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**

**Examples**: English, Chinese, Hindi, Arabic, Spanish, ...

**Examples**: 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)
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

\[
2 + 3 \times 4 \\
\]
\[
(5 + 6) \times (7 + 8) \\
\]
\[
\text{if true then (2+3) else 5} \\
\]

Abstract syntax and grammars
Grammars are a **metalanguage** for describing syntax. The language we’re defining is called the **object language**.

Grammars define syntactic categories and nonterminal symbols. The syntax for sentences is defined as:

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

Nouns and verbs are defined as:

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

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

\[
\begin{align*}
    s &\Rightarrow n \ v \ n \\
    n &\Rightarrow \text{cats} \ v \ n \\
    n &\Rightarrow \text{cats} \ v \ \text{ducks} \\
    n &\Rightarrow \text{cats} \ \text{cuddle} \ \text{ducks}
\end{align*}
\]
Exercise

Animal behavior language

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

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

\[
\begin{array}{c}
\text{true} \\
\text{if} \\
\text{true} \\
\text{false} \\
\text{true} \\
\text{not} \\
\text{false}
\end{array}
\]

Language generated by grammar: set of all ASTs

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

   ![AST diagrams]

   1. \[ \text{neg} \quad \mid \quad \text{add} \]
      \[ \quad \mid \quad \text{add} \]
      \[ \quad \mid \quad \text{neg} \]
      \[ \quad \mid \quad 3 \]
      \[ \quad \mid \quad 5 \]
      \[ \quad \mid \quad 3 \]
      \[ \quad \mid \quad 5 \]
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 covered in this class … (CS 480)
Pretty printing: transforms abstract syntax into concrete syntax

Inverse of parsing!

Abstract syntax vs. concrete syntax
Abstract grammar vs. concrete grammar

**Abstract grammar**

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

**Concrete grammar**

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

Our focus is on **abstract syntax**

- we’re always writing **trees**, even if it looks like text
- we 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

```
true
  \text{if}
  \quad | \quad \text{not}
  \quad | \quad true
  \quad | \quad false
  \quad | \quad true
  \quad | \quad false
```

Haskell data type definition

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

Haskell values

- \(\text{Lit True}\)
- \(\text{If (Lit True)}\)
  \(\text{(Lit False)}\)
  \(\text{(Lit True)}\)
- \(\text{Not (Not (Lit False))}\)
Translating grammars into Haskell data types

Strategy: grammar $\rightarrow$ Haskell

1. For each basic nonterminal, choose a built-in type, e.g. \texttt{Int}, \texttt{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:

\begin{verbatim}
data Term = ...
type Sentence = [Term]
\end{verbatim}
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 Comment Expr
          | Add Expr Expr
          | Mul Expr Expr
          | Lit Int
```

Encoding grammars as Haskell data types