

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

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

Programming language

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

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

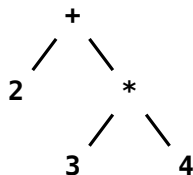
Abstract syntax vs. concrete syntax

Encoding grammars as Haskell data types

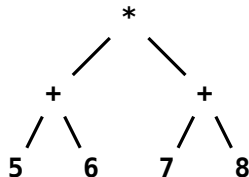
Programs are trees!

Abstract syntax tree (AST): captures the essential structure of a program

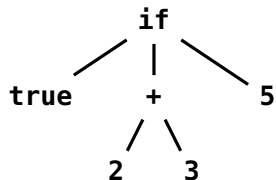
- everything needed to determine its semantics



`2 + 3 * 4`



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



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

Grammars

Grammars are a **metalanguage** for describing syntax

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



syntactic category 

 nonterminal symbol

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

} 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

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

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?

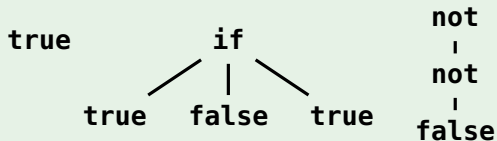
- **cats chase dogs**
- **cats and dogs chase ducks**
- **dogs cuddle cats and ducks chase dogs**
- **dogs chase cats and cats chase ducks and ducks chase dogs**

Abstract syntax trees

Grammar (BNF notation)

$$t \in Term ::= \begin{array}{l} \mathbf{true} \\ \mathbf{false} \\ \mathbf{not} \ t \\ \mathbf{if} \ t \ t \ t \end{array}$$

Example ASTs



Language generated by grammar: set of all ASTs

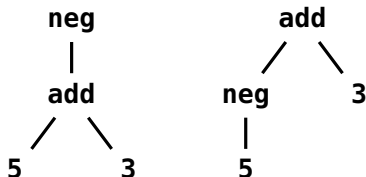
$$Term = \{\mathbf{true}, \mathbf{false}\} \cup \left\{ \begin{array}{c} \mathbf{not} \\ | \\ t \end{array} \mid t \in Term \right\} \cup \left\{ \begin{array}{c} \mathbf{if} \\ / \quad | \quad \backslash \\ t_1 \quad t_2 \quad t_3 \end{array} \mid t_1, t_2, t_3 \in Term \right\}$$

Exercise

Arithmetic expression language

$i \in Int ::= \mathbf{1} \mid \mathbf{2} \mid \dots$
 $e \in Expr ::= \mathbf{add} \ e \ e$
 $\mid \mathbf{mul} \ e \ e$
 $\mid \mathbf{neg} \ e$
 $\mid i$

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:



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

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

Pretty printing: transforms abstract syntax into concrete syntax

Inverse of parsing!



Abstract grammar vs. concrete grammar

Abstract grammar

```
 $t \in \text{Term} ::= \text{true}$   
                  |  $\text{false}$   
                  |  $\text{not } t$   
                  |  $\text{if } t \ t \ t$ 
```

Concrete grammar

```
 $t \in \text{Term} ::= \text{true}$   
                  |  $\text{false}$   
                  |  $\text{not } t$   
                  |  $\text{if } t \text{ then } t \text{ else } t$   
                  |  $( t )$ 
```

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

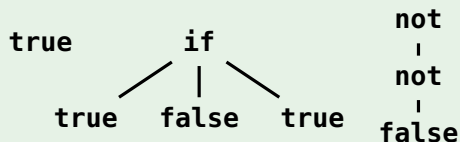
Abstract grammar

$b \in Bool ::= \mathbf{true} \mid \mathbf{false}$
 $t \in Term ::= \mathbf{not} \ t$
 $\quad \mid \mathbf{if} \ t \ t \ t$
 $\quad \mid b$

Haskell data type definition

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

Abstract syntax trees



defines set

linear
encoding

Haskell values

- Lit True
- If (Lit True)
 (Lit False)
 (Lit True)
- Not (Not (Lit False))

defines set

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: $s ::= \epsilon \mid t s$ 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

$n \in Nat ::=$ (natural number)

$c \in Comm ::=$ (comment string)

$e \in Expr ::=$

neg e	negation
e @ c	comment
$e + e$	addition
$e * e$	multiplication
n	literal

Haskell encoding

```
type Comment = String

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