Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2010/2011, periode 2

Johan Jeuring

Department of Information and Computing Sciences
Utrecht University

November 15, 2010

2. Grammars and Parsing

This lecture

Grammars and Parsing

- Recap
- Grammars
- Examples of context-free grammars
- Ambiguity
- Parsing, concrete and abstract syntax

2.1 Recap

Last lecture

Alphabet A finite set of symbols.

Language A set of words/sentences, i.e., sequences of symbols from the alphabet.

We have discussed different ways to define languages:

- by enumerating all elements,
- using a predicate,
- using an inductive definition

Example: palindromes

The language of palindromes PAL is defined as follows:

- ε is in PAL,
- $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$ are in PAL,
- if P is in PAL, then $\mathtt{a}P\mathtt{a}$, $\mathtt{b}P\mathtt{b}$ and $\mathtt{c}P\mathtt{c}$ are also in PAL.

2.2 Grammars

Grammars

A **grammar** is a formalism to describe a language inductively.

Grammars consist of rewrite rules, called **productions**.

A grammar for palindromes

$P \rightarrow \varepsilon$
$P \rightarrow \mathtt{a}$
$P \rightarrow \mathtt{b}$
$P \rightarrow \mathtt{c}$
$P \rightarrow \mathtt{a}P\mathtt{a}$
$P \rightarrow \mathtt{b}P\mathtt{b}$
$P \rightarrow \mathtt{c}P\mathtt{c}$

The language of palindromes PAL is defined as follows:

- ε is in PAL,
- a, b, c are in PAL,
- if P is in PAL, then aPa, bPb and cPc are also in PAL.

A grammar for palindromes

$$
\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow \texttt{a} \\
P &\rightarrow \texttt{b} \\
P &\rightarrow \texttt{c} \\
P &\rightarrow \texttt{a}P\texttt{a} \\
P &\rightarrow \texttt{b}P\texttt{b} \\
P &\rightarrow \texttt{c}P\texttt{c}
\end{aligned}
$$

The language of palindromes PAL is defined as follows:

- ε is in PAL,
- a, b, c are in PAL,
- if P is in PAL, then aPa, bPb and cPc are also in PAL.

Very close to the inductive definition.

Grammars

$P \to \varepsilon$
$P \to \mathtt{a}$
$P \to \mathtt{b}$
$P \to \mathtt{c}$
$\boxed{P \to \mathtt{a}P\mathtt{a}}$
$P \to \mathtt{b}P\mathtt{b}$
$P \to \mathtt{c}P\mathtt{c}$

- A grammar consists of multiple **productions**. Productions can be seen as rewrite rules. If the left hand side matches, it can be replaced by the right hand side.

Grammars

$$
\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow a \\
P &\rightarrow b \\
P &\rightarrow c \\
\boxed{P} &\rightarrow aPa \\
P &\rightarrow bPb \\
P &\rightarrow cPc
\end{aligned}
$$

- A grammar consists of multiple **productions**. Productions can be seen as rewrite rules. If the left hand side matches, it can be replaced by the right hand side.
- The grammar makes use of auxiliary symbols – called **nonterminals** – that are not part of the alphabet and hence cannot be part of the final word/sentence.

Grammars

$$\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow \mathtt{a} \\
P &\rightarrow \mathtt{b} \\
P &\rightarrow \mathtt{c} \\
P &\rightarrow \mathtt{a}P\boxed{\mathtt{a}} \\
P &\rightarrow \mathtt{b}P\mathtt{b} \\
P &\rightarrow \mathtt{c}P\mathtt{c}
\end{aligned}$$

- A grammar consists of multiple **productions**. Productions can be seen as rewrite rules. If the left hand side matches, it can be replaced by the right hand side.
- The grammar makes use of auxiliary symbols – called **nonterminals** – that are not part of the alphabet and hence cannot be part of the final word/sentence.
- The symbols from the alphabet are also called **terminals**.

Derivation

$$
\begin{aligned}
P &\to \varepsilon \\
P &\to \mathtt{a} \\
P &\to \mathtt{b} \\
P &\to \mathtt{c} \\
&\boxed{P \to \mathtt{a}P\mathtt{a}} \\
P &\to \mathtt{b}P\mathtt{b} \\
P &\to \mathtt{c}P\mathtt{c}
\end{aligned}
$$

Starting from a nonterminal, we can rewrite successively until we reach a string of terminals:

$$P$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Derivation

Starting from a nonterminal, we can rewrite successively until we reach a string of terminals:

$$
\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow \mathtt{a} \\
P &\rightarrow \mathtt{b} \\
P &\rightarrow \mathtt{c} \\
P &\rightarrow \mathtt{a}P\mathtt{a} \\
P &\rightarrow \mathtt{b}P\mathtt{b} \\
&\boxed{P \rightarrow \mathtt{c}P\mathtt{c}}
\end{aligned}
$$

$$
\begin{aligned}
& P \\
\Rightarrow\ & \mathtt{a}P\mathtt{a}
\end{aligned}
$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Derivation

$$
\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow a \\
P &\rightarrow b \\
P &\rightarrow c \\
P &\rightarrow aPa \\
P &\rightarrow bPb \\
&\boxed{P \rightarrow cPc}
\end{aligned}
$$

Starting from a nonterminal, we can rewrite successively until we reach a string of terminals:

$$
\begin{aligned}
& P \\
\Rightarrow\ & aPa \\
\Rightarrow\ & acPca
\end{aligned}
$$

Derivation

Starting from a nonterminal, we can rewrite successively until we reach a string of terminals:

$$
\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow a \\
\boxed{P \rightarrow b} \\
P &\rightarrow c \\
P &\rightarrow aPa \\
P &\rightarrow bPb \\
P &\rightarrow cPc
\end{aligned}
$$

$$
\begin{aligned}
& P \\
\Rightarrow\ & aPa \\
\Rightarrow\ & acPca \\
\Rightarrow\ & accPcca
\end{aligned}
$$

Derivation

Starting from a nonterminal, we can rewrite successively until we reach a string of terminals:

$$
\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow \mathtt{a} \\
P &\rightarrow \mathtt{b} \\
P &\rightarrow \mathtt{c} \\
P &\rightarrow \mathtt{a}P\mathtt{a} \\
P &\rightarrow \mathtt{b}P\mathtt{b} \\
P &\rightarrow \mathtt{c}P\mathtt{c}
\end{aligned}
$$

$$
\begin{aligned}
& P \\
\Rightarrow\ & \mathtt{a}P\mathtt{a} \\
\Rightarrow\ & \mathtt{ac}P\mathtt{ca} \\
\Rightarrow\ & \mathtt{acc}P\mathtt{cca} \\
\Rightarrow\ & \mathtt{accbcca}
\end{aligned}
$$

Derivation

$$\begin{aligned}
P &\rightarrow \varepsilon \\
P &\rightarrow \mathtt{a} \\
P &\rightarrow \mathtt{b} \\
P &\rightarrow \mathtt{c} \\
P &\rightarrow \mathtt{a}P\mathtt{a} \\
P &\rightarrow \mathtt{b}P\mathtt{b} \\
P &\rightarrow \mathtt{c}P\mathtt{c}
\end{aligned}$$

Starting from a nonterminal, we can rewrite successively until we reach a string of terminals:

$$\begin{aligned}
& P \\
\Rightarrow\ & \mathtt{a}P\mathtt{a} \\
\Rightarrow\ & \mathtt{ac}P\mathtt{ca} \\
\Rightarrow\ & \mathtt{acc}P\mathtt{cca} \\
\Rightarrow\ & \mathtt{accbcca}
\end{aligned}$$

We call such a sequence a **derivation**. All strings that can be derived from a nonterminal are in the **language generated by the nonterminal**.

Multiple nonterminals

Grammars can have multiple nonterminals:

$$
\begin{aligned}
S &\rightarrow A\\
S &\rightarrow B\\
A &\rightarrow \mathtt{a}\\
A &\rightarrow AA\\
B &\rightarrow \mathtt{b}\\
B &\rightarrow BB
\end{aligned}
$$

Multiple nonterminals

Grammars can have multiple nonterminals:

$$
\begin{aligned}
\mathsf{S} &\rightarrow \mathsf{A}\\
\mathsf{S} &\rightarrow \mathsf{B}\\
\mathsf{A} &\rightarrow \mathtt{a}\\
\mathsf{A} &\rightarrow \mathsf{AA}\\
\mathsf{B} &\rightarrow \mathtt{b}\\
\mathsf{B} &\rightarrow \mathsf{BB}
\end{aligned}
$$

One nonterminal in the grammar is called the **start symbol**.

Multiple nonterminals

Grammars can have multiple nonterminals:

$$
\begin{aligned}
\boxed{S} &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow \mathtt{a} \\
A &\rightarrow AA \\
B &\rightarrow \mathtt{b} \\
B &\rightarrow BB
\end{aligned}
$$

One nonterminal in the grammar is called the **start symbol**.

If not otherwise mentioned, we implicitly assume that the nonterminal on the left hand side of the first production is the start symbol (and we often, but not always, call it 'S').

Multiple nonterminals

Grammars can have multiple nonterminals:

$$
\begin{aligned}
\boxed{S} &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow a \\
A &\rightarrow AA \\
B &\rightarrow b \\
B &\rightarrow BB
\end{aligned}
$$

Question

What is the language generated by this grammar (i.e., generated by S)?

One nonterminal in the grammar is called the **start symbol**.

If not otherwise mentioned, we implicitly assume that the nonterminal on the left hand side of the first production is the start symbol (and we often, but not always, call it 'S').

Context-free grammars

The grammars we consider are restricted:

- the left hand side of a production always consists of a single nonterminal

Grammars with this restriction are called **context-free**.

Remarks about grammars

- Not all languages can be generated/described by a grammar.
- Even fewer languages can be described by a context-free grammar.
- Languages that can be described by a context-free grammar are called **context-free languages**.
- Context-free languages are relatively easy to deal with algorithmically, and therefore most programming languages are context-free languages.
- Multiple grammars may describe the same language.

Multiple grammars for one language

$$S \rightarrow aS \qquad S \rightarrow Sa \qquad S \rightarrow SS \qquad S \rightarrow AS$$
$$S \rightarrow a \qquad S \rightarrow a \qquad S \rightarrow a \qquad S \rightarrow A$$
$$A \rightarrow a$$

2.3 Examples of context-free grammars

Language of (single) digits

$$
\begin{aligned}
\mathit{Dig} &\to 0\\
\mathit{Dig} &\to 1\\
\mathit{Dig} &\to 2\\
\mathit{Dig} &\to 3\\
\mathit{Dig} &\to 4\\
\mathit{Dig} &\to 5\\
\mathit{Dig} &\to 6\\
\mathit{Dig} &\to 7\\
\mathit{Dig} &\to 8\\
\mathit{Dig} &\to 9
\end{aligned}
$$

Language of (single) digits

$$\begin{aligned}
\text{Dig} &\rightarrow 0\\
\text{Dig} &\rightarrow 1\\
\text{Dig} &\rightarrow 2\\
\text{Dig} &\rightarrow 3\\
\text{Dig} &\rightarrow 4\\
\text{Dig} &\rightarrow 5\\
\text{Dig} &\rightarrow 6\\
\text{Dig} &\rightarrow 7\\
\text{Dig} &\rightarrow 8\\
\text{Dig} &\rightarrow 9
\end{aligned}$$

Multiple productions for the same nonterminal can be joined:

$$\text{Dig} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

(We still count ten productions!)

Sequences of digits

$\text{Digs} \rightarrow \varepsilon \mid \text{Dig Digs}$

Sequences of digits

$$\text{Digs} \rightarrow \varepsilon \mid \text{Dig Digs}$$

This grammar allows sequences with leading zeros:

$$\begin{aligned} \text{Digs} &\Rightarrow \text{Dig Digs} \Rightarrow \text{Dig Dig Digs} \Rightarrow \text{Dig Dig Dig Digs} \\ &\Rightarrow \text{Dig Dig Dig}\ \varepsilon \Rightarrow^* 007 \end{aligned}$$

The symbol '$\Rightarrow^*$' means that we make multiple (zero or more, but finitely many) derivation steps at once.

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Sequences of digits

$$\text{Digs} \rightarrow \varepsilon \mid \text{Dig Digs}$$

This grammar allows sequences with leading zeros:

$$\begin{aligned}\text{Digs} &\Rightarrow \text{Dig Digs} \Rightarrow \text{Dig Dig Digs} \Rightarrow \text{Dig Dig Dig Digs}\\ &\Rightarrow \text{Dig Dig Dig}\ \varepsilon \Rightarrow^{*} 007\end{aligned}$$

The symbol '$\Rightarrow^{*}$' means that we make multiple (zero or more, but finitely many) derivation steps at once.

We also allow the star notation on the right hand side of a grammar to abbreviate zero or more occurrences of symbols:

$$\text{Digs} \rightarrow \text{Dig}^{*}$$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Natural numbers

To disallow leading zeros we introduce another nonterminal:

$\text{Dig-0} \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Natural numbers

To disallow leading zeros we introduce another nonterminal:

$$\text{Dig-0} \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\text{Nat} \rightarrow 0 \mid \text{Dig-0 Digs}$$

Integers

$$\begin{aligned} \mathsf{Sign} &\rightarrow \mathtt{+} \mid \mathtt{-} \\ \mathsf{Int} &\rightarrow \mathsf{Sign}\ \mathsf{Nat} \mid \mathsf{Nat} \end{aligned}$$

The sign is optional.

Integers

$$\begin{aligned} \text{Sign} &\rightarrow + \mid - \\ \text{Int} &\rightarrow \text{Sign Nat} \mid \text{Nat} \end{aligned}$$

The sign is optional.

There is an abbreviation for optional symbols as well:

$$\text{Int} \rightarrow \text{Sign}?\ \text{Nat}$$

Letters

Letters are much like digits.

$$\begin{aligned} \text{SLetter} &\rightarrow \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \\ \text{CLetter} &\rightarrow \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \end{aligned}$$

(52 productions in total.)

Letters

Letters are much like digits.

$$\begin{aligned}\mathsf{SLetter} &\rightarrow \mathtt{a} \mid \mathtt{b} \mid \ldots \mid \mathtt{z}\\ \mathsf{CLetter} &\rightarrow \mathtt{A} \mid \mathtt{B} \mid \ldots \mid \mathtt{Z}\end{aligned}$$

(52 productions in total.)

$$\mathsf{Letter} \rightarrow \mathsf{SLetter} \mid \mathsf{CLetter}$$

Identifiers

In many languages, identifiers must not start with a number, but can have numbers following an initial letter.

$$
\begin{aligned}
\text{Identifier} &\rightarrow \text{Letter AlphaNum}^* \\
\text{AlphaNum} &\rightarrow \text{Letter} \mid \text{Dig}
\end{aligned}
$$

Variations are easy to define (such as allowing certain symbols, for example '_', as well).

A fragment of Java

$$
\begin{array}{lcl}
\mathsf{Stat} & \rightarrow & \mathsf{Var} = \mathsf{Expr}\ ; \\
 & | & \mathtt{if}\ (\ \mathsf{Expr}\ )\ \mathsf{Stat}\ \mathtt{else}\ \mathsf{Stat} \\
 & | & \mathtt{while}\ (\ \mathsf{Expr}\ )\ \mathsf{Stat} \\
\mathsf{Expr} & \rightarrow & \mathsf{Integer} \\
 & | & \mathsf{Var} \\
 & | & \mathsf{Expr}\ \mathsf{Op}\ \mathsf{Expr} \\
\mathsf{Var} & \rightarrow & \mathsf{Identifier} \\
\mathsf{Op} & \rightarrow & \mathsf{Sign} \mid *
\end{array}
$$

2.4 Ambiguity

Multiple derivations for one sentence

Consider the grammar:

> $S \rightarrow SS$
> $S \rightarrow a$

These are three derivations of aaa:

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aSa \Rightarrow aaa \tag{1}$$

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa \tag{2}$$

$$S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow aSa \Rightarrow aaa \tag{3}$$

Multiple derivations for one sentence

Consider the grammar:

$$
\begin{aligned}
&\mathsf{S} \rightarrow \mathsf{SS}\\
&\mathsf{S} \rightarrow \mathsf{a}
\end{aligned}
$$

These are three derivations of aaa:

$$\mathsf{S} \Rightarrow \mathsf{SS} \Rightarrow \mathsf{aS} \Rightarrow \mathsf{aSS} \Rightarrow \mathsf{aSa} \Rightarrow \mathsf{aaa} \tag{1}$$

$$\mathsf{S} \Rightarrow \mathsf{SS} \Rightarrow \mathsf{aS} \Rightarrow \mathsf{aSS} \Rightarrow \mathsf{aaS} \Rightarrow \mathsf{aaa} \tag{2}$$

$$\mathsf{S} \Rightarrow \mathsf{SS} \Rightarrow \mathsf{Sa} \Rightarrow \mathsf{SSa} \Rightarrow \mathsf{aSa} \Rightarrow \mathsf{aaa} \tag{3}$$

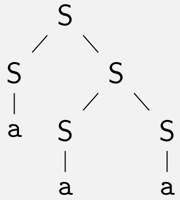
Question

Can you see that (1) and (2) are less fundamentally different than either (1) and (3) or (2) and (3)?

Parse trees

We can visualize a derivation as a **parse tree**:

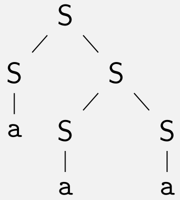
> $S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aSa \Rightarrow aaa$



Parse trees

We can visualize a derivation as a **parse tree**:

$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aSa \Rightarrow aaa$



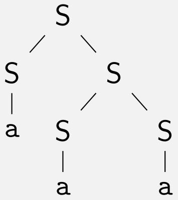
Same tree:

$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa$

Parse trees

We can visualize a derivation as a **parse tree**:

> S ⇒ SS ⇒ aS ⇒ aSS ⇒ aSa ⇒ aaa

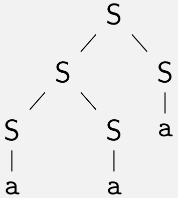


Same tree:

> S ⇒ SS ⇒ aS ⇒ aSS ⇒ aaS ⇒ aaa

Different tree:

> S ⇒ SS ⇒ Sa ⇒ SSa ⇒ aSa ⇒ aaa



Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Ambiguity

A grammar where every sentence corresponds to a unique parse tree is called **unambiguous**.

If this is not the case, the grammar is called **ambiguous**.

Ambiguity

A grammar where every sentence corresponds to a unique parse tree is called **unambiguous**.

If this is not the case, the grammar is called **ambiguous**.

The grammar

$$
\begin{aligned}
&\mathsf{S} \rightarrow \mathsf{SS}\\
&\mathsf{S} \rightarrow \mathtt{a}
\end{aligned}
$$

is thus ambiguous.

Universiteit Utrecht
[Faculty of **Science**
Information and Computing Sciences]

Ambiguity

A grammar where every sentence corresponds to a unique parse tree is called **unambiguous**.

If this is not the case, the grammar is called **ambiguous**.

The grammar

$$S \to SS$$
$$S \to \mathtt{a}$$

is thus ambiguous.

Question

Why are ambiguous grammars bad?

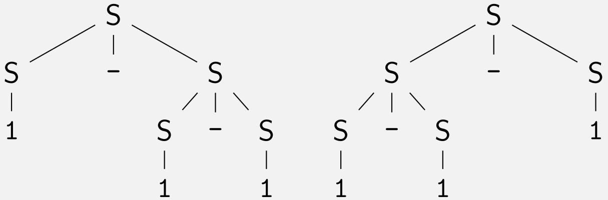
Ambiguity and semantics

Let's look ahead for a moment. Later we are going to assign **semantics** to parse trees.

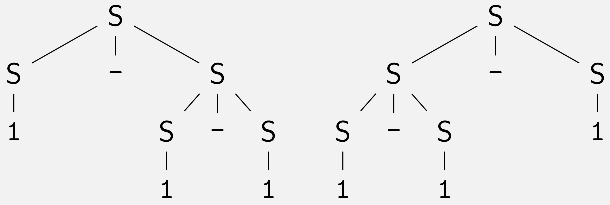
Assume the (ambiguous) grammar:

> S $\rightarrow$ S–S
> S $\rightarrow$ 1

Now the sentence 1–1–1 corresponds to two parse trees:



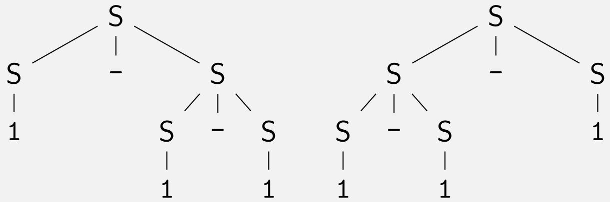
Ambiguity and semantics – contd.



Using the standard semantics,

- the left tree corresponds to the value 1,
- the right tree corresopnds to the value −1.

Ambiguity and semantics – contd.

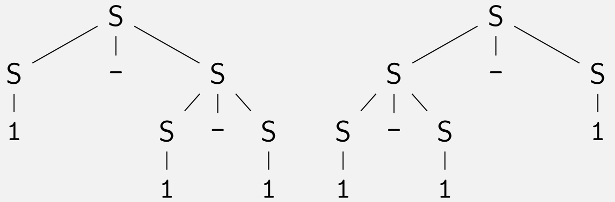


Using the standard semantics,

- the left tree corresponds to the value 1,
- the right tree corresopnds to the value −1.

Hence, **ambiguous grammars lead to ambiguous semantics**.

Ambiguity and semantics – contd.



Using the standard semantics,

- the left tree corresponds to the value 1,
- the right tree corresopnds to the value −1.

Hence, **ambiguous grammars lead to ambiguous semantics**.

Later, we will also see that ambiguous grammars can cause inefficiency.

Words of warning: syntax vs. semantics

Do not immediately associate semantics with a sentence:

> 1−1−1

A language defines which sentences are syntactically correct. Assigning meaning to these sentences is a separate step.

Words of warning: syntax vs. semantics

Do not immediately associate semantics with a sentence:

```
1-1-1
```

A language defines which sentences are syntactically correct. Assigning meaning to these sentences is a separate step.

Depending on the semantics we assign, a string such as `1-1-1` can have many different meanings:

- it could mean the value 1 or -1,
- it could mean the 1st of January in year 1,
- it could mean that the first item in a table should be copied three times,
- ...

Dangling else

A famous ambiguity problem, demonstrated using a simplified grammar:

$$
\begin{aligned}
\mathsf{S} \rightarrow\ & \texttt{if b then}\ \mathsf{S}\ \texttt{else}\ \mathsf{S} \\
\mid\ & \texttt{if b then}\ \mathsf{S} \\
\mid\ & \texttt{a}
\end{aligned}
$$

Dangling else

A famous ambiguity problem, demonstrated using a simplified grammar:

$$
\begin{aligned}
\mathsf{S} \rightarrow\; & \texttt{if b then } \mathsf{S} \texttt{ else } \mathsf{S} \\
\mid\; & \texttt{if b then } \mathsf{S} \\
\mid\; & \texttt{a}
\end{aligned}
$$

Consider:

```
if b then if b then a else a
```

Dangling else

A famous ambiguity problem, demonstrated using a simplified grammar:

$$
\begin{aligned}
\mathsf{S} \rightarrow\ & \texttt{if b then } \mathsf{S} \texttt{ else } \mathsf{S} \\
\mid\ & \texttt{if b then } \mathsf{S} \\
\mid\ & \texttt{a}
\end{aligned}
$$

Consider:

```
if b then if b then a else a
```

Exercise 2.17

Ambiguity is a property of grammars

(…and not of languages!)

All of these grammars describe the same language:

$$S \rightarrow aS \qquad S \rightarrow Sa \qquad S \rightarrow SS \qquad S \rightarrow AS$$
$$S \rightarrow a \qquad S \rightarrow a \qquad S \rightarrow a \qquad S \rightarrow A$$
$$A \rightarrow a$$

Are all of them ambiguous?

Grammar transformations

A **grammar transformation** is a mapping from one grammar to another, such that the generated language remains the same.

Grammar transformations

A **grammar transformation** is a mapping from one grammar to another, such that the generated language remains the same.

Formally:

A grammar transformation maps a grammar G to another grammar G' such that

$$L(\mathsf{G}) = L(\mathsf{G}')$$

Grammar transformations

A **grammar transformation** is a mapping from one grammar to another, such that the generated language remains the same.

Formally:

A grammar transformation maps a grammar G to another grammar G′ such that

$$L(\mathsf{G}) = L(\mathsf{G}')$$

Grammar transformations can help us to transform grammars with undesirable properties (such as ambiguity) into grammars with other (hopefully better) properties.

Grammar transformations

A **grammar transformation** is a mapping from one grammar to another, such that the generated language remains the same.

Formally:

A grammar transformation maps a grammar G to another grammar G' such that

$$L(\mathsf{G}) = L(\mathsf{G}')$$

Grammar transformations can help us to transform grammars with undesirable properties (such as ambiguity) into grammars with other (hopefully better) properties.

Most grammar transformations are motivated by facilitating parsing.

2.5 Parsing, concrete and abstract syntax

Parsing problem

Given a grammar G and a string s, the **parsing problem** is to decide whether or not s $\in L(\mathsf{G})$.

Parsing problem

Given a grammar G and a string s, the **parsing problem** is to decide whether or not s $\in L(\mathsf{G})$.

Furthermore, if s $\in L(\mathsf{G})$, we want evidence/proof/an explanation why this is the case, usually in the form of a parse tree.

Parse trees in Haskell

Consider this grammar (do you recognize the language?)

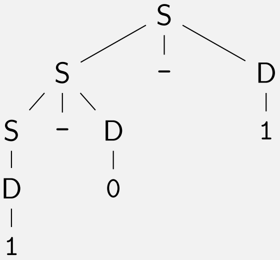
$$
\begin{aligned}
S &\rightarrow S\text{–}D \mid D \\
D &\rightarrow 0 \mid 1
\end{aligned}
$$

Parse trees in Haskell

Consider this grammar (do you recognize the language?)

$$
\begin{aligned}
S &\to S\text{-}D \mid D \\
D &\to 0 \mid 1
\end{aligned}
$$

The string 1-0-1 corresponds to the parse tree

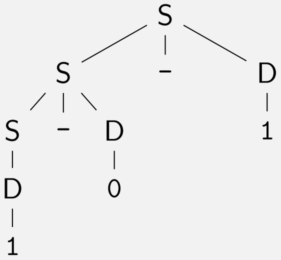


Parse trees in Haskell

Consider this grammar (do you recognize the language?)

> S → S–D | D
> D → 0 | 1

The string 1–0–1 corresponds to the parse tree



Question

How do we best represent such a tree in Haskell?

Parse trees in Haskell – contd.

Idea

Let us represent nonterminals as **datatypes**:

Parse trees in Haskell – contd.

Idea

Let us represent nonterminals as **datatypes**:

- In every node of the parse tree, we have a choice between one of the productions for the nonterminal in question.

Parse trees in Haskell – contd.

Idea

Let us represent nonterminals as **datatypes**:

- In every node of the parse tree, we have a choice between one of the productions for the nonterminal in question.
- If we want to build a value of a Haskell datatype, we have a choice between any of that datatype's **constructors**.

Parse trees in Haskell – contd.

Idea

Let us represent nonterminals as **datatypes**:

- In every node of the parse tree, we have a choice between one of the productions for the nonterminal in question.
- If we want to build a value of a Haskell datatype, we have a choice between any of that datatype's **constructors**.

Hence, productions become constructors.

Parse trees in Haskell – contd.

$S \rightarrow S{-}D \mid D$	**data** $S = \ldots \qquad \mid \ldots$
$D \rightarrow 0 \mid 1$	**data** $D = \ldots \qquad \mid \ldots$

What names to choose for the constructors?

Parse trees in Haskell – contd.

$S \to S\text{-}D \mid D$	**data** $S = \ldots \qquad \mid \ldots$
$D \to 0 \mid 1$	**data** $D = \ldots \qquad \mid \ldots$

What names to choose for the constructors? – Our choice, but let's try to pick somewhat meaningful names.

Parse trees in Haskell – contd.

$S \rightarrow S\text{-}D \mid D$ $D \rightarrow 0 \mid 1$	**data** S = Minus ... | SingleDigit ... **data** D = Zero ... | One ...

What names to choose for the constructors? – Our choice, but let's try to pick somewhat meaningful names.

And what do we do for each of the nonterminals on the right hand sides of the productions?

Parse trees in Haskell – contd.

$S \rightarrow S\text{-}D \mid D$ $D \rightarrow 0 \mid 1$	**data** S = Minus ...	SingleDigit ... **data** D = Zero ...	One ...

What names to choose for the constructors? – Our choice, but let's try to pick somewhat meaningful names.

And what do we do for each of the nonterminals on the right hand sides of the productions? – They become arguments of the constructor.

Parse trees in Haskell – contd.

$S \rightarrow S\text{-}D \mid D$ $D \rightarrow 0 \mid 1$	**data** S = Minus S ... D | SingleDigit D **data** D = Zero ... | One ...

What names to choose for the constructors? – Our choice, but let's try to pick somewhat meaningful names.

And what do we do for each of the nonterminals on the right hand sides of the productions? – They become arguments of the constructor.

And what do we do with the terminals on the right hand sides of the productions?

Parse trees in Haskell – contd.

$S \rightarrow S\text{-}D \mid D$	**data** S = Minus S ... D | SingleDigit D
$D \rightarrow 0 \mid 1$	**data** D = Zero ... | One ...

What names to choose for the constructors? – Our choice, but let's try to pick somewhat meaningful names.

And what do we do for each of the nonterminals on the right hand sides of the productions? – They become arguments of the constructor.

And what do we do with the terminals on the right hand sides of the productions? – Do we actually need them?

Parse trees in Haskell – contd.

$S \rightarrow S\text{-}D \mid D$ $D \rightarrow 0 \mid 1$	**data** S = Minus S D | SingleDigit D **data** D = Zero | One

What names to choose for the constructors? – Our choice, but let's try to pick somewhat meaningful names.

And what do we do for each of the nonterminals on the right hand sides of the productions? – They become arguments of the constructor.

And what do we do with the terminals on the right hand sides of the productions? – Do we actually need them? – No, the choice of the constructor already contains enough information to **reconstruct** the terminals.

Concrete and abstract syntax

We call the grammar the **concrete**, the datatype the **abstract** syntax of the described language.

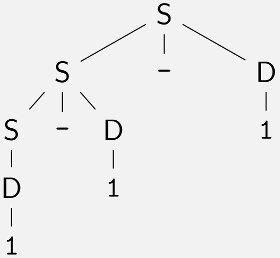
$$\begin{array}{l} S \rightarrow S\text{-}D \mid D \\ D \rightarrow 0 \mid 1 \end{array} \qquad \begin{array}{l} \textbf{data}\ S = \text{Minus}\ S\ D \mid \text{SingleDigit}\ D \\ \textbf{data}\ D = \text{Zero} \mid \text{One} \end{array}$$

Concrete and abstract syntax

We call the grammar the **concrete**, the datatype the **abstract** syntax of the described language.

$$
\begin{array}{l|l}
S \rightarrow S\text{-}D \mid D & \textbf{data}\ S = \text{Minus}\ S\ D \mid \text{SingleDigit}\ D \\
D \rightarrow 0 \mid 1 & \textbf{data}\ D = \text{Zero} \mid \text{One}
\end{array}
$$

The string 1−0−1 corresponds to the parse tree



Haskell

```
Minus (Minus (SingleDigit One)
                          Zero)
      One
```

Semantic functions

$$\begin{array}{lcl} S & \rightarrow & S\text{-}D \mid D \\ D & \rightarrow & 0 \mid 1 \end{array}$$

```
data S = Minus S D | SingleDigit D
data D = Zero | One
```

Back to the string representation:

```
printS :: S → String
printS (Minus s d)       = printS s ++ "-" ++ printD d
printS (SingleDigit d) = printD d

printD :: D → String
printD Zero              = "0"
printD One               = "1"
```

Semantic functions

$$\begin{aligned} S &\rightarrow S\text{-}D \mid D \\ D &\rightarrow 0 \mid 1 \end{aligned}$$

```
data S = Minus S D | SingleDigit D
data D = Zero | One
```

Back to the string representation:

```
printS :: S → String
printS (Minus s d)       = printS s ++ "-" ++ printD d
printS (SingleDigit d) = printD d
printD :: D → String
printD Zero              = "0"
printD One               = "1"
```

```
sample = Minus (Minus (SingleDigit One) Zero) One
printS sample   evaluates to   "1-0-1"
```

Semantic functions – contd.

$$
\begin{array}{l}
S \rightarrow S\text{–}D \mid D \\
D \rightarrow 0 \mid 1
\end{array}
\qquad
\begin{array}{l}
\textbf{data}\ S = \text{Minus}\ S\ D \mid \text{SingleDigit}\ D \\
\textbf{data}\ D = \text{Zero} \mid \text{One}
\end{array}
$$

Another semantic function – evaluation:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{evalS} :: S \rightarrow \text{Int}} \\
\text{evalS}\ (\text{Minus}\ s\ d) & = \text{evalS}\ s - \text{evalD}\ d \\
\text{evalS}\ (\text{SingleDigit}\ d) & = \text{evalD}\ d \\
\multicolumn{2}{l}{\text{evalD} :: D \rightarrow \text{Int}} \\
\text{evalD Zero} & = 0 \\
\text{evalD One} & = 1
\end{array}
$$

Semantic functions – contd.

$S \rightarrow S\text{–}D \mid D$
$D \rightarrow 0 \mid 1$

```
data S = Minus S D | SingleDigit D
data D = Zero | One
```

Another semantic function – evaluation:

```
evalS :: S → Int
evalS (Minus s d)       = evalS s − evalD d
evalS (SingleDigit d) = evalD d
evalD :: D → Int
evalD Zero              = 0
evalD One               = 1
```

```
sample = Minus (Minus (SingleDigit One) Zero) One
evalS sample    evaluates to    0
```

Summary

Grammar	A way to describe a language inductively.
Production	A rewrite rule in a grammar.
Context-free	The class of grammars/languages we consider.
Nonterminal	Auxiliary symbols in a grammar.
Terminal	Alphabet symbols in a grammar.
Derivation	Successively rewriting from a grammar until we reach a sentence.
Parse tree	Tree representation of a derivation.
Ambiguity	Multiple parse trees for the same sentence.
Abstract syntax	(Haskell) Datatype corresponding to a grammar.
Semantic function	Function defined on the abstract syntax.