Levels of Syntax

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

Abstract syntax.
- The deep structure of a language.
- Representation of phrases as trees (terms).
- Concerned with fundamental structure.

Two levels of abstract syntax:
- First-order (FOAS): abstract syntax tree (ast)
- Higher-order (HOAS): abstract binding tree (abt)

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 $\Sigma$ 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Member</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digits</td>
<td>d</td>
<td>::= 0</td>
</tr>
<tr>
<td>Numbers</td>
<td>n</td>
<td>::= d</td>
</tr>
<tr>
<td>Expressions</td>
<td>e</td>
<td>::= n</td>
</tr>
</tbody>
</table>

#### Example: Arithmetic Expressions

0 digit · · · 9 digit

d digit n number d digit

\[
e_{\text{n expr}} \quad e_{1 \text{ expr}} e_{2 \text{ expr}}
\]

### 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.

---

### Ambiguity

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

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

- \( e \rightarrow e₁*e₂ \rightarrow e₁*e₂ → e₁*e₂ → 1+2*3 \).
- \( e \rightarrow e₁*e₂ \rightarrow e₁*e₂ → e₁*e₂ → 1+2*3 \).

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

### Ambiguity

Suppose we wish to define an evaluator by rule induction:

\[
\begin{align*}
\text{eval}_d(0) &= 0 \\
\text{eval}_d(9) &= 9 \\
\text{eval}_d(d) &= \text{eval}_d(d) \\
\text{eval}_{\text{n expr}}(n d) &= 10 \times \text{eval}_{\text{n expr}}(n) + \text{eval}_d(d) \\
\text{eval}_{\text{expr}}(n) &= \text{eval}_{\text{num}}(n) \\
\text{eval}_{\text{expr}}(e₁*e₂) &= \text{eval}_{\text{expr}}(e₁) + \text{eval}_{\text{expr}}(e₂) \\
\text{eval}_{\text{expr}}(e₁*e₂) &= \text{eval}_{\text{expr}}(e₁) \times \text{eval}_{\text{expr}}(e₂)
\end{align*}
\]

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.
Resolving Ambiguity

An equation such as
\[ \text{eval}_{n}(e_1 + e_2) = \text{eval}_{n}(e_1) + \text{eval}_{n}(e_2) \]
means "decompose the string \( e \) into the string \( e_1 + e_2 \)."

Similarly,
\[ \text{eval}_{n}(e_1 \cdot e_2) = \text{eval}_{n}(e_1) \times \text{eval}_{n}(e_2) \]
means "decompose the string \( e \) into the string \( e_1 \cdot e_2 \)."

For some strings there is more than one decomposition!

Resolving Ambiguity

<table>
<thead>
<tr>
<th>Category</th>
<th>Member</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digits</td>
<td>( d )</td>
<td>( 0</td>
</tr>
<tr>
<td>Numbers</td>
<td>( n )</td>
<td>( d</td>
</tr>
<tr>
<td>Expressions</td>
<td>( e )</td>
<td>( t</td>
</tr>
<tr>
<td>Terms</td>
<td>( t )</td>
<td>( f</td>
</tr>
<tr>
<td>Factors</td>
<td>( f )</td>
<td>( n</td>
</tr>
</tbody>
</table>

Resolving Ambiguity

Each well-formed string is derived in exactly one way.

- Heavy “layering” of grammar.
- Introduction of additional notation.

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 \( \text{eval}_{n} \) are well-defined.

Resolving Ambiguity

Even once ambiguity is resolved, it’s not very practical to define functions such as \( \text{eval} \).

- Must analyze strings according to the grammar to find decomposition.
- Must repeat this work for every such function.
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.

Abstract Syntax

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

```
num[n] expr

expr =
  e1 expr e2 expr
  plus(e1, e2) expr

expr =
  e1 expr e2 expr
  times(e1, e2) 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 `num[n]`.
- if \( P \) holds of \( e_1 \) and \( e_2 \), then \( P \) holds of `plus(e_1, e_2)`.
- if \( P \) holds of \( e_1 \) and \( e_2 \), then \( P \) holds of `times(e_1, e_2)`. 
Structural Induction

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

\[
\begin{align*}
\text{eval}(\text{num}[n]) & = n \\
\text{eval}(\text{plus}(e_1, e_2)) & = \text{eval}(e_1) + \text{eval}(e_2) \\
\text{eval}(\text{times}(e_1, e_2)) & = \text{eval}(e_1) \times \text{eval}(e_2)
\end{align*}
\]

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

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): 530/630.
  - Bottom-up parsers (LR parsers): 410/610.

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

These equations determine a parsing function:

\[
\begin{align*}
\text{parse}_{\text{dig}}(0) & = 0 \\
\text{parse}_{\text{dig}}(9) & = 9 \\
\text{parse}_{\text{num}}(d) & = \text{num}[\text{parse}_{\text{dig}}(d)] \\
\text{parse}_{\text{num}}(n \ d) & = \text{num}[10 \times k + \text{parse}_{\text{dig}}(d)], \text{ where } \text{parse}_{\text{num}} n = \text{num}[k] \\
\text{parse}_{\text{num}}(t) & = \text{parse}_{\text{num}}(t) \\
\text{parse}_{\text{num}}(t) & = \text{parse}_{\text{num}}(t) \times \text{parse}_{\text{num}}(e) \\
\text{parse}_{\text{num}}(f) & = \text{parse}_{\text{dig}}(f) \\
\text{parse}_{\text{num}}(f) & = \text{times}(\text{parse}_{\text{num}}(f), \text{parse}_{\text{num}}(t)) \\
\text{parse}_{\text{num}}(n) & = \text{parse}_{\text{num}}(n) \\
\text{parse}_{\text{num}}(e) & = \text{parse}_{\text{num}}(e)
\end{align*}
\]

A Revised Grammar

Change the productions for numbers:

\[
\text{Numbers } n :: = d \mid d n
\]

This rule is right-recursive, admits a left-to-right analysis.

Parsing

But there is one more problem! The equation

\[
\text{parse}_{\text{num}}(n \ d) = \text{num}[10 \times k + \text{parse}_{\text{dig}} d],
\]

where

\[
\text{parse}_{\text{num}} n = \text{num}[k]
\]

requires reading the input from right to left!
Disclaimer

These guidelines are enough to build hand-coded parsers.

- Layer to resolve ambiguity.
- Replace “left recursive” rules by “right recursive” rules.

There is a well-developed theory of all of this! (See Compilers course.)

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.