Programming Paradigms

Syntax (Part 3)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

Overview

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars
- **Scanning** 
- **Parsing**

Implementing a Scanner

General idea

- Read one character at a time
- Whenever a full token is recognized, return it
- When no token can be recognized, report an error
- Sometimes, need to look multiple characters ahead to determine next token

Option 1: Ad-hoc Scanners

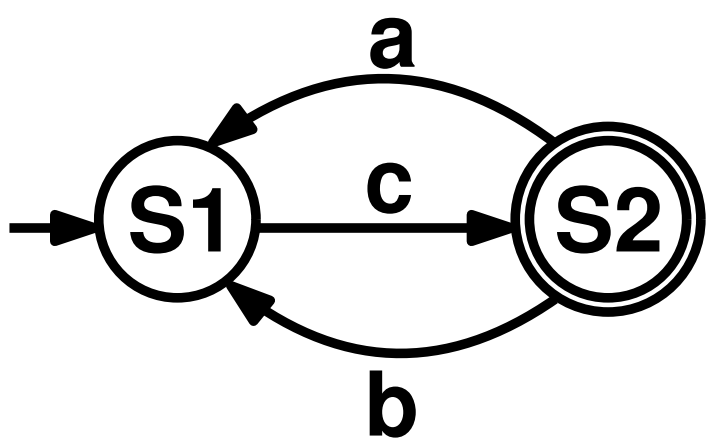
- **Manually implemented**
- **Handle common tokens first**
- **Used in many production compilers**
  - Compact code
  - Efficient scanning

Option 2: Finite Automata

- Each token specified by a regular expression
- Finite automata = Recognizers of regular expressions

Example:

c ((**a** | **b**) **c**)*



Definition: DFA

Deterministic finite automaton (DFA):

$(Q, \Sigma, \delta, q_0, F)$

- Finite set Q of states
- Finite set Σ of input symbols
- Transition function $\delta : Q \times \Sigma \to Q$
- Start state q_0
- Set of accept states $F \subseteq Q$

DFA versus NFA

- **Deterministic finite automaton (DFA):**
  - At most one outgoing transition for each input symbol
  - No ϵ transitions (empty word)
- **Non-deterministic finite automaton (NFA)**
  - Multiple outgoing transitions for same character
  - May have ϵ transitions

From Reg. Expr. to DFA

- **Regular expression to NFA**
- **NFA to DFA**
  - To avoid exploring multiple possible next states during scanning
- **DFA to minimal DFA**
  - Simplifies a DFA-based scanner
  - Remove unreachable and non-distiguishable states

See course on theoretical computer science or Chapter 2 of "Programming Language Pragmatics" for details

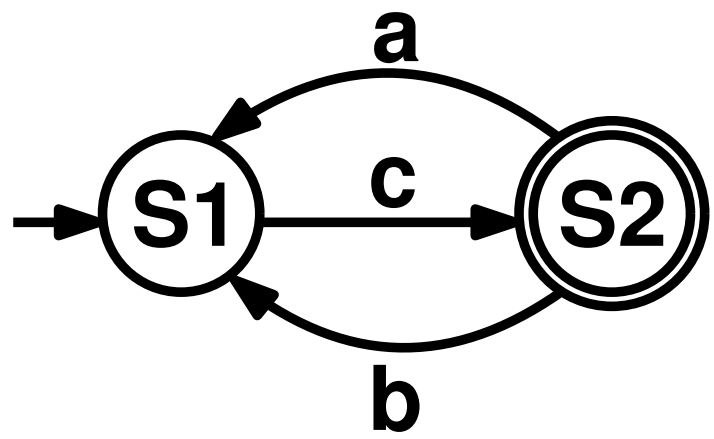
From DFA to Scanner

Two popular options

- Implement the DFA using switch statements
  - Mostly in hand-written scanners
- Table-based scanners
  - Table represents states and transitions
  - Driver program indexes the table
  - Mostly in auto-generated scanners

Switch Statement Style

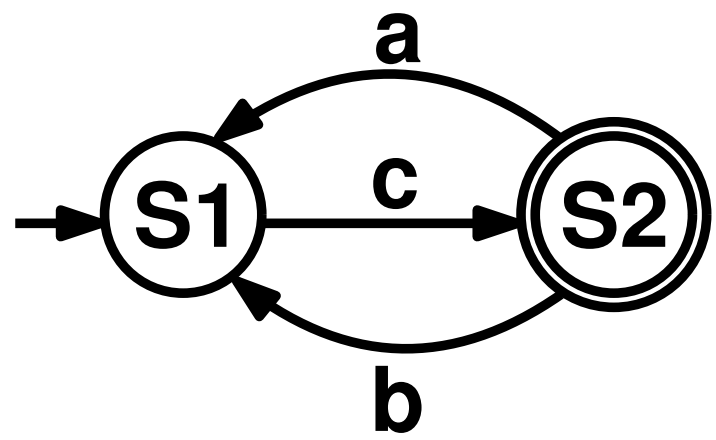
```
state = S1
```



Starting state: S1

Switch Statement Style

```
state = S1
token = ""
loop:
```

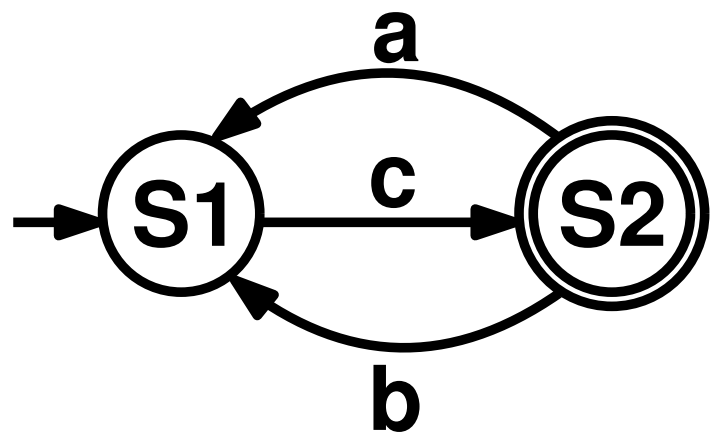


Loop reads one character at a time and builds the token

```
token = token + in_char
read next in_char
```

Switch Statement Style

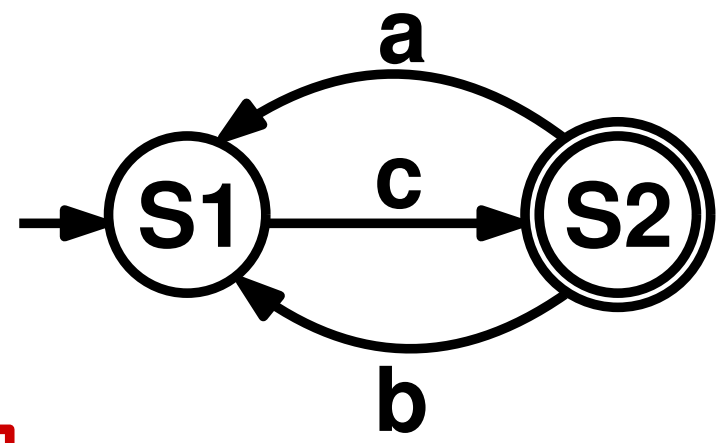
```
state = S1
token = ""
loop:
  switch state:
    case S1:


    case S2:


  token = token + in_char
  read next in_char
```



Switch statement that handles the current state

Switch Statement Style

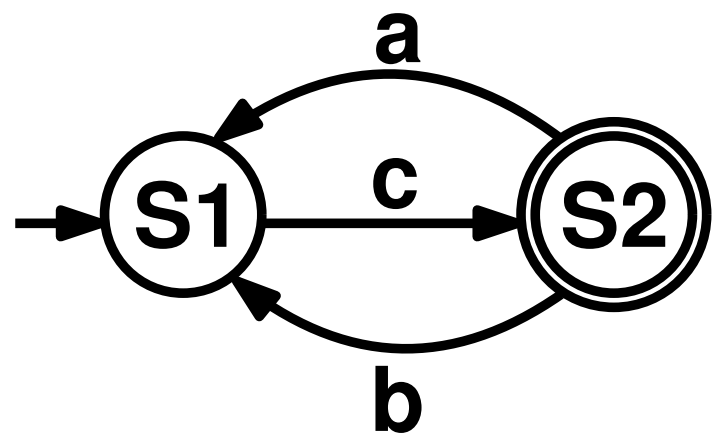
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char
```



Switch statements to handle the current character

Switch Statement Style

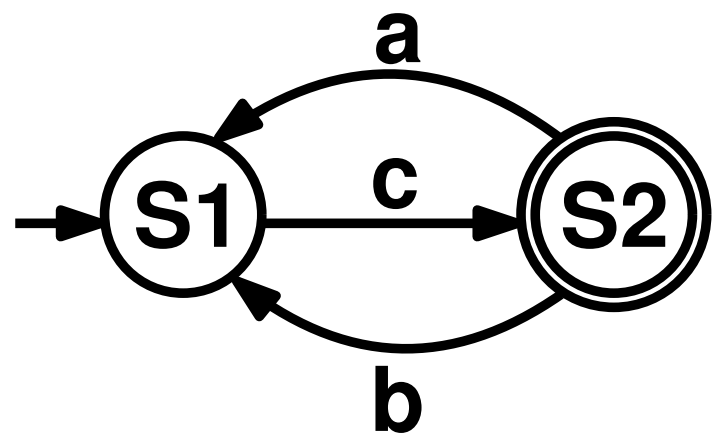
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char
```



Move to next state if character accepted

Switch Statement Style

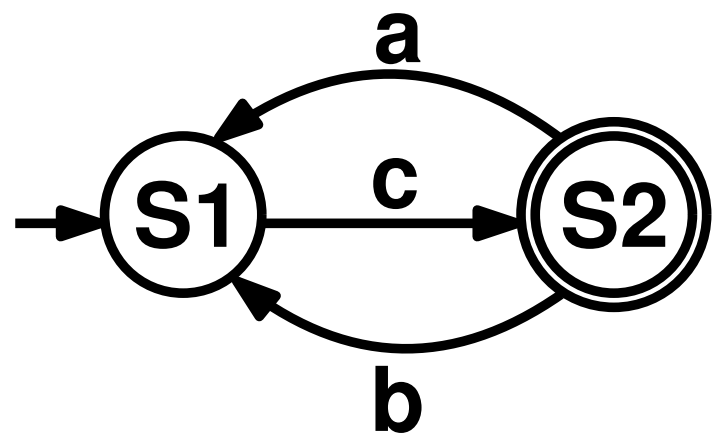
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char
```



Return the token when a space occurs

Switch Statement Style

```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char
```

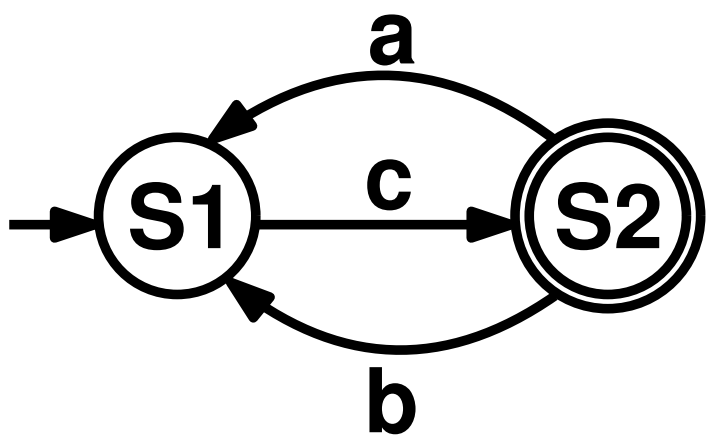


Raise an error for any illegal character

Table-based Scanning

Transition table indexed by state and input:

State	'a'	'b'	'c'	Return
S1	-	-	S2	-
S2	S1	S1	-	token



Driver program

- moves to a new state,
- returns a token, or
- raises an error

Longest Possible Token Rule

- **What if one token is a prefix of another?**
  - Number `3.1` vs. number `3.141`
- **Accept the longest possible token**
  - `3.141` for the above example
- **How to decide whether token has ended?**
  - Scanner looks ahead (at least one character)

Quiz: Automata and Scanners

Which of these statements is true?

- A scanner produces a sequence of tokens.
- A scanner produces a syntax tree.
- Efficient scanners are based on NFAs.
- A scanner for C will turn ”ifStmt” into two tokens ”if” and ”Stmt”.

Please vote in Ilias.

Quiz: Automata and Scanners

Which of these statements is true?

- A scanner produces a sequence of tokens.
- ~~A scanner produces a syntax tree.~~
- ~~Efficient scanners are based on NFAs.~~
- ~~A scanner for C will turn "ifStmt" into two tokens "if" and "Stmt".~~

Please vote in Ilias.