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**

We use a broad interpretation of “programming language”

English, Chinese, Hindi, Arabic, Spanish, ...

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

\[
\begin{align*}
2 + 3 & \ast 4 \\
(5 + 6) & \ast (7 + 8) \\
\text{if true then } (2+3) & \text{ else 5}
\end{align*}
\]
Grammars

Grammars are a metalanguage for describing syntax.

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

syntactic category \( s \in \text{Sentence} \) ::= \( n \ v \ n \ | \ s \ \text{and} \ s \)

nonterminal symbol

\( n \in \text{Noun} \) ::= \text{cats} | \text{dogs} | \text{ducks} \n
\( v \in \text{Verb} \) ::= \text{chase} | \text{cuddle} \n
production rules

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

\[
\begin{align*}
  s &\in \text{Sentence} ::= n \; v \; n \mid s \; \text{and} \; s \\
n &\in \text{Noun} ::= \text{cats} \mid \text{dogs} \mid \text{ducks} \\
v &\in \text{Verb} ::= \text{chase} \mid \text{cuddle}
\end{align*}
\]

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

Animal behavior language

\[
\begin{align*}
\text{n} & \in \text{Noun} \quad ::= \quad \text{cats} \mid \text{dogs} \mid \text{ducks} \\
\text{v} & \in \text{Verb} \quad ::= \quad \text{chase} \mid \text{cuddle} \\
\text{s} & \in \text{Sentence} \quad ::= \quad \text{n} \ \text{v} \ \text{n} \mid \text{s} \ \text{and} \ \text{s} \ \\
\end{align*}
\]

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)

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

#### Example ASTs

![Example ASTs]

#### Language generated by grammar: set of all ASTs

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

Arithmetic expression language

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

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:

\[
\begin{array}{c}
\text{neg} \\
\text{add} \\
5 \quad \text{add} \\
5 \\
\end{array}
\]

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

**Abstract grammar**

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

**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
- use parentheses to **disambiguate** textual representation of ASTs but they are **not** part of the syntax
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

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

Abstract syntax trees

```
        true
       /
      /   \   
     /     \     
    /       \     
   /         \      
  true  false  true  false
```

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))

Encodings of set linear encoding
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:

```haskell
data Term = ...

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

Abstract syntax

\[ n \in \text{Nat} ::= \text{(natural number)} \]

\[ c \in \text{Comm} ::= \text{(comment string)} \]

\[ e \in \text{Expr} ::= \text{neg } e \quad \text{negation} \]
\[ e @ c \quad \text{comment} \]
\[ e + e \quad \text{addition} \]
\[ e * e \quad \text{multiplication} \]
\[ n \quad \text{literal} \]

Haskell encoding

```
type Comment = String

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