510: Programming Languages

Concrete and Abstract Syntax

David Walker

Fall, 2002

Levels of Syntax

Concrete syntax.

- The **surface structure** of a language.
- Representation of phrases as strings.
- Concerned with readability, ambiguity.

Levels of Syntax

Abstract syntax.

- The **deep structure** of a language.
- Representation of phrases as ast's (terms).
- Concerned with fundamental structure.

Levels of Syntax

Two levels of abstract syntax:

- **First-order (FOAS)**: tree structure.
- **Higher-order (HOAS)**: binding and scope.

Specifying Concrete Syntax

Concrete syntax is an inductively defined set of **strings**.

- Alphabet of **tokens** (*e.g.*, identifiers, numbers).

- Context-free grammar notation (aka BNF).

Context-Free Grammars

A **context-free grammar** consists of three things:

1. An alphabet Σ of **terminals**, or **symbols**.

2. A finite set $\mathcal{N}$ of **non-terminals**, or **categories**.

3. A finite set of **productions** of the form $A ::= \alpha$, where $A \in \mathcal{N}$ and $\alpha \in (\Sigma \cup \mathcal{N})^*$.

Example: Arithmetic Expressions

Category	*Member*		*Elements*
Digits	d	$::=$	$\mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9}$
Numbers	n	$::=$	$d \mid n\,d$
Expressions	e	$::=$	$n \mid e_1\mathtt{+}e_2 \mid e_1\mathtt{*}e_2$

Grammars as Inductive Definitions

A grammar is a **simultaneous inductive definition** of several sets of strings.

- Each category determines a set of strings.

- Each production determines a rule.

Example: Arithmetic Expressions

$$\frac{}{\textsf{0 digit}} \quad \cdots \quad \frac{}{\textsf{9 digit}}$$

$$\frac{d \ \textsf{digit}}{d \ \textsf{number}} \qquad \frac{n \ \textsf{number} \qquad d \ \textsf{digit}}{n\,d \ \textsf{number}}$$

$$\frac{n \ \textsf{number}}{n \ \textsf{expr}}$$

$$\frac{e_1 \ \textsf{expr} \qquad e_2 \ \textsf{expr}}{e_1\texttt{+}e_2 \ \textsf{expr}}$$

$$\frac{e_1 \ \textsf{expr} \qquad e_2 \ \textsf{expr}}{e_1\texttt{*}e_2 \ \textsf{expr}}$$

A note on BNF

- Grammars, which define sets of **strings**, are often given in BNF:

$$\textit{Expressions} \quad e \quad ::= \quad n \mid e_1\texttt{+}e_2 \mid e_1\texttt{*}e_2$$

- but this is really a short form for an inductive definition.

- conversely, there are sets of **terms** (as opposed to strings) which we might define using inductive definitions, but we often end up using BNF as a short form

Ambiguity

This grammar is **ambiguous**: the same string arises in different ways!

The string 1+2*3 may be thought of in two ways:

- $e \to e_1$*$e_2 \to e_{1,1}$+$e_{1,2}$*$e_2 \to^* $ 1+2*3.

- $e \to e_1$+$e_2 \to e_1$+$e_{2,1}$*$e_{2,2} \to^* $ 1+2*3.

You cannot tell by looking at the string which derivation was used!

Ambiguity

Suppose we wish to define an evaluator by rule induction:

$$
\begin{array}{rcl}
eval_{\mathsf{dig}}(\mathsf{0}) & = & 0 \\
 & \vdots & \\
eval_{\mathsf{dig}}(\mathsf{9}) & = & 9 \\
 & & \\
eval_{\mathsf{num}}(d) & = & eval_{\mathsf{dig}}(d) \\
eval_{\mathsf{num}}(n\,d) & = & 10 \times eval_{\mathsf{num}}(n) + eval_{\mathsf{dig}}(d) \\
 & & \\
eval_{\mathsf{exp}}(n) & = & eval_{\mathsf{num}}(n) \\
eval_{\mathsf{exp}}(e_1 \texttt{+} e_2) & = & eval_{\mathsf{exp}}(e_1) + eval_{\mathsf{exp}}(e_2) \\
eval_{\mathsf{exp}}(e_1 \texttt{*} e_2) & = & eval_{\mathsf{exp}}(e_1) \times eval_{\mathsf{exp}}(e_2)
\end{array}
$$

Ambiguity

Do these equations determine well-defined evaluation functions **on strings**?

No! Strings with two derivations have two (different) values!

- If we knew the rule to use, we would know the decomposition.

- We cannot tell just by looking at the string.

Ambiguity

An equation such as

$$eval_{\mathsf{exp}}(e_1\texttt{+}e_2) = eval_{\mathsf{exp}}(e_1) + eval_{\mathsf{exp}}(e_2)$$

means "decompose the **string** e into the **string** $e_1\texttt{+}e_2$".

Similarly,

$$eval_{\mathsf{exp}}(e_1\texttt{*}e_2) = eval_{\mathsf{exp}}(e_1) \times eval_{\mathsf{exp}}(e_2)$$

means "decompose the **string** e into the **string** $e_1\texttt{*}e_2$".

For some strings there is more than one decomposition!

Resolving Ambiguity

Introduce precedence conventions and parenthesization so that

- Every string has a **unique** derivation.

- User can override defaults by grouping.

The decompositions are unique, so functions such as $eval_{\mathsf{exp}}$ are well-defined.

A Better Idea

Separate the **deep structure** from the **surface syntax**.

- Surface syntax: human-oriented, string representation.

- Deep structure: machine-oriented, tree (term) representation.

The deep structure reveals "this is an addition" directly, rather than by precedence conventions.

Syntax Analysis, or Parsing

A job of a **parser** is to analyse the surface syntax to determine the deep structure of an expression.

- Take apart the string once and for all.

- Translate to abstract syntax to reveal structure.

Perform syntactic analysis once and for all.

Abstact Syntax

The abstract syntax of a language is an inductively-defined set of terms.

$$\frac{}{\texttt{number}(n)\ \textsf{expr}}$$

$$\frac{e_1\ \textsf{expr} \quad e_2\ \textsf{expr}}{\texttt{plus}(e_1, e_2)\ \textsf{expr}}$$

$$\frac{e_1\ \textsf{expr} \quad e_2\ \textsf{expr}}{\texttt{times}(e_1, e_2)\ \textsf{expr}}$$

Abstract Syntax

A term wears its structure on its sleeve! There is no ambiguity when decomposing a term into its parts.

- The operators are **injective**, meaning that we can recover the sub-terms from the term.

- String concatenation is **non-injective**: many pairs of strings map to the same string.

Abstract Syntax in ML

The same idea, written in ML notation:

```
datatype expr =
  Num of int |
  Plus of expr * expr |
  Times of expr * expr
```

The values of type `expr` are pieces of abstract syntax.

Structural Induction

Rule induction over abstract syntax is called **structural induction**.

To show P holds of every arithmetic expression e, it is enough to show

- P holds of $\texttt{number}(n)$.
- if P holds of e_1 and e_2, then P holds of $\texttt{plus}(e_1, e_2)$.
- if P holds of e_1 and e_2, then P holds of $\texttt{times}(e_1, e_2)$.

Structural Induction

We may define evaluation by structural induction over the abstract syntax:

$$
\begin{aligned}
eval(\mathtt{number}(n)) &= n \\
eval(\mathtt{plus}(e_1, e_2)) &= eval(e_1) + eval(e_2) \\
eval(\mathtt{times}(e_1, e_2)) &= eval(e_1) \times eval(e_2)
\end{aligned}
$$

These equations determine a function. (Easy proof by structural induction.)

Terminology

The terms representing abstract syntax are sometimes called

- **abstract syntax trees**, or **ast**'s, emphasizing the tree structure.

- **parse trees**, reflecting the result of a parse (syntactic analysis).

Parsing

A parser is a function mapping concrete to abstract syntax.

- Use a well-structured grammar (in several senses).

- Analyze and decompose strings using one of several methods.
  - Top-down parsers (recursive descent)
  - Bottom-up parsers (LR parsers)

Parsing

These equations determine a parsing function:

$$
\begin{array}{rcl}
parse_{\mathsf{dig}}(0) & = & 0 \\
 & \vdots & \\
parse_{\mathsf{dig}}(9) & = & 9 \\
\\
parse_{\mathsf{num}}(d) & = & \mathtt{number}(parse_{\mathsf{dig}}(d)) \\
parse_{\mathsf{num}}(n\,d) & = & \mathtt{number}(10 \times k + parse_{\mathsf{dig}}\,d), \\
 & & \textit{where} \;\; parse_{\mathsf{num}}\,n = \mathtt{number}(k) \\
\\
parse_{\mathsf{exp}}(t) & = & parse_{\mathsf{trm}}(t) \\
parse_{\mathsf{exp}}(t\mathtt{+}e) & = & \mathtt{plus}(parse_{\mathsf{trm}}(t), parse_{\mathsf{exp}}(e)) \\
\\
parse_{\mathsf{trm}}(f) & = & parse_{\mathsf{fct}}(f) \\
parse_{\mathsf{trm}}(f\mathtt{*}t) & = & \mathtt{times}(parse_{\mathsf{fct}}(f), parse_{\mathsf{trm}}(t)) \\
\\
parse_{\mathsf{fct}}(n) & = & parse_{\mathsf{num}}(n) \\
parse_{\mathsf{fct}}(\mathtt{(}e\mathtt{)}) & = & parse_{\mathsf{exp}}(e)
\end{array}
$$

Summary

It is useful to separate **concrete** from **abstract** syntax.

- Surface features.
- Deep structure.

Parsers translate from concrete to abstract syntax.

- Using various grammar "tricks".
- Often generated automatically from the grammar.