Computer Languages

Finite automata & Lexical analysis

January 21st

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

Plan

What we know

1. Regular expressions can be used to describe some (not so elaborate) languages.
   - Numbers in a programming language, keywords in a programming language, e-mail addresses, dates.
2. Scanner Generators can be used to create a program that reads a sequence of characters and identifies the words of the language described by a regular expression!

What we will learn today

1. What kind of program is a scanner?
2. How can such a program be generated by another program?
3. How can a scanner be used to do lexical analysis?

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:

One word

We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

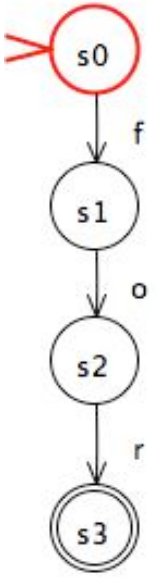
We can represent the code fragment using the diagram:

One word

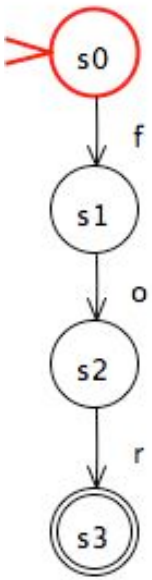
We want to recognize whether a string forms the word `for`

```
readChar(c);
if(c!='f') do something else
else
  readChar(c);
  if(c!='o') do something else
  else
    readChar(c);
    if(c!='r') do something else
    else report success
```

We can represent the code fragment using the diagram:



States and transitions



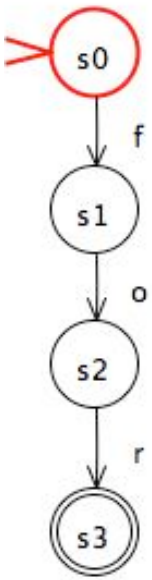
s0, s1, s2, s3 are called states

s0 is marked as the initial state.

s3 is marked as one final state

$\xrightarrow{x}$ represent transitions from state to state based on the input character.

States and transitions



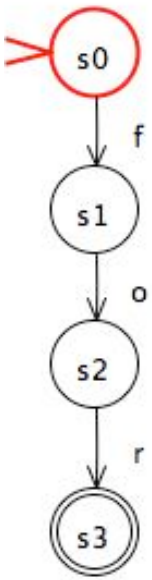
s0, s1, s2, s3 are called states

s0 is marked as the initial state.

s3 is marked as one final state

$\xrightarrow{x}$ represent transitions from state to state based on the input character.

States and transitions



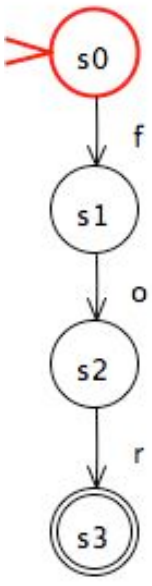
s0, s1, s2, s3 are called states

s0 is marked as the initial state.

s3 is marked as one final state

$\xrightarrow{x}$ represent transitions from state to state based on the input character.

States and transitions



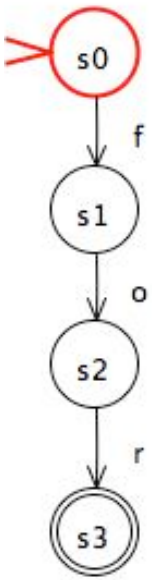
s0, s1, s2, s3 are called states

s0 is marked as the initial state.

s3 is marked as one final state

$\xrightarrow{x}$ represent transitions from state to state based on the input character.

States and transitions



s0, s1, s2, s3 are called states

s0 is marked as the initial state.

s3 is marked as one final state

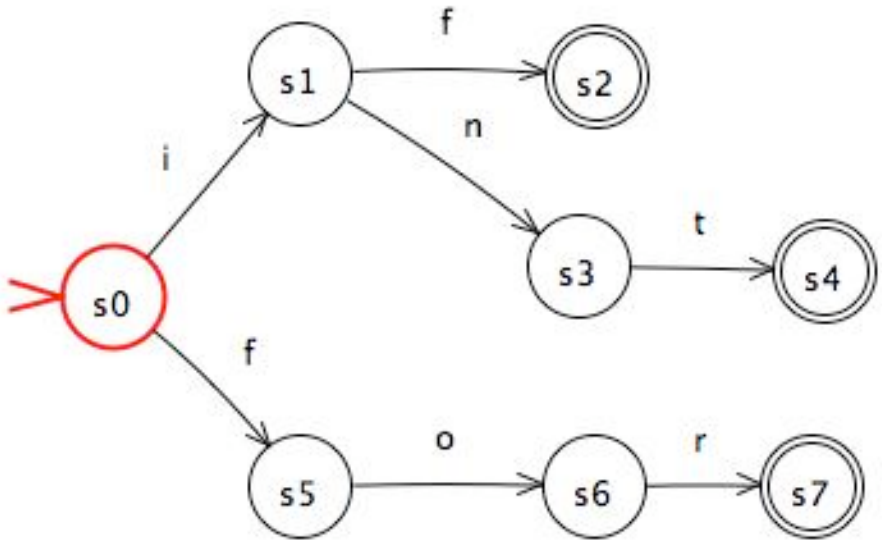
$\xrightarrow{x}$ represent transitions from state to state based on the input character.

More than one word

Just add proper code in the *do something else* fragments!

More than one word

Just add proper code in the *do something else* fragments!



Encoding an automata

Transitions (δ)

δ	**f**	**i**	**n**	**o**	**r**	**t**	**Other**
s0	s5	s1	sE	sE	sE	sE	sE
s1	s2	sE	s3	sE	sE	sE	sE
s2	sE	sE	sE	sE	sE	sE	sE
s3	sE	sE	sE	sE	sE	s4	sE
s4	sE	sE	sE	sE	sE	sE	sE
s5	sE	sE	sE	s6	sE	sE	sE
s6	sE	sE	sE	sE	s7	sE	sE
sE	sE	sE	sE	sE	sE	sE	sE

The automaton accepts or rejects a string as follows.

Starting in the start state, for each char c in the input change state according to $\delta(s, c)$. After making n transitions for an *n-character* string accept if the state is a final state. Reject otherwise.

Encoding an automata

Transitions (δ)

δ	**f**	**i**	**n**	**o**	**r**	**t**	**Other**
s0	s5	s1	sE	sE	sE	sE	sE
s1	s2	sE	s3	sE	sE	sE	sE
s2	sE	sE	sE	sE	sE	sE	sE
s3	sE	sE	sE	sE	sE	s4	sE
s4	sE	sE	sE	sE	sE	sE	sE
s5	sE	sE	sE	s6	sE	sE	sE
s6	sE	sE	sE	sE	s7	sE	sE
sE	sE	sE	sE	sE	sE	sE	sE

The automaton accepts or rejects a string as follows.

Starting in the start state, for each char c in the input change state according to $\delta(s, c)$. After making n transitions for an *n-character* string accept if the state is a final state. Reject otherwise.

Encoding an automata

Transitions (δ)

δ	**f**	**i**	**n**	**o**	**r**	**t**	**Other**
s0	s5	s1	sE	sE	sE	sE	sE
s1	s2	sE	s3	sE	sE	sE	sE
s2	sE	sE	sE	sE	sE	sE	sE
s3	sE	sE	sE	sE	sE	s4	sE
s4	sE	sE	sE	sE	sE	sE	sE
s5	sE	sE	sE	s6	sE	sE	sE
s6	sE	sE	sE	sE	s7	sE	sE
sE	sE	sE	sE	sE	sE	sE	sE

The automaton accepts or rejects a string as follows.

Starting in the start state, for each char c in the input change state according to $\delta(s, c)$. After making n transitions for an *n-character* string accept if the state is a final state. Reject otherwise.

More complex words

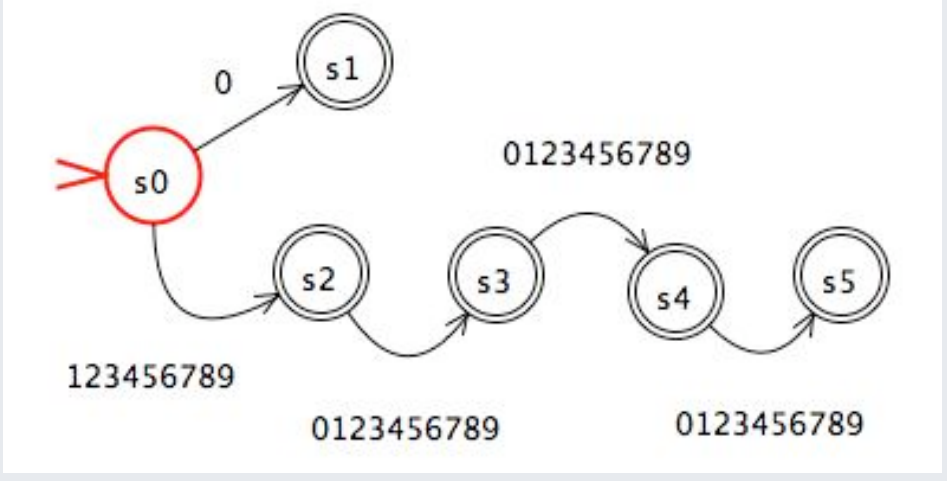
What automaton recognizes numbers?

We want to say that $\delta(s_2, 0) = s_2$, $\delta(s_2, 1) = s_2$, $\delta(s_2, 2) = s_2, \ldots \delta(s_2, 9) = s_2$.

Doesn't work for numbers with more than 4 digits!

More complex words

What automaton recognizes numbers?

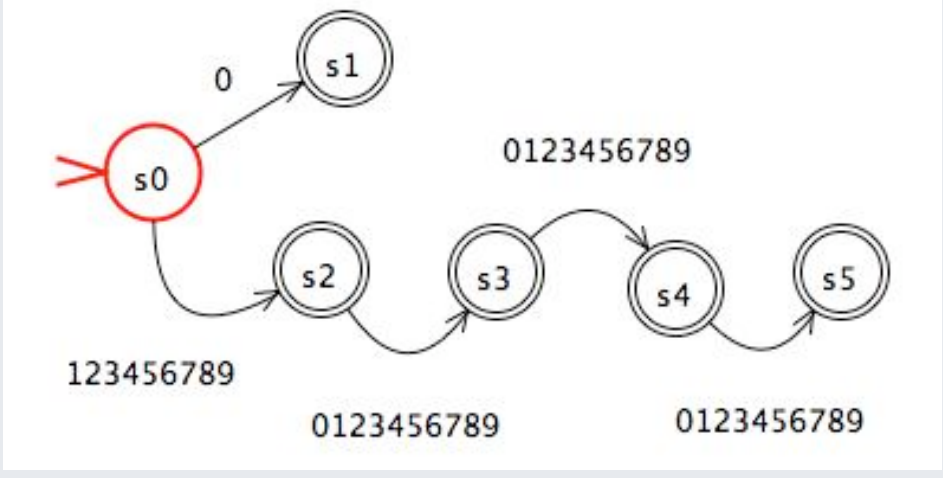


We want to say that $\delta(s_2, 0) = s_2$, $\delta(s_2, 1) = s_2$, $\delta(s_2, 2) = s_2, \ldots \delta(s_2, 9) = s_2$.

Doesn't work for numbers with more than 4 digits!

More complex words

What automaton recognizes numbers?

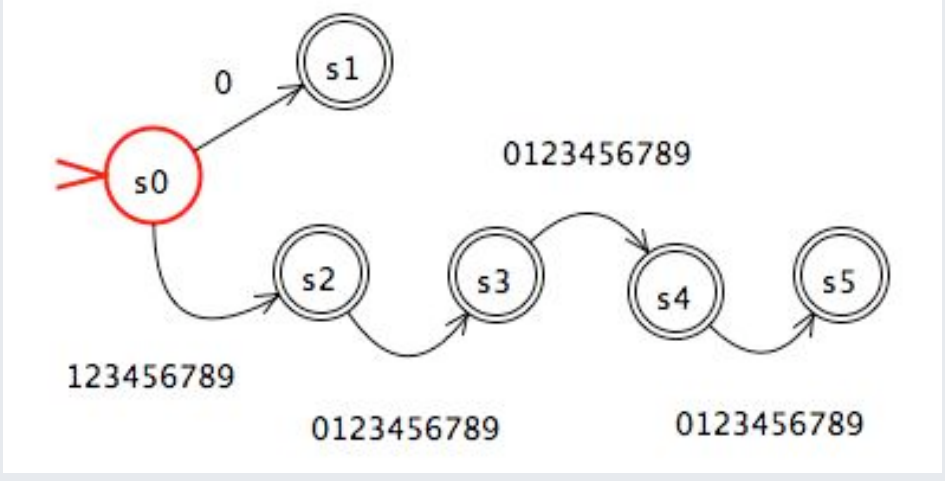


We want to say that $\delta(s_2, 0) = s_2$, $\delta(s_2, 1) = s_2$, $\delta(s_2, 2) = s_2, \ldots \delta(s_2, 9) = s_2$.

Doesn't work for numbers with more than 4 digits!

More complex words

What automaton recognizes numbers?

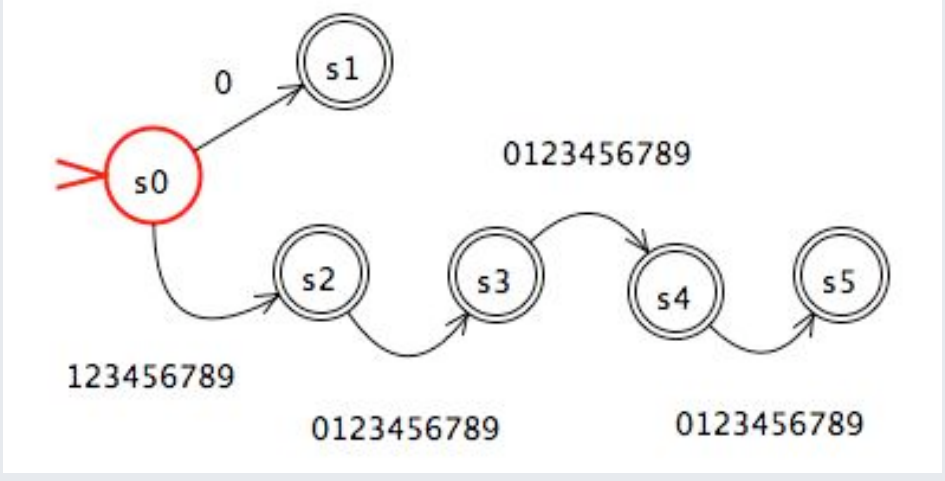


We want to say that
$\delta(s_2, 0) = s_2$, $\delta(s_2, 1) = s_2$,
$\delta(s_2, 2) = s_2, \ldots \delta(s_2, 9) = s_2$.

Doesn't work for numbers with more than 4 digits!

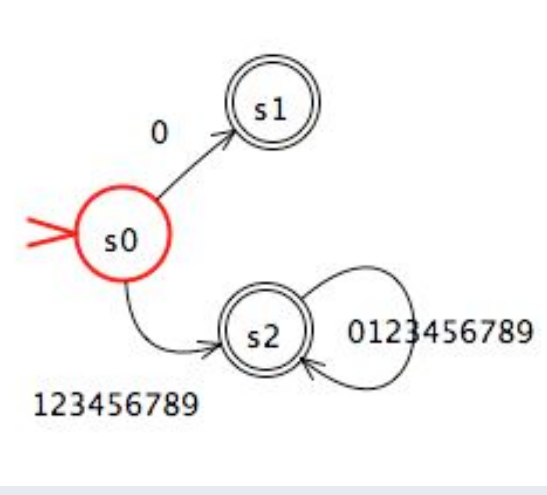
More complex words

What automaton recognizes numbers?



Doesn't work for numbers with more than 4 digits!

We want to say that
$\delta(s_2, 0) = s_2$, $\delta(s_2, 1) = s_2$,
$\delta(s_2, 2) = s_2, \ldots \delta(s_2, 9) = s_2$.



Deterministic finite state automata

1. Finite number of states
2. From each state only one transition for a given character.

For every regular expression there is a deterministic finite state automata that recognizes its language.

The proof of this theorem is the algorithm that is used by a scanner generator!

For every finite automata there is a regular expression that describes the language it recognizes.

Deterministic finite state automata

1. Finite number of states
2. From each state only one transition for a given character.

For every regular expression there is a deterministic finite state automata that recognizes its language.

The proof of this theorem is the algorithm that is used by a scanner generator!

For every finite automata there is a regular expression that describes the language it recognizes.

Deterministic finite state automata

1. Finite number of states
2. From each state only one transition for a given character.

For every regular expression there is a deterministic finite state automata that recognizes its language.

The proof of this theorem is the algorithm that is used by a scanner generator!

For every finite automata there is a regular expression that describes the language it recognizes.

Deterministic finite state automata

1. Finite number of states
2. From each state only one transition for a given character.

For every regular expression there is a deterministic finite state automata that recognizes its language.

For every finite automata there is a regular expression that describes the language it recognizes.

The proof of this theorem is the algorithm that is used by a scanner generator!

Sketch of the proof (algorithm)

We will see how to associate a nondeterministic finite automata to each regular expression!

- There might be more than one edge labeled with the same symbol leaving a state.
- There might be transitions on the empty string (labeled $\in$)
- They are easier to construct, but they do not help as programs!

We will then see how to associate a deterministic finite automata to a nondeterministic one!

Sketch of the proof (algorithm)

We will see how to associate a nondeterministic finite automata to each regular expression!

- There might be more than one edge labeled with the same symbol leaving a state.
- There might be transitions on the empty string (labeled $\in$)
- They are easier to construct, but they do not help as programs!

We will then see how to associate a deterministic finite automata to a nondeterministic one!

Sketch of the proof (algorithm)

We will see how to associate a nondeterministic finite automata to each regular expression!

- There might be more than one edge labeled with the same symbol leaving a state.
- There might be transitions on the empty string (labeled $\in$)
- They are easier to construct, but they do not help as programs!

We will then see how to associate a deterministic finite automata to a nondeterministic one!

Thompson's construction

In the following arbitrary NFAs can be used in place of the NFAs for a and b

a is a RE for $a \in \Sigma$

is a NFA for a

b is a RE for $b \in \Sigma$

is a NFA for b

ab is a regular expression

is a NFA for ab

Thompson's construction

In the following arbitrary NFAs can be used in place of the NFAs for a and b

a is a RE for $a \in \Sigma$

is a NFA for a

b is a RE for $b \in \Sigma$

is a NFA for b

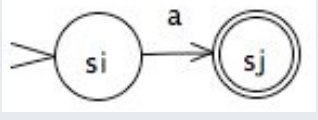
ab is a regular expression

is a NFA for ab

Thompson's construction

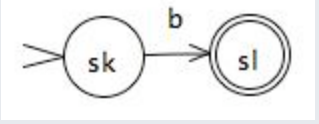
In the following arbitrary NFAs can be used in place of the NFAs for a and b

a is a RE for $a \in \Sigma$



is a NFA for a

b is a RE for $b \in \Sigma$



is a NFA for b

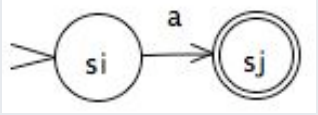
ab is a regular expression

is a NFA for ab

Thompson's construction

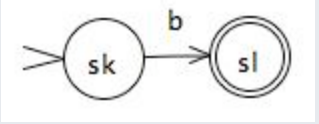
In the following arbitrary NFAs can be used in place of the NFAs for a and b

a is a RE for $a \in \Sigma$



is a NFA for a

b is a RE for $b \in \Sigma$



is a NFA for b

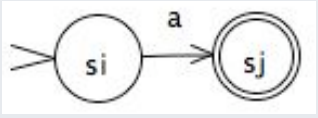
ab is a regular expression

is a NFA for ab

Thompson's construction

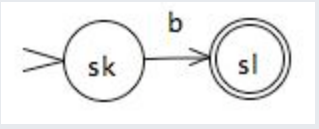
In the following arbitrary NFAs can be used in place of the NFAs for a and b

a is a RE for $a \in \Sigma$



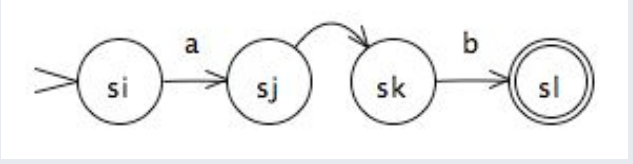
is a NFA for a

b is a RE for $b \in \Sigma$



is a NFA for b

ab is a regular expression



is a NFA for ab

Thompson's construction

$a|b$ is a regular expression

is a NFA for $a|b$

a^* is a regular expression

is a NFA for a^*

Thompson's construction

$a|b$ is a regular expression

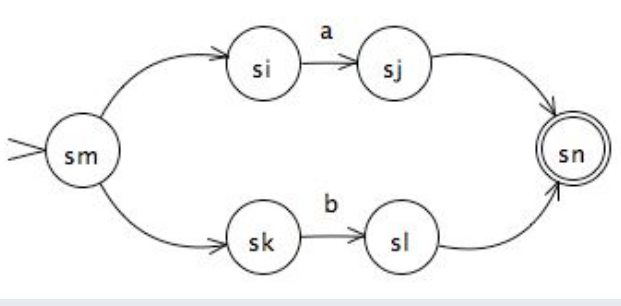
is a NFA for $a|b$

a^* is a regular expression

is a NFA for a^*

Thompson's construction

$a|b$ is a regular expression



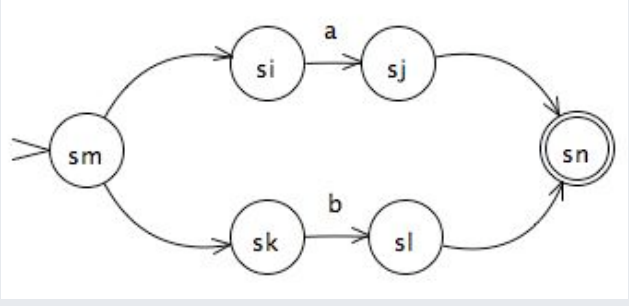
is a NFA for $a|b$

a^* is a regular expression

is a NFA for a^*

Thompson's construction

$a|b$ is a regular expression



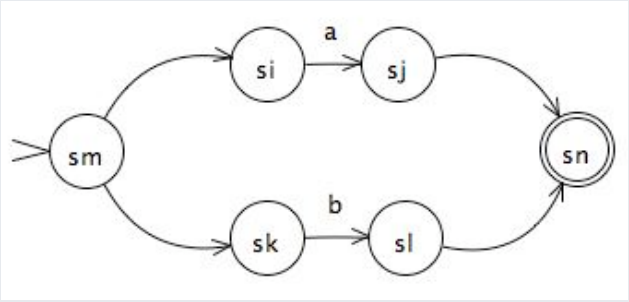
is a NFA for $a|b$

a^* is a regular expression

is a NFA for a^*

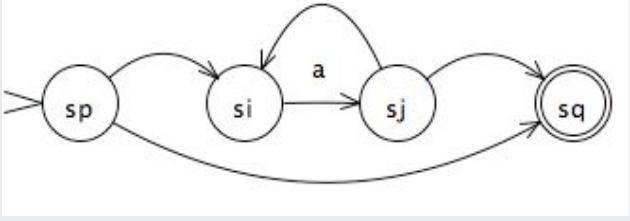
Thompson's construction

$a|b$ is a regular expression



is a NFA for $a|b$

a^* is a regular expression



is a NFA for a^*

And now?

As we see, NFAs recognizing the languages generated by regular expressions are easy to construct!

However, they are no so easy to implement! (How do we deal with guessing?)

And now?

As we see, NFAs recognizing the languages generated by regular expressions are easy to construct!

However, they are no so easy to implement! (How do we deal with guessing?)

And now?

As we see, NFAs recognizing the languages generated by regular expressions are easy to construct!

However, they are no so easy to implement! (How do we deal with guessing?)

NFA to DFA

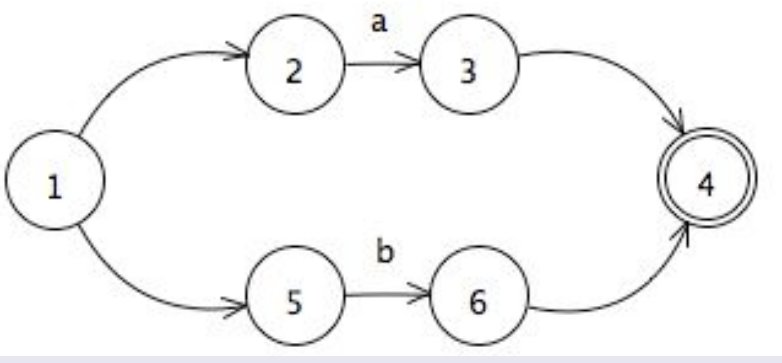
There is a way to transform a NFA to a deterministic automata by simulating that we make all possible choices at once!

Example

NFA to DFA

There is a way to transform a NFA to a deterministic automata by simulating that we make all possible choices at once!

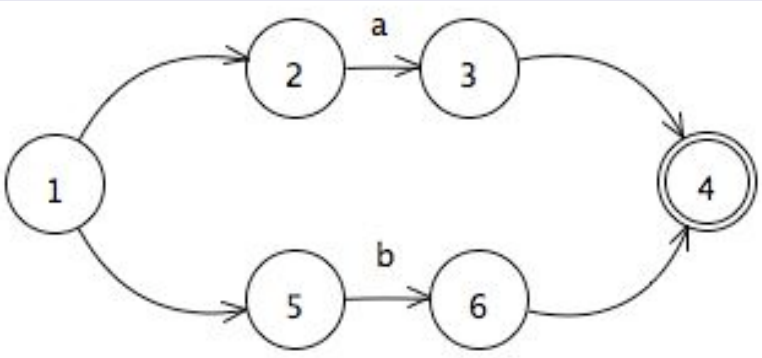
Example

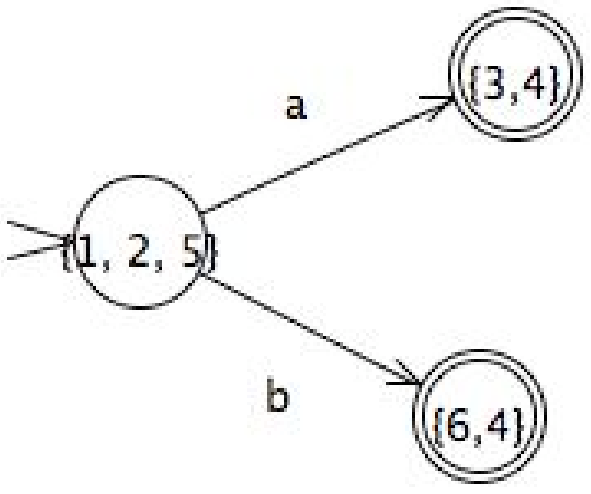


NFA to DFA

There is a way to transform a NFA to a deterministic automata by simulating that we make all possible choices at once!

Example





Why all this?

The lexical structure of computer languages is described using regular expressions.

The first part of the compiler reads a sequence of characters

- Ignores comments and white spaces.
- Finds lexemes that correspond to the lexical structure of the language.
- Generates a sequence of tokens for the rest of the compiler!

It is a deterministic finite state automaton generated by a scanner generator!

Why all this?

The lexical structure of computer languages is described using regular expressions.

The first part of the compiler reads a sequence of characters

- Ignores comments and white spaces.
- Finds lexemes that correspond to the lexical structure of the language.
- Generates a sequence of tokens for the rest of the compiler!

It is a deterministic finite state automaton generated by a scanner generator!

Why all this?

The lexical structure of computer languages is described using regular expressions.

The first part of the compiler reads a sequence of characters

- Ignores comments and white spaces.
- Finds lexemes that correspond to the lexical structure of the language.
- Generates a sequence of tokens for the rest of the compiler!

It is a deterministic finite state automaton generated by a scanner generator!

Why all this?

The lexical structure of computer languages is described using regular expressions.

The first part of the compiler reads a sequence of characters

- Ignores comments and white spaces.
- Finds lexemes that correspond to the lexical structure of the language.
- Generates a sequence of tokens for the rest of the compiler!

It is a deterministic finite state automaton generated by a scanner generator!

Why all this?

The lexical structure of computer languages is described using regular expressions.

The first part of the compiler reads a sequence of characters

- Ignores comments and white spaces.
- Finds lexemes that correspond to the lexical structure of the language.
- Generates a sequence of tokens for the rest of the compiler!

It is a deterministic finite state automaton generated by a scanner generator!

Why all this?

The lexical structure of computer languages is described using regular expressions.

The first part of the compiler reads a sequence of characters

- Ignores comments and white spaces.
- Finds lexemes that correspond to the lexical structure of the language.
- Generates a sequence of tokens for the rest of the compiler!

It is a deterministic finite state automaton generated by a scanner generator!

Some notation

Lexeme

A legal word in a language.

For example, in Java the words

`while`, `class`, `A`, `empty`, `{`

are all lexeme.

Token

A category of lexeme. For example, in Java, there are tokens for

- WHILE where the only lexeme is `while`
- IDENTIFIER where there are infinitely many lexemes, for example `A`, `empty`.
- OPENBRACE where the only lexeme is `{`

Some notation

Lexeme

A legal word in a language.
For example, in Java the words

`while`, `class`, `A`, `empty`, `{`

are all lexeme.

Token

A category of lexeme. For example, in Java, there are tokens for

- WHILE where the only lexeme is `while`
- IDENTIFIER where there are infinitely many lexemes, for example `A`, `empty`.
- OPENBRACE where the only lexeme is `{`

Some notation

Lexeme

A legal word in a language.
For example, in Java the words

`while`, `class`, `A`, `empty`, `{`

are all lexeme.

Token

A category of lexeme. For example, in Java, there are tokens for

- WHILE where the only lexeme is `while`
- IDENTIFIER where there are infinitely many lexemes, for example `A`, `empty`.
- OPENBRACE where the only lexeme is `{`

Some notation

Lexeme

A legal word in a language.
For example, in Java the words

`while`, `class`, `A`, `empty`, `{`

are all lexeme.

Token

A category of lexeme. For example, in Java, there are tokens for

- WHILE where the only lexeme is `while`
- IDENTIFIER where there are infinitely many lexemes, for example `A`, `empty`.
- OPENBRACE where the only lexeme is `{`

Some notation

Lexeme

A legal word in a language.
For example, in Java the words

`while`, `class`, `A`, `empty`, `{`

are all lexeme.

Token

A category of lexeme. For example, in Java, there are tokens for

- WHILE where the only lexeme is `while`
- IDENTIFIER where there are infinitely many lexemes, for example `A`, `empty`.
- OPENBRACE where the only lexeme is `{`

Some notation

Lexeme

A legal word in a language.
For example, in Java the words

`while`, `class`, `A`, `empty`, `{`

are all lexeme.

Token

A category of lexeme. For example, in Java, there are tokens for

- WHILE where the only lexeme is `while`
- IDENTIFIER where there are infinitely many lexemes, for example `A`, `empty`.
- OPENBRACE where the only lexeme is `{`

Some notation

Regular expressions are used to describe the legal lexeme that belong to a token. There will be a regular expression for WHILE, one for IDENTIFIER, one for OPENBRACE.

When we represent tokens, we can use integers (for the token) and some extra info if needed for further understanding of the source (for example, it is not enough with knowing that we saw an identifier, we need to keep track of the lexeme!)

Some notation

Regular expressions are used to describe the legal lexeme that belong to a token. There will be a regular expression for WHILE, one for IDENTIFIER, one for OPENBRACE.

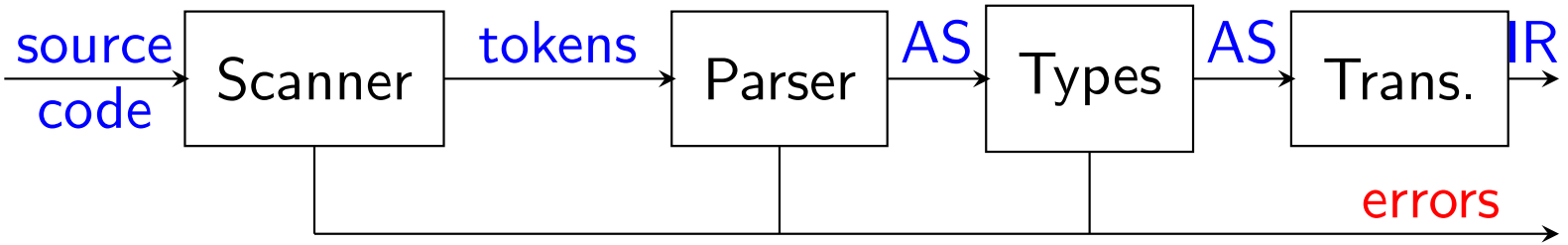
When we represent tokens, we can use integers (for the token) and some extra info if needed for further understanding of the source (for example, it is not enough with knowing that we saw an identifier, we need to keep track of the lexeme!)

Some notation

Regular expressions are used to describe the legal lexeme that belong to a token. There will be a regular expression for WHILE, one for IDENTIFIER, one for OPENBRACE.

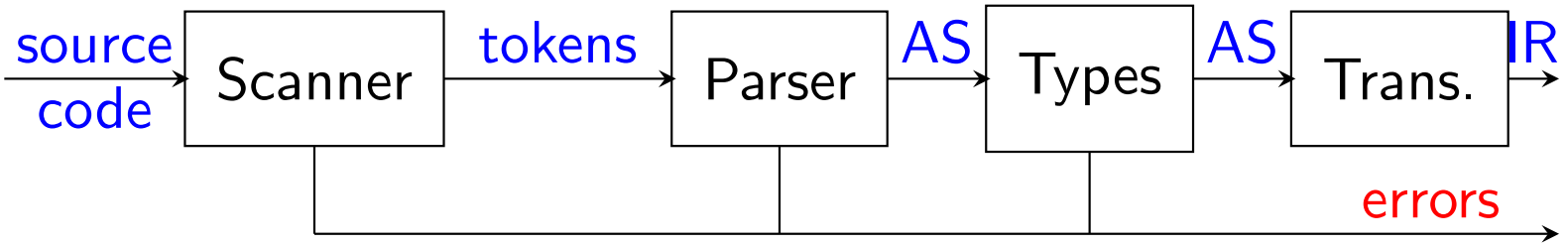
When we represent tokens, we can use integers (for the token) and some extra info if needed for further understanding of the source (for example, it is not enough with knowing that we saw an identifier, we need to keep track of the lexeme!)

The Front End



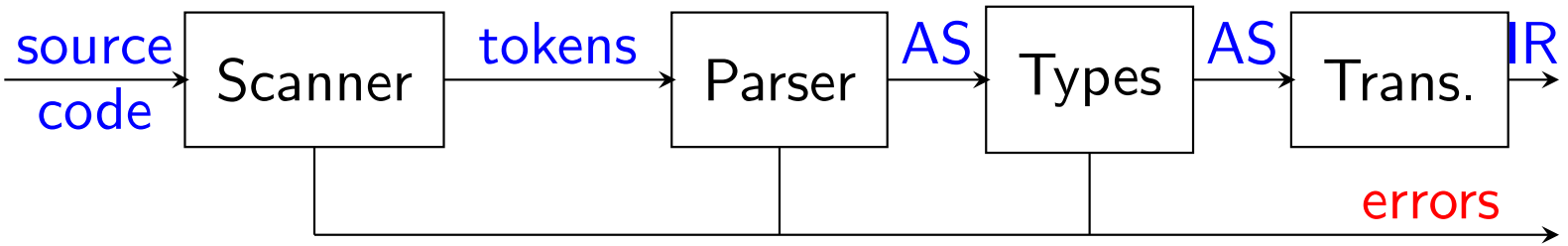
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End



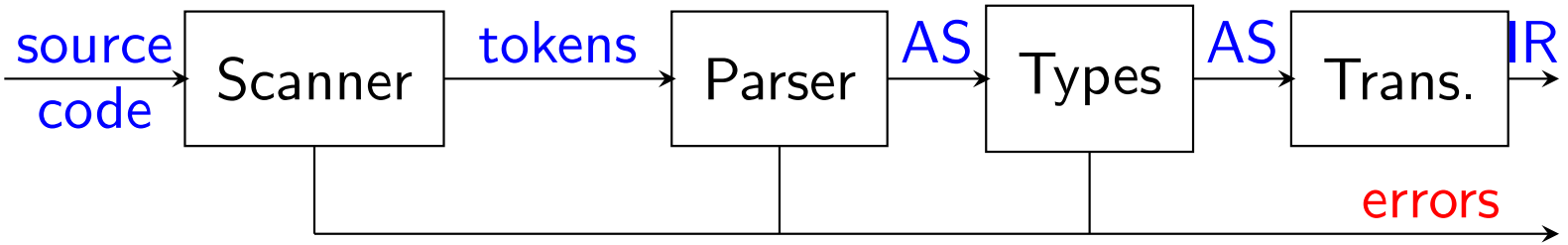
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End



- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End



- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
class ' ' Factorial { '\n' '\t' public ' '
```

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ’ ’ F a c t o r i a l { ’\n’ ’\t’ p u b l i c ’ ’
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ’ ’ F a c t o r i a l { ’\n’ ’\t’ p u b l i c ’ ’
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ’ ’ F a c t o r i a l { ’\n’ ’\t’ p u b l i c ’ ’
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer

Example

```
class Factorial{
   public static void main(String[] a)}
      System.out.println(new Fac().ComputeFac(10));
   }
}
```

```
c l a s s ' ' F a c t o r i a l { '\n' '\t' p u b l i c ' '
```

scanner

```
CLASS   (ID, Factorial)  {   PUBLIC
```

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*?
  www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

The lexical analyzer - cont.

- What are the tokens of *minijava*? www.cambridge.org/resources/052182060X/
  - each keyword is a token
  - each punctuation symbol is a token
  - each operator is a token
  - an identifier is a token *(and we are interested in its value!)*
  - an integer literal is a token *(and we are interested in its value!)*
  - spaces, new lines, tabs and comments are ignored!
- look at the appendices at the end of the book! The terminals for the grammar are the tokens!

Source code for JFlex

JFlex specification

```
usercode
%%
options and declarations
%%
lexical rules
```

Code to be placed at the begining of the class with the generated lexer. (`package` and `imports`.)

%%

Directives to adapt the lexer class to other programs.

Code that will be included in the generated class

Definitions used in regular expressions

%%

Regular expressions and the actions to be taken on recognizing tokens.

Source code for JFlex

JFlex specification

```
usercode
%%
options and declarations
%%
lexical rules
```

Code to be placed at the begining of the class with the generated lexer. (`package` and `imports`.)

%%

Directives to adapt the lexer class to other programs.

Code that will be included in the generated class

Definitions used in regular expressions

%%

Regular expressions and the actions to be taken on recognizing tokens.

Source code for JFlex

JFlex specification

```
usercode
%%
options and declarations
%%
lexical rules
```

Code to be placed at the begining of the class with the generated lexer. (`package` and `imports`.)

%%

Directives to adapt the lexer class to other programs.

Code that will be included in the generated class

Definitions used in regular expressions

%%

Regular expressions and the actions to be taken on recognizing tokens.

Source code for JFlex

JFlex specification

```
usercode
%%
options and declarations
%%
lexical rules
```

Code to be placed at the begining of the class with the generated lexer. (`package` and `imports`.)

%%

Directives to adapt the lexer class to other programs.

Code that will be included in the generated class

Definitions used in regular expressions

%%

Regular expressions and the actions to be taken on recognizing tokens.

Source code for JFlex

JFlex specification

```
usercode
%%
options and declarations
%%
lexical rules
```

Code to be placed at the begining of the class with the generated lexer. (`package` and `imports`.)

%%

Directives to adapt the lexer class to other programs.

Code that will be included in the generated class

Definitions used in regular expressions

%%

Regular expressions and the actions to be taken on recognizing tokens.

Source code for JFlex

JFlex specification

```
usercode
%%
options and declarations
%%
lexical rules
```

Code to be placed at the begining of the class with the generated lexer. (`package` and `imports`.)

%%

Directives to adapt the lexer class to other programs.

Code that will be included in the generated class

Definitions used in regular expressions

%%

Regular expressions and the actions to be taken on recognizing tokens.

Source code for JFlex

JFlex specification

```
usercode
%%
options and declarations
%%
lexical rules
```

Code to be placed at the begining of the class with the generated lexer. (`package` and `imports`.)

%%

Directives to adapt the lexer class to other programs.

Code that will be included in the generated class

Definitions used in regular expressions

%%

Regular expressions and the actions to be taken on recognizing tokens.

```
%%
%debug
%class      minijavaLexer
%implements minijavaTokens
%int
%unicode
%line
%column
%{
Object semanticValue;
int token;
%}
nl  = \n | \r | \r \n
nls = nl | [ \ f \ t]
%%
"class"      {return token = CLASS;}
"+"          {return token = '+';}
{nls}        {/* ignore new lines and spaces */}
```

```
interface minijavaTokens {
    int ENDINPUT = 0;
    int CLASS = 1;
    int error = 2;
    // '+' (code=43)
}
```