



Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Talen en Compilers

2019 - 2020, period 2

Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University

2019-11-04

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



Previous 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,
- ▶ a, b, c are in PAL,
- ▶ if P is in PAL, then aPa, bPb and cPc 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

$P \rightarrow \varepsilon$

$P \rightarrow a$

$P \rightarrow b$

$P \rightarrow c$

$P \rightarrow aPa$

$P \rightarrow bPb$

$P \rightarrow cPc$

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

$$P \rightarrow \varepsilon$$

$$P \rightarrow a$$

$$P \rightarrow b$$

$$P \rightarrow c$$

$$P \rightarrow aPa$$

$$P \rightarrow bPb$$

$$P \rightarrow cPc$$

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.



$P \rightarrow \varepsilon$

$P \rightarrow a$

$P \rightarrow b$

$P \rightarrow c$

$P \rightarrow aPa$

$P \rightarrow bPb$

$P \rightarrow cPc$

- ▶ 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.



$P \rightarrow \varepsilon$

$P \rightarrow a$

$P \rightarrow b$

$P \rightarrow c$

$P \rightarrow aPa$

$P \rightarrow bPb$

$P \rightarrow cPc$

- ▶ 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.



$P \rightarrow \varepsilon$

$P \rightarrow a$

$P \rightarrow b$

$P \rightarrow c$

$P \rightarrow aPa$

$P \rightarrow bPb$

$P \rightarrow cPc$

- ▶ 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

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

$$P \rightarrow \varepsilon$$

$$P \rightarrow a$$

$$P \rightarrow b$$

$$P \rightarrow c$$

$$P \rightarrow aPa$$

$$P \rightarrow bPb$$

$$P \rightarrow cPc$$

P



Derivation

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

$$P \rightarrow \varepsilon$$

$$P \rightarrow a$$

$$P \rightarrow b$$

$$P \rightarrow c$$

$$P \rightarrow aPa$$

$$P \rightarrow bPb$$

$$P \rightarrow cPc$$

$$P \\ \Rightarrow aPa$$



Derivation

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

$$P \rightarrow \varepsilon$$

$$P \rightarrow a$$

$$P \rightarrow b$$

$$P \rightarrow c$$

$$P \rightarrow aPa$$

$$P \rightarrow bPb$$

$$P \rightarrow cPc$$

$$P$$

$$\Rightarrow aPa$$

$$\Rightarrow acPca$$



Derivation

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

$$P \rightarrow \varepsilon$$

$$P \rightarrow a$$

$$P \rightarrow b$$

$$P \rightarrow c$$

$$P \rightarrow aPa$$

$$P \rightarrow bPb$$

$$P \rightarrow cPc$$

$$P$$

$$\Rightarrow aPa$$

$$\Rightarrow acPca$$

$$\Rightarrow accPcca$$



Derivation

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

$$P \rightarrow \varepsilon$$

$$P \rightarrow a$$

$$P \rightarrow b$$

$$P \rightarrow c$$

$$P \rightarrow aPa$$

$$P \rightarrow bPb$$

$$P \rightarrow cPc$$

$$P$$

$$\Rightarrow aPa$$

$$\Rightarrow acPca$$

$$\Rightarrow accPcca$$

$$\Rightarrow accbcca$$



Derivation

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

$$P \rightarrow \varepsilon$$

$$P \rightarrow a$$

$$P \rightarrow b$$

$$P \rightarrow c$$

$$P \rightarrow aPa$$

$$P \rightarrow bPb$$

$$P \rightarrow cPc$$

P

$$\Rightarrow aPa$$

$$\Rightarrow acPca$$

$$\Rightarrow accPcca$$

$$\Rightarrow accbcca$$

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:

$$S \rightarrow A$$
$$S \rightarrow B$$
$$A \rightarrow a$$
$$A \rightarrow AA$$
$$B \rightarrow b$$
$$B \rightarrow BB$$


Multiple nonterminals

Grammars can have multiple nonterminals:

$$S \rightarrow A$$
$$S \rightarrow B$$
$$A \rightarrow a$$
$$A \rightarrow AA$$
$$B \rightarrow b$$
$$B \rightarrow BB$$

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



Multiple nonterminals

Grammars can have multiple nonterminals:

$$\begin{array}{l} S \rightarrow A \\ S \rightarrow B \\ A \rightarrow a \\ A \rightarrow AA \\ B \rightarrow b \\ B \rightarrow BB \end{array}$$

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:

$S \rightarrow A$
 $S \rightarrow B$
 $A \rightarrow a$
 $A \rightarrow AA$
 $B \rightarrow b$
 $B \rightarrow BB$

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

$$\left\{ \begin{array}{l} S \rightarrow aS \\ S \rightarrow a \end{array} \right.$$

$$\left\{ \begin{array}{l} S \rightarrow Sa \\ S \rightarrow a \end{array} \right.$$

$$\left\{ \begin{array}{l} S \rightarrow SS \\ S \rightarrow a \end{array} \right.$$

$$\left\{ \begin{array}{l} S \rightarrow AS \\ S \rightarrow A \\ A \rightarrow a \end{array} \right.$$



2.3 Examples of context-free grammars



Language of (single) digits

Dig \rightarrow 0

Dig \rightarrow 1

Dig \rightarrow 2

Dig \rightarrow 3

Dig \rightarrow 4

Dig \rightarrow 5

Dig \rightarrow 6

Dig \rightarrow 7

Dig \rightarrow 8

Dig \rightarrow 9



Language of (single) digits

Dig \rightarrow 0

Dig \rightarrow 1

Dig \rightarrow 2

Dig \rightarrow 3

Dig \rightarrow 4

Dig \rightarrow 5

Dig \rightarrow 6

Dig \rightarrow 7

Dig \rightarrow 8

Dig \rightarrow 9

Multiple productions for the same nonterminal can be joined:

Dig \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

(We still count ten productions!)



Sequences of digits

| Digs $\rightarrow \varepsilon$ | Dig Digs



Sequences of digits

| Digs $\rightarrow \epsilon$ | Dig Digs

This grammar allows sequences with leading zeros:

| Digs \Rightarrow Dig Digs \Rightarrow Dig Dig Digs \Rightarrow Dig Dig Dig Digs
 \Rightarrow Dig Dig Dig $\epsilon \Rightarrow^*$ 007

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



Sequences of digits

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

This grammar allows sequences with leading zeros:

| $\text{Digs} \Rightarrow \text{Dig Digs} \Rightarrow \text{Dig Dig Digs} \Rightarrow \text{Dig Dig Dig Digs}$
 $\Rightarrow \text{Dig Dig Dig } \varepsilon \Rightarrow^* 007$

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:

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



Natural numbers

To disallow leading zeros we introduce another nonterminal:

| Dig-0 \rightarrow 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Natural numbers

To disallow leading zeros we introduce another nonterminal:

| Dig-0 \rightarrow 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

| Nat \rightarrow 0 | Dig-0 Digs



Integers

Sign $\rightarrow + \mid -$
Int \rightarrow Sign Nat \mid Nat

The sign is optional.



Integers

Sign $\rightarrow + \mid -$

Int \rightarrow Sign Nat \mid Nat

The sign is optional.

There is an abbreviation for optional symbols as well:

Int \rightarrow Sign? Nat



Letters

Letters are much like digits.

| SLetter \rightarrow a | b | ... | z
| CLetter \rightarrow A | B | ... | Z

(52 productions in total.)



Letters

Letters are much like digits.

| SLetter \rightarrow a | b | ... | z
| CLetter \rightarrow A | B | ... | Z

(52 productions in total.)

| Letter \rightarrow SLetter | CLetter



Identifiers

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

Identifier \rightarrow Letter AlphaNum*

AlphaNum \rightarrow Letter | Dig

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



A fragment of C#

Stat \rightarrow Var = Expr ;
| if (Expr) Stat else Stat
| while (Expr) Stat

Expr \rightarrow Integer
| Var
| Expr Op Expr

Var \rightarrow Identifier

Op \rightarrow Sign | *



2.4 Ambiguity



Multiple derivations for one sentence

Consider the grammar:

$$\begin{array}{l} S \rightarrow SS \\ S \rightarrow a \end{array}$$

These are three derivations of aaa:

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aSa \Rightarrow aaa \quad (1)$$

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa \quad (2)$$

$$S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow aSa \Rightarrow aaa \quad (3)$$



Multiple derivations for one sentence

Consider the grammar:

$$\begin{array}{l} S \rightarrow SS \\ S \rightarrow a \end{array}$$

These are three derivations of aaa:

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aSa \Rightarrow aaa \quad (1)$$

$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa \quad (2)$$

$$S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow aSa \Rightarrow aaa \quad (3)$$

Question

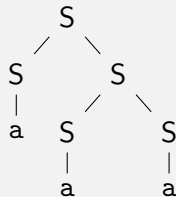
Can you see that (1) and (2) are less fundamentally different than either (1) and (3) or (2) and (3)?



Parse trees

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

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



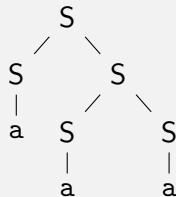
Parse trees

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**:

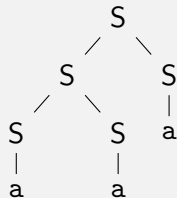
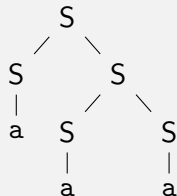
| $S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aSa \Rightarrow aaa$

Same tree:

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

Different tree:

| $S \Rightarrow SS \Rightarrow Sa \Rightarrow SSa \Rightarrow aSa \Rightarrow aaa$



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

$$\begin{array}{l} S \rightarrow SS \\ S \rightarrow a \end{array}$$

is thus 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

$$\begin{array}{l} S \rightarrow SS \\ S \rightarrow a \end{array}$$

is thus ambiguous.

Question

Why are ambiguous grammars bad?



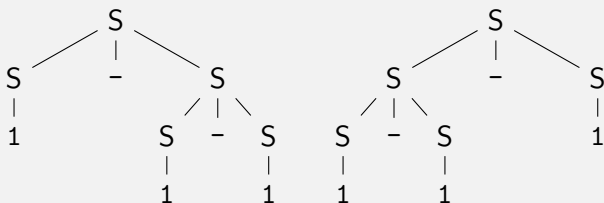
Ambiguity and semantics

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

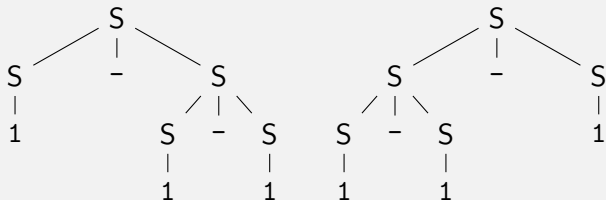
Assume the (ambiguous) grammar:

$S \rightarrow S-S$
 $S \rightarrow 1$

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



Ambiguity and semantics – contd.

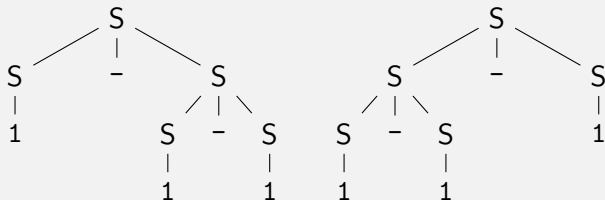


Using the standard semantics,

- ▶ the left tree corresponds to the value 1,
- ▶ the right tree corresponds to the value -1.



Ambiguity and semantics – contd.



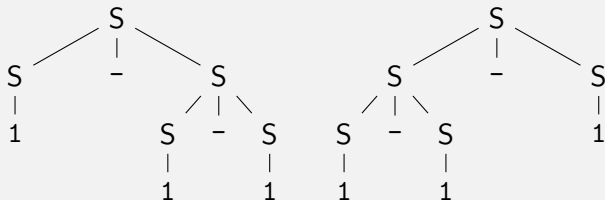
Using the standard semantics,

- ▶ the left tree corresponds to the value 1,
- ▶ the right tree corresponds 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 corresponds 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:

```
S → if b then S else S
   | if b then S
   | a
```



Dangling else

A famous ambiguity problem, demonstrated using a simplified grammar:

```
S → if b then S else S
   | if b then S
   | a
```

Consider:

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



Dangling else

A famous ambiguity problem, demonstrated using a simplified grammar:

```
S → if b then S else S
   | if b then S
   | a
```

Consider:

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

Exercise 2.17



Ambiguity is a property of grammars

All of these grammars describe the same language:

$$\left| \begin{array}{l} S \rightarrow aS \\ S \rightarrow a \end{array} \right.$$

$$\left| \begin{array}{l} S \rightarrow Sa \\ S \rightarrow a \end{array} \right.$$

$$\left| \begin{array}{l} S \rightarrow SS \\ S \rightarrow a \end{array} \right.$$

$$\left| \begin{array}{l} S \rightarrow AS \\ S \rightarrow A \\ A \rightarrow a \end{array} \right.$$

Are all of them ambiguous?



Ambiguity is a property of grammars

All of these grammars describe the same language:

$$\left| \begin{array}{l} S \rightarrow aS \\ S \rightarrow a \end{array} \right| \quad \left| \begin{array}{l} S \rightarrow Sa \\ S \rightarrow a \end{array} \right| \quad \left| \begin{array}{l} S \rightarrow SS \\ S \rightarrow a \end{array} \right| \quad \left| \begin{array}{l} S \rightarrow AS \\ S \rightarrow A \\ A \rightarrow a \end{array} \right|$$

Are all of them ambiguous?

Note: some CF languages have only ambiguous grammars.



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

$$L(G) = L(G')$$



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

$$L(G) = L(G')$$

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

$$L(G) = L(G')$$

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 $s \in L(G)$.



Parsing problem

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

Furthermore, if $s \in L(G)$, 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 (what is the language?)

$$\begin{array}{l} S \rightarrow S-D \mid D \\ D \rightarrow 0 \mid 1 \end{array}$$

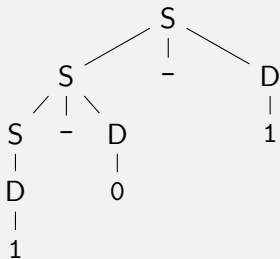


Parse trees in Haskell

Consider this grammar (what is the language?)

$$\begin{array}{l} S \rightarrow S-D \mid D \\ D \rightarrow 0 \mid 1 \end{array}$$

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

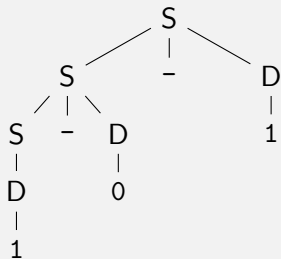


Parse trees in Haskell

Consider this grammar (what is the language?)

$$\begin{array}{l} S \rightarrow S-D \mid D \\ D \rightarrow 0 \mid 1 \end{array}$$

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 =
$D \rightarrow 0 \mid 1$	data D =

What names to choose for the constructors?



Parse trees in Haskell – contd.

$S \rightarrow S-D \mid D$	<code>data S = ...</code>	$\mid \dots$
$D \rightarrow 0 \mid 1$	<code>data D = ...</code>	$\mid \dots$

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



Parse trees in Haskell – contd.

$S \rightarrow S-D \mid D$	data S = Minus ...	SingleDigit ...
$D \rightarrow 0 \mid 1$	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?



Parse trees in Haskell – contd.

$S \rightarrow S-D \mid D$	data S = Minus ...	SingleDigit ...
$D \rightarrow 0 \mid 1$	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.



Parse trees in Haskell – contd.

$S \rightarrow S-D \mid D$	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?



Parse trees in Haskell – contd.

$S \rightarrow S-D \mid D$	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.

$S \rightarrow S-D \mid D$	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? – No, the choice of the constructor already contains enough information to **reconstruct** the terminals.



Concrete and abstract syntax

The grammar and the datatype describe the language.

concrete: **abstract** syntax:

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



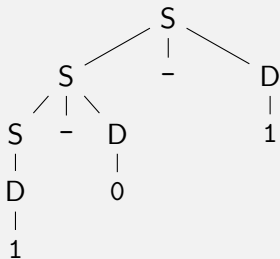
Concrete and abstract syntax

The grammar and the datatype describe the language.

concrete: **abstract** syntax:

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

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



Haskell

Minus (Minus (SingleDigit One)	Zero)
One	



Semantic functions

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

Back to the string representation:

$\text{printS} :: S \rightarrow \text{String}$
 $\text{printS } (\text{Minus } s \ d) = \text{printS } s \ ++ \ \text{"-"} \ ++ \ \text{printD } d$
 $\text{printS } (\text{SingleDigit } d) = \text{printD } d$
 $\text{printD} :: D \rightarrow \text{String}$
 $\text{printD } \text{Zero} = \text{"0"}$
 $\text{printD } \text{One} = \text{"1"}$



Semantic functions

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

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.

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

Another semantic function – evaluation:

$\text{evalS} :: S \rightarrow \text{Int}$	
$\text{evalS } (\text{Minus } s \ d)$	$= \text{evalS } s - \text{evalD } d$
$\text{evalS } (\text{SingleDigit } d)$	$= \text{evalD } d$
$\text{evalD} :: D \rightarrow \text{Int}$	
$\text{evalD } \text{Zero}$	$= 0$
$\text{evalD } \text{One}$	$= 1$



Semantic functions – contd.

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

Another semantic function – evaluation:

$\text{evalS} :: S \rightarrow \text{Int}$
 $\text{evalS } (\text{Minus } s \ d) = \text{evalS } s - \text{evalD } d$
 $\text{evalS } (\text{SingleDigit } d) = \text{evalD } d$
 $\text{evalD} :: D \rightarrow \text{Int}$
 $\text{evalD } \text{Zero} = 0$
 $\text{evalD } \text{One} = 1$

$\text{sample} = \text{Minus } (\text{Minus } (\text{SingleDigit } \text{One}) \ \text{Zero}) \ \text{One}$
 $\text{evalS } \text{sample}$ 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.

