Programming Paradigms
Lecture 2: Syntax (Part 1)

Join the course on Ilias! See link on
http://software-lab.org/teaching/winter2019/pp/

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2019/2020

Wake-up Exercise

What does the following C code print?

```
#include <stdio.h>

int main() {
    int arr[3] = {5, 42, 23};
    int n = 2[arr];
    printf("%d", n);
}
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following C code print?

Result: 23

```
#include <stdio.h>

int main() {
    int arr[3] = {5, 42, 23};
    int n = 2[arr];
    printf("%d", n);
}
```

https://ilias3.uni-stuttgart.de/vote/0ZT9

Wake-up Exercise

What does the following C code print?

Result: 23

```
#include <stdio.h>

int main() {
    int arr[3] = {5, 42, 23};
    int n = 2[arr];
    printf("%d", n);
}
```

Same as arr[2]:

Computes a pointer `*(arr + 2)` and returns value stored at the address

https://ilias3.uni-stuttgart.de/vote/0ZT9

Motivation

- **Goal: Specify a programming language**
  - What code is part of the language?
  - What is the meaning of a piece of code?
- **Important for both developers and tools**
- **In contrast: Natural languages not formally specified**

Syntax vs. Semantics

Structure of code

Meaning of code

Example:

Grammar to define a language:

digit $\rightarrow$ **0** | **1** | **...** | **9**

non_zero_digit $\rightarrow$ **1** | **...** | **9**

number $\rightarrow$ **non_zero_digit digit***

Could mean

- Natural numbers
- Days of a 10-day week
- Colors
- ...

Syntax vs. Semantics

Structure of code

Meaning of code

Example:

Grammar to define a language:

digit $\rightarrow$ **0** | **1** | **...** | **9**

non_zero_digit $\rightarrow$ **1** | **...** | **9**

number $\rightarrow$ **non_zero_digit digit***

Focus of this and next lecture

Could mean

- Natural numbers
- Days of a 10-day week
- Colors
- ...

Syntax of Different PLs

Common: Different syntax, same semantics

Java:

```
if (foo > 100) {
  ...
}
```

Bash:

```
if [ $foo -gt 100 ]
then
  ...
fi
```

Syntax of Different PLs

Common: Different syntax, same semantics

Java:

```
if (foo > 100) {
   ...
}
```

Bash:

```
if [ $foo -gt 100 ]
then
   ...
fi
```

Sometimes: Same syntax, different semantics

Java:

Type error at compile time

JavaScript:

Branch is executed

```
if ("abc" != 5) {
   ...
}
```

4 concepts to specify syntax

concatenation

alternation / choice

repetition ("Kleene closure")

recursion

regular expressions
↳ recognized by scanners

context-free grammars (CFG)
↳ recognized by parsers

Plan for Today

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

Tokens

Basic building blocks of every PL

- Keywords, identifiers, constants, operators
- Think: ”Words” of the language

Example: C has more than 100 tokens

- Keywords, e.g., `double`, `if`, `return`, `struct`
- Identifiers, e.g., `my_var`, `printf`
- Literals, e.g., `6.022e23`, `'x'`
- Punctuators, e.g., `(`, `}`, `&&`

Regular Expressions

- **Used to specify tokens**
- **A regular expression is one of:**
  - A character
  - The empty string ϵ
  - The concatentation of two regular expressions
  - Two regular expressions separated by |
    - Means a string generated by one or the other
  - A reg. expression followed by the Kleene star *
    - Means zero or more repetitions

Regular Expressions

- **Used to specify tokens**
- **A regular expression is one of:**
  - A character
  - The empty string ϵ
  - The concatentation of two regular expressions

    No recursion!
  - Two regular expressions separated by |
    - Means a string generated by one or the other
  - A reg. expression followed by the Kleene star *
    - Means zero or more repetitions

Example

Numeric constants accepted by a calculator

number → integer | real

integer → digit digit*

real → integer exponent | decimal (exponent | ε)

decimal → digit* (. digit | digit.) digit*

exponent → (e | E) (+ | − | ε) integer

digit → 0 | 1 | 2 ... | 9

Quiz

Which of the following strings is accepted by the regular expression *number*?

https://ilias3.uni-stuttgart.de/vote/0ZT9

More compact syntax f. regular expressions

Language of tokens: { c, cac, cbc, cacac, cbcbc, cacbc, cbcac, etc. }

Token → c More*

More → AorB c

AorB → a | b

Shorter notation: Nest all in one

↳ $\underbrace{c\,\underbrace{(\underbrace{a|b}_{AorB})\,c}_{More}}_{Token}{}^{*}$

Identifiers in Popular PLs

Different PLs allow different identifiers

- Case-sensitive vs. case-insensitive
  - E.g., `foo`, `Foo`, and `FOO` are the same in Ada and Common Lisp, but not in Perl and C
- Letters and digits: Almost always allowed
- Underscore: Allowed in most languages

In addition to syntax rules: Conventions

- E.g., Java: `ClassName`, `variableName`

Identifiers in Popular PLs

Different PLs allow different identifiers

- Case-sensitive vs. case-insensitive
  - E.g., `foo`, `Foo`, and `FOO` are the same in Ada and Common Lisp, but not in Perl and C
- Letters and digits: Almost always allowed
- Underscore: Allowed in most languages

In addition to syntax rules: Conventions

- E.g., Java: `ClassName`, `variableName`

Know the rules of the language you use!

White space in Popular PLs

Free format vs. formatting as syntax

- Spaces and tabs sometimes matter
  - E.g., in Python
- Line breaks sometimes matter
  - E.g., to separate statements in JavaScript or Python

Demo

[demos/whitespace: show both stmts on one line; insert semi-colon; show not indenting print (after inverting condition)]

Are regular expressions enough?

Specifying arithmetic expressions

E.g., $\boxed{5+7}$

$\boxed{(\boxed{5+7})+6}$

$\boxed{((\boxed{5+7})+6)-23}$

Each $\boxed{}$ is an arithmetic expression

→ Want: Nesting

→ Need recursion

Context-free Grammars

$\approx$ Regular expressions + Recursion

Example: Arithmetic expressions

expr $\rightarrow$ `id` | `number` | **expr op expr** | `(` **expr** `)`

op $\rightarrow$ `+` | `-` | `*` | `/`

Context-free Grammars

≈ **Regular expressions + Recursion**

Example: Arithmetic expressions

expr → `id` | `number` | **expr op expr** | `(` **expr** `)`

op → `+` | `-` | `*` | `/`

Non-terminals

Context-free Grammars

≈ **Regular expressions + Recursion**

Example: Arithmetic expressions

expr → `id` | `number` | **expr op expr** | `(` **expr** `)`

op → `+` | `-` | `*` | `/`

Non-terminals

Terminals = tokens of the PL

Context-free Grammars

≈ **Regular expressions + Recursion**

Example: Arithmetic expressions

expr → `id` | `number` | **expr op expr** | `(` **expr** `)`

op → `+` | `-` | `*` | `/`

Recursion allows for nesting expressions

Definition of CFGs

$G = (N, T, R, s)$

N .. finite set of non-terminals

T .. finite set of terminals

= alphabet of language

= (for PLs) tokens of language

R .. finite relation from N to $(N \cup T)^*$

= production rules

s .. start symbol

Extensions of basic definition

- Kleene star
  - ↳ E.g., id_list → id (, id)* ← means zero or more
    - ↳ Short-hand for

      id_list → id

      id_list → id_list, id

- Kleene plus
  - ↳ Same, but one or more

- Vertical bar
  - ↳ E.g., op → + | - | * | /
    - ↳ Short-hand for

      op → +

      op → -

      ⋮

Derivations

Create concrete strings from the grammar

- Begin with start symbol
- Repeat until no non-terminals remain:
  - Choose production with a remaining non-terminal on the left-hand side
  - Replace it with right-hand side of the production (choose one option if multiple options)

<u>Example:</u>

expr → id | number | expr | (expr) | expr op expr

op → + | − | * | /

Derivation of foo * x + bar

"derives"

$$
\begin{aligned}
\text{expr} &\Longrightarrow \text{expr op } \underline{\text{expr}} \\
&\Longrightarrow \text{expr } \underline{\text{op}} \text{ id} \\
&\Longrightarrow \underline{\text{expr}} + \text{id} \\
&\Longrightarrow \text{expr op } \underline{\text{expr}} + \text{id} \\
&\Longrightarrow \text{expr } \underline{\text{op}} \text{ id} + \text{id} \\
&\Longrightarrow \underline{\text{expr}} * \text{id} + \text{id} \\
&\Longrightarrow \text{id} * \text{id} + \text{id}
\end{aligned}
$$

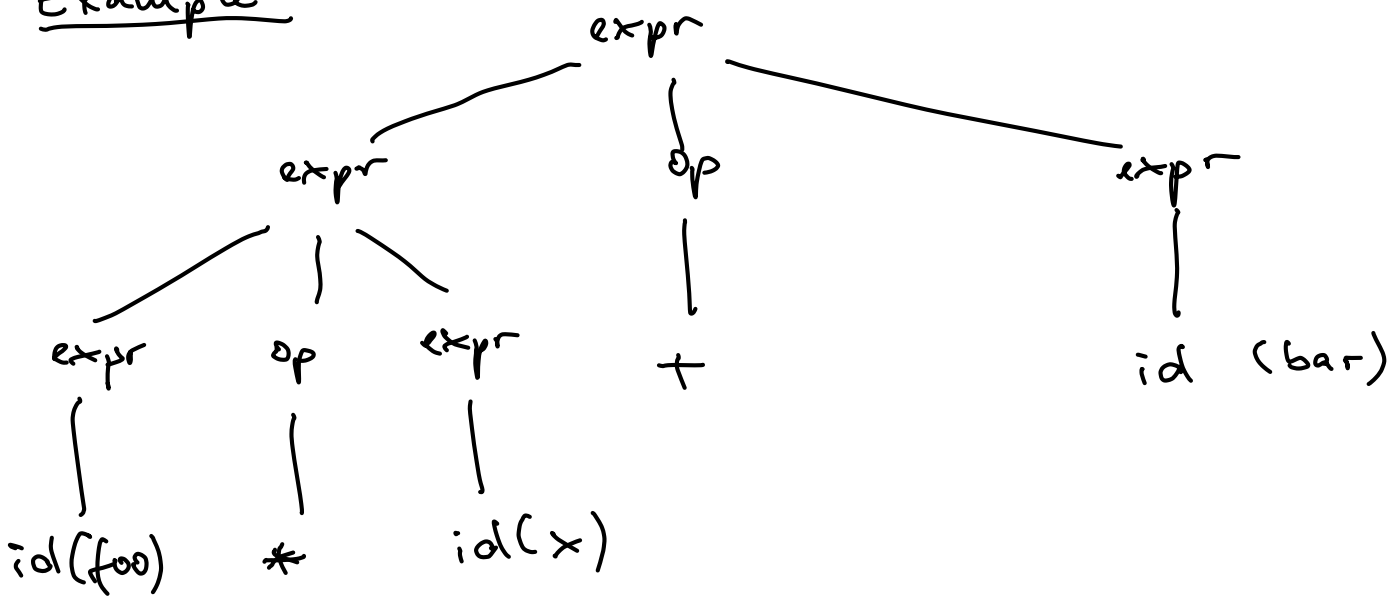
foo x bar

Parse Trees

Tree-structured representation of a derivation

- Root = Start symbol
- Leaf nodes = Tokens that result from derivation
- Intermediate nodes = Application of a production

Example



Not All Grammars are Equal

Each language has infinitely many grammars

Some grammars are ambiguous

- A single string may have multiple derivations
- Unambiguous grammars facilitate parsing

Grammar should reflect the internal structure of the PL

- E.g., associativity and precedence of operators

Example: Revised Grammar

A better version of the grammar of arithmetic expressions:

expr $\rightarrow$ **term** | **expr add_op term**

term $\rightarrow$ **factor** | **term mult_op factor**

factor $\rightarrow$ `id` | `number` | **–** **factor** | `(` **expr** `)`

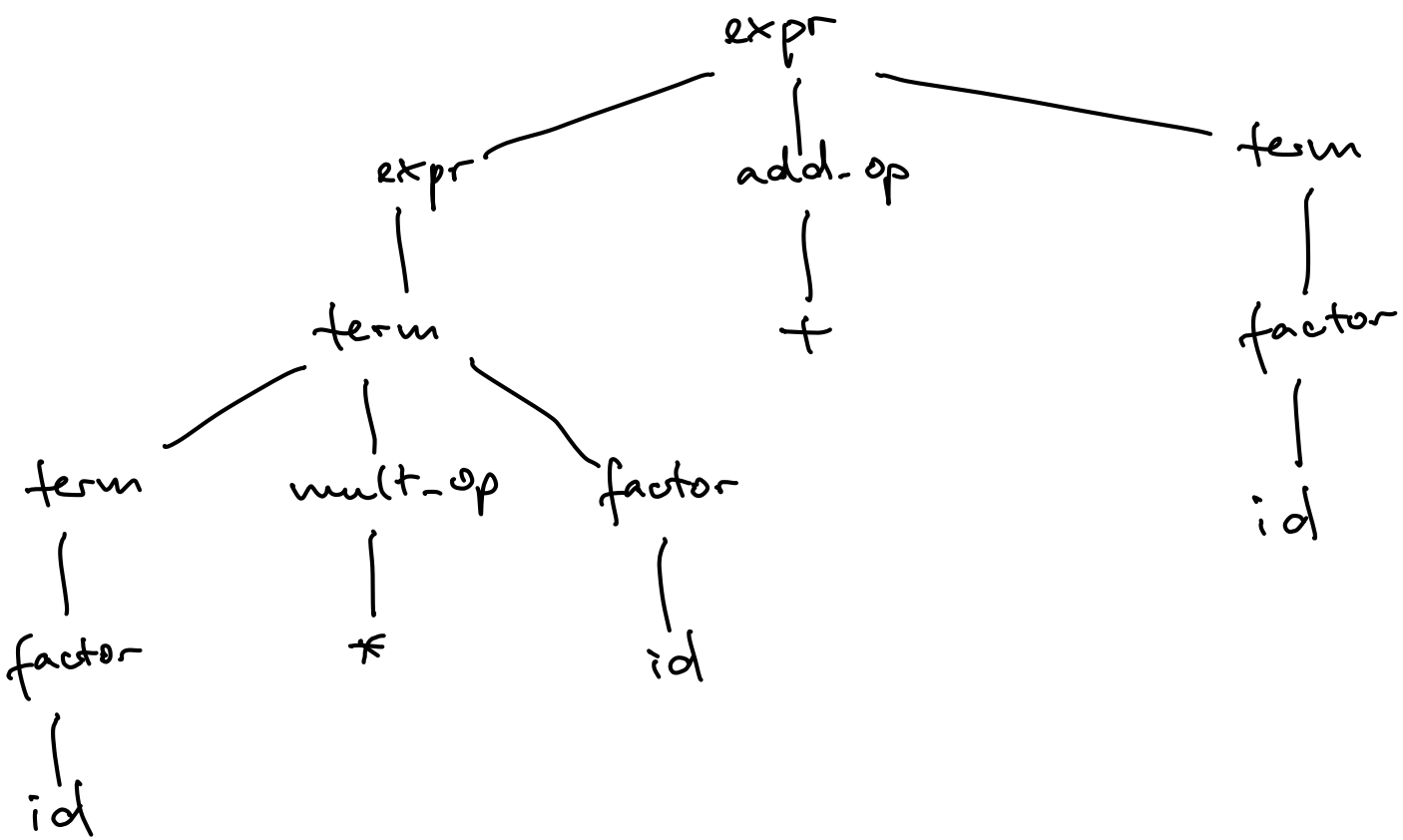
add_op $\rightarrow$ **+** | **–**

mult_op $\rightarrow$ `*` | `/`

Quiz: Context-free Grammars

Draw the parse tree of `foo * x + bar` with the revised grammar. How many nodes and edges does the tree have?

https://ilias3.uni-stuttgart.de/vote/0ZT9


expr
expr
add_op
term
term
+
factor
term
mult_op
factor
id
factor
*
id
id

Plan for Today

- **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

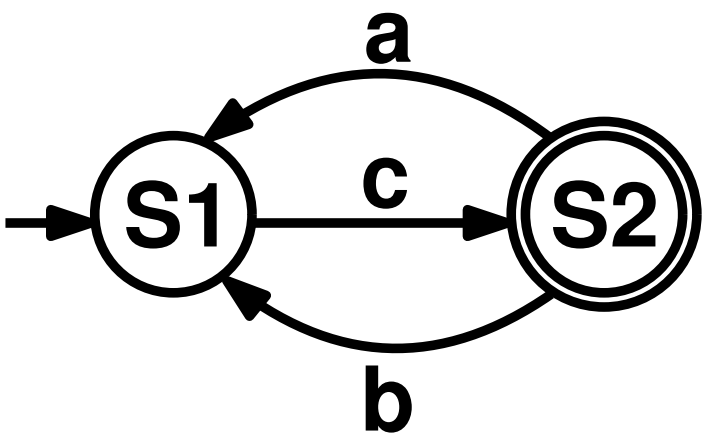
- **Manually implemented**
- **Handle common tokens first**
- **Used in many production compilers**
  - Compact code
  - Efficient scanning

Option 2: Finite Automata

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

Example:

$\mathbf{c}\ (\ (\ \mathbf{a} \mid \mathbf{b}\ )\ \mathbf{c}\ )*$



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. Expressions to DFAs

See course on theoretical computer science or Chapter 2 of *Programming Language Pragmatics*

From NFA to Minimal DFA

- **NFA to DFA**
  - Why? To avoid exploring multiple possible next states during scanning
  - How? Powerset construction method (not covered here)
- **DFA to minimal DFA**
  - Why? Simplifies a DFA-based scanner
  - How? Remove unreachable and non-distiguishable states

From DFA to Scanner

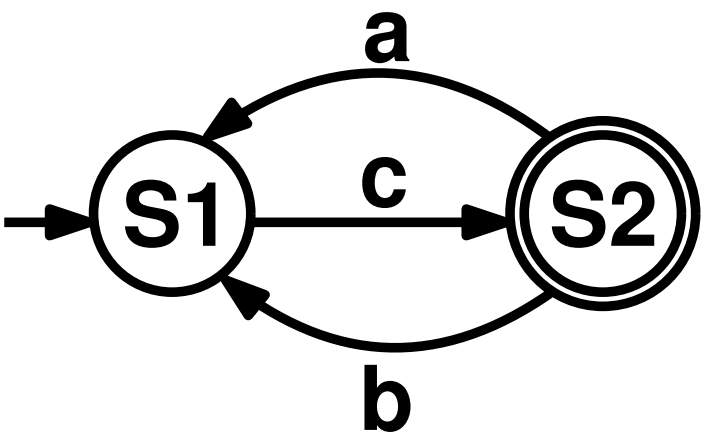
Two popular options

- 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

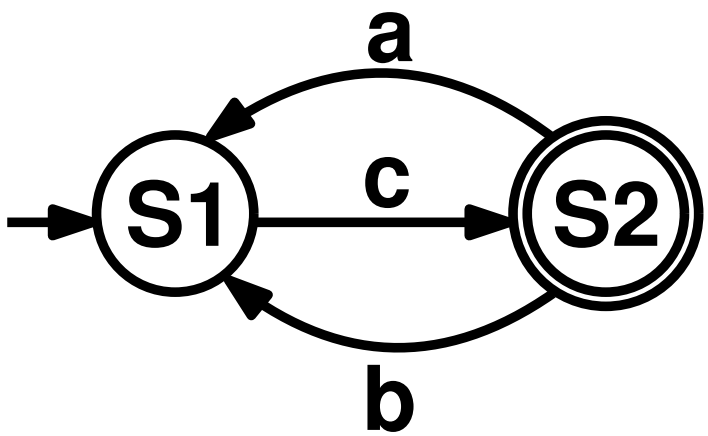
```
state = S1
```

Starting state: S1



Switch Statement Style

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

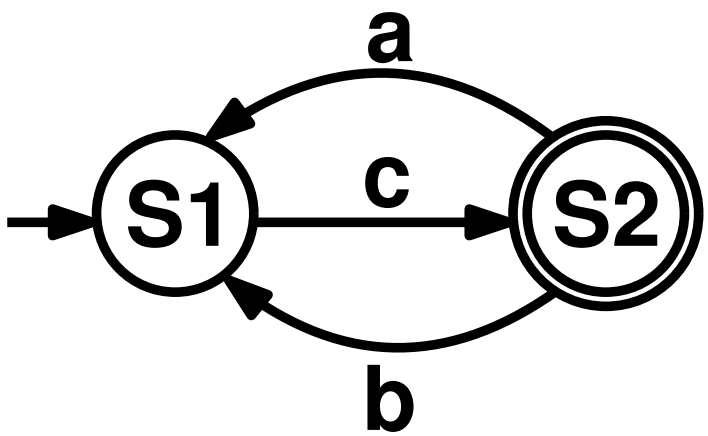


Loop reads one character at a time and builds the token

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

Switch Statement Style

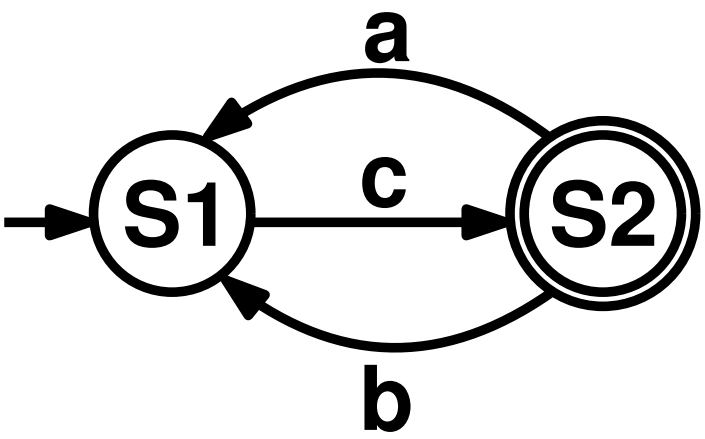
```
state = S1
token = ""
loop:
  switch state:
    case S1:


    case S2:


  token = token + in_char
  read next in_char
```



Switch statement that handles the current state

Switch Statement Style

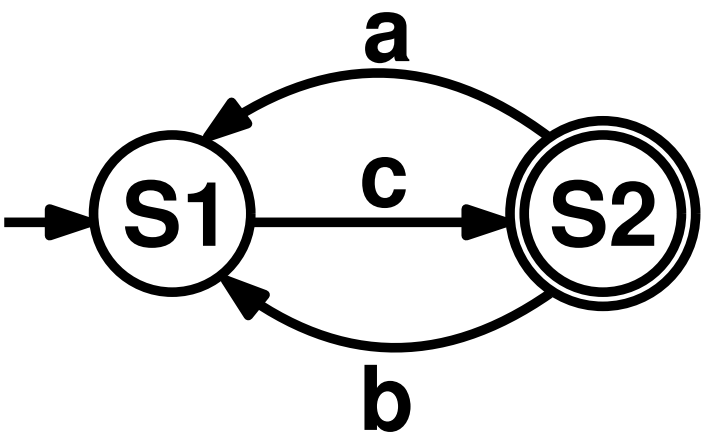
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read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



Switch statements to handle the current character

Switch Statement Style

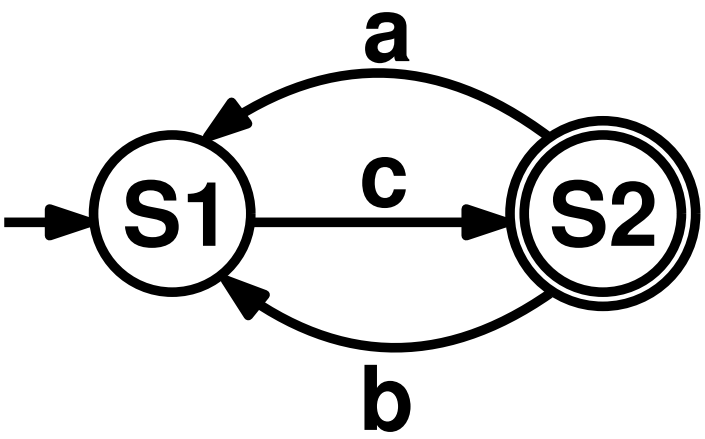
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read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



Move to next state if character accepted

Switch Statement Style

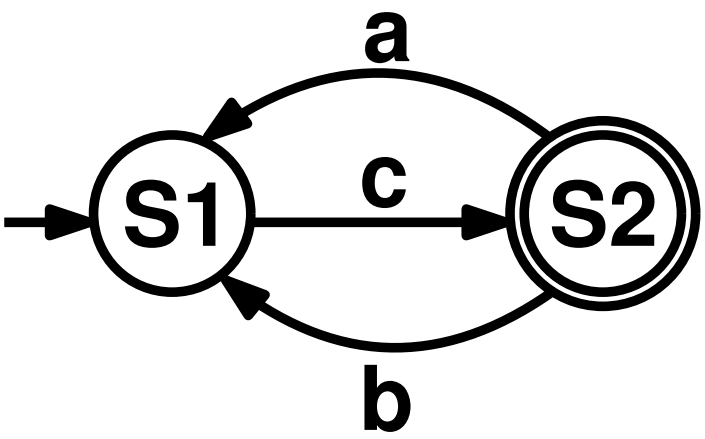
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read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

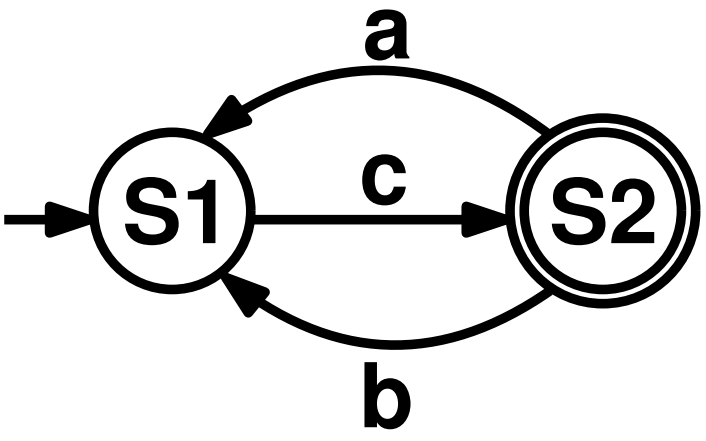


Raise an error for any illegal character

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”.

https://ilias3.uni-stuttgart.de/vote/0ZT9

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".~~

https://ilias3.uni-stuttgart.de/vote/0ZT9

Plan for Today

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