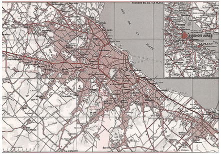
Computer Languages

Describing Syntax
Context Free Grammars

January 27

Plan



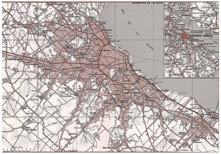
What we know

1. How we can describe the words that can be used in a computer language.
2. How to generate programs that recognize legal words in source code.
3. How to use such a program to generate a sequence of tokens and eliminate irrelevant fragments (white spaces, new-lines, comments).

What we will learn today

1. How we can describe the valid sentences on a computer language.
2. That we can use a parser generator to use this descriptions and get a program that does things while recognizing legal source code.

Plan



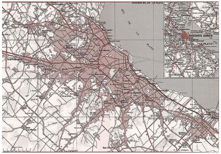
What we know

1. How we can describe the words that can be used in a computer language.
2. How to generate programs that recognize legal words in source code.
3. How to use such a program to generate a sequence of tokens and eliminate irrelevant fragments (white spaces, new-lines, comments).

What we will learn today

1. How we can describe the valid sentences on a computer language.
2. That we can use a parser generator to use this descriptions and get a program that does things while recognizing legal source code.

Plan



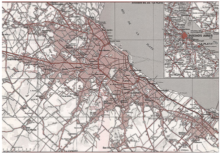
What we know

1. How we can describe the words that can be used in a computer language.
2. How to generate programs that recognize legal words in source code.
3. How to use such a program to generate a sequence of tokens and eliminate irrelevant fragments (white spaces, new-lines, comments).

What we will learn today

1. How we can describe the valid sentences on a computer language.
2. That we can use a parser generator to use this descriptions and get a program that does things while recognizing legal source code.

Plan



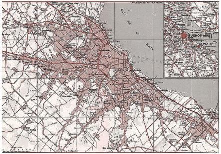
What we know

1. How we can describe the words that can be used in a computer language.
2. How to generate programs that recognize legal words in source code.
3. How to use such a program to generate a sequence of tokens and eliminate irrelevant fragments (white spaces, new-lines, comments).

What we will learn today

1. How we can describe the valid sentences on a computer language.
2. That we can use a parser generator to use this descriptions and get a program that does things while recognizing legal source code.

Plan



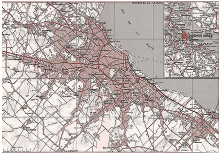
What we know

1. How we can describe the words that can be used in a computer language.
2. How to generate programs that recognize legal words in source code.
3. How to use such a program to generate a sequence of tokens and eliminate irrelevant fragments (white spaces, new-lines, comments).

What we will learn today

1. How we can describe the valid sentences on a computer language.
2. That we can use a parser generator to use this descriptions and get a program that does things while recognizing legal source code.

Plan



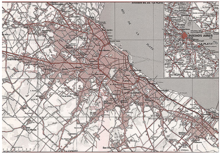
What we know

1. How we can describe the words that can be used in a computer language.
2. How to generate programs that recognize legal words in source code.
3. How to use such a program to generate a sequence of tokens and eliminate irrelevant fragments (white spaces, new-lines, comments).

What we will learn today

1. How we can describe the valid sentences on a computer language.
2. That we can use a parser generator to use this descriptions and get a program that does things while recognizing legal source code.

Plan



What we know

1. How we can describe the words that can be used in a computer language.
2. How to generate programs that recognize legal words in source code.
3. How to use such a program to generate a sequence of tokens and eliminate irrelevant fragments (white spaces, new-lines, comments).

What we will learn today

1. How we can describe the valid sentences on a computer language.
2. That we can use a parser generator to use this descriptions and get a program that does things while recognizing legal source code.

Context-Free Grammars

We need a notation

- that can capture the syntactic structure of computer languages
- and that leads to efficient recognizers.

Regular expressions are not powerful enough!

A context-free grammar G is a set of rules describing how to form sentences. The set of all these sentences L(G) is *the language defined by G*.

The rules are of a special form!

Example

$$SN \rightarrow \texttt{mbää}\ SN \mid \texttt{mbää}$$

Context-Free Grammars

We need a notation

- that can capture the syntactic structure of computer languages
- and that leads to efficient recognizers.

Regular expressions are not powerful enough!

A context-free grammar G is a set of rules describing how to form sentences. The set of all these sentences L(G) is *the language defined by G*.

The rules are of a special form!

Example

$$SN \rightarrow \texttt{mbää}\ SN \mid \texttt{mbää}$$

Context-Free Grammars

We need a notation

- that can capture the syntactic structure of computer languages
- and that leads to efficient recognizers.

Regular expressions are not powerful enough!

A context-free grammar G is a set of rules describing how to form sentences. The set of all these sentences L(G) is *the language defined by G*.

The rules are of a special form!

Example

$$SN \rightarrow \texttt{mbää}\ SN \mid \texttt{mbää}$$

Context-Free Grammars

We need a notation

- that can capture the syntactic structure of computer languages
- and that leads to efficient recognizers.

Regular expressions are not powerful enough!

A context-free grammar G is a set of rules describing how to form sentences. The set of all these sentences L(G) is *the language defined by G*.

The rules are of a special form!

Example

SN	→	`mbää` *SN*
	|	`mbää`

Context-Free Grammars

We need a notation

- that can capture the syntactic structure of computer languages
- and that leads to efficient recognizers.

Regular expressions are not powerful enough!

A context-free grammar G is a set of rules describing how to form sentences. The set of all these sentences L(G) is *the language defined by G*.

The rules are of a special form!

Example

$$SN \rightarrow \texttt{mbää}\ SN \mid \texttt{mbää}$$

Context-Free Grammars

We need a notation

- that can capture the syntactic structure of computer languages
- and that leads to efficient recognizers.

Regular expressions are not powerful enough!

A context-free grammar G is a set of rules describing how to form sentences. The set of all these sentences L(G) is *the language defined by G*.

The rules are of a special form!

Example

$$SN \rightarrow \texttt{mbää}\ SN \mid \texttt{mbää}$$

Context-Free Grammars

We need a notation

- that can capture the syntactic structure of computer languages
- and that leads to efficient recognizers.

Regular expressions are not powerful enough!

A context-free grammar G is a set of rules describing how to form sentences. The set of all these sentences L(G) is *the language defined by G*.

The rules are of a special form!

Example

$$
\begin{array}{lcl}
SN & \rightarrow & \texttt{mbää}\ SN \\
 & | & \texttt{mbää}
\end{array}
$$

Example

$$SN \rightarrow \texttt{mbää}\ SN \mid \texttt{mbää}$$

$SN \rightarrow$ mbää SN
is called a production and is said to derive sentences built by the word mbää followed by more SN.

SN is like a variable standing for a *type of sentences* or *syntactic category*. It is called a non-terminal.

The words that appear in the sentences, e.g. mbää, are called terminals.

Example

$$SN \rightarrow \texttt{mbää}\ SN$$
$$\mid \texttt{mbää}$$

$SN \rightarrow$ `mbää` SN
is called a production and is said to derive sentences built by the word `mbää` followed by more SN.

SN is like a variable standing for a *type of sentences* or *syntactic category*. It is called a non-terminal.

The words that appear in the sentences, e.g. `mbää`, are called terminals.

Example

$$SN \rightarrow \texttt{mbää}\ SN$$
$$\mid\ \texttt{mbää}$$

$SN \rightarrow \texttt{mbää}\ SN$
is called a production and is said to derive sentences built by the word mbää followed by more *SN*.

SN is like a variable standing for a *type of sentences* or *syntactic category*. It is called a non-terminal.

The words that appear in the sentences, e.g. mbää, are called terminals.

Example

$$SN \rightarrow \texttt{mbää}\ SN \mid \texttt{mbää}$$

$SN \rightarrow$ `mbää` SN
is called a production and is said to derive sentences built by the word `mbää` followed by more SN.

SN is like a variable standing for a *type of sentences* or *syntactic category*. It is called a non-terminal.

The words that appear in the sentences, e.g. `mbää`, are called terminals.

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	→	mbää *SN*
2		|	mbää

Example

Prod.	String
	SN
1	mbää *SN*
1	mbää mbää *SN*
1	mbää mbää mbää *SN*
2	mbää mbää mbää mbää

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	→	`mbää` *SN*
2		|	`mbää`

Example

Prod.	String
	SN
1	`mbää` *SN*
1	`mbää mbää` *SN*
1	`mbää mbää mbää` *SN*
2	`mbää mbää mbää mbää`

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	SN	$\rightarrow$	`mbää` SN
2		$\mid$	`mbää`

Example

Prod.	String
	SN
1	`mbää` SN
1	`mbää mbää` SN
1	`mbää mbää mbää` SN
2	`mbää mbää mbää mbää`

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	→	`mbää` *SN*
2		|	`mbää`

Example

Prod.	String
	SN
1	mbää *SN*
1	mbää mbää *SN*
1	mbää mbää mbää *SN*
2	mbää mbää mbää mbää

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	$\longrightarrow$	`mbää` *SN*
2		|	`mbää`

Example

Prod.	**String**
	SN
1	`mbää` *SN*
1	`mbää mbää` *SN*
1	`mbää mbää mbää` *SN*
2	`mbää mbää mbää mbää`

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	$\rightarrow$	`mbää` *SN*
2		|	`mbää`

Example

Prod.	String
	SN
1	`mbää` *SN*
1	`mbää mbää` *SN*
1	`mbää mbää mbää` *SN*
2	`mbää mbää mbää mbää`

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	$\rightarrow$	`mbää` *SN*
2		|	`mbää`

Example

Prod.	String
	SN
1	`mbää` *SN*
1	`mbää mbää` *SN*
1	`mbää mbää mbää` *SN*
2	`mbää mbää mbää mbää`

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	→	`mbää` *SN*
2		|	`mbää`

Example

Prod.	String
	SN
1	`mbää` *SN*
1	`mbää mbää` *SN*
1	`mbää mbää mbää` *SN*
2	`mbää mbää mbää mbää`

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	→	`mbää` *SN*
2		|	`mbää`

Example

Prod.	**String**
	SN
1	`mbää` *SN*
1	`mbää mbää` *SN*
1	`mbää mbää mbää` *SN*
2	`mbää mbää mbää mbää`

Deriving sentences

To derive a sentence

1. Start with the start symbol (*one non-terminal!*) and replace it by one right hand side in a production.
2. Pick a non-terminal in the string and replace it by the right hand side of one of its productions.
3. Continue like this until there are no more non-terminals in the string.

Example

1	*SN*	$\rightarrow$	`mbää` *SN*
2		|	`mbää`

Example

Prod.	String
	SN
1	`mbää` *SN*
1	`mbää mbää` *SN*
1	`mbää mbää mbää` *SN*
2	`mbää mbää mbää mbää`

More formally

A context-free grammar consists of four parts T, NT, s and P.

T, the set of terminal symbols (words, tokens).

NT, the set of non-terminal symbols (syntactic categories)

s, the start symbol (goal), one non-terminal standing for the syntactic category whose sentences we are describing.

P, the set of productions. Each member of P maps one non terminal onto a string formed by terminals and nonterminals.

More formally

A context-free grammar consists of four parts T, NT, s and P.

T, the set of terminal symbols (words, tokens).

NT, the set of non-terminal symbols (syntactic categories)

s, the start symbol (goal), one non-terminal standing for the syntactic category whose sentences we are describing.

P, the set of productions. Each member of P maps one non terminal onto a string formed by terminals and nonterminals.

More formally

A context-free grammar consists of four parts T, NT, s and P.

T, the set of terminal symbols (words, tokens).

NT, the set of non-terminal symbols (syntactic categories)

s, the start symbol (goal), one non-terminal standing for the syntactic category whose sentences we are describing.

P, the set of productions. Each member of P maps one non terminal onto a string formed by terminals and nonterminals.

More formally

A context-free grammar consists of four parts T, NT, s and P.

T, the set of terminal symbols (words, tokens).

NT, the set of non-terminal symbols (syntactic categories)

s, the start symbol (goal), one non-terminal standing for the syntactic category whose sentences we are describing.

P, the set of productions. Each member of P maps one non terminal onto a string formed by terminals and nonterminals.

More formally

A context-free grammar consists of four parts T, NT, s and P.

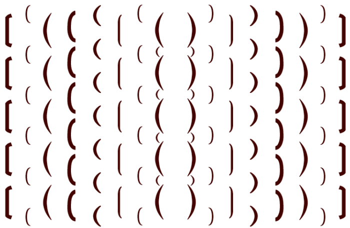
T, the set of terminal symbols (words, tokens).

NT, the set of non-terminal symbols (syntactic categories)

s, the start symbol (goal), one non-terminal standing for the syntactic category whose sentences we are describing.

P, the set of productions. Each member of P maps one non terminal onto a string formed by terminals and nonterminals.

Balanced parentheses



Example

$$
\begin{array}{lcl}
\textit{Paren} & \rightarrow & (\ \textit{Bracket}\ ) \\
 & | & (\ ) \\
\textit{Bracket} & \rightarrow & [\ \textit{Paren}\ ] \\
 & | & [\ ]
\end{array}
$$

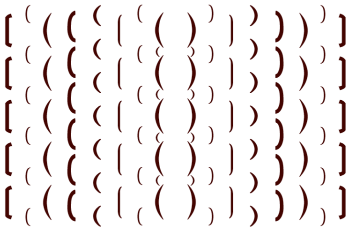
Depending on what start symbol we choose we get different languages! *Paren* forces outermost parentheses. *Bracket* forces outermost brackets.

Example

$$
\begin{array}{lcl}
S & \rightarrow & \textit{Paren} \\
 & | & \textit{Bracket}
\end{array}
$$

allows both!

Balanced parentheses



Example

$$
\begin{array}{lcl}
\textit{Paren} & \rightarrow & (\ \textit{Bracket}\ ) \\
 & \mid & (\ ) \\
\textit{Bracket} & \rightarrow & [\ \textit{Paren}\ ] \\
 & \mid & [\ ]
\end{array}
$$

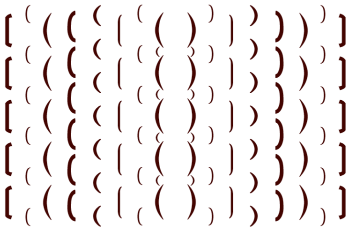
Depending on what start symbol we choose we get different languages!
Paren forces outermost parentheses.
Bracket forces outermost brackets.

Example

$$
\begin{array}{lcl}
S & \rightarrow & \textit{Paren} \\
 & \mid & \textit{Bracket}
\end{array}
$$

allows both!

Balanced parentheses



Example

$$
\begin{array}{lcl}
\mathit{Paren} & \rightarrow & (\ \mathit{Bracket}\ ) \\
 & | & (\ ) \\
\mathit{Bracket} & \rightarrow & [\ \mathit{Paren}\ ] \\
 & | & [\ ]
\end{array}
$$

Depending on what start symbol we choose we get different languages!
Paren forces outermost parentheses.
Bracket forces outermost brackets.

Example

$$
\begin{array}{lcl}
S & \rightarrow & \mathit{Paren} \\
 & | & \mathit{Bracket}
\end{array}
$$

allows both!

Boolean expressions

Example

```
true & true | false
true | true & false
¬ true & ¬ false
```

A CFG for boolean expressions

1	*Bexp*	→	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	`true`
5		|	`false`

Example

Prod.	String
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & `false`
4	*Bexp* | `true` & `false`
4	`true` | `true` & `false`

Boolean expressions

Example

`true & true | false`

`true | true & false`

`¬ true & ¬ false`

A CFG for boolean expressions

1	*Bexp*	→	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	`true`
5		|	`false`

Example

Prod.	String
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & `false`
4	*Bexp* | `true & false`
4	`true	true & false`

Boolean expressions

Example

true & true | false
true | true & false
¬ true & ¬ false

A CFG for boolean expressions

1	*Bexp*	→	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	true
5		|	false

Example

Prod.	String
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & false
4	*Bexp* | true & false
4	true | true & false

Boolean expressions

A CFG for boolean expressions

1	*Bexp*	$\rightarrow$	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	`true`
5		|	`false`

Example

`true & true | false`
`true | true & false`
`¬ true & ¬ false`

Example

Prod.	**String**
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & `false`
4	*Bexp* | `true & false`
4	`true	true & false`

Boolean expressions

Example

```
true & true | false
true | true & false
¬ true & ¬ false
```

A CFG for boolean expressions

1	*Bexp*	$\rightarrow$	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	`true`
5		|	`false`

Example

Prod.	**String**
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & `false`
4	*Bexp* | `true & false`
4	`true` | `true & false`

Boolean expressions

Example

`true & true | false`
`true | true & false`
`¬ true & ¬ false`

A CFG for boolean expressions

1	*Bexp*	$\rightarrow$	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	`true`
5		|	`false`

Example

Prod.	**String**
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & `false`
4	*Bexp* | `true` & `false`
4	`true` | `true` & `false`

Boolean expressions

A CFG for boolean expressions

1	*Bexp*	$\rightarrow$	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	`true`
5		|	`false`

Example

`true & true | false`
`true | true & false`
`¬ true & ¬ false`

Example

Prod.	String
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & `false`
4	*Bexp* | `true & false`
4	`true	true & false`

Boolean expressions

Example

true & true | false
true | true & false
¬ true & ¬ false

A CFG for boolean expressions

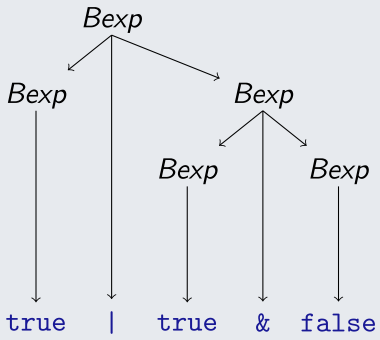
1	*Bexp*	→	*Bexp* & *Bexp*
2		|	*Bexp* | *Bexp*
3		|	¬ *Bexp*
4		|	true
5		|	false

Example

Prod.	String
	Bexp
2	*Bexp* | *Bexp*
1	*Bexp* | *Bexp* & *Bexp*
5	*Bexp* | *Bexp* & false
4	*Bexp* | true & false
4	true | true & false

Parse trees

We can depict the derivation



Structure and meaning

This parse tree will lead the way we understand the source code!

Example

What is the value of `true | true & false`?

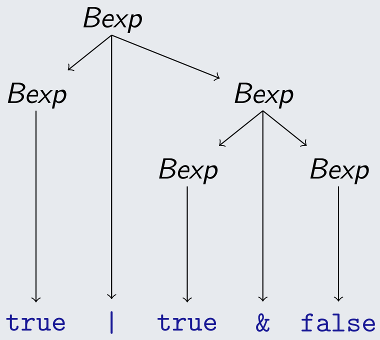
Traverse the tree in post-order calculating values!

Example

`true`

Parse trees

We can depict the derivation



Structure and meaning

This parse tree will lead the way we understand the source code!

Example

What is the value of `true | true & false`?

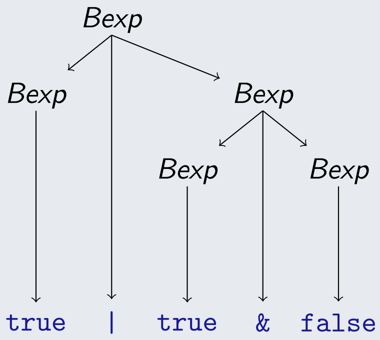
Traverse the tree in post-order calculating values!

Example

`true`

Parse trees

We can depict the derivation



Structure and meaning

This parse tree will lead the way we understand the source code!

Example

What is the value of `true | true & false`?

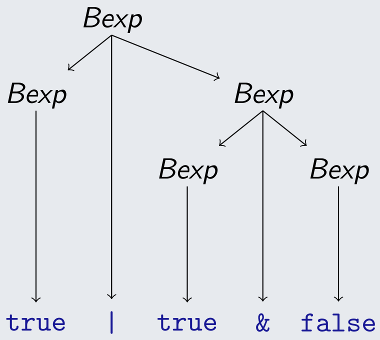
Traverse the tree in post-order calculating values!

Example

`true`

Parse trees

We can depict the derivation



Structure and meaning

This parse tree will lead the way we understand the source code!

Example

What is the value of `true | true & false`?

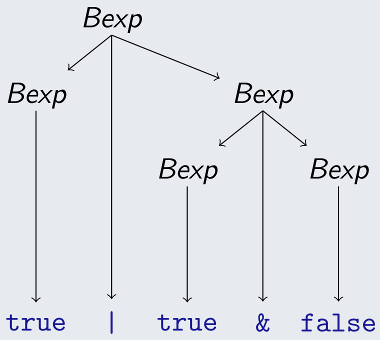
Traverse the tree in post-order calculating values!

Example

`true`

Parse trees

We can depict the derivation



Structure and meaning

This parse tree will lead the way we understand the source code!

Example

What is the value of `true | true & false`?

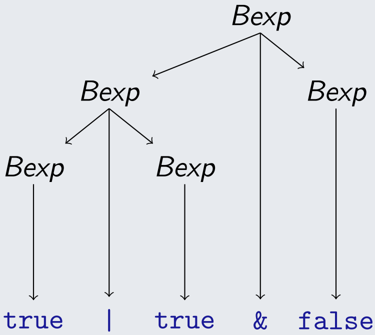
Traverse the tree in post-order calculating values!

Example

`true`

Ambiguity

What about this parse tree?



Example

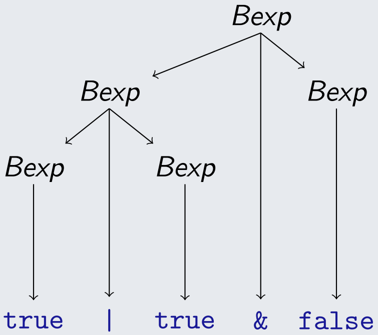
Results in `false`!

A grammar where more than one parse tree is possible for a given sentence is said to be ambiguous.

We will try to avoid them! Source code writers could not be certain about how the compiler interpreted the source!

Ambiguity

What about this parse tree?



Example

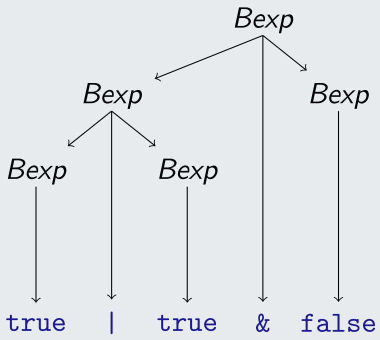
Results in `false`!

A grammar where more than one parse tree is possible for a given sentence is said to be ambiguous.

We will try to avoid them! Source code writers could not be certain about how the compiler interpreted the source!

Ambiguity

What about this parse tree?



Example

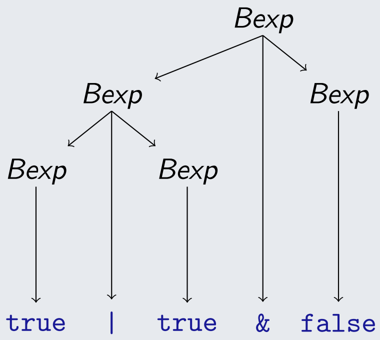
Results in `false`!

A grammar where more than one parse tree is possible for a given sentence is said to be ambiguous.

We will try to avoid them! Source code writers could not be certain about how the compiler interpreted the source!

Ambiguity

What about this parse tree?



Example

Results in `false`!

A grammar where more than one parse tree is possible for a given sentence is said to be ambiguous.

We will try to avoid them! Source code writers could not be certain about how the compiler interpreted the source!

Rearranging the grammar

A new grammar can be designed for the same language.

We have in mind the conventions we are used to for

- associativity
- precedence

to avoid the need for too many parenthesis.

- & and | associate to the left.
- & has higher precedence than |.
- ¬ has higher precedence than &.

$$
\begin{array}{lcl}
Bexp & \rightarrow & Bexp \mid Conj \\
 & \mid & Conj \\
Conj & \rightarrow & Conj \,\&\, Neg \\
 & \mid & Neg \\
Neg & \rightarrow & \neg\, Atom \\
 & \mid & Atom \\
Atom & \rightarrow & \texttt{true} \\
 & \mid & \texttt{false} \\
 & \mid & (Bexp)
\end{array}
$$

Try the parse tree for the same sentence we inspected before!

Rearranging the grammar

A new grammar can be designed for the same language.

We have in mind the conventions we are used to for

- associativity
- precedence

to avoid the need for too many parenthesis.

- & and | associate to the left.
- & has higher precedence than |.
- ¬ has higher precedence than &.

$$
\begin{array}{lcl}
Bexp & \rightarrow & Bexp \mid Conj \\
 & \mid & Conj \\
Conj & \rightarrow & Conj \;\&\; Neg \\
 & \mid & Neg \\
Neg & \rightarrow & \neg\ Atom \\
 & \mid & Atom \\
Atom & \rightarrow & \texttt{true} \\
 & \mid & \texttt{false} \\
 & \mid & (Bexp)
\end{array}
$$

Try the parse tree for the same sentence we inspected before!

Rearranging the grammar

A new grammar can be designed for the same language.

We have in mind the conventions we are used to for

- associativity
- precedence

to avoid the need for too many parenthesis.

- & and | associate to the left.
- & has higher precedence than |.
- ¬ has higher precedence than &.

$$
\begin{array}{lcl}
Bexp & \rightarrow & Bexp \mid Conj \\
 & \mid & Conj \\
Conj & \rightarrow & Conj \; \& \; Neg \\
 & \mid & Neg \\
Neg & \rightarrow & \neg \; Atom \\
 & \mid & Atom \\
Atom & \rightarrow & \texttt{true} \\
 & \mid & \texttt{false} \\
 & \mid & (Bexp)
\end{array}
$$

Try the parse tree for the same sentence we inspected before!

Rearranging the grammar

A new grammar can be designed for the same language.

We have in mind the conventions we are used to for

- associativity
- precedence

to avoid the need for too many parenthesis.

- & and | associate to the left.
- & has higher precedence than |.
- ¬ has higher precedence than &.

$$
\begin{array}{lcl}
Bexp & \rightarrow & Bexp \mid Conj \\
 & | & Conj \\
Conj & \rightarrow & Conj \;\&\; Neg \\
 & | & Neg \\
Neg & \rightarrow & \neg\ Atom \\
 & | & Atom \\
Atom & \rightarrow & \texttt{true} \\
 & | & \texttt{false} \\
 & | & (Bexp)
\end{array}
$$

Try the parse tree for the same sentence we inspected before!

Rearranging the grammar

A new grammar can be designed for the same language.

We have in mind the conventions we are used to for

- associativity
- precedence

to avoid the need for too many parenthesis.

- & and | associate to the left.
- & has higher precedence than |.
- ¬ has higher precedence than &.

$$
\begin{array}{lcl}
Bexp & \rightarrow & Bexp \mid Conj \\
 & | & Conj \\
Conj & \rightarrow & Conj \ \& \ Neg \\
 & | & Neg \\
Neg & \rightarrow & \neg \ Atom \\
 & | & Atom \\
Atom & \rightarrow & \texttt{true} \\
 & | & \texttt{false} \\
 & | & (Bexp)
\end{array}
$$

Try the parse tree for the same sentence we inspected before!

Using a parser generator

From a context free grammar a program can be generated that recognizes the sentences of the language described by the grammar!

The generated program is called a parser. The generating program is called a parser generator

We will use jacc with special permission of Mark P. Jones from OHSU.

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg  : '-' atom
     | atom
     ;
atom : TRUE|FALSE|'('bexp')';
%%
```

Using a parser generator

From a context free grammar a program can be generated that recognizes the sentences of the language described by the grammar!

The generated program is called a parser. The generating program is called a parser generator

We will use jacc with special permission of Mark P. Jones from OHSU.

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg  : '-' atom
     | atom
     ;
atom : TRUE|FALSE|'('bexp')';
%%
```

Using a parser generator

From a context free grammar a program can be generated that recognizes the sentences of the language described by the grammar!

The generated program is called a parser. The generating program is called a parser generator

We will use `jacc` with special permission of Mark P. Jones from OHSU.

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg  : '-' atom
     | atom
     ;
atom : TRUE|FALSE|'('bexp')';
%%
```

Using a parser generator

From a context free grammar a program can be generated that recognizes the sentences of the language described by the grammar!

The generated program is called a parser. The generating program is called a parser generator

We will use jacc with special permission of Mark P. Jones from OHSU.

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%%
bexp : bexp '|' conj
     | conj
     ;
conj : conj '&' neg
     | neg
     ;
neg  : '-' atom
     | atom
     ;
atom : TRUE|FALSE|'('bexp')';
%%
```

Using a parser generator

From a context free grammar a program can be generated that recognizes the sentences of the language described by the grammar!

The generated program is called a parser. The generating program is called a parser generator

We will use `jacc` with special permission of Mark P. Jones from OHSU.

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%%
bexp  : bexp '|' conj
      | conj
      ;
conj  : conj '&' neg
      | neg
      ;
neg   : '-' atom
      | atom
      ;
atom  : TRUE|FALSE|'('bexp')';
%%
```

Precedence and Associativity

The kind of ambiguity we discussed for binary expressions arised from using binary infix operators.

The solution, including conventions for avoiding too many parenthesis, is standard.

The modified grammar can be generated automatically if proper directives are given!

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'

%left '|'
%left '&'
%nonassoc '-'
%%
bexp : bexp '|' bexp
     | bexp '&' bexp
     | '-' bexp
     | TRUE
     | FALSE
     | '(' bexp ')'
     ;
%%
```

Precedence and Associativity

The kind of ambiguity we discussed for binary expressions arised from using binary infix operators.

The solution, including conventions for avoiding too many parenthesis, is standard.

The modified grammar can be generated automatically if proper directives are given!

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'

 %left '|'
 %left '&'
 %nonassoc '-'
%%
bexp : bexp '|' bexp
     | bexp '&' bexp
     | '-' bexp
     | TRUE
     | FALSE
     | '(' bexp ')'
     ;
%%
```

Precedence and Associativity

The kind of ambiguity we discussed for binary expressions arised from using binary infix operators.

The solution, including conventions for avoiding too many parenthesis, is standard.

The modified grammar can be generated automatically if proper directives are given!

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
 %left '|'
 %left '&'
 %nonassoc '-'
%%
bexp : bexp '|' bexp
     | bexp '&' bexp
     | '-' bexp
     | TRUE
     | FALSE
     | '(' bexp ')'
     ;
%%
```

Precedence and Associativity

The kind of ambiguity we discussed for binary expressions arised from using binary infix operators.

The solution, including conventions for avoiding too many parenthesis, is standard.

The modified grammar can be generated automatically if proper directives are given!

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%left '|'
%left '&'
%nonassoc '-'
%%
bexp : bexp '|' bexp
     | bexp '&' bexp
     | '-' bexp
     | TRUE
     | FALSE
     | '(' bexp ')'
     ;
%%
```

Precedence and Associativity

The kind of ambiguity we discussed for binary expressions arised from using binary infix operators.

The solution, including conventions for avoiding too many parenthesis, is standard.

The modified grammar can be generated automatically if proper directives are given!

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%left '|'
%left '&'
%nonassoc '-'
%%
bexp : bexp '|' bexp
     | bexp '&' bexp
     | '-' bexp
     | TRUE
     | FALSE
     | '(' bexp ')'
     ;
%%
```

Precedence and Associativity

The kind of ambiguity we discussed for binary expressions arised from using binary infix operators.

The solution, including conventions for avoiding too many parenthesis, is standard.

The modified grammar can be generated automatically if proper directives are given!

```
%token TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%left '|'
%left '&'
%nonassoc '-'
%%
bexp : bexp '|' bexp
     | bexp '&' bexp
     | '-' bexp
     | TRUE
     | FALSE
     | '(' bexp ')'
     ;
%%
```

Using jacc

As it is we have not said how to connect to a lexer generating tokens!

We can anyway test our grammar without generating a parser and connecting it to a lexer!

The file containing our sentence must consist of tokens and nonterminals

Example

An input file for the *parser* could be

```
TRUE '&' bexp '|' FALSE
```

Example

And the way of using jacc for recognizing the sentences described with the cfg is to use the command

```
jacc -pt bexpP.jacc -r test1
```

Using jacc

As it is we have not said how to connect to a lexer generating tokens!

We can anyway test our grammar without generating a parser and connecting it to a lexer!

The file containing our sentence must consist of tokens and nonterminals

Example

An input file for the *parser* could be

```
TRUE '&' bexp '|' FALSE
```

Example

And the way of using jacc for recognizing the sentences described with the cfg is to use the command

```
jacc -pt bexpP.jacc -r test1
```

Using jacc

As it is we have not said how to connect to a lexer generating tokens!

We can anyway test our grammar without generating a parser and connecting it to a lexer!

The file containing our sentence must consist of tokens and nonterminals

Example

An input file for the *parser* could be

```
TRUE '&' bexp '|' FALSE
```

Example

And the way of using `jacc` for recognizing the sentences described with the cfg is to use the command

```
jacc -pt bexpP.jacc -r test1
```

Using jacc

As it is we have not said how to connect to a lexer generating tokens!

We can anyway test our grammar without generating a parser and connecting it to a lexer!

The file containing our sentence must consist of tokens and nonterminals

Example

An input file for the *parser* could be

```
TRUE '&' bexp '|' FALSE
```

Example

And the way of using jacc for recognizing the sentences described with the cfg is to use the command

```
jacc -pt bexpP.jacc -r test1
```

Using jacc

As it is we have not said how to connect to a lexer generating tokens!

We can anyway test our grammar without generating a parser and connecting it to a lexer!

The file containing our sentence must consist of tokens and nonterminals

Example

An input file for the *parser* could be

```
TRUE '&' bexp '|' FALSE
```

Example

And the way of using `jacc` for recognizing the sentences described with the cfg is to use the command

```
jacc -pt bexpP.jacc -r test1
```

```
Running example from "test1.be"
start  :  _ TRUE ...
shift  :  TRUE _ '&' ...
reduce :  _ bexp '&' ...
goto   :  bexp _ '&' ...
shift  :  bexp '&' _ bexp ...
goto   :  bexp '&' bexp _ '|' ...
reduce :  _ bexp '|' ...
goto   :  bexp _ '|' ...
shift  :  bexp '|' _ FALSE ...
shift  :  bexp '|' FALSE _
reduce :  bexp '|' _ bexp $end
goto   :  bexp '|' bexp _ $end
reduce :  _ bexp $end
goto   :  bexp _ $end
Accept!
```

Generating a parser

If we want to generate a parser, we have to connect it to a lexer that provides the tokens!

We have to use directives and program a little to do so!

We will at the same time see how to use the parser to compute while recognizing structure!

Directives:

Generating a parser

If we want to generate a parser, we have to connect it to a lexer that provides the tokens!

We have to use directives and program a little to do so!

We will at the same time see how to use the parser to compute while recognizing structure!

Directives:

Generating a parser

If we want to generate a parser, we have to connect it to a lexer that provides the tokens!

We have to use directives and program a little to do so!

We will at the same time see how to use the parser to compute while recognizing structure!

Generating a parser

If we want to generate a parser, we have to connect it to a lexer that provides the tokens!

We have to use directives and program a little to do so!

We will at the same time see how to use the parser to compute while recognizing structure!

Directives:

```
%class      Evaluator
%interface  BooleanTokens
%next       nextToken()
%get        lexer.token
%semantic   boolean: lexer.val
%token <boolean> TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%left '|'
%left '&'
%left '-'
%type <boolean> bexp
%%
```

Generating a parser

If we want to generate a parser, we have to connect it to a lexer that provides the tokens!

We have to use directives and program a little to do so!

We will at the same time see how to use the parser to compute while recognizing structure!

Directives:

```
%class      Evaluator
%interface  BooleanTokens
%next       nextToken()
%get        lexer.token
%semantic   boolean: lexer.val
%token <boolean> TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%left '|'
%left '&'
%left '-'
%type <boolean> bexp
%%
```

Generating a parser

If we want to generate a parser, we have to connect it to a lexer that provides the tokens!

We have to use directives and program a little to do so!

We will at the same time see how to use the parser to compute while recognizing structure!

Directives:

```
%class      Evaluator
%interface  BooleanTokens
%next       nextToken()
%get        lexer.token
%semantic   boolean: lexer.val
%token <boolean> TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%left '|'
%left '&'
%left '-'
%type <boolean> bexp
%%
```

Generating a parser

If we want to generate a parser, we have to connect it to a lexer that provides the tokens!

We have to use directives and program a little to do so!

We will at the same time see how to use the parser to compute while recognizing structure!

Directives:

```
%class      Evaluator
%interface  BooleanTokens
%next       nextToken()
%get        lexer.token
%semantic   boolean: lexer.val
%token <boolean> TRUE FALSE
%token '-' '&' '|'
%token '(' ')'
%left '|'
%left '&'
%left '-'
%type <boolean> bexp
%%
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p    : bexp               {System.out.println($1);};
bexp : bexp '|' bexp      {$$ = $1 || $3;}
     | bexp '&' bexp      {$$ = $1 && $3;}
     | '-' bexp           {$$ = ! $2;}
     | TRUE               {$$ = $1;}
     | FALSE              {$$ = $1;}
     | '(' bexp ')'       {$$ = $2;}
     ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p    : bexp                {System.out.println($1);};
bexp : bexp '|' bexp       {$$ = $1 || $3;}
     | bexp '&' bexp       {$$ = $1 && $3;}
     | '-' bexp            {$$ = ! $2;}
     | TRUE                {$$ = $1;}
     | FALSE               {$$ = $1;}
     | '(' bexp ')'        {$$ = $2;}
     ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p     : bexp                {System.out.println($1);};
bexp  : bexp '|' bexp       {$$ = $1 || $3;}
      | bexp '&' bexp       {$$ = $1 && $3;}
      | '-' bexp            {$$ = ! $2;}
      | TRUE                {$$ = $1;}
      | FALSE               {$$ = $1;}
      | '(' bexp ')'        {$$ = $2;}
      ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p     : bexp                {System.out.println($1);};
bexp  : bexp '|' bexp       {$$ = $1 || $3;}
      | bexp '&' bexp       {$$ = $1 && $3;}
      | '-' bexp            {$$ = ! $2;}
      | TRUE                {$$ = $1;}
      | FALSE               {$$ = $1;}
      | '(' bexp ')'        {$$ = $2;}
      ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p     : bexp              {System.out.println($1);};
bexp  : bexp '|' bexp     {$$ = $1 || $3;}
      | bexp '&' bexp     {$$ = $1 && $3;}
      | '-' bexp          {$$ = ! $2;}
      | TRUE              {$$ = $1;}
      | FALSE             {$$ = $1;}
      | '(' bexp ')'      {$$ = $2;}
      ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p     : bexp                 {System.out.println($1);};
bexp  : bexp '|' bexp        {$$ = $1 || $3;}
      | bexp '&' bexp        {$$ = $1 && $3;}
      | '-' bexp             {$$ = ! $2;}
      | TRUE                 {$$ = $1;}
      | FALSE                {$$ = $1;}
      | '(' bexp ')'         {$$ = $2;}
      ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p     : bexp               {System.out.println($1);};
bexp  : bexp '|' bexp      {$$ = $1 || $3;}
      | bexp '&' bexp      {$$ = $1 && $3;}
      | '-' bexp           {$$ = ! $2;}
      | TRUE               {$$ = $1;}
      | FALSE              {$$ = $1;}
      | '(' bexp ')'       {$$ = $2;}
      ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p     : bexp              {System.out.println($1);};
bexp  : bexp '|' bexp     {$$ = $1 || $3;}
      | bexp '&' bexp     {$$ = $1 && $3;}
      | '-' bexp          {$$ = ! $2;}
      | TRUE              {$$ = $1;}
      | FALSE             {$$ = $1;}
      | '(' bexp ')'      {$$ = $2;}
      ;
```

Semantic actions

We might want to use an extra non-terminal as start symbol to use a special action when the complete phrase has been recognized.

The actions refer to the values calculated for the sub-phrases.

```
%%
p    : bexp              {System.out.println($1);};
bexp : bexp '|' bexp     {$$ = $1 || $3;}
     | bexp '&' bexp     {$$ = $1 && $3;}
     | '-' bexp          {$$ = ! $2;}
     | TRUE              {$$ = $1;}
     | FALSE             {$$ = $1;}
     | '(' bexp ')'      {$$ = $2;}
     ;
```

Connecting to the lexer

```
%%
private Scanner lexer;
Evaluator(Scanner s)lexer = s;

public static void main(String[] cmdLn){
   try{
    Scanner scanner =
       new Scanner(new java.io.FileReader(cmdLn[0]));
    scanner.yylex();
    Evaluator eval = new Evaluator(scanner);
    eval. parse() ;
  }catch(Exception e){System.out.println(e.getMessage());}
}
```

Connecting to the lexer

```
%%
private Scanner lexer;
Evaluator(Scanner s)lexer = s;

public static void main(String[] cmdLn){
   try{
    Scanner scanner =
       new Scanner(new java.io.FileReader(cmdLn[0]));
    scanner.yylex();
    Evaluator eval = new Evaluator(scanner);
    eval. parse() ;
  }catch(Exception e){System.out.println(e.getMessage());}
}
```

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!

We have studied

1. Context-free grammars as a formalism to describe the syntax of computer languages.
2. Parse trees and ambiguity.
3. How to use a parser generator to *test* our grammars.

How to connect a lexer generated by JFlex to a parser generated by jacc is described in detail in the computer based exercises today afternoon.

What is comming

1. A lecture on what kind of programs parsers are.
2. An assignment on understanding the workings of a parser.
3. A lecture on abstract syntax (*one internal representation of the source code*)
4. The rest of a language processor!