Syntax and Grammars
June 23, 2015
Outline

Syntax and grammars

Representing abstract syntax

Is it abstract or concrete syntax?

Relating abstract syntax to Haskell
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,…
Syntax vs. semantics

Two main aspects of a language:

- **syntax**: the structure of its programs
- **semantics**: the meaning of its programs

Example: well-structured sentences

- **syntax** defines the set of all sentences

How can we define a syntax?

1. enumerate all sentences
2. define rules to construct sentences (grammar)
Grammars

Grammars are a **metalanguage** for describing syntax.

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

\[
\begin{align*}
\text{syntactic category} & \quad \text{nonterminal symbol} \\
S & \in \text{Sentence} \quad ::= \quad n \; v \; n \quad | \quad s \; \text{and} \; s \\
N & \in \text{Noun} \quad ::= \quad \text{cats} \quad | \quad \text{dogs} \quad | \quad \text{ducks} \\
V & \in \text{Verb} \quad ::= \quad \text{chase} \quad | \quad \text{cuddle} \\
\end{align*}
\]
Generating sentences from grammars

How to generate a sentence from a grammar
1. start with a nonterminal $s$
2. find production rules with $s$ on the LHS
3. replace $s$ by one possible RHS case

A sentence is in the language if and only if it can be generated by the grammar!

Animal behavior language

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

$s 
\Rightarrow n \lor n
\Rightarrow \text{cats} \lor n
\Rightarrow \text{cats} \lor \text{ducks}
\Rightarrow \text{cats cuddle ducks}$
Colorless green ideas sleep furiously

Just because a sentence is grammatically correct, doesn’t mean it is semantically correct!

Derivation order

The order of rule application is *not* fixed

**Animal behavior language**

\[
\begin{align*}
S \in \text{Sentence} & ::= n \; v \; n \mid s \; \text{and} \; s \quad (R1) \\
n \in \text{Noun} & ::= \text{cats} \mid \text{dogs} \mid \text{ducks} \quad (R2) \\
v \in \text{Verb} & ::= \text{chase} \mid \text{cuddle} \quad (R3)
\end{align*}
\]

\[
\begin{align*}
S & \Rightarrow n \; v \; n \quad R1 \\
\Rightarrow \text{cats} \; v \; n \quad R2 \\
\Rightarrow \text{cats} \; v \; \text{ducks} \quad R3 \\
\Rightarrow \text{cats} \; \text{cuddle} \; \text{ducks}
\end{align*}
\]
Exercise: animal behavior language

Animal behavior language
\[
S \in \text{Sentence} ::= \text{n} \ \text{v} \ \text{n} \mid \text{s} \ \text{and} \ \text{s} \\
\text{n} \in \text{Noun} ::= \text{cats} \mid \text{dogs} \mid \text{ducks} \\
\text{v} \in \text{Verb} ::= \text{chase} \mid \text{cuddle}
\]

Which of the following sentences are well defined in the animal behavior language?

- cats chase dogs  \text{Yes}
- cats and dogs chase ducks  \text{No}
- dogs cuddle cats and ducks chase dogs  \text{Yes}
- dogs chase cats and cats chase ducks and ducks chase dogs  \text{Yes}
Exercise: boolean language

Write a grammar for boolean expressions built from the terms true and false and the logical operation not

Boolean language
\[ t \in \text{Term} ::= \begin{array}{l}
\text{true} \ (R1) \\
\text{false} \ (R2) \\
\text{not } t \ (R3)
\end{array} \]

Derive the sentence not not false

\[
\begin{align*}
& \Rightarrow t \ t \ t \ \text{R3} \\
& \Rightarrow t \ \text{not } t \ \text{R2} \\
& \Rightarrow t \ \text{not false} \ \text{R3} \\
& \Rightarrow \text{not not false}
\end{align*}
\]
Outline

Syntax and grammars

Representing abstract syntax

Is it abstract or concrete syntax?

Relating abstract syntax to Haskell
Abstract syntax trees — ASTs

**Syntax tree**: a structure to *represent* derivations

**Derivation**: a process of producing a sentence according to the rules of a grammar

**Grammar (BNF notation)**

```
t ∈ Term ::= true
    | false
    | not t
    | if t t t
```

**Example ASTs**

```
true

if

true false true
```

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

```
Term = \{true, false\} \cup \{ t \mid t ∈ Term\} \cup \{ if t \mid t1, t2, t3 ∈ Term\}
```

Representing abstract syntax
Programs are trees!

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

- everything needed to determine its semantics

```
2 + 3 x 4
  /   \
 3   4

(5 + 6) x (7 + 8)
  /  \
 5   6
  /  \
 7   8

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

`2 + 3 x 4`  
`(5 + 6) x (7 + 8)`  
`if true then (2 + 3) else 5`
Observations about ASTs

Leaves contain *terminal symbols*

Internal nodes contain *nonterminal symbols*

Nonterminal in the root node indicates the *type* of the syntax tree

Derivation order is *not represented* — which is a good thing, because it is *not important*
Outline

Syntax and grammars

Representing abstract syntax

Is it abstract or concrete syntax?

Relating abstract syntax to Haskell
Abstract syntax vs. concrete syntax

**Abstract syntax**: captures the *essential structure* of programs

**Concrete syntax**: describes how programs are written down (linear representation)

Abstract grammar

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

Concrete grammar

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

We will focus on **abstract syntax** — always constructing **trees**

- use parentheses to disambiguate linear representations of ASTs
Example: animal behavior language

Animal behavior language

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

set of sentences/strings (linear)

\[ \text{Sentence} = \{ \text{cats chase dogs, dogs cuddle ducks,} \ldots \} \]

Is it abstract or concrete syntax?
Exercise: arithmetic expression language

1. Draw two different ASTs for the expression: $2 + 3 \times 4$

2. Draw an AST for the expression: $-5 \times (6 + 7)$

3. What are the integer *results* of evaluating the following ASTs?

```
\[
\begin{array}{c}
\text{neg} \\
\text{add} \\
\text{add} \\
\text{add} \\
5 \\
3 \\
5 \\
5 \\
3 \\
\end{array}
\]
```

-8

```
\[
\begin{array}{c}
\text{neg} \\
\text{add} \\
\text{add} \\
\text{neg} \\
1 \\
2 \\
3 \\
\end{array}
\]
```

-2

Arithmetic expression language

$i \in Int ::= 1 | 2 | ...$

$e \in Expr ::= \text{add} \ e \ e$

$\ | \text{mul} \ e \ e$

$\ | \text{neg} \ e$

$\ | \ i$
Outline

Syntax and grammars

Representing abstract syntax

Is it abstract or concrete syntax?

Relating abstract syntax to Haskell
Encoding abstract syntax in Haskell

Abstract grammar

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

defines set

Abstract syntax trees

\[
\begin{align*}
    \text{true} \quad & \quad \text{if} \quad \text{true} \quad \text{false} \quad \text{true} \\
    \text{false} \quad & \quad \text{not} \quad \text{false} \\
\end{align*}
\]

Haskell data type definition

```hs
data Bool = True | False

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

defines set

Haskell values

```hs
Lit True
If (Lit True)
  (Lit False)
  (Lit True)
Not (Not (Lit False))
```

linear encoding
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 rule, define a data constructor
4. The nonterminals in the production rules determine the arguments to the constructor

Special rule for lists:

- in grammars, \( s ::= t^* \) is shorthand for: \( s ::= \varepsilon | t \ s \) or \( s ::= \varepsilon | t, s \n
- can translate any of these to a Haskell list:

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

Abstract syntax

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

Haskell encoding

\[
\begin{align*}
type \text{Comment} & = \text{String} \\
data \text{Expr} & = \text{Neg } \text{Expr} \\
& \quad | \quad \text{Annot } \text{Comment } \text{Expr} \\
& \quad | \quad \text{Add } \text{Expr } \text{Expr} \\
& \quad | \quad \text{Mul } \text{Expr } \text{Expr} \\
& \quad | \quad \text{Lit } \text{Int}
\end{align*}
\]