Lexical Analysis

Outline

1. Scanning Tokens
2. Regular Expressions
3. Finite State Automata
4. Non-deterministic Versus Deterministic Finite State Automata
5. Regular Expressions to NFA
6. NFA to DFA
7. DFA to Minimal DFA
8. JavaCC

Scanning Tokens

Scanning Tokens

The first step in compiling a program is to break it into tokens

Scanning Tokens

The first step in compiling a program is to break it into tokens

Example

HelloWorld.java

```
// Copyright 2012- Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
//
// Writes to standard output the message "Hello, World".

import java.lang.System;

public class HelloWorld {
    // Entry point.
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Tokens: `import, java, ., lang, ., System,;, public, class, HelloWorld, {, ..., ;, }, }`

Scanning Tokens

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

- `import`, `public`, `class`, `static`, etc are reserved words

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

- `import`, `public`, `class`, `static`, etc are reserved words
- `java`, `lang`, `System`, `HelloWorld`, etc are identifiers

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

- `import`, `public`, `class`, `static`, etc are reserved words
- `java`, `lang`, `System`, `HelloWorld`, etc are identifiers
- `"Hello, World"` is a (string) literal

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

- `import`, `public`, `class`, `static`, etc are reserved words
- `java`, `lang`, `System`, `HelloWorld`, etc are identifiers
- `"Hello, World"` is a (string) literal
- `.`, `;`, `{`, `[`, etc are separators

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

- `import`, `public`, `class`, `static`, etc are reserved words
- `java`, `lang`, `System`, `HelloWorld`, etc are identifiers
- `"Hello, World"` is a (string) literal
- `.`, `;`, `{`, `[`, etc are separators
- There are no operators

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

- `import`, `public`, `class`, `static`, etc are reserved words
- `java`, `lang`, `System`, `HelloWorld`, etc are identifiers
- `"Hello, World"` is a (string) literal
- `.`, `;`, `{`, `[`, etc are separators
- There are no operators

A program that breaks the source language program into a sequence of tokens is called a lexical analyzer or a scanner

Scanning Tokens

Tokens are separated into categories such as reserved words, identifiers, literals, separators, and operators

For example, in `HelloWorld.java`:

- `import`, `public`, `class`, `static`, etc are reserved words
- `java`, `lang`, `System`, `HelloWorld`, etc are identifiers
- `"Hello, World"` is a (string) literal
- `.`, `;`, `{`, `[`, etc are separators
- There are no operators

A program that breaks the source language program into a sequence of tokens is called a lexical analyzer or a scanner

A scanner may be hand-crafted or generated from a specification consisting of regular expressions

Scanning Tokens

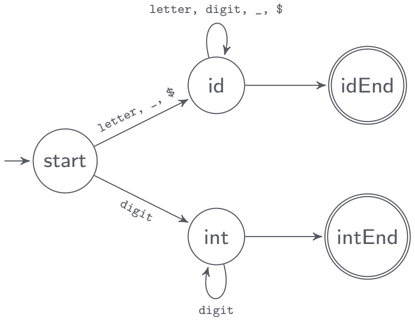
Scanning Tokens

State transition diagrams can be used for describing scanners

Scanning Tokens

State transition diagrams can be used for describing scanners

A state transition diagram for recognizing identifiers and integers



Scanning Tokens

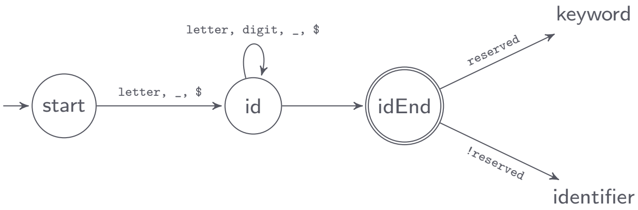
Scanning Tokens

Scanner.java

```java
    if (isLetter(ch) || ch == '_' || ch == '$') {
        buffer = new StringBuffer();
        do {
            buffer.append(ch);
            nextCh();
        } while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$');
        return new TokenInfo(IDENTIFIER, buffer.toString(), line);
    } else if (isDigit(ch)){
        buffer = new StringBuffer();
        do {
            buffer.append(ch);
            nextCh();
        } while (isDigit(ch));
        return new TokenInfo(INT_LITERAL, buffer.toString(), line);
    }
```

Scanning Tokens

A state transition diagram for recognizing keywords



Scanning Tokens

Scanning Tokens

Scanner.java

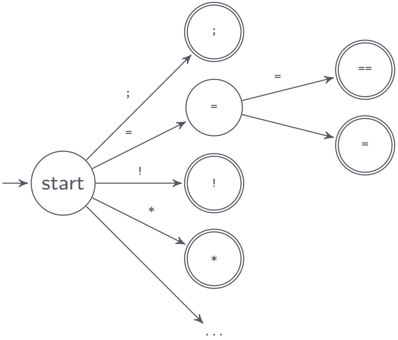
```
reserved = new Hashtable<String, Integer>();
reserved.put("abstract", ABSTRACT);
reserved.put("boolean", BOOLEAN);
reserved.put("char", CHAR);
...
reserved.put("while", WHILE);

...

if (isLetter(ch) || ch == '_' || ch == '$') {
    buffer = new StringBuffer();
    do {
        buffer.append(ch);
        nextCh();
    } while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$');
    String identifier = buffer.toString();
    if (reserved.containsKey(identifier)) {
        return new TokenInfo(reserved.get(identifier), line);
    } else {
        return new TokenInfo(IDENTIFIER, identifier, line);
    }
}
```

Scanning Tokens

A state transition diagram for recognizing separators and operators



Scanning Tokens

Scanning Tokens

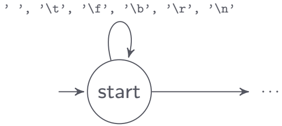
Scanner.java

```
switch (ch) {
    case ';':
        nextCh();
        return new TokenInfo(SEMI, line);
    case '=':
        nextCh();
        if (ch == '=') {
            nextCh();
            return new TokenInfo(EQUAL, line);
        } else {
            return new TokenInfo(ASSIGN, line);
        }
    case '!':
        nextCh();
        return new TokenInfo(LNOT, line);
    case '*':
        nextCh();
        return new TokenInfo(STAR, line);
    ...
}
```

Scanning Tokens

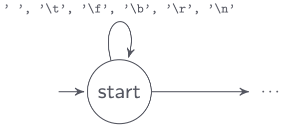
Scanning Tokens

A state transition diagram for recognizing whitespace



Scanning Tokens

A state transition diagram for recognizing whitespace

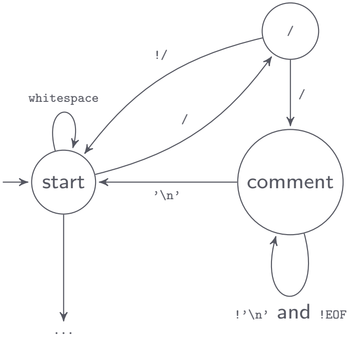


Scanner.java

```
while (isWhitespace(ch)) {
    nextCh();
}
```

Scanning Tokens

A state transition diagram for recognizing comments



Scanning Tokens

Scanning Tokens

Scanner.java

```
boolean moreWhiteSpace = true;
while (moreWhiteSpace) {
    while (isWhitespace(ch)) {
        nextCh();
    }
    if (ch == '/') {
        nextCh();
        if (ch == '/') {
            while (ch != '\n' && ch != EOFCH) {
                nextCh();
            }
        } else {
            reportScannerError("Operator / is not supported in j--.");
        }
    } else {
        moreWhiteSpace = false;
    }
}
```

Regular Expressions

Regular Expressions

A regular expression describes a language of strings over an alphabet Σ

Regular Expressions

A regular expression describes a language of strings over an alphabet Σ

ϵ (epsilon) describes the language consisting of only the empty string

Regular Expressions

A regular expression describes a language of strings over an alphabet Σ

ϵ (epsilon) describes the language consisting of only the empty string

If $a \in \Sigma$, then a describes the language $L(a)$ consisting of the string a

Regular Expressions

A regular expression describes a language of strings over an alphabet Σ

ϵ (epsilon) describes the language consisting of only the empty string

If $a \in \Sigma$, then a describes the language $L(a)$ consisting of the string a

If r and s are regular expressions, then their concatenation rs describes the language $L(rs)$ consisting of strings obtained by concatenating a string from $L(r)$ to a string from $L(s)$

Regular Expressions

A regular expression describes a language of strings over an alphabet Σ

ϵ (epsilon) describes the language consisting of only the empty string

If $a \in \Sigma$, then a describes the language $L(a)$ consisting of the string a

If r and s are regular expressions, then their concatenation rs describes the language $L(rs)$ consisting of strings obtained by concatenating a string from $L(r)$ to a string from $L(s)$

If r and s are regular expressions, then their alternation $r|s$ describes the language $L(r|s)$ consisting of strings from $L(r)$ or $L(s)$

Regular Expressions

A regular expression describes a language of strings over an alphabet Σ

ϵ (epsilon) describes the language consisting of only the empty string

If $a \in \Sigma$, then a describes the language $L(a)$ consisting of the string a

If r and s are regular expressions, then their concatenation rs describes the language $L(rs)$ consisting of strings obtained by concatenating a string from $L(r)$ to a string from $L(s)$

If r and s are regular expressions, then their alternation $r|s$ describes the language $L(r|s)$ consisting of strings from $L(r)$ or $L(s)$

If r is a regular expression, then the Kleene closure $r*$ describes the language $L(r*)$ consisting of strings obtained by concatenating zero or more instances of strings from $L(r)$

Regular Expressions

A regular expression describes a language of strings over an alphabet Σ

ϵ (epsilon) describes the language consisting of only the empty string

If $a \in \Sigma$, then a describes the language $L(a)$ consisting of the string a

If r and s are regular expressions, then their concatenation rs describes the language $L(rs)$ consisting of strings obtained by concatenating a string from $L(r)$ to a string from $L(s)$

If r and s are regular expressions, then their alternation $r|s$ describes the language $L(r|s)$ consisting of strings from $L(r)$ or $L(s)$

If r is a regular expression, then the Kleene closure $r*$ describes the language $L(r*)$ consisting of strings obtained by concatenating zero or more instances of strings from $L(r)$

Both r and (r) describe the same language, ie, $L(r) = L((r))$

Regular Expressions

Regular Expressions

For example, given an alphabet $\Sigma = \{a, b\}$:

Regular Expressions

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a
- $aa|ab|ba|bb$ describes the language of all two-symbol strings over the alphabet

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a
- $aa|ab|ba|bb$ describes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ describes the language of all strings of a's and b's ending in ab

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a
- $aa|ab|ba|bb$ describes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ describes the language of all strings of a's and b's ending in ab

As another example, in a programming language such as *j--*:

Regular Expressions

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a
- $aa|ab|ba|bb$ describes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ describes the language of all strings of a's and b's ending in ab

As another example, in a programming language such as *j--*:

- Reserved words may be described as

```
"abstract" | "boolean" | "char" | ...
```

Regular Expressions

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a
- $aa|ab|ba|bb$ describes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ describes the language of all strings of a's and b's ending in ab

As another example, in a programming language such as *j--*:

- Reserved words may be described as

  ```
  "abstract" | "boolean" | "char" | ...
  ```

- Separators and operators may be described as

  ```
  "," | "." | "[" | ... | "=" | "==" | ">" | ...
  ```

Regular Expressions

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a
- $aa|ab|ba|bb$ describes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ describes the language of all strings of a's and b's ending in ab

As another example, in a programming language such as *j--*:

- Reserved words may be described as
  ```
  "abstract" | "boolean" | "char" | ...
  ```
- Separators and operators may be described as
  ```
  "," | "." | "[" | ... | "=" | "==" | ">" | ...
  ```
- Identifiers may be described as
  ```
  ( "a"..."z" | "A"..."Z" | "_" | "$" ) ( "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" )*
  ```

For example, given an alphabet $\Sigma = \{a, b\}$:

- $a(a|b)*$ describes the language of non-empty strings of a's and b's beginning with an a
- $aa|ab|ba|bb$ describes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ describes the language of all strings of a's and b's ending in ab

As another example, in a programming language such as *j--*:

- Reserved words may be described as
  `"abstract" | "boolean" | "char" | ...`
- Separators and operators may be described as
  `"," | "." | "[" | ... | "=" | "==" | ">" | ...`
- Identifiers may be described as
  `( "a"..."z" | "A"..."Z" | "_" | "$" ) ( "a"..."z" | "A"..."Z" | "_" | "0"..."9" | "$" )*`
- Integer literals may be described as
  `( "0"..."9" ) ( "0"..."9" )*`

Finite State Automata

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can recognize strings in the language

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can recognize strings in the language

A finite state automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, F, M)$, where:

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can recognize strings in the language

A finite state automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, F, M)$, where:

1. Σ is the input alphabet

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can recognize strings in the language

A finite state automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, F, M)$, where:

1. Σ is the input alphabet
2. S is a set of states

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can recognize strings in the language

A finite state automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, F, M)$, where:

1. Σ is the input alphabet
2. S is a set of states
3. $s_0 \in S$ is a special start state

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can recognize strings in the language

A finite state automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, F, M)$, where:

1. Σ is the input alphabet
2. S is a set of states
3. $s_0 \in S$ is a special start state
4. $F \in S$ is a set of final states

Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can recognize strings in the language

A finite state automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, F, M)$, where:

1. Σ is the input alphabet
2. S is a set of states
3. $s_0 \in S$ is a special start state
4. $F \in S$ is a set of final states
5. M is a set of moves (aka transitions) of the form $m(r, a) = s$, where $r, s \in S$ and $a \in \Sigma$

Finite State Automata

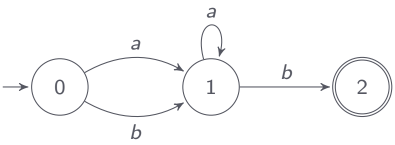
Finite State Automata

For example, consider the regular expression $(a|b)a*b$ over the alphabet $\{a, b\}$

Finite State Automata

For example, consider the regular expression $(a|b)a*b$ over the alphabet $\{a, b\}$

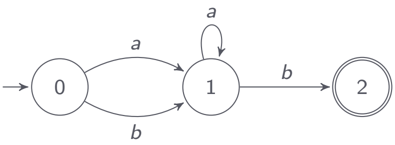
An FSA F that recognizes the language described by the regular expression



Finite State Automata

For example, consider the regular expression $(a|b)a{*}b$ over the alphabet $\{a, b\}$

An FSA F that recognizes the language described by the regular expression



Formally, $F = (\Sigma, S, s_0, F, M)$, where $\Sigma = \{a, b\}$, $S = \{0, 1, 2\}$, $s_0 = 0$, $F = \{2\}$, and M is

r	a	$m(r, a)$
0	a	1
0	b	1
1	a	1
1	b	2

Non-deterministic Versus Deterministic Finite State Automata

Non-deterministic Versus Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is one that allows:

Non-deterministic Versus Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is one that allows:

- An ϵ-move defined on the empty string ϵ, ie, $m(r, \epsilon) = s$

Non-deterministic Versus Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is one that allows:

- An ϵ-move defined on the empty string ϵ, ie, $m(r, \epsilon) = s$
- More than one move from a state r on an input symbol a, ie, $m(r, a) = s$ and $m(r, a) = t$, where $s \neq t$

Non-deterministic Versus Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is one that allows:

- An ϵ-move defined on the empty string ϵ, ie, $m(r, \epsilon) = s$
- More than one move from a state r on an input symbol a, ie, $m(r, a) = s$ and $m(r, a) = t$, where $s \neq t$

An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

Non-deterministic Versus Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is one that allows:

- An ϵ-move defined on the empty string ϵ, ie, $m(r, \epsilon) = s$
- More than one move from a state r on an input symbol a, ie, $m(r, a) = s$ and $m(r, a) = t$, where $s \neq t$

An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

A deterministic finite state automaton (DFA) is one in which:

Non-deterministic Versus Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is one that allows:

- An ϵ-move defined on the empty string ϵ, ie, $m(r, \epsilon) = s$
- More than one move from a state r on an input symbol a, ie, $m(r, a) = s$ and $m(r, a) = t$, where $s \neq t$

An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

A deterministic finite state automaton (DFA) is one in which:

- There are no ϵ-moves

Non-deterministic Versus Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is one that allows:

- An ϵ-move defined on the empty string ϵ, ie, $m(r, \epsilon) = s$
- More than one move from a state r on an input symbol a, ie, $m(r, a) = s$ and $m(r, a) = t$, where $s \neq t$

An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

A deterministic finite state automaton (DFA) is one in which:

- There are no ϵ-moves
- There is a unique move from any state r on an input symbol a, ie, if $m(r, a) = s$ and $m(r, a) = t$, then $s = t$

Non-deterministic Versus Deterministic Finite State Automata

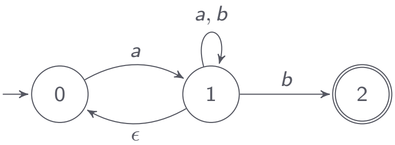
Non-deterministic Versus Deterministic Finite State Automata

For example, consider the regular expression $a(a|b)*b$ over the alphabet $\{a, b\}$

Non-deterministic Versus Deterministic Finite State Automata

For example, consider the regular expression $a(a|b)*b$ over the alphabet $\{a, b\}$

An NFA N that recognizes the language described by the regular expression



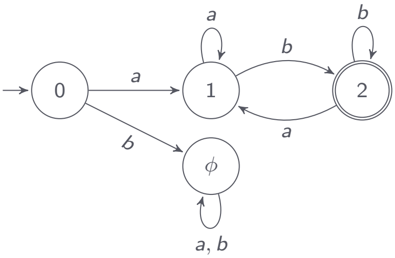
$N = (\Sigma, S, s_0, F, M)$ where $\Sigma = \{a, b\}$, $S = \{0, 1, 2\}$, $s_0 = 0$, $F = \{2\}$, and M is

r	a	$m(r, a)$
0	a	1
1	ϵ	0
1	a	1
1	b	1
1	b	2

Non-deterministic Versus Deterministic Finite State Automata

Non-deterministic Versus Deterministic Finite State Automata

And a DFA D that recognizes the same language



$D = (\Sigma, S, s_0, F, M)$ where $\Sigma = \{a, b\}$, $S = \{0, 1, 2, \phi\}$, $s_0 = 0$, $F = \{2\}$, and M is

r	a	$m(r, a)$
0	a	1
0	b	ϕ
1	a	1
1	b	2
2	a	1
2	b	2
ϕ	a, b	ϕ

Regular Expressions to NFA

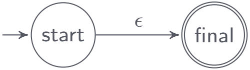
Regular Expressions to NFA

Given any regular expression r, we can construct (using Thompson's construction procedure) an NFA N that recognizes the same language; ie, $L(N) = L(r)$

Regular Expressions to NFA

Given any regular expression r, we can construct (using Thompson's construction procedure) an NFA N that recognizes the same language; ie, $L(N) = L(r)$

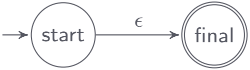
(Rule 1) NFA N_r for recognizing $L(r = \epsilon)$



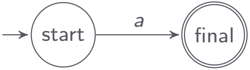
Regular Expressions to NFA

Given any regular expression r, we can construct (using Thompson's construction procedure) an NFA N that recognizes the same language; ie, $L(N) = L(r)$

(Rule 1) NFA N_r for recognizing $L(r = \epsilon)$



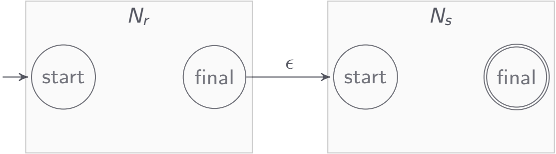
(Rule 2) NFA N_r for recognizing $L(r = a)$



Regular Expressions to NFA

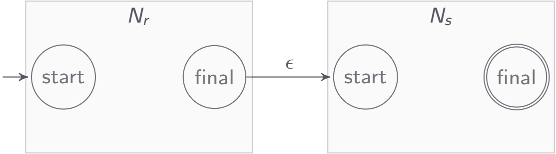
Regular Expressions to NFA

(Rule 3) NFA N_{rs} for recognizing $L(rs)$

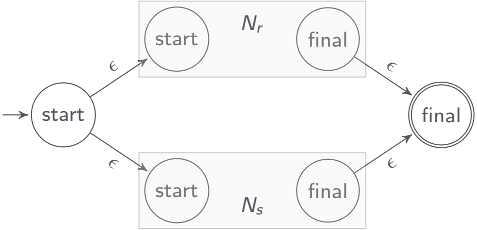


Regular Expressions to NFA

(Rule 3) NFA N_{rs} for recognizing $L(rs)$



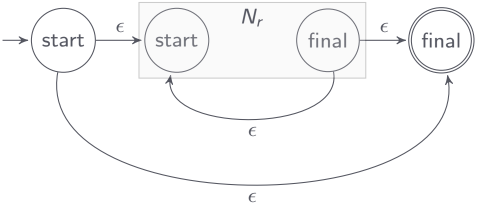
(Rule 4) NFA $N_{r|s}$ for recognizing $L(r|s)$



Regular Expressions to NFA

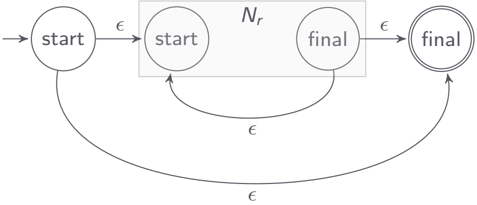
Regular Expressions to NFA

(Rule 5) NFA N_{r*} for recognizing $L(r*)$



Regular Expressions to NFA

(Rule 5) NFA N_{r*} for recognizing $L(r*)$



(Rule 6) NFA N_r for recognizing $L(r)$ also recognizes $L((r))$

Regular Expressions to NFA

Regular Expressions to NFA

As an example, let's construct an NFA for the regular expression $(a|b)a*b$

Regular Expressions to NFA

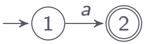
As an example, let's construct an NFA for the regular expression $(a|b)a*b$

Using Rule 2, we get the NFAs N_a and N_b for recognizing a and b as

Regular Expressions to NFA

As an example, let's construct an NFA for the regular expression $(a|b)a*b$

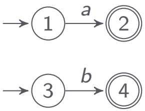
Using Rule 2, we get the NFAs N_a and N_b for recognizing a and b as



Regular Expressions to NFA

As an example, let's construct an NFA for the regular expression $(a|b)a*b$

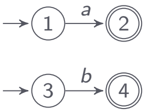
Using Rule 2, we get the NFAs N_a and N_b for recognizing a and b as



Regular Expressions to NFA

As an example, let's construct an NFA for the regular expression $(a|b)a*b$

Using Rule 2, we get the NFAs N_a and N_b for recognizing a and b as

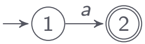


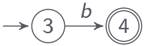
Using Rules 4 and 6, we get the NFA $N_{(a|b)}$ for recognizing $(a|b)$ as

Regular Expressions to NFA

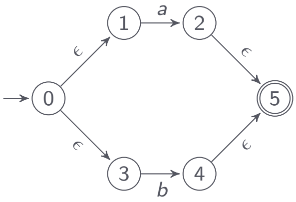
As an example, let's construct an NFA for the regular expression $(a|b)a*b$

Using Rule 2, we get the NFAs N_a and N_b for recognizing a and b as





Using Rules 4 and 6, we get the NFA $N_{(a|b)}$ for recognizing $(a|b)$ as



Regular Expressions to NFA

Regular Expressions to NFA

Using Rule 2, we get the NFAs N_a for recognizing the second instance of a as

Regular Expressions to NFA

Using Rule 2, we get the NFAs N_a for recognizing the second instance of a as

Regular Expressions to NFA

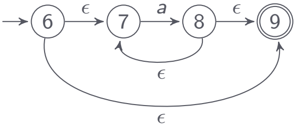
Using Rule 2, we get the NFAs N_a for recognizing the second instance of a as

Using Rule 5, we get the NFA N_{a*} for recognizing $a*$ as

Regular Expressions to NFA

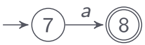
Using Rule 2, we get the NFAs N_a for recognizing the second instance of a as

Using Rule 5, we get the NFA N_{a*} for recognizing $a*$ as

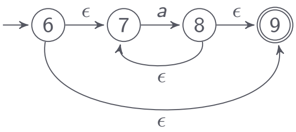


Regular Expressions to NFA

Using Rule 2, we get the NFAs N_a for recognizing the second instance of a as



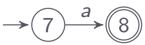
Using Rule 5, we get the NFA N_{a*} for recognizing $a*$ as



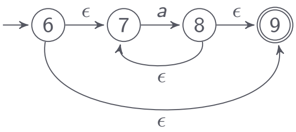
Using Rule 3, we get the NFA $N_{(a|b)a*}$ for recognizing $(a|b)a*$

Regular Expressions to NFA

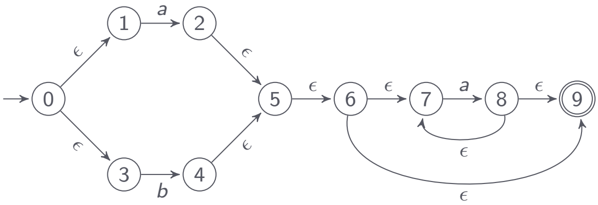
Using Rule 2, we get the NFAs N_a for recognizing the second instance of a as



Using Rule 5, we get the NFA N_{a*} for recognizing $a*$ as



Using Rule 3, we get the NFA $N_{(a|b)a*}$ for recognizing $(a|b)a*$



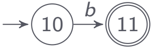
Regular Expressions to NFA

Regular Expressions to NFA

Using Rule 2, we get the NFAs N_b for recognizing the second instance of b as

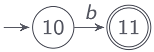
Regular Expressions to NFA

Using Rule 2, we get the NFAs N_b for recognizing the second instance of b as



Regular Expressions to NFA

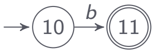
Using Rule 2, we get the NFAs N_b for recognizing the second instance of b as



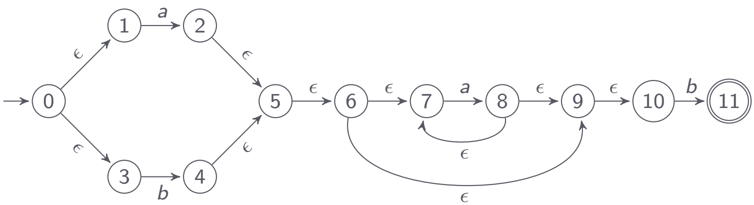
Finally, using Rule 3, we get the NFA $N_{(a|b)a*b}$ for recognizing $(a|b)a*b$ as

Regular Expressions to NFA

Using Rule 2, we get the NFAs N_b for recognizing the second instance of b as



Finally, using Rule 3, we get the NFA $N_{(a|b)a*b}$ for recognizing $(a|b)a*b$ as



NFA to DFA

NFA to DFA

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

NFA to DFA

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

The DFA is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input

NFA to DFA

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

The DFA is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input

The computation of all states reachable from a given state s based on ϵ-moves alone is called taking the ϵ-closure of that state

NFA to DFA

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

The DFA is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input

The computation of all states reachable from a given state s based on ϵ-moves alone is called taking the ϵ-closure of that state

The ϵ-closure(s) for a state s includes s and all states reachable from s using ϵ-moves alone, ie,
ϵ-closure(s) $= \{s\} \cup \{r \in S |$ there is a path of only ϵ-moves from s to $r\}$

NFA to DFA

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

The DFA is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input

The computation of all states reachable from a given state s based on ϵ-moves alone is called taking the ϵ-closure of that state

The ϵ-closure(s) for a state s includes s and all states reachable from s using ϵ-moves alone, ie,
ϵ-closure(s) $= \{s\} \cup \{r \in S |$ there is a path of only ϵ-moves from s to $r\}$

The ϵ-closure(S) for a set of states S includes S and all states reachable from any state $s \in S$ using ϵ-moves alone

NFA to DFA

Algorithm ϵ-closure(S) for a set of states S

Input: a set of states S
Output: ϵ-closure(S)

```
P ← Stack(S)
C ← Set(S)
while not P.isEmpty() do
    r ← P.pop()
    for s ∈ m(r, ϵ) do
        if s ∉ C then
            P.push(s)
            C.add(s)
        end if
    end for
end while
return C
```

NFA to DFA

NFA to DFA

Algorithm ϵ-closure(s) for a state s

Input: a state s

Output: ϵ-closure(s)

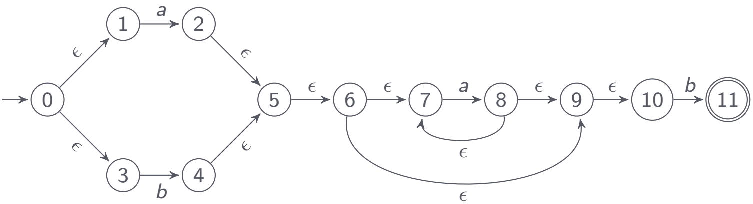
1: $S \leftarrow$ Set(s)

2: **return** ϵ-closure(S)

NFA to DFA

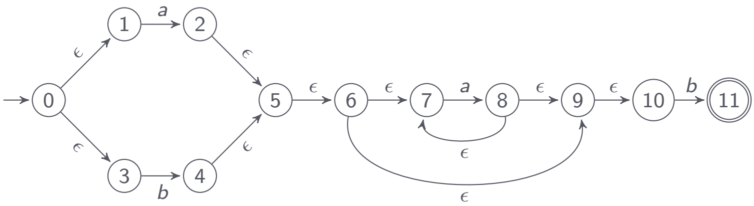
NFA to DFA

As an example, let's convert the NFA $N_{(a|b)a*b}$ to a DFA



NFA to DFA

As an example, let's convert the NFA $N_{(a|b)a*b}$ to a DFA

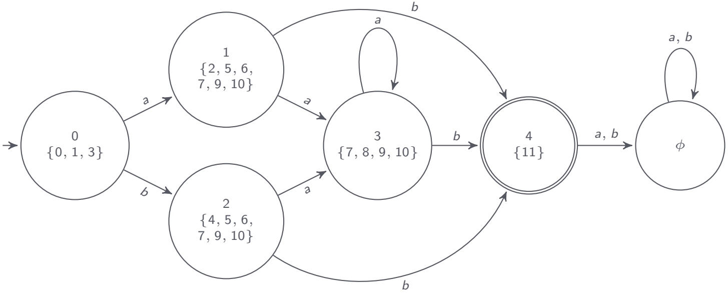


r	a	$m(r,a)$
$\{0,1,3\} = 0$ (start state)	a	$\{2,5,6,7,9,10\} = 1$
0	b	$\{4,5,6,7,9,10\} = 2$
1	a	$\{7,8,9,10\} = 3$
1	b	$\{11\} = 4$ (accept state)
2	a	3
2	b	4
3	a	3
3	b	4
4	a, b	ϕ
ϕ	a, b	ϕ

NFA to DFA

NFA to DFA

The DFA for recognizing $(a|b)a*b$



NFA to DFA

Algorithm NFA to DFA construction

Input: an NFA $N = (\Sigma, S, s_0, M, F)$
Output: an equivalent DFA $D = (\Sigma, S_D, s_{D0}, M_D, F_D)$

$s_{D0} \leftarrow \epsilon\text{-closure}(s_0)$
$S_D \leftarrow \text{Set}(s_{D0})$
$M_D \leftarrow Moves()$
$stk \leftarrow \text{Stack}(s_{D0})$
$i \leftarrow 0$
while not stk.isEmpty() **do**
    $r \leftarrow stk$.pop()
    for $a \in \Sigma$ **do**
        $s_{Di+1} \leftarrow \epsilon\text{-closure}(m(r, a))$
        if $s_{Di+1} \neq \{\}$ **then**
            if $s_{Di+1} \notin S_D$ **then**
                S_D.add(s_{Di+1})
                stk.push(s_{Di+1})
                $i \leftarrow i + 1$
                M_D.add($(r, a) \rightarrow s_{Di+1}$)
            else if $\exists s_j \in S_D$ such that $s_{Di+1} = s_j$ **then**
                M_D.add($(r, a) \rightarrow s_j$)
            end if
        end if
    end for
end while
$F_D \leftarrow \text{Set}()$
for $s_D \in S_D$ **do**
    for $s \in s_D$ **do**
        if $s \in F$ **then**
            F_D.add(s_D)
        end if
    end for
end for
return $D = (\Sigma, S_D, s_{D0}, M_D, F_D)$

DFA to Minimal DFA

DFA to Minimal DFA

To obtain a smaller but equivalent DFA, partition the states such that the states in the new DFA are subsets of the states in the original (perhaps larger) DFA

DFA to Minimal DFA

To obtain a smaller but equivalent DFA, partition the states such that the states in the new DFA are subsets of the states in the original (perhaps larger) DFA

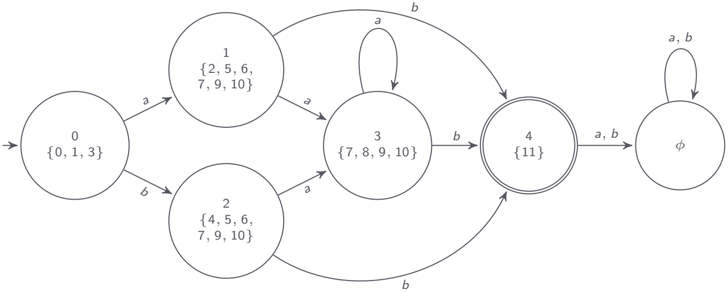
The initial partition contains two subsets: the non-final states and the final states

DFA to Minimal DFA

To obtain a smaller but equivalent DFA, partition the states such that the states in the new DFA are subsets of the states in the original (perhaps larger) DFA

The initial partition contains two subsets: the non-final states and the final states

For example, consider the DFA for $(a|b)a*b$

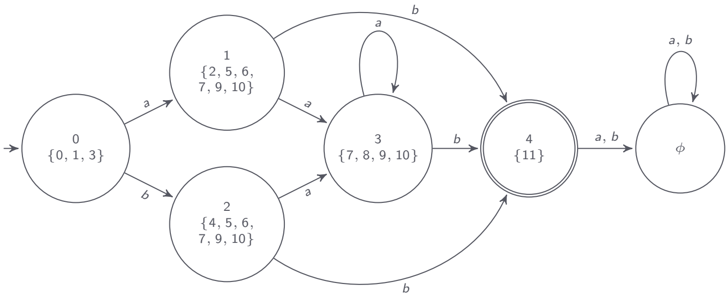


DFA to Minimal DFA

To obtain a smaller but equivalent DFA, partition the states such that the states in the new DFA are subsets of the states in the original (perhaps larger) DFA

The initial partition contains two subsets: the non-final states and the final states

For example, consider the DFA for $(a|b)a*b$



The initial partition contains the subsets $\{0, 1, 2, 3, \phi\}$ and $\{4\}$

DFA to Minimal DFA

DFA to Minimal DFA

Make sure that from a particular subset, on each input symbol, you transition into an identical subset; if not, split the subset

DFA to Minimal DFA

Make sure that from a particular subset, on each input symbol, you transition into an identical subset; if not, split the subset

The symbol *a* does not split the subset $\{0, 1, 2, 3, \phi\}$, since

$$
\begin{aligned}
m(0, a) &= 1 \\
m(1, a) &= 3 \\
m(2, a) &= 3 \\
m(3, a) &= 3 \\
m(\phi, a) &= \phi
\end{aligned}
$$

DFA to Minimal DFA

Make sure that from a particular subset, on each input symbol, you transition into an identical subset; if not, split the subset

The symbol a does not split the subset $\{0, 1, 2, 3, \phi\}$, since

$$m(0, a) = 1$$
$$m(1, a) = 3$$
$$m(2, a) = 3$$
$$m(3, a) = 3$$
$$m(\phi, a) = \phi$$

The symbol b splits the subset $\{0, 1, 2, 3, \phi\}$ into subsets $\{0, \phi\}$ and $\{1, 2, 3\}$, since

$$m(0, b) = 2$$
$$m(1, b) = 4$$
$$m(2, b) = 4$$
$$m(3, b) = 4$$
$$m(\phi, b) = \phi$$

DFA to Minimal DFA

DFA to Minimal DFA

The symbol *a* splits the subset $\{0, \phi\}$ into subsets $\{0\}$ and $\{\phi\}$

DFA to Minimal DFA

The symbol *a* splits the subset $\{0, \phi\}$ into subsets $\{0\}$ and $\{\phi\}$

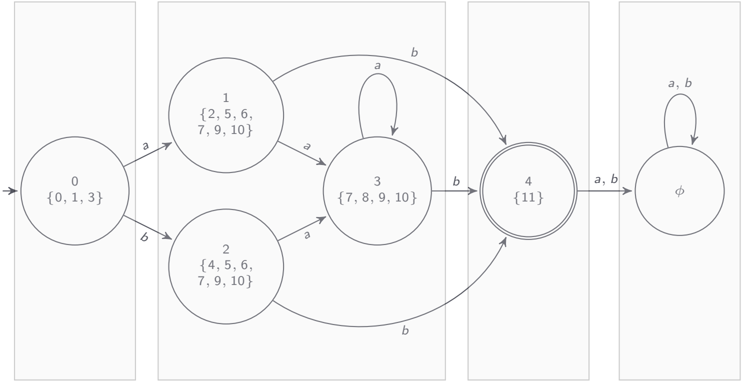
Neither *a* nor *b* splits the subset $\{1, 2, 3\}$

DFA to Minimal DFA

The symbol a splits the subset $\{0, \phi\}$ into subsets $\{0\}$ and $\{\phi\}$

Neither a nor b splits the subset $\{1, 2, 3\}$

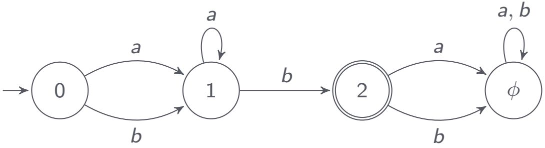
The final partition is therefore $\{\{0\}, \{1, 2, 3\}, \{4\}, \{\phi\}\}$



DFA to Minimal DFA

DFA to Minimal DFA

Minimal DFA for recognizing $(a|b)a*b$



DFA to Minimal DFA

DFA to Minimal DFA

Algorithm Minimizing a DFA

Input: a DFA $D = (\Sigma, S, s_0, M, F)$

Output: a partition of S

$partition \leftarrow \{S - F, F\}$
while splitting occurs **do**
  for $subset \in partition$ **do**
    if $subset$.size() > 1 **then**
      for $a \in \Sigma$ **do**
        $r \leftarrow$ a state chosen from $subset$
        $targetSet \leftarrow$ the subset in the partition containing $m(r, a)$
        $set1 \leftarrow \{s \in subset | m(s, a) \in targetSet\}$
        $set2 \leftarrow \{s \in subset | m(s, a) \notin targetSet\}$
        if $set2 \neq \{\}$ **then**
          replace $subset$ in $partition$ by $set1$ and $set2$
          break
        end if
      end for
    end if
  end for
end while

JavaCC

JavaCC

JavaCC is a tool for generating scanners from regular expressions and parsers from context-free grammars

JavaCC

JavaCC is a tool for generating scanners from regular expressions and parsers from context-free grammars

A lexical grammar consists a set of regular expressions and a set of lexical states

JavaCC is a tool for generating scanners from regular expressions and parsers from context-free grammars

A lexical grammar consists a set of regular expressions and a set of lexical states

From a particular state, only certain regular expressions may be matched by the input

JavaCC is a tool for generating scanners from regular expressions and parsers from context-free grammars

A lexical grammar consists a set of regular expressions and a set of lexical states

From a particular state, only certain regular expressions may be matched by the input

There is a DEFAULT state in which scanning begins; one may specify additional states as required

JavaCC

JavaCC is a tool for generating scanners from regular expressions and parsers from context-free grammars

A lexical grammar consists a set of regular expressions and a set of lexical states

From a particular state, only certain regular expressions may be matched by the input

There is a `DEFAULT` state in which scanning begins; one may specify additional states as required

Scanning proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

JavaCC

JavaCC is a tool for generating scanners from regular expressions and parsers from context-free grammars

A lexical grammar consists a set of regular expressions and a set of lexical states

From a particular state, only certain regular expressions may be matched by the input

There is a `DEFAULT` state in which scanning begins; one may specify additional states as required

Scanning proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, the scanner goes into a specified state or stays in the current state

JavaCC

There are four kinds of regular expressions that determine what happens when the regular expression has been matched:

There are four kinds of regular expressions that determine what happens when the regular expression has been matched:

1. SKIP: throws away the matched string

There are four kinds of regular expressions that determine what happens when the regular expression has been matched:

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along

There are four kinds of regular expressions that determine what happens when the regular expression has been matched:

1. `SKIP`: throws away the matched string
2. `MORE`: continues to the next state, taking the matched string along
3. `TOKEN`: creates a token from the matched string and returns it to the parser

There are four kinds of regular expressions that determine what happens when the regular expression has been matched:

1. `SKIP`: throws away the matched string
2. `MORE`: continues to the next state, taking the matched string along
3. `TOKEN`: creates a token from the matched string and returns it to the parser
4. `SPECIAL_TOKEN`: creates a special token that does not participate in the parsing

JavaCC

There are four kinds of regular expressions that determine what happens when the regular expression has been matched:

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

JavaCC generates a scanner for *j--* from regular expressions defined in `$j/j--/src/jminusminus/j--.jj`

`j--.jj` → JavaCC → `TokenManager.java`

JavaCC

Scanning whitespace

```
SKIP: { " " | "\t" | "\n" | "\r" | "\f" }
```

Scanning whitespace

```
SKIP: { " " | "\t" | "\n" | "\r" | "\f" }
```

Scanning single-line comments

```
SKIP: { <BEGIN_COMMENT: "//">: IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <END_COMMENT: "\n" | "\r" | "\r\n">: DEFAULT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <COMMENT: ~[]> }
```

Scanning whitespace

```
SKIP: { " " | "\t" | "\n" | "\r" | "\f" }
```

Scanning single-line comments

```
SKIP: { <BEGIN_COMMENT: "//">: IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <END_COMMENT: "\n" | "\r" | "\r\n">: DEFAULT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <COMMENT: ~[]> }
```

Alternative way of scanning single-line comments

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" ( ~[ "\n", "\r" ] )* ( "\n" | "\r" | "\r\n" )>
}
```

Scanning whitespace

```
SKIP: { " " | "\t" | "\n" | "\r" | "\f" }
```

Scanning single-line comments

```
SKIP: { <BEGIN_COMMENT: "//">: IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <END_COMMENT: "\n" | "\r" | "\r\n">: DEFAULT }
<IN_SINGLE_LINE_COMMENT>
SKIP: { <COMMENT: ~[]> }
```

Alternative way of scanning single-line comments

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" ( ~[ "\n", "\r" ] )* ( "\n" | "\r" | "\r\n" )>
}
```

Scanning reserved words, separators, and operators

```
TOKEN: {
  <ABSTRACT: "abstract">
| <BOOLEAN: "boolean">
...
| <COMMA: ",">
| <DOT: "." >
...
| <ASSIGN: "=">
| <DEC: "--">
...
}
```

JavaCC

Scanning identifiers

```
TOKEN: {
  <IDENTIFIER: ( <LETTER> | "_" | "$" ) ( <LETTER> | <DIGIT> | "_" | "$" )*>
| <#LETTER: [ "a"-"z", "A"-"Z" ]>
| <#DIGIT: [ "0"-"9" ]>
}
```

Scanning identifiers

```
TOKEN: {
  <IDENTIFIER: ( <LETTER> | "_" | "$" ) ( <LETTER> | <DIGIT> | "_" | "$" )*>
| <#LETTER: [ "a"-"z", "A"-"Z" ]>
| <#DIGIT: [ "0"-"9" ]>
}
```

Scanning literals

```
TOKEN: {
  <INT_LITERAL: <DIGIT> ( <DIGIT> )*>
| <CHAR_LITERAL: "'" ( <ESC> | ~[ "'", "\\" ]) "'">
| <STRING_LITERAL: "\"" ( <ESC> | ~[ "\"", "\\" ] )* "\"">
| <#ESC: "\\" [ "n", "t", "b", "r", "f", "\\", "'", "\"" ]>
}
```