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>
</table>
| • used for arbitrary communication | • used to describe aspects of computation  
  i.e. systematic transformation of representation  
  • programs have a precise **structure** and **meaning** |
| • complex, nuanced, and imprecise |                           |

English, Chinese, Hindi, Arabic, Spanish, …

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

We use a broad interpretation of “programming language”
Object vs. metalanguage

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, …
Outline

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
Grammars are a **metalanguage** for describing syntax

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

syntactic category  →  nonterminal symbol

\[
s \in \text{Sentence} \quad ::= \quad n \circlearrowleft v \circlearrowright n \mid s \ 	ext{and} \ s
\]

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

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

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

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

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

Animal behavior language

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

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

Y: Yes \quad N: No \quad ?: \_\_\_(ツ)_/\_\_
Abstract syntax trees

<table>
<thead>
<tr>
<th>Grammar (BNF notation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \in \text{Term} ::= \text{true} \mid \text{false} \mid \text{not } t \mid \text{if } t t t$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example ASTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>true</td>
</tr>
<tr>
<td>not</td>
</tr>
</tbody>
</table>

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 ∈ Int** ::=
  - 1
  - 2
  - ...

- **e ∈ Expr** ::=
  - add e e
  - mul e e
  - neg e
  - i

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

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

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

   ![AST Diagram](image-url)
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

<table>
<thead>
<tr>
<th>Abstract grammar</th>
<th>Concrete grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \in \text{Term} ::= \text{true} \mid \text{false} \mid \text{not } t \mid \text{if } t \ 	ext{then } t \ 	ext{else } t$</td>
<td>$t \in \text{Term} ::= \text{true} \mid \text{false} \mid \text{not } t \mid \text{if } t \ 	ext{then } t \ 	ext{else } t \mid (t)$</td>
</tr>
</tbody>
</table>

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

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

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))

Abstract syntax trees

```
true
   /  \
  /    \ 
/        \
not      not
   \
true   false   true
         \
           /
          false
```

defines set

defines set

linear encoding
Translating grammars into Haskell data types

Strategy: grammar → 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: $s ::= \epsilon | ts$ or $s ::= \epsilon | t, s$
- can translate any of these to a Haskell list:
  ```haskell
data Term = ...
type Sentence = [Term]
```
Example: Annotated arithmetic expression language

Abstract syntax

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

Haskell encoding

```haskell
type Comment = String

data Expr = Neg Expr
| Annot Expr Comment
| Add Expr Expr
| Mul Expr Expr
| Lit Int```

Encoding grammars as Haskell data types