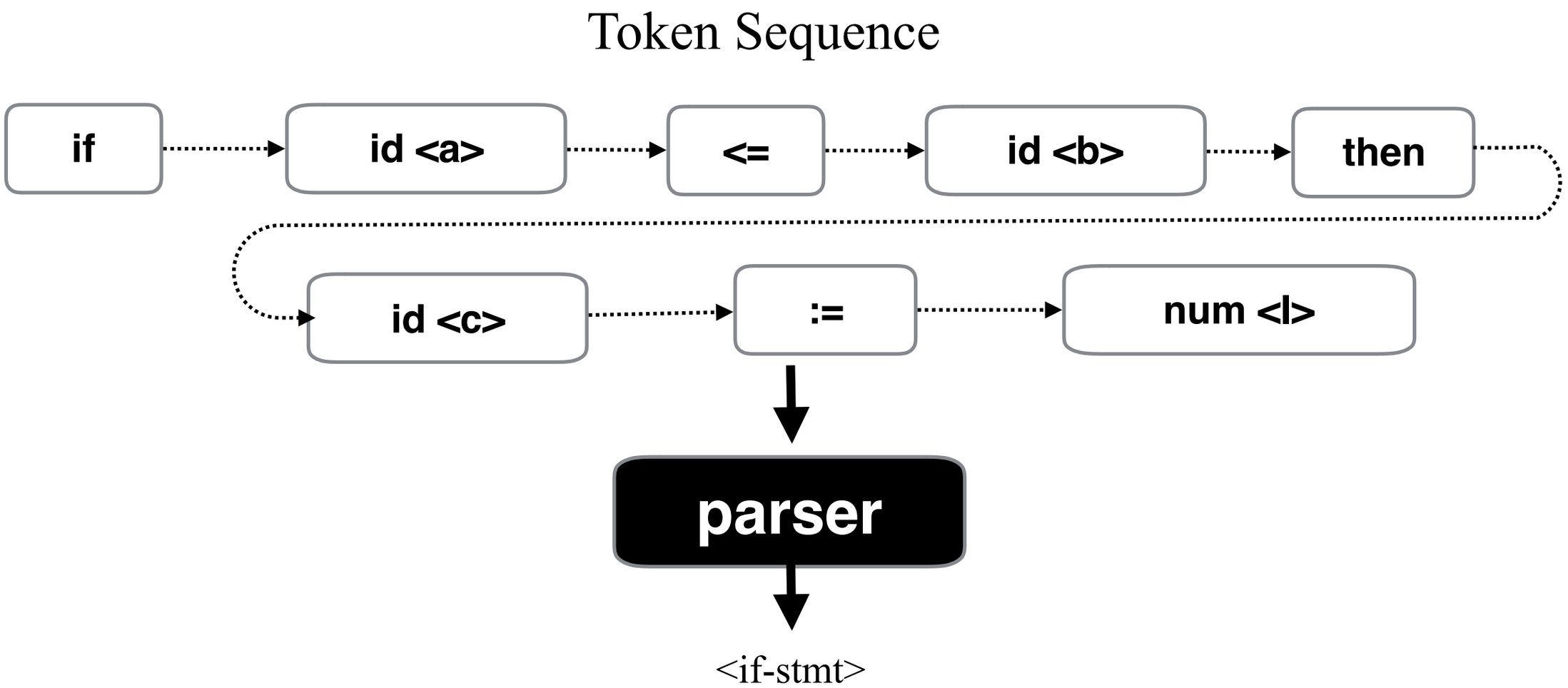
- Recognize legal programs
- Report errors
- Produce IR
- Preliminary storage map
- Shape the code for the back end

Much of front end construction can be automated

# Syntax Analysis ( Scott, Chapter 2.3)

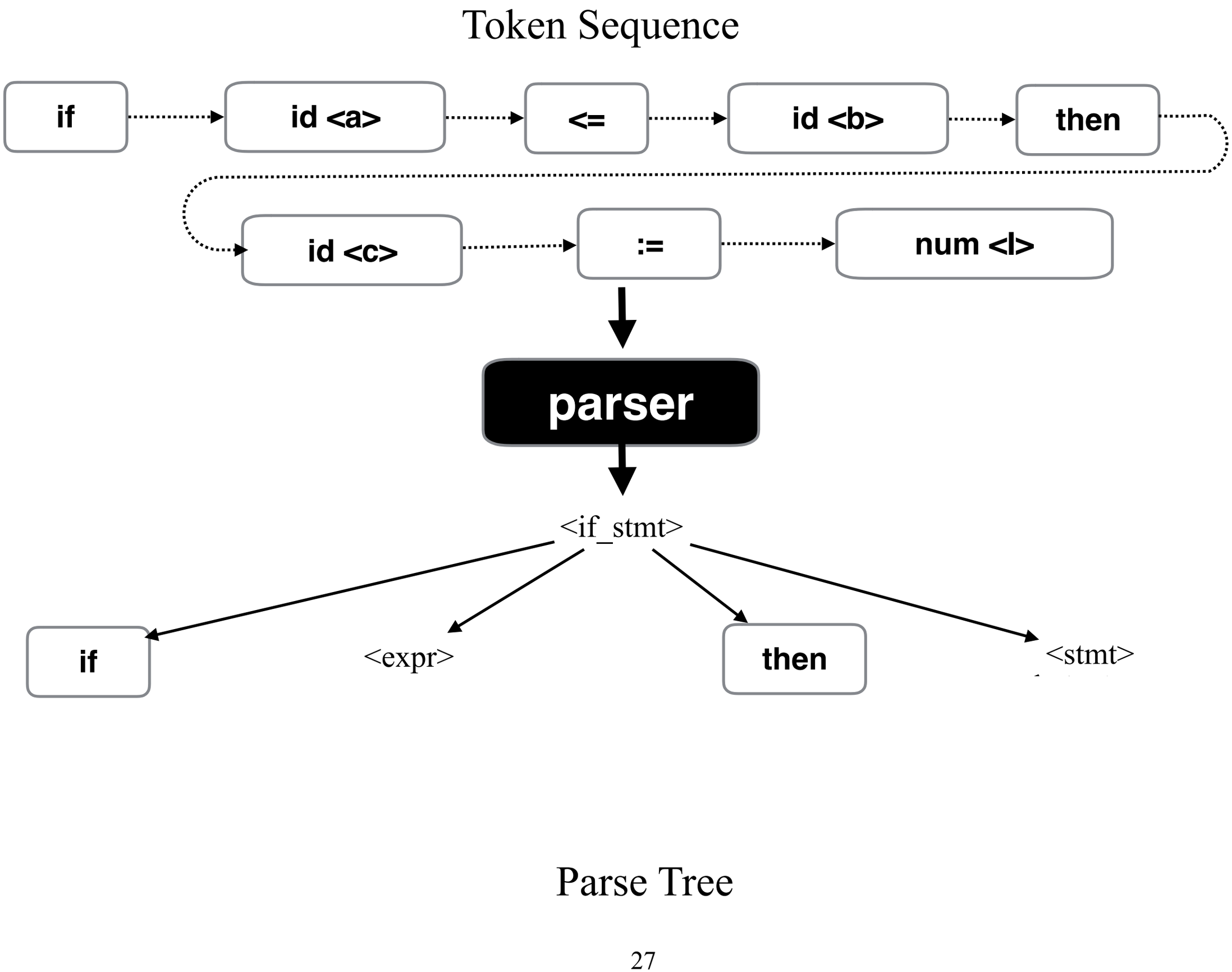


# Syntax Analysis ( Scott, Chapter 2.3)



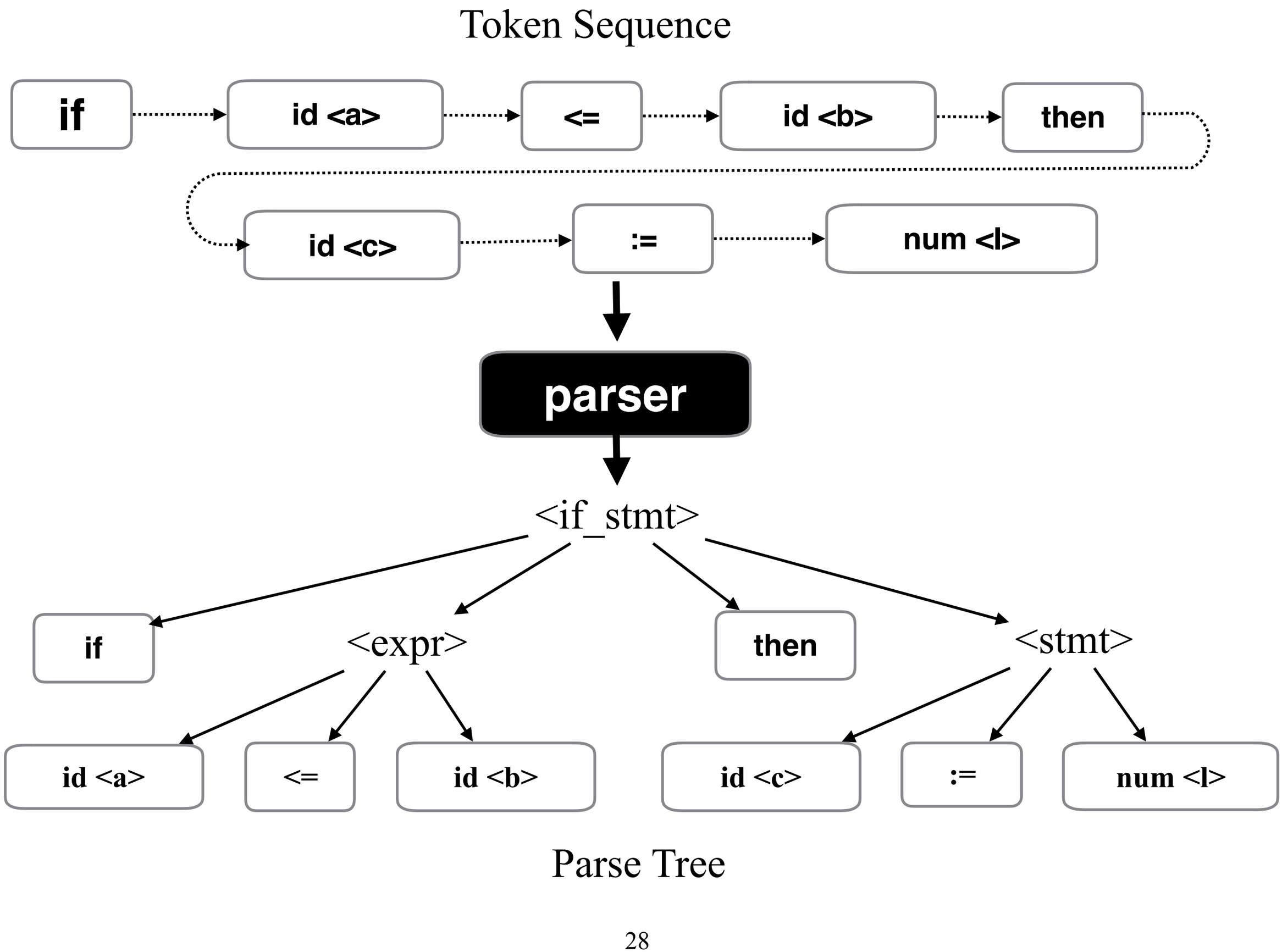

Parse Tree

# Syntax Analysis ( Scott, Chapter 2.3)




Parse Tree

# Syntax Analysis ( Scott, Chapter 2.3)



# Context Free Grammars (CFGs)

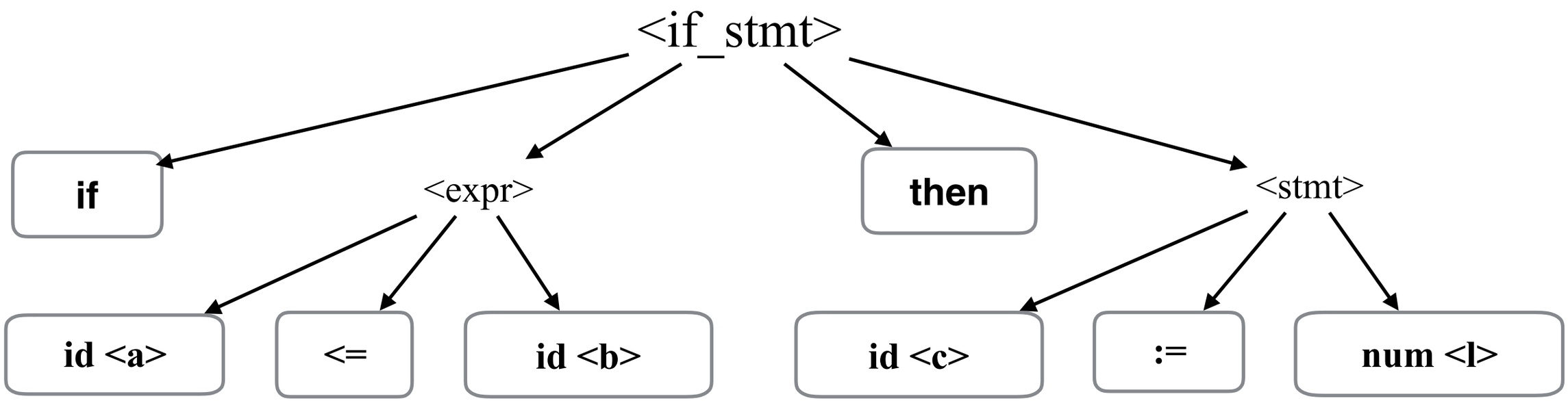
- A formalism to for describing languages
- A CFG $\mathcal{G}$ = < T, N, P, S >:
  1. A set T of terminal symbols (tokens).
  2. A set N of nonterminal symbols.
  3. A set P production (rewrite) rules.
  4. A special start symbol S.
- The language L($\mathcal{G}$) is the set of sentences of terminal symbols in T* that can be derived from the start symbol S:

$$L(\mathcal{G}) = \{w \in T^* \mid S \Rightarrow^* w\}$$

CFGs are rewrite systems with restrictions on the form of rewrite (production) rules that can be used.

# An Example of a Partial Context Free Grammar

<if-stmt> ::= **if** <expr> **then** <stmt>
<expr> ::= **id** <= **id**
<stmt> ::= **id** := **num**




Parse Tree

# Elements of BNF Syntax

Terminal Symbol: **Symbol-in-Boldface**

Non-Terminal Symbol: *Symbol-in-Angle-Brackets*

Production Rule:

Non-Terminal ::= Sequence of Symbols

or

Non-Terminal ::= Sequence | Sequence | Sequence | …

Alternative Symbol: |

Empty String: ε

Example:

…

<if-stmt> ::= **if** <expr> **then** <stmt>

<expr> ::= **id** <= **id**

<stmt> ::= **id** := **num**

# How a BNF Grammar Describes a Language

**A sentence is a sequence of terminal symbols (tokens).**

**The language L($\mathcal{G}$) of a BNF grammar $\mathcal{G}$ is the set of sentences generated using the grammar:**

- Begin with start symbol.
- Iteratively replace non-terminals with terminals according to the rules (rewrite system).

*This replacing process is called a derivation ($\Rightarrow$).*
*Zero or multiple derivation steps are written as $\Rightarrow^*$.*

Formally, $L(\mathcal{G}) = \{w \in T^* \mid S \Rightarrow^* w\}$

# A Complete Simple BNF Grammar

**Terminals** **letters, digits, :=**

**Non-Terminals** <letter> <digit> <identifier> <stmt>

**Productions**

1. < letter > ::= **A** | **B** | **C** | … | **Z**
2. < digit > ::= **0** | **1** | **2** | … | **9**
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > **:=** 0

**Start Symbol** <stmt>

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

In which order to apply the rules?

Typically, there are three options:

- leftmost ($\Rightarrow_L$)
- rightmost ($\Rightarrow_R$)
- any ($\Rightarrow$)

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 $\in$ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

| rightmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_R$ | |

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

| rightmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_R$ | 6 |
| <identifier> := 0 | $\Rightarrow_R$ | |

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

| rightmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_R$ | 6 |
| <identifier> := 0 | $\Rightarrow_R$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_R$ | |

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

| rightmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_R$ | 6 |
| <identifier> := 0 | $\Rightarrow_R$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_R$ | 2 |
| <identifier> 2 := 0 | $\Rightarrow_R$ | |

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

| rightmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_R$ | 6 |
| <identifier> := 0 | $\Rightarrow_R$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_R$ | 2 |
| <identifier> 2 := 0 | $\Rightarrow_R$ | 4 |
| <letter> 2 := 0 | $\Rightarrow_R$ | |

# Derivation in a Grammar ($\mathcal{G}$)

Is X2 := 0 ∈ L($\mathcal{G}$), i.e., can X2 := 0 be derived in $\mathcal{G}$?

| leftmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_L$ | 6 |
| <identifier> := 0 | $\Rightarrow_L$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_L$ | 3 |
| <letter> <digit> := 0 | $\Rightarrow_L$ | 1 |
| X <digit> := 0 | $\Rightarrow_L$ | 2 |
| X2 := 0 | | |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

| rightmost derivation | | rule |
|---|---|---|
| <stmt> | $\Rightarrow_R$ | 6 |
| <identifier> := 0 | $\Rightarrow_R$ | 5 |
| <identifier> <digit> := 0 | $\Rightarrow_R$ | 2 |
| <identifier> 2 := 0 | $\Rightarrow_R$ | 4 |
| <letter> 2 := 0 | $\Rightarrow_R$ | 1 |
| X2 := 0 | | |

# Parsing of a Language L($\mathcal{G}$)

Can we **recognize** X2 := 0 as being in L($\mathcal{G}$)?

| X2 := 0 | Rule |
|---|---|
| <letter> 2 := 0 | 1 |
| <identifier> 2 := 0 | 3 |
| <identifier> <digit> := 0 | 2 |
| <identifier> := 0 | 5 |
| <stmt> | 4 |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

# Parsing of a Language L($\mathcal{G}$)

Can we **recognize** X2 := 0 as being in L($\mathcal{G}$)?

| X2 := 0 | Rule |
|---|---|

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

# Parsing of a Language L($\mathcal{G}$)

Can we **recognize** X2 := 0 as being in L($\mathcal{G}$)?

| X2 := 0 | Rule |
|---|---|
| <letter> 2 := 0 | 1 |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

# Parsing of a Language L($\mathcal{G}$)

Can we **recognize** X2 := 0 as being in L($\mathcal{G}$)?

| X2 := 0 | Rule |
|---|---|
| <letter> 2 := 0 | 1 |
| <identifier> 2 := 0 | 3 |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

# Parsing of a Language L($\mathcal{G}$)

Can we **recognize** X2 := 0 as being in L($\mathcal{G}$)?

| X2 := 0 | Rule |
|---|---|
| <letter> 2 := 0 | 1 |
| <identifier> 2 := 0 | 3 |
| <identifier> <digit> := 0 | 2 |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

# Parsing of a Language L($\mathcal{G}$)

Can we **recognize** X2 := 0 as being in L($\mathcal{G}$)?

| X2 := 0 | Rule |
|---|---|
| <letter> 2 := 0 | 1 |
| <identifier> 2 := 0 | 3 |
| <identifier> <digit> := 0 | 2 |
| <identifier> := 0 | 5 |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

# Parsing of a Language L($\mathcal{G}$)

Can we **recognize** X2 := 0 as being in L($\mathcal{G}$)?

| X2 := 0 | Rule |
|---|---|
| <letter> 2 := 0 | 1 |
| <identifier> 2 := 0 | 3 |
| <identifier> <digit> := 0 | 2 |
| <identifier> := 0 | 5 |
| <stmt> | 4 |

1. < letter > ::= A | B | C | … | Z
2. < digit > ::= 0 | 1 | 2 | … | 9
3. < identifier > ::= < letter > |
4. < identifier > < letter > |
5. < identifier > < digit >
6. < stmt > ::= < identifier > := 0

We will talk about LL(1) grammars and an example parser for a small language (tinyL) that is implemented using mutually recursive procedures (recursive descent parser).

# Next Lecture

Things to do:

- Read Scott, Chapter 2.3 - 2.5 (skip 2.3.3 bottom - up Parsing)