Syntax and Grammars
Outline

What is a language?

Abstract syntax and grammars

Abstract syntax vs. concrete syntax

Encoding grammars as Haskell data types
**Language**: a system of communication using “words” in a structured way

<table>
<thead>
<tr>
<th>Natural language</th>
<th>Programming language</th>
</tr>
</thead>
<tbody>
<tr>
<td>• used for arbitrary communication</td>
<td>• used to describe aspects of computation</td>
</tr>
<tr>
<td>• complex, nuanced, and imprecise</td>
<td>i.e. systematic transformation of representation</td>
</tr>
<tr>
<td></td>
<td>• programs have a precise <strong>structure</strong> and <strong>meaning</strong></td>
</tr>
</tbody>
</table>

English, Chinese, Hindi, Arabic, Spanish, …

Haskell, Java, C, Python, SQL, XML, HTML, CSS, …

We use a broad interpretation of “programming language”
Important to distinguish two kinds of languages:

- **Object language**: the language we’re defining
- **Metalanguage**: the language we’re using to define the structure and meaning of the object language!

A single language can fill both roles at different times! (e.g. Haskell)
Syntax vs. semantics

Two main aspects of a language:

- **syntax**: the structure of its programs
- **semantics**: the meaning of its programs

Metalanguages for defining syntax: grammars, Haskell, ...

Metalanguages for defining semantics: mathematics, inference rules, Haskell, ...
What is a language?

Abstract syntax and grammars

Abstract syntax vs. concrete syntax

Encoding grammars as Haskell data types
Programs are trees!

**Abstract syntax tree (AST):** captures the essential structure of a program
- everything needed to determine its semantics

```
+  \
|   +  \
|    |   +  \
|     |    |   +  \
|      |     |    |   +  \
|       |      |     |    |   +  \
|        |       |      |     |    |   +  \
|         |        |       |      |     |    |   + 5
```

```
2 + 3 * 4  
(5 + 6) * (7 + 8)  
if true then (2+3) else 5
```
Grammars are a **metalanguage** for describing syntax.

The language we’re defining is called the **object language**

syntactic category  →  nonterminal symbol

\[ s \in \text{Sentence} ::= n \ v \ n \quad \text{and} \quad s \quad \text{and} \quad s \]

\[ n \in \text{Noun} ::= \text{cats} \quad | \quad \text{dogs} \quad | \quad \text{ducks} \]

\[ v \in \text{Verb} ::= \text{chase} \quad | \quad \text{cuddle} \]

**production rules**
Generating programs from grammars

How to generate a program from a grammar

1. start with a nonterminal $s$
2. find production rules with $s$ on the LHS
3. replace $s$ by one possible case on the RHS

A program is in the language if and only if it can be generated by the grammar!

Animal behavior language

$s \in Sentence ::= n \ v \ n \ | \ s \ and \ s$
$n \in Noun ::= \text{cats} \ | \ \text{dogs} \ | \ \text{ducks}$
$v \in Verb ::= \text{chase} \ | \ \text{cuddle}$

$s$
$\Rightarrow n \ v \ n$
$\Rightarrow \text{cats} \ v \ n$
$\Rightarrow \text{cats} \ v \ \text{ducks}$
$\Rightarrow \text{cats cuddle ducks}$
Exercise

Animal behavior language

\[ s \in Sentence ::= n \ v \ n \mid s \ \text{and} \ s \]
\[ n \in Noun ::= \text{cats} \mid \text{dogs} \mid \text{ducks} \]
\[ v \in Verb ::= \text{chase} \mid \text{cuddle} \]

Is each “program” in the animal behavior language?

- cats chase dogs
- cats and dogs chase ducks
- dogs cuddle cats and ducks chase dogs
- dogs chase cats and cats chase ducks and ducks chase dogs
Abstract syntax trees

Grammar (BNF notation)

\[ t \in \text{Term} ::= \text{true} \mid \text{false} \mid \text{not } t \mid \text{if } t \ t \ t \]

Example ASTs

\[
\begin{align*}
\text{true} & \quad \text{if} \\
\text{true} & \quad \text{false} \quad \text{true} \\
\text{not} & \quad \text{not} \\
\text{false} & \quad \text{false}
\end{align*}
\]

Language generated by grammar: set of all ASTs

\[
\text{Term} = \{\text{true, false}\} \cup \{ t \mid t \in \text{Term} \} \cup \{ \text{if } t_1 \ t_2 \ t_3 \mid t_1, t_2, t_3 \in \text{Term} \}
\]
Exercise

Arithmetic expression language

\[ i \in \text{Int} \quad ::= \quad 1 \mid 2 \mid \ldots \]
\[ e \in \text{Expr} \quad ::= \quad \text{add} \ e \ e \]
\[ \quad | \quad \text{mul} \ e \ e \]
\[ \quad | \quad \text{neg} \ e \]
\[ \quad | \quad i \]

1. Draw two different ASTs for the expression: \(2+3+4\)

2. Draw an AST for the expression: \(-5*(6+7)\)

3. What are the integer results of evaluating the following ASTs:

\[
\text{neg}
\quad \text{add}
\quad \text{add}
\quad \text{neg}
\quad 5
\]

Abstract syntax and grammars 12 / 21
Outline

What is a language?

Abstract syntax and grammars

Abstract syntax vs. concrete syntax

Encoding grammars as Haskell data types
Abstract syntax vs. concrete syntax

**Abstract syntax** captures the **essential structure** of programs
- typically **tree-structured**
- what we use when defining the semantics

**Concrete syntax** describes how programs are **written** down
- typically **linear** (e.g. as text in a file)
- what we use when we’re writing programs in the language
**Parsing**: transforms concrete syntax into abstract syntax

Typically several steps:

- **lexical analysis**: chunk character stream into *tokens*
- **generate parse tree**: parse token stream into intermediate “concrete syntax tree”
- **convert to AST**: convert parse tree into AST

_Not a focus of this class!_
Pretty printing: transforms abstract syntax into concrete syntax

Inverse of parsing!
Abstract grammar vs. concrete grammar

Abstract grammar
\[
t ∈ \text{Term} ::= \text{true} \\
| \text{false} \\
| \text{not } t \\
| \text{if } t t t
\]

Concrete grammar
\[
t ∈ \text{Term} ::= \text{true} \\
| \text{false} \\
| \text{not } t \\
| \text{if } t \text{ then } t \text{ else } t \\
| ( t )
\]

Our focus is on abstract syntax
- we’re always writing trees, even if it looks like text
- use parentheses to disambiguate textual representation of ASTs but they are not part of the syntax
Outline

What is a language?

Abstract syntax and grammars

Abstract syntax vs. concrete syntax

Encoding grammars as Haskell data types
Encoding abstract syntax in Haskell

**Abstract grammar**

\[
\begin{align*}
  b & \in \text{Bool} \quad ::= \quad \text{true} \mid \text{false} \\
  t & \in \text{Term} \quad ::= \quad \text{not} \ t \\
   & \quad \mid \quad \text{if} \ t \ t \ t \\
   & \quad \mid \quad b
\end{align*}
\]

**Abstract syntax trees**

\[
\begin{tikzpicture}
  \node {true} [sibling distance=2.5cm]
  child {node {if}
    child {node {true}
      child {node {not}}
      child {node {false}}
    }
    child {node {if}
      child {node {true}}
      child {node {false}}
    }
  }
  child {node {false}};
\end{tikzpicture}
\]

**Haskell data type definition**

```haskell
data Term = Not Term
           | If Term Term Term
           | Lit Bool
```

**Haskell values**

- Lit True
- If (Lit True) (Lit False) (Lit True)
- Not (Not (Lit False))
Translating grammars into Haskell data types

Strategy: grammar $\rightarrow$ Haskell

1. For each **basic nonterminal**, choose a **built-in type**, e.g. `Int`, `Bool`;
2. For each other **nonterminal**, define a **data type**;
3. For each **production**, define a **data constructor**;
4. The **nonterminals in the production** determine the **arguments to the constructor**;

Special rule for lists:

- in grammars, $s ::= t^*$ is shorthand for: $s ::= \epsilon \mid ts$ or $s ::= \epsilon \mid t, s$;
- can translate any of these to a Haskell list:

```
data Term = ...
type Sentence = [Term]
```
Example: Annotated arithmetic expression language

Abstract syntax

\[
\begin{align*}
    n & \in \text{Nat} \ ::= \text{(natural number)} \\
    c & \in \text{Comm} \ ::= \text{(comment string)} \\
    e & \in \text{Expr} \ ::= \text{neg } e \quad \text{negation} \\
        & | \text{e } @ \text{c} \quad \text{comment} \\
        & | \text{e } + \text{e} \quad \text{addition} \\
        & | \text{e } * \text{e} \quad \text{multiplication} \\
        & | \text{n} \quad \text{literal}
\end{align*}
\]

Haskell encoding

\[
\begin{align*}
\text{type Comment} & = \text{String} \\
\text{data Expr} & = \text{Neg} \text{Expr} \\
        & | \text{Annot} \text{Expr} \text{Comment} \\
        & | \text{Add} \text{Expr} \text{Expr} \\
        & | \text{Mul} \text{Expr} \text{Expr} \\
        & | \text{Lit} \text{Int}
\end{align*}
\]