Syntax and Grammars

Outline

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, …

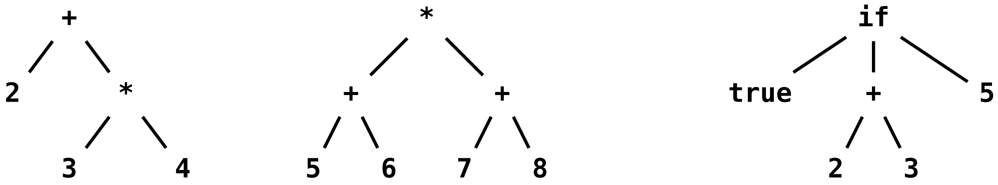
Metalanguages for defining semantics: mathematics, inference rules, Haskell, …

Outline

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

$$
\left.\begin{array}{rcl}
s \in \mathit{Sentence} & ::= & n \;\; v \;\; n \;\mid\; s \;\, \mathtt{and} \;\, s \\
n \in \mathit{Noun} & ::= & \mathtt{cats} \;\mid\; \mathtt{dogs} \;\mid\; \mathtt{ducks} \\
v \in \mathit{Verb} & ::= & \mathtt{chase} \;\mid\; \mathtt{cuddle}
\end{array}\right\} \text{production rules}
$$

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

$$
\begin{array}{rcl}
s \in \mathit{Sentence} & ::= & n \;\; v \;\; n \;\mid\; s \;\; \texttt{and} \;\; s \\
n \in \mathit{Noun} & ::= & \texttt{cats} \mid \texttt{dogs} \mid \texttt{ducks} \\
v \in \mathit{Verb} & ::= & \texttt{chase} \mid \texttt{cuddle}
\end{array}
$$

$$
\begin{array}{ll}
 & s \\
\Rightarrow & n \;\; v \;\; n \\
\Rightarrow & \texttt{cats} \;\; v \;\; n \\
\Rightarrow & \texttt{cats} \;\; v \;\; \texttt{ducks} \\
\Rightarrow & \texttt{cats cuddle ducks}
\end{array}
$$

Exercise

Animal behavior language

$$\begin{aligned} s \in \mathit{Sentence} &::= n\ v\ n \mid s\ \texttt{and}\ s \\ n \in \mathit{Noun} &::= \texttt{cats} \mid \texttt{dogs} \mid \texttt{ducks} \\ v \in \mathit{Verb} &::= \texttt{chase} \mid \texttt{cuddle} \end{aligned}$$

Is each "program" in the animal behavior language?

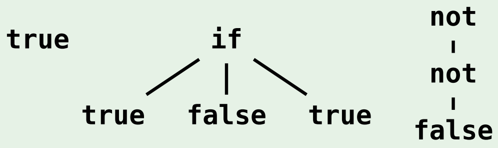
- `cats chase dogs`
- `cats and dogs chase ducks`
- `dogs cuddle cats and ducks chase dogs`
- `dogs chase cats and cats chase ducks and ducks chase dogs`

Abstract syntax trees

Grammar (BNF notation)

$$
\begin{array}{lcl}
t \in \mathit{Term} & ::= & \texttt{true} \\
 & | & \texttt{false} \\
 & | & \texttt{not}\ t \\
 & | & \texttt{if}\ t\ t\ t
\end{array}
$$

Example ASTs



Language generated by grammar: set of all ASTs

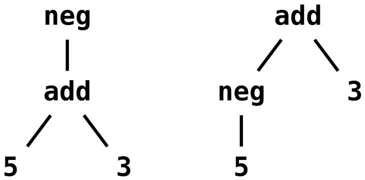
$$
\mathit{Term} = \{\texttt{true}, \texttt{false}\} \cup \left\{ \begin{array}{c} \texttt{not} \\ | \\ t \end{array} \;\middle|\; t \in \mathit{Term} \right\} \cup \left\{ \begin{array}{ccc} & \texttt{if} & \\ / & | & \backslash \\ t_1 & t_2 & t_3 \end{array} \;\middle|\; t_1, t_2, t_3 \in \mathit{Term} \right\}
$$

Exercise

Arithmetic expression language

$$
\begin{array}{rcl}
i \in \mathit{Int} & ::= & \mathtt{1} \mid \mathtt{2} \mid \ldots \\
e \in \mathit{Expr} & ::= & \mathtt{add}\ e\ e \\
 & \mid & \mathtt{mul}\ e\ e \\
 & \mid & \mathtt{neg}\ e \\
 & \mid & i
\end{array}
$$

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

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

source code (concrete syntax) → **Parser** → **abstract syntax tree**

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

$$
\begin{array}{rcl}
t \in \mathit{Term} & ::= & \texttt{true} \\
& | & \texttt{false} \\
& | & \texttt{not}\ t \\
& | & \texttt{if}\ t\ t\ t
\end{array}
$$

Concrete grammar

$$
\begin{array}{rcl}
t \in \mathit{Term} & ::= & \texttt{true} \\
& | & \texttt{false} \\
& | & \texttt{not}\ t \\
& | & \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\
& | & \texttt{(}\ t\ \texttt{)}
\end{array}
$$

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

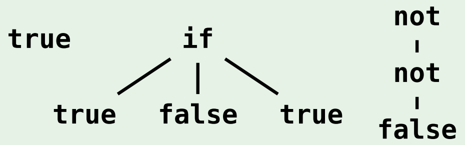
Encoding abstract syntax in Haskell

defines set

Abstract grammar

$$
\begin{array}{rcl}
b \in \mathit{Bool} & ::= & \texttt{true} \mid \texttt{false} \\
t \in \mathit{Term} & ::= & \texttt{not}\ t \\
& \mid & \texttt{if}\ t\ t\ t \\
& \mid & b
\end{array}
$$

Abstract syntax trees



linear encoding

Haskell data type definition

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

defines set

Haskell values

- `Lit True`
- `If (Lit True)`
  `(Lit False)`
  `(Lit True)`
- `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. **`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 $s ::= \epsilon \mid t\ \texttt{,}\ s$
- can translate any of these to a Haskell list:

```
data Term = ...
type Sentence = [Term]
```

Example: Annotated arithmetic expression language

Abstract syntax

$$
\begin{array}{rcll}
n \in \mathit{Nat} & ::= & \text{(natural number)} & \\
c \in \mathit{Comm} & ::= & \text{(comment string)} & \\
e \in \mathit{Expr} & ::= & \mathbf{neg}\ e & \text{negation} \\
& \mid & e\ \texttt{@}\ c & \text{comment} \\
& \mid & e\ \texttt{+}\ e & \text{addition} \\
& \mid & e\ \texttt{*}\ e & \text{multiplication} \\
& \mid & n & \text{literal}
\end{array}
$$

Haskell encoding

```haskell
type Comment = String

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