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.

Specifying Concrete Syntax

Concrete syntax is an inductively defined set of strings.

- Lexical Level
  - Alphabet of tokens (e.g., identifiers, numbers).
  - Described by regular expressions or inductive judgements

- Grammatical Level
  - Language phrases made up of tokens
  - Described by context-free grammar notation (aka BNF).

A Bare Bones Start

We'll start by studying $L\{\text{num, str}\}$, a very simple language of expressions.

- Natural numbers and strings.
- Variables, simple operations, binding.

Context-Free Grammars

A context-free grammar consists of three things:

1. An alphabet $\Sigma$ of terminals, or symbols.
2. A finite set $N$ of non-terminals, or categories.
3. A finite set of productions of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (\Sigma \cup N)^*$. 
Lexical Structure

First phase of syntactic processing, called lexical analysis or **lexing**, is converting stream of characters to tokens. For $L(\text{num str})$ this involves:

1. Identifying numbers and strings
2. Identifying keywords and operators
3. Discarding "white space" (spaces, tabs, comments, etc.)

Lexical Structure for $L(\text{num str})$ (cont.)

<table>
<thead>
<tr>
<th>Item</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letter</td>
<td>ltr ::= a</td>
</tr>
<tr>
<td>Digit</td>
<td>dig ::= 0</td>
</tr>
<tr>
<td>Quote</td>
<td>qum ::= &quot;</td>
</tr>
</tbody>
</table>

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.

Lexical Analysis Judgements

- $s \text{ inp } \longleftarrow \text{ tokstr}$: Scan input
- $s \text{ item } \longleftarrow \text{ tok}$: Scan an item
- $s \text{ kwid } \longleftarrow \text{ tok}$: Scan a keyword
- $s \text{ id } \longleftarrow \text{ tok}$: Scan an identifier
- $s \text{ num } \longleftarrow \text{ tok}$: Scan a number
- $s \text{ spl } \longleftarrow \text{ tok}$: Scan a symbol
- $s \text{ lit } \longleftarrow \text{ tok}$: Scan a string literal
- $s \text{ whs } \longleftarrow \text{ tok}$: Skip whitespace
Inductive Definition of Lexical Analysis

\[
\begin{align*}
\text{Exp} & ::= \text{num} \mid \text{id} \mid \text{LP} \text{exp} \text{RP} \mid \text{exp ADD exp} \\
\text{LET id BE exp IN exp} \\
\text{Number} & ::= \text{NUM}[n] \quad (n \text{ nat}) \\
\text{String} & ::= \text{LIT}[s] \quad (s \text{ str})
\end{align*}
\]

Grammatical Structure for \(L(\text{num str})\)

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

Inductive Definition of Grammar for \(L(\text{num str})\)

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 expression evaluation is well-defined.
Resolving Ambiguity

Factor \( fct \ ::= \) num | lit | id | LP prg RP

Term \( trm \ ::= \) fct | fct MUL trm | VB fct VB

Expression \( exp \ ::= \) trm | trm ADD exp | trm CAT exp

Program \( prg \ ::= \) exp | LET id BE exp IN prg

Abstract Syntax

Even once ambiguity is resolved, it’s not very practical to use concrete syntax for studying or processing programming languages.

- Must analyze strings according to the grammar to create complicated representation that indirectly reveals meaning.

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.

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

Abstract Syntax

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 meaning (e.g., “this is an addition”) directly, rather than by precedence conventions.

Abstract Syntax Tree Signature for \( L(\text{num,str}) \)

\[
\begin{align*}
\text{ar}(\text{num}[n]) & = 0 \ (n \ \text{nat}) \\
\text{ar}(\text{str}[s]) & = 0 \ (s \ \text{str}) \\
\text{ar}(\text{id}[s]) & = 0 \ (s \ \text{str}) \\
\text{ar}(\text{plus}) & = 2 \\
\text{ar}(\text{times}) & = 2 \\
\text{ar}(\text{cat}) & = 2 \\
\text{ar}(\text{len}) & = 1 \\
\text{ar}(\text{let}[s]) & = 2 \ \text{[identifier-indexed family of operators]}
\end{align*}
\]
Inductive Definition of Abstract Syntax for \( L\{\text{num str}\} \)

<table>
<thead>
<tr>
<th></th>
<th>s num</th>
<th>s str</th>
<th>s id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1 ast</td>
<td>a2 ast</td>
<td>a3 ast</td>
<td>a ast</td>
</tr>
</tbody>
</table>

Syntax Analysis, or Parsing

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.

- Notice that the abstract syntax is independent from the particular grammar used to describe the concrete syntax.

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 Judgements for \( L\{\text{num str}\} \)

- \( s \text{ prg} \rightarrow a \text{ ast} \) Parse as a program
- \( s \text{ exp} \rightarrow a \text{ ast} \) Parse as an expression
- \( s \text{ trm} \rightarrow a \text{ ast} \) Parse as a term
- \( s \text{ fct} \rightarrow a \text{ ast} \) Parse as a factor
- \( s \text{ num} \rightarrow a \text{ ast} \) Parse as a number
- \( s \text{ lit} \rightarrow a \text{ ast} \) Parse as a literal
- \( s \text{ id} \rightarrow a \text{ ast} \) Parse as an identifier

Inductive Definition of Parsing for \( L\{\text{num str}\} \)

<table>
<thead>
<tr>
<th></th>
<th>s num</th>
<th>s str</th>
<th>s id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1 ast</td>
<td>a2 ast</td>
<td>a3 ast</td>
<td>a ast</td>
</tr>
</tbody>
</table>

Parsing

A parser is a function mapping concrete to abstract syntax.

- Use a well-structured grammar (in several senses).
  - At least unambiguous – or parsing won’t be a function!

- Analyze and decompose strings using one of several methods.
  - Top-down parsers (recursive descent): 630.
  - Bottom-up parsers (LR parsers): 610.
Inductive Definition of Parsing (cont.)

\[ s \text{ trm} \longrightarrow a \text{ ast} \]
\[ s \text{ exp} \longrightarrow a \text{ ast} \]
\[ s_1 \text{ trm} \longrightarrow s_2 \text{ ast} \]
\[ s_2 \text{ exp} \longrightarrow a \text{ ast} \]
\[ s_1 \text{ CAT} s_2 \text{ exp} \longrightarrow \text{cat}(a_1, a_2) \text{ ast} \]
\[ s \text{ exp} \longrightarrow a \text{ ast} \]
\[ s_1 \text{ id} \longrightarrow \text{id}[s] \text{ ast} \]
\[ s_2 \text{ exp} \longrightarrow a_2 \text{ ast} \]
\[ s_3 \text{ prg} \longrightarrow s_3 \text{ ast} \]
\[ \text{LET} s_1 \text{ BL} s_2 \text{ IN} s_3 \text{ prg} \longrightarrow \text{let}[s](a_1; a_2; a_3) \text{ ast} \]

Inductive Definition of Parsing

A successful parse implies that the token string must have been
derived according to an unambiguous grammar and the result is
a well-formed abstract syntax tree.

Theorem 1 (Parsing Judgements)
If \( s \text{ prg} \longrightarrow a \text{ ast} \), then \( s \text{ prg} \) and \( a \text{ ast} \) and similarly for the other
parsing judgements.

Proof: The proof proceeds by rule induction on the rules for
the inductive definition of parsing.

Inductive Definition of Parsing

Moreover, if a string is generated according to the rules of the
grammar, then it will parse to a unique abstract syntax tree.

Theorem 2 (Unique Parse)
If \( s \text{ prg} \), then there is a unique \( a \) such that \( s \text{ prg} \longrightarrow a \text{ ast} \), and
similarly for the other parsing judgements. That is, the pars-
ing judgements have mode \( \forall, \exists \) over the class of well-formed
strings and abstract syntax trees.

Proof: The proof proceeds by rule induction on the rules cor-
responding to the unambiguous grammar for \( L\{\text{num str}\} \).

Inductive Definition of Parsing

And conversely, an abstract syntax tree can be un-parsed into a
string that would parse back to the abstract syntax tree.

Theorem 3 (Pretty Printing)
If \( a \text{ ast} \), then there exists a (not necessarily unique) string \( s \) such
that \( s \text{ prg} \) and \( s \text{ prg} \longrightarrow a \text{ ast} \). That is, the parsing judgement has
mode \( \exists, \forall \).

Proof: The proof proceeds by rule induction on the rules cor-
responding to the unambiguous grammar for \( L\{\text{num str}\} \).

Parsing to Abstract Binding Trees

A parser is a function mapping concrete to abstract syntax.

- But abstract syntax trees don't account for scope and bind-
ing.

- So we revise parsing to produce abstract binding trees.

- Identifiers are no longer operators, but are translated into
variables by the parser.

Abstract Binding Tree Signature for \( L\{\text{num str}\} \)

\[ \text{ar}(\text{num}[n]) = () \]
\[ \text{ar}(\text{str}[s]) = () \]
\[ \text{ar}(\text{plus}) = (0, 0) \]
\[ \text{ar}(\text{times}) = (0, 0) \]
\[ \text{ar}(\text{Cat}) = (0, 0) \]
\[ \text{ar}(\text{len}) = (0) \]
\[ \text{ar}(\text{let}) = (0, 1) \]
Parsing to Abstract Binding Trees

First approach: revised parsing judgements are parametric hypothetical, rather than categorical. For example:

\[ \text{ID}[:x] \vdash \text{id} \rightarrow x \text{ abst} \]
\[ \text{LET} \underbrace{\text{id}}_{s_1} \text{ BE} \underbrace{\text{IN}}_{s_3} \text{ prg} \vdash \text{let}(s_2, x, s_3) \rightarrow \text{ abst} \]

Updating hypotheses in parsing rules records associations of bound identifiers and corresponding variables:

\[ \Gamma \vdash s_1 \text{ id} \rightarrow x \text{ abst} \]
\[ \Gamma \vdash s_2 \text{ exp} \rightarrow s_3 \text{ abst} \]
\[ \Gamma, s_1, s_2 \vdash s_3 \text{ prg} \rightarrow s_3 \text{ abst} \]

\[ \Gamma \vdash \text{LET} \underbrace{s_1} \text{ BE} \underbrace{s_3} \text{ prg} \vdash \text{let}(s_2, x, s_3) \rightarrow \text{ abst} \]

Representing a symbol table as a finite sequence of ordered pairs adds a few more judgements:

\[ \sigma \text{ symtab} \quad \text{well-formed symbol table} \]
\[ \sigma' = \sigma[s] \rightarrow x \quad \text{add new association} \]
\[ \sigma(\text{ID}[s]) = x \quad \text{lookup identifier} \]

Tabular Specification of Abstract and Concrete Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>(e)</td>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\sigma)</td>
<td>(\sigma)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(s)</td>
<td>(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{let}(e_1, e_2))</td>
<td>(\text{let}(e_1, e_2))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{plus}(e_1; e_2))</td>
<td>(\text{plus}(e_1; e_2))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{times}(e_1; e_2))</td>
<td>(\text{times}(e_1; e_2))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{cat}(e_1; e_2))</td>
<td>(\text{cat}(e_1; e_2))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{len}(e))</td>
<td>(\text{len}(e))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{let}(e_1, x, e_2))</td>
<td>(\text{let}(e_1, x, e_2))</td>
</tr>
</tbody>
</table>

Implicitly specifies two signatures, \(\Omega_{\text{sys}}\) and \(\Omega_{\text{exp}}\).

Parsing to Abstract Binding Trees

Unfortunately, this doesn’t quite work. For example, if two nested LETs bind the same name \(x\), we could get:

\[ \text{ID}[s] \rightarrow x_1 \text{ abst}, \ldots, \text{ID}[s] \rightarrow x_n \text{ abst} \vdash s \text{ prg} \rightarrow a \text{ abst} \]

So we’re forced to return to categorical judgements, including management of an explicit symbol table:

\[ s_1 \text{ id} \rightarrow x [s] \quad s_2 \text{ exp} \rightarrow s_3 \text{ abst} [s] \]
\[ \sigma' = \sigma[s_1 \rightarrow x] \quad s_3 \text{ prg} \rightarrow s_3 \text{ abst} [s'] \]

\[ \text{LET} s_1 \text{ BE} s_3 \text{ prg} \vdash \text{let}(s_2, x, s_3) \rightarrow [s'] \]

Informal Specifications of Syntax

We will usually specify abstract syntax informally. For example:

\[ \text{Type } r ::= \text{ num } \mid \text{ str } \]
\[ \text{Expr } e ::= x \mid \text{ num}[n] \mid \text{ str}[s] \mid \text{ plus}(e_1; e_2) \mid \text{ times}(e_1; e_2) \mid \text{ cat}(e_1; e_2) \mid \text{ len}(e) \mid \text{ let}(e_1, x, e_2) \]

This defines two judgements, \(\Gamma \vdash e\) and \(\text{c exp}\), asserting that two categories of objects are well formed. It also implicitly specifies two sets of operators and their corresponding arities and (binding tree) valences (i.e., signatures).

It is often useful to have a more readable representation, so we usually provide a table matching concrete to abstract syntax. For example:

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.