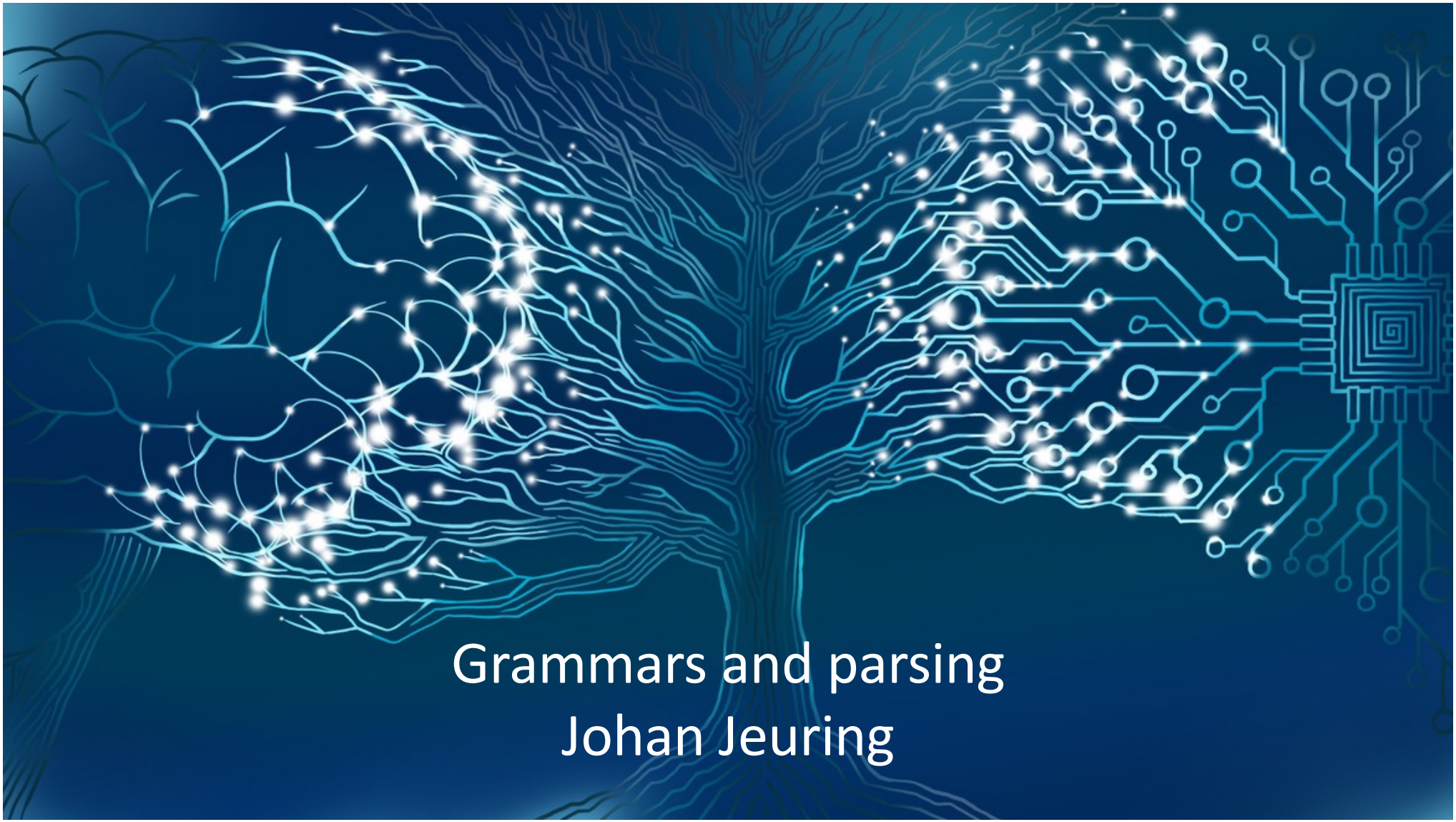

Grammars and parsing
Johan Jeuring

Last lecture

Alphabet: a finite set of symbols

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

Ways to define languages:

- Enumerate all elements:
  CSCourses = {INFOB3TC, INFOFP, INFOIMP, …}
- Use a predicate:
  MobileNrs = { nr ∈ character*
  | each character in nr is a number
  , nr starts with 06
  }
- Use an inductive definition

Example inductive definition

Inductive definition of DNA palindromes:

- ε is a DNA-palindrome
- A, C, T, G are DNA-palindromes
- If P is a DNA-palindrome, then so are APA, CPC, TPT, GPG

This lecture: **Grammars**, a formalism for inductive definitions

Addresses the first learning goal of the course:

> Describe language structures using grammars

and a bit of the third learning goal:

> Analyse grammars to see whether or not specific properties hold

Wat helpt om naar het werk/hoorcollege te komen?

Bonus

College's die:
- moeilijke onderwerpen goed uitleggen
- voorbeelden geven
- interactie mbv wooclap (oid) organiseren
- interessante onderwerpen
- niet om 9 uur beginnen

Werkcollege's
- interactief
- uitwerking van oefeningen
- nuttige TAs

Sociaal
- met medestudenten omgaan
- (in verschillende varianten)

Wooclap: eventcode HHMJRGT

A grammar for DNA palindromes

- ε is a DNA-palindrome
- A, C, T, G are DNA-palindromes
- If P is a DNA-palindrome, then so are APA, CPC, TPT, GPG

This definition is translated into rewrite rules or **productions**:

$P \rightarrow \varepsilon$

$P \rightarrow A$
$P \rightarrow C$
$P \rightarrow T$
$P \rightarrow G$

The alphabet consists of A, C, T, G, the **terminal symbols**

$P \rightarrow APA$
$P \rightarrow CPC$
$P \rightarrow TPT$
$P \rightarrow GPG$

The symbol P is not in the alphabet, a **nonterminal symbol**

Derivation

$P \rightarrow \varepsilon$
$P \rightarrow A$
$P \rightarrow C$
$P \rightarrow T$
$P \rightarrow G$
$P \rightarrow APA$
$P \rightarrow CPC$
$P \rightarrow TPT$
$P \rightarrow GPG$

P
$\Rightarrow CPC$
$\Rightarrow CAPAC$
$\Rightarrow CATAC$

This definition is called a **derivation**

All strings derived from P form the **language generated by** P

Multiple nonterminals

Z → NNNN LL
N → 0
N → 1
…
L → A
L → B
…

This grammar has three nonterminals

N generates a single digit

L generates a (capital) letter

Z is the **start-symbol**

The language of a grammar is the language of its start-symbol

Context-free grammars

$Z \rightarrow NNNN\ LL$
$N \rightarrow 0$
$N \rightarrow 1$
...
$L \rightarrow A$
$L \rightarrow B$
...

The left-hand side of a production always consists of a single nonterminal, and has no **context**

This is a **context-free grammar**

Examples

Language of (single) digits

Dig → 0
Dig → 1
Dig → 2
Dig → 3
Dig → 4
Dig → 5
Dig → 6
Dig → 7
Dig → 8
Dig → 9

Combining multiple productions for the same nonterminal:

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

(Still 10 productions!)

Sequences of digits

Digs → ε | Dig Digs

This grammar allows sequences with leading zeroes

Digs
⇒ Dig Digs
⇒ Dig Dig Digs
⇒ Dig Dig Dig Digs
⇒ Dig Dig Dig ε
⇒* 007

The symbol ⇒* means that we make zero or more (finitely many) derivation steps at once

We can also use * in the right-hand side of a production for zero or more occurrences of an element

Digs → Dig*

Natural numbers

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

Nat → 0 | Dig-0 Digs

Integers

Sign → + | -

Int → Sign Nat | Nat

The Sign is optional

We have an abbreviation for optional elements:

Int → Sign? Nat

Letters

Letters are like digits

SLetter → a| b | c | d |... | z

CLetter → A| B | C | D |... | Z

(52 productions in total)

Letter → Sletter | CLetter

Identifiers

Many languages do not allow an identifier to start with a number, but a number may appear later in an identifier

Identifier → Letter AlphaNum*

AlphaNum → Letter | Dig

Additional symbols (like '_') are easily added

A fragment of C

Stat → Var = Expr ;
| if (Expr) Stat else Stat
| while (Expr) Stat

Expr → Int
| Var
| Expr Op Expr

Var → Identifier

Op → + | - | *

Wooclap: eventcode HHMJRGT

About context-free languages

Languages described by a context-free grammar are called **context-free languages**

Context-free languages are relatively easy to deal with from a computational perspective

Most programming languages are (almost) context-free

Many languages cannot be described by a context-free grammar

Multiple grammars for the same language

$Digs^1 \rightarrow \varepsilon \mid Dig\ Digs^1$

$Digs^2 \rightarrow \varepsilon \mid Digs^2\ Dig$

$Digs^3 \rightarrow \varepsilon \mid Dig \mid Digs^3\ Digs^3$

Ambiguity and Parsing

Multiple derivations for the same number

Digs → ε | Dig | Digs Digs

I want to parse 101 (nr of students registered for the course on Oct 9, 2026)

Digs ⇒ Digs Digs ⇒ Dig Digs ⇒ 1Digs ⇒ 1Digs Digs ⇒ 1Dig Digs ⇒ 10Digs ⇒ 10Dig ⇒ 101
Digs ⇒ Digs Digs ⇒ Dig Digs ⇒ 1Digs ⇒ 1Digs Digs ⇒ 1Digs Dig ⇒ 1Digs1 ⇒ 1Dig1 ⇒ 101
Digs ⇒ Digs Digs ⇒ Digs Dig ⇒ Digs1 ⇒ Digs Digs1 ⇒ Digs Dig1 ⇒ Digs01 ⇒ Dig01 ⇒ 101

How do these derivations differ?

Look at the **parse trees**
(On the whiteboard)

Ambiguity

If there exists a unique derivation using a grammar for every sentence, the grammar is **unambiguous**

If not, the grammar is **ambiguous**

Thus the following is ambiguous

Digs → ε | Dig | Digs Digs

Why is the distinction important?

Syntax versus semantics

Wooclap

A language specifies which sentences are syntactically correct (which sentences belong to the language)

What these sentences mean (semantics) is a different step

Expressions

Expr → Int
| Var
| Expr Op Expr

Op → + | - | *

Ambiguous grammars lead to ambiguous semantics (meaning)

(And cause inefficiency, but that is for later)

Dangling else

Stat → if Expr then Stat else Stat
| if Expr then Stat
| ...

How do you parse if ... then if ... then ... else ...?

Exercise 2.17

Ambiguity is a property of grammars

Wooclap

There are context-free languages for which all grammars are ambiguous

Grammar transformations

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

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 some form of parsing

Parsing, concrete vs abstract syntax

Parsing and recognizing: definition

Given a grammar G and an input string s:

The **recognising problem** is to decide whether or not s in an element of the language of G

The **parsing problem** is to decide whether or not s is an element of the language of G, and to return a parse tree for s if it is

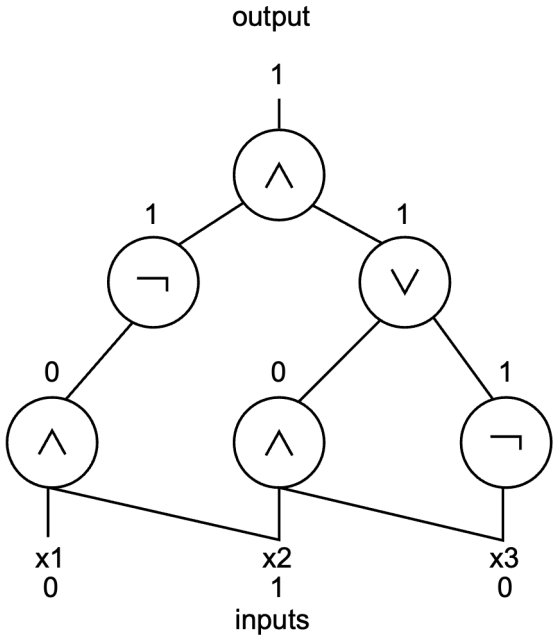
Representing parse trees: example

Suppose we want to parse strings representing boolean expressions

$\neg(x_1 \wedge x_2) \wedge (x_2 \wedge x_3) \vee \neg x_3$

BExpr → BExpr ∧ BExpr
| BExpr ∨ BExpr
| ¬BExpr
| Var

Is this grammar ambiguous?



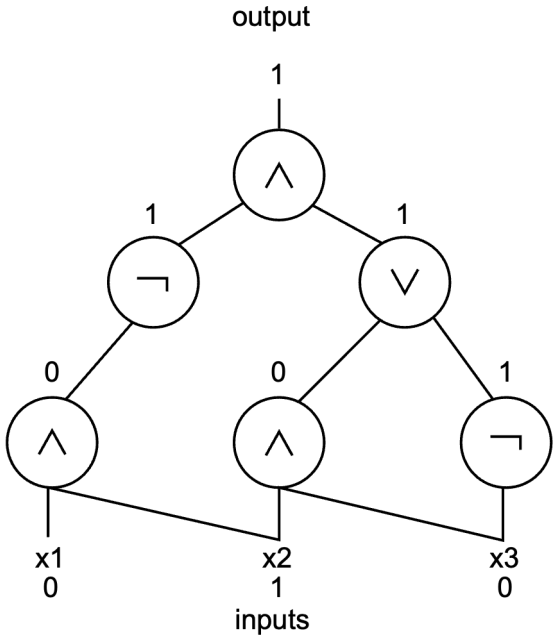
Representing parse trees in Haskell

BExpr → BExpr ∧ BExpr
| BExpr ∨ BExpr
| ¬BExpr
| Var

Idea: represent a nonterminal as a datatype

In every node of the parse tree, we have a choice for one of the productions for the nonterminal

If a nonterminal is represented by a Haskell datatype, a choice between different productions becomes a choice between different constructors



Representing parse trees in Haskell

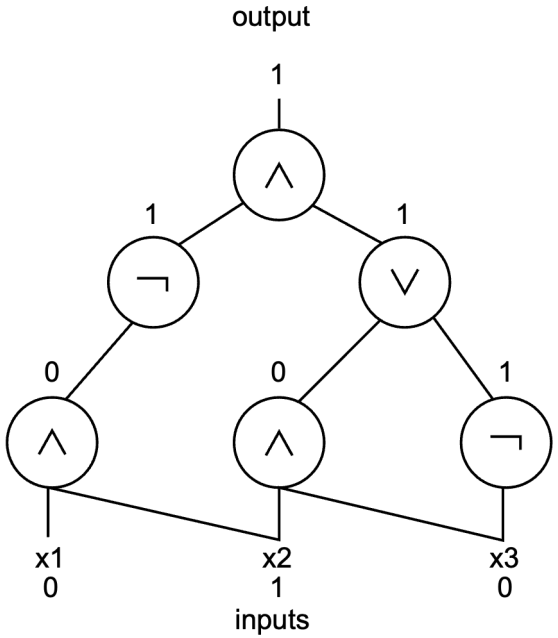
BExpr → BExpr ∧ BExpr
| BExpr ∨ BExpr
| ¬BExpr
| Var

```
data BExpr = And BExpr BExpr
           | Or BExpr BExpr
           | Not BExpr
           | Id Identifier
```

$\neg(x_1 \wedge x_2) \wedge (x_2 \wedge x_3) \vee \neg x_3$

is represented by

And (Not (And x_1 x_2)) (Or (And x_2 x_3) (Not x_3))



Concrete and abstract syntax

```
BExpr → BExpr ∧ BExpr
      |  BExpr ∨ BExpr
      |  ¬BExpr
      |  Var
```

concrete

```
data BExpr = And BExpr BExpr
           | Or BExpr BExpr
           | Not BExpr
           | Id Identifier
```

abstract

Semantic functions I

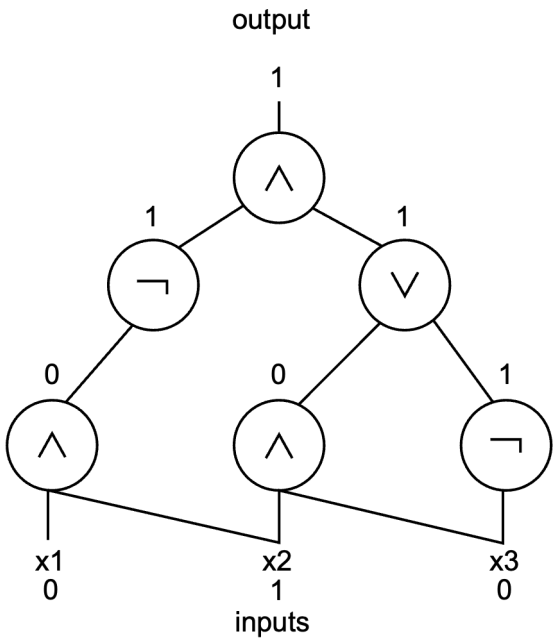
A semantic function takes an abstract syntax value, and produces some semantics

```
data BExpr = And BExpr BExpr
           | Or BExpr BExpr
           | Not BExpr BExpr
           | Id Identifier

circuit_result  ::  BExpr → Bool
circuit_result (And x y) = circuit_result x && circuit_result y
circuit_result (Or x y)    = circuit_result x ||  circuit_result y
circuit_result (Not x)     = not (circuit_result x)
circuit_result (Id v)      =  lookup v
```

where lookup looks up the boolean value

lookup x_1 = False
lookup x_2 = True
lookup x_3 = False



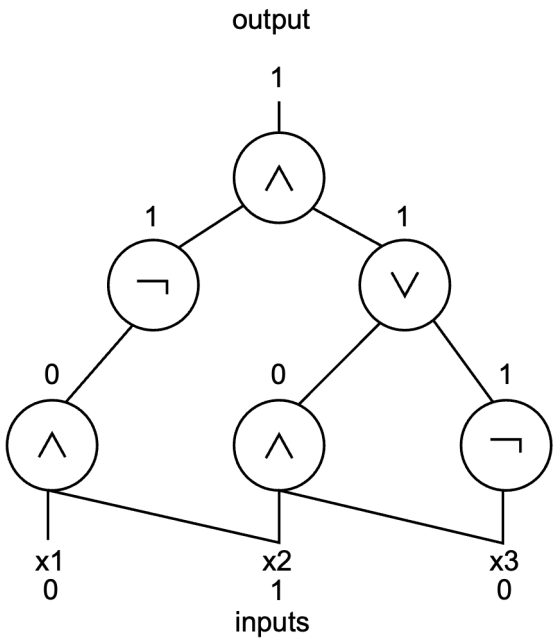
The value of the circuit is 1 (=True)

Semantic functions II

A semantic function takes an abstract syntax value, and produces some semantics

```
data BExpr = And BExpr BExpr
           | Or BExpr BExpr
           | Not BExpr BExpr
           | Id Identifier

print  ::  BExpr → String
print (And x y) = print x ++ "∧" ++ print y
print (Or x y)  = print x ++ "∨" ++ print y
print (Not x)   = "¬" ++ print x
print (Id v)    =  v
```

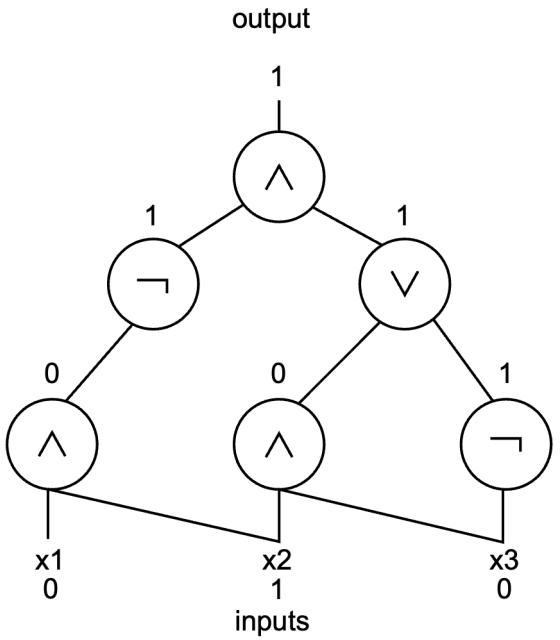


The print of the circuit is:
"$\neg x_1 \wedge x_2 \wedge x_2 \wedge x_3 \vee \neg x_3$"

Semantic functions III

A better printing function:

print :: BExpr → String
print (And x y) = “(“ ++ print x ++ “∧” ++ print y ++ “)”
print (Or x y) = “(“ ++ print x ++ “∨” ++ print y ++ “)”
print (Not x) = “¬” ++ print x
print (Id v) = v



The print of the circuit is:
“$(\neg(x_1 \wedge x_2) \wedge ((x_2 \wedge x_3) \vee \neg x_3))$”

Beware: *object* & *meta* language

We use language to talk about other languages

We may use English words in a formal language; they are different from the English words we use to talk about them

We may use Haskell to implement a language like Haskell, or C# to implement a language like C#

Be aware of the different levels

Terminology

Formal languages	
Grammar	A way to describe a language inductively
Production	A rewrite rule in a grammar
Context-free	The class of grammars we consider
Nonterminal	Auxiliary symbol in a grammar
Terminal	Alphabet symbols in a grammar
Derivation	Rewriting using 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 taking abstract syntax as input

Next week

Functions for parsing: parser combinators

Start of P1, in which you will write a parser