Syntax and Grammars
Outline

What is a language?

Abstract syntax and grammars

Abstract syntax vs. concrete syntax

Encoding grammars as Haskell data types
What is a language?

**Language**: a system of communication using “words” in a structured way

**Natural language**
- used for arbitrary communication
- complex, nuanced, and imprecise

**Programming language**
- used to describe aspects of computation
  i.e. systematic transformation of representation
- programs have a precise **structure** and **meaning**

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

```
2 + 3 * 4  
/ 
3  4

(5 + 6) * (7 + 8)
/ 
5  6  
/ 
7  8

if true then (2+3) else 5
/ 
2  3
/ 
true
```

2 + 3 * 4  
/ 
3  4

(5 + 6) * (7 + 8)
/ 
5  6  
/ 
7  8

if true then (2+3) else 5
/ 
2  3
/ 
true
Grammars are a **metalanguage** for describing syntax

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

\[
\text{production rules}
\]

syntactic category → nonterminal symbol

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

terminal symbol
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

$s \in Sentence ::= n \; v \; n \mid s \; \text{and} \; s$

$n \in Noun ::= \text{cats} \mid \text{dogs} \mid \text{ducks}$

$v \in Verb ::= \text{chase} \mid \text{cuddle}$

$s$
$
\Rightarrow n \; v \; n$

$
\Rightarrow \text{cats} \; v \; n$

$
\Rightarrow \text{cats} \; v \; \text{ducks}$

$
\Rightarrow \text{cats cuddle ducks}$
Exercise

Animal behavior language

$$s \in \text{Sentence} ::= n \; v \; n \mid s \; \text{and} \; s$$

$$n \in \text{Noun} ::= \text{cats} \mid \text{dogs} \mid \text{ducks}$$

$$v \in \text{Verb} ::= \text{chase} \mid \text{cuddle}$$

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
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

Language generated by grammar: set of all ASTs

\[ \text{Term} = \{\text{true, false}\} \cup \{ t \mid t \in \text{Term}\} \cup \{ \text{not} \ t \mid t \in \text{Term}\} \cup \{ \text{if} \ t_1 \ t_2 \ t_3 \mid t_1, t_2, t_3 \in \text{Term}\} \]
Exercise

Arithmetic expression language

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

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:

\[
\begin{align*}
\text{neg} & \quad \text{add} \\
\text{add} & \quad \text{neg} \\
5 & \quad 3 \\
5 & \quad \text{3}
\end{align*}
\]
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 syntax tree → Pretty Printer → source code
(concrete syntax)
Abstract grammar vs. concrete grammar

Abstract grammar

\[ t \in \text{Term} ::= \begin{array}{c} \text{true} \\ \text{false} \\ \text{not} \; t \\ \text{if} \; t \; t \; t \end{array} \]

Concrete grammar

\[ t \in \text{Term} ::= \begin{array}{c} \text{true} \\ \text{false} \\ \text{not} \; t \\ \text{if} \; t \; \text{then} \; t \; \text{else} \; t \\ ( \; t \; ) \end{array} \]

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**

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

**Abstract syntax trees**

```
true       if
    |     \\
  not    |     \\
  false  true false
```

**Haskell data type definition**

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

**Haskell values**

- \(\text{Lit True}\)
- \(\text{If (Lit True)}\)
  - \(\text{Lit False}\)
  - \(\text{Lit True}\)
- \(\text{Not (Not (Lit False))}\)

Encoding grammars as Haskell data types
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: $s ::= \epsilon \mid ts$ or $s ::= \epsilon \mid t, s$
- can translate any of these to a Haskell list:

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

Abstract syntax

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

Haskell encoding

```haskell
type Comment = String

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