1 Lexing and Parsing

1.1 Core and Full Languages

The X³ Language Specification

Ross Tate

October 11, 2013

1 Lexing and Parsing

1.1 Core and Full Languages

\[ \nu_c := \text{variable/function/method names} \]

\[ \nu_e := \text{class/interface names} \]

\[ \nu_p := \text{type-parameter names} \]

\[ \nu_{vc} := \nu_v \mid \nu_c \]

- \[ \text{kind context} \quad \Theta ::= \nu_p, \ldots, \nu_p \]
- \[ \text{type context} \quad \Gamma ::= \nu_e : \tau, \ldots, \nu_e : \tau \]
- \[ \text{type} \quad \tau ::= \nu_p \mid \nu_c \langle \tau, \ldots, \tau \rangle \mid \tau \cap \tau \mid \top \mid \bot \]
- \[ \text{type scheme} \quad \sigma ::= \langle \Theta \rangle : \tau \]
- \[ \text{expression} \quad e ::= \nu_e | \nu_e \langle \tau, \ldots, \tau \rangle(e, \ldots, e) | e.\nu_e \langle \tau, \ldots, \tau \rangle(e, \ldots, e) | \langle e, \ldots, e \rangle | e + e \mid \text{true} \mid \text{false} \mid n \mid \text{"string"} \]
- \[ \text{statement} \quad s ::= \{ s \ldots s \} | \nu_e := e; | \text{if } (e) \text{ s else } s \mid \text{while } (e) \text{ s } | \text{for } (\nu_e \text{ in } e) \text{ s } | \text{return } e \mid \text{interface } \nu_c(\Theta) \text{ extends } \tau \{ \text{fun } \nu_e \sigma : \ldots \text{fun } \nu_e \sigma ; \text{fun } \nu_e \sigma s \ldots \text{fun } \nu_e \sigma s \} \]
- \[ \text{class} \quad c ::= \text{class } \nu_c(\Theta)(\Gamma) \text{ extends } \tau \{ s \ldots s \text{ super}(e, \ldots, e) ; \text{fun } \nu_e \sigma s \ldots \text{fun } \nu_e \sigma s \} \]
- \[ \text{program} \quad p ::= s | s \ldots s p | \text{fun } \nu_e \sigma s \ldots \text{fun } \nu_e \sigma s p | i p | c p \]
- \[ \text{function context} \quad \Delta ::= \emptyset | \Delta, \nu_e \sigma \]
- \[ \text{class context} \quad \Psi ::= \emptyset | \Psi, \text{interface } \nu_c(\Theta) \text{ extends } \tau \{ \Delta; \nu_v, \ldots, \nu_v \} | \Psi, \text{class } \nu_c(\Theta) \text{ extends } \tau \{ \Delta \} \]

Figure 1: Cubex Core Language Grammar, which gives the grammar for the Cubex core language, along with definitions for the formal constructs \( \Delta \) and \( \Psi \), which are not part of the language per se but are useful in our formalization of the type system. Note that \( p \) stands for “program” and defines a syntactically-valid program in the Cubex core language. Note also that lists, represented with an elipsis, may consist of zero, one, or more elements. So \( a, \ldots, a \) may be the empty string, \( a \), or some list of \( a \)s separated by commas.

We distinguish between the Cubex core language, which is specified by the grammar in Figure 1.1, and the full language. The full language differs from the core language in a number of ways:

- It includes the following unary and binary operators, listed in order of precedence, which are (except for ++) short for method calls on arbitrary classes (their signatures are those that are given for the built-in classes at the end of this document). All operators are left-associative.
  1. Unary prefixes \(-\) and \(!\) short for negative and negate respectively.
  2. Multiplicative operators \(*\), \(/\), and \(\%\) short for times, divide, and modulo respectively.
  3. Additive \(+\) and \(-\) short for plus and minus respectively.
  4. Range operators \(\ldots, <, <, <=, <, \ldots\), and \(<\), with the last two being unary suffixes; the binary operators are shorthand for through using additional Boolean parameters to indicate whether to include the lower and/or upper bounds; the unary suffixes are shorthand for onwards using an additional Boolean parameter to indicate inclusiveness.
  5. The binary operator \(\oplus\), (actually part of the core language), which implements appending of iterables, denoted in the grammar by \(\top\).
  6. Inequality operators \(<\), \(<\), \(>=\), and \(>\) short for lessThan with an additional Boolean parameter indicating strictness, and with order reversed for \(\geq\) and \(\leq\).
  7. Equational operators \(==\) and \(!=\), short for equals and equals followed by negate.
  8. Boolean operators \&\& short for and followed by | short for or.
- Characters in the core language which have no obvious ASCII equivalent are represented in the full language in ASCII as follows:
- Thing for \( \top \) and Nothing for \( \bot \).
- \& for \( \land \) and | for \( \lor \).
- \& for \( \cap \).
- < for \( ( \) and > for \( ) \).

- Expressions (denoted \( e \) in the grammar) may be surrounded by parentheses.
- If there are no type parameters (i.e. when \( \langle \tau, \ldots, \tau \rangle \) is the empty list), a class/interface/function specification can drop the \( \langle \rangle \).
- When calling a function/method/constructor with no type parameters, the \( \langle \rangle \) may be dropped.
- If the else case is \( \{ \} \), it can be dropped.
- In the full language, unimplemented and implemented methods of interfaces may mixed together, unlike in the core language where all of the latter must follow all of the former.
- If a class or interface just extends \( \top \), the extends clause can be omitted.
- If the call to super has no arguments, it can be omitted.
- If a function implementation is simply return of some expression, the return can be abbreviated to \( = \). For example,
  ```
  fun foo(y : Integer) : Integer return y * 2;
  ```
  may be abbreviated to
  ```
  fun foo(y : Integer) : Integer = y * 2;
  ```

  Finally, there is no name hiding. If a judgement requires binding a name twice, that judgement is ill formed and does not hold. However, all the different kinds of context are different namespaces. So a name can be bound both in \( \Delta \) and \( \Gamma \). To clarify, \( \Gamma \) and \( \hat{\Gamma} \) use the same namespace, so a variable must not be bound in both.

1.2 Name, Keywords, Literals, etc.

The grammar uses \( \nu \) with subscripts to denote both various kinds of names. We enforce a distinction between these kinds. Variables and function/method names (\( \nu_v \)) are a lower-case letter followed by letters, numbers, and/or underscores. Class/interface names (\( \nu_c \)) are an upper-case letter followed by one or more letters, numbers, and/or underscores. Type parameters (\( \nu_p \)) are an upper-case letter alone. In the formalism we refer to all of these collectively as just \( \nu \).

Key words (i.e. any words receiving special treatment in the core language) cannot be used as names or variables.

String literals (denoted in the grammar by "string") have no escaping and cannot contain white space besides the space character. Integer literals (denoted in the grammar by \( n \)) are in decimal only.

Finally, note that the grammar contains various other syntactic artifacts (semicolons, brackets, etc.) whose only legal uses in the language (outside of comments and string literals) are fully specified by their role in the grammar.

1.3 Whitespace and Comments

Unless it occurs inside a string literal, white space is disregarded except for its role of separating names and other tokens (e.g. " := " is not the same as " := ", and only the latter can be used in a variable assignment). Unless it occurs inside a string literal, # starts a comment that extends to the end of the line. Unless it occurs inside a string literal, \( ^ \) starts a comment that extends to the matching \( ^ \), where these comments can be nested inside each other. Comments are disregarded except for their role of separating names and other tokens.
2 Validating

The following formalizes when a program is valid, incorporating type validity, type checking, and class/interface validation. Judgements take the form “(context) ⊢ (property)”. Ψ indicates the classes and interfaces in scope, what they directly inherit, and what their methods are. Θ indicates the type parameters in scope. Γ indicates the immutable variable names in scope and what their type is. ˆΓ typically indicates the mutable variable names in scope and what their type is. Γ typically indicates the mutable variable names in scope and what their type is. The following is a summary of the judgements used in this formalism and where their definitions can be found.

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ψ</td>
<td>Θ</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
<tr>
<td>Ψ</td>
<td>⊢</td>
</tr>
</tbody>
</table>

Figure 2: Subtyping
Figure 3: Method Lookup

Figure 4: Type Validity

Figure 5: Type Checking Expressions
Figure 6: Type Checking Statements

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow s : \Gamma$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \mathcal{T}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \mathcal{T}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash s : \hat{\Gamma}$$

for all $i$, $\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma}_{i-1} \vdash s_i : \hat{\Gamma}_i$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma}_{0} \vdash \{ s_1 \ldots s_n \} : \hat{\Gamma}_n$$

Figure 7: Type Checking Returns

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash e : \mathcal{T}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \nu \Rightarrow e : \mathcal{T}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash e : \text{Boolean}()$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash e : \text{Iterable}(\mathcal{T})$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \text{for} (\nu \text{ in } e) s : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash b : s : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \text{true} : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \text{false} : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash e : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \nu \Rightarrow \text{false} : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \text{false} : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash (e) s_1 \text{ else } s_2 : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \text{while} (e) s : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \text{for} (\nu \text{ in } e) s : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash e : \mathcal{T}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \nu \Rightarrow e : \mathcal{T}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash e : \text{Iterable}(\mathcal{T})$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash e : \text{for} (\nu \text{ in } e) s : \hat{\Gamma}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash c : \mathcal{T}$$

$$\Psi \Rightarrow \Delta \Rightarrow \Gamma \Rightarrow \hat{\Gamma} \vdash \text{true} \Rightarrow c : ; \hat{\Gamma}$$
Figure 8: Method-Context Lookup

Figure 9: Method-Implemented Check

Figure 10: Class and Interface Checking
\[ \Psi \mid \Delta \mid \Gamma \vdash p \vdash p \]

\[
\frac{\Psi \mid \varnothing \mid \Delta \mid \Gamma \vdash \text{true}_{\text{Iterable(String)}} \ s : \hat{\Gamma}}{\Psi \mid \Delta \mid \Gamma \vdash s}
\]

\[ \hat{\Gamma}_0 = \varnothing \quad \text{for all } i, \quad \Psi \mid \varnothing \mid \Delta \mid \Gamma \mid \hat{\Gamma}_{i-1} \vdash \text{true}_{\text{Iterable(String)}} \ s_i : \hat{\Gamma}_i \quad \Psi \mid \Delta \mid \Gamma, \hat{\Gamma}_n \vdash p \]

\[ \Delta' = \Delta, \nu_1(\Theta_1)(\Gamma_1) : \tau_1, \ldots, \nu_n(\Theta_n)(\Gamma_n) : \tau_n \quad \text{for all } i, \quad \Psi \mid \Theta_i \vdash \Gamma_i \quad \Psi \mid \Theta_i \vdash \tau_i \quad \Psi \mid \Theta_i \mid \Delta' \mid \Gamma \mid \Gamma_i \vdash \text{true}_s \ s_i : \hat{\Gamma}_i \]

\[ \Psi \mid \Delta \mid \Gamma \vdash \text{fun} \ \nu_1(\Theta_1)(\Gamma_1) : \tau_1 \ s_1 \ldots \text{fun} \ \nu_n(\Theta_n)(\Gamma_n) : \tau_n \ s_n \ p \]

\[ \Psi \mid \Delta \mid \Gamma \vdash i : \Psi' \quad \Psi, \Psi' \mid \Delta \mid \Gamma \vdash p \quad \Psi \mid \Delta \mid \Gamma \vdash c : \Psi' \quad \Psi, \Psi' \mid \Delta' \mid \Gamma \vdash p \quad \Psi \mid \Delta \mid \Gamma \vdash c \ p \]

\[ \Psi_0 = \quad \text{class } \text{Iterable}(E) \text{ extends } T \{ \}, \]

\[ \text{class } \text{Boolean}() \text{ extends } T \{ \]
\[ \quad \text{negate}() : \text{Boolean}(); \]
\[ \quad \text{and}(\text{that} : \text{Boolean}()) : \text{Boolean}(); \]
\[ \quad \text{or}(\text{that} : \text{Boolean}()) : \text{Boolean}(); \]
\[ \quad \text{through}(\text{upper} : \text{Boolean}(), \text{includeLower} : \text{Boolean}(), \text{includeUpper} : \text{Boolean}()) : \text{Iterable}(!\text{Boolean}()); \]
\[ \quad \text{onwards}(\text{inclusive} : \text{Boolean}()) : \text{Iterable}(!\text{Boolean}()); \]
\[ \quad \text{lessThan}(\text{that} : \text{Boolean}(), \text{strict} : \text{Boolean}()) : \text{Boolean}(); \]
\[ \quad \text{equals}(\text{that} : \text{Boolean}()) : \text{Boolean}(); \]
\[ \}, \quad \text{class } \text{Integer}() \text{ extends } T \{ \]
\[ \quad \text{negative}() : \text{Integer}(); \]
\[ \quad \text{times}(\text{factor} : \text{Integer}()) : \text{Integer}(); \]
\[ \quad \text{divide}(\text{divisor} : \text{Integer}()) : \text{Iterable}(!\text{Integer}()); \]
\[ \quad \text{modulo}(\text{modulus} : \text{Integer}()) : \text{Iterable}(!\text{Integer}()); \]
\[ \quad \text{plus}(\text{summand} : \text{Integer}()) : \text{Integer}(); \]
\[ \quad \text{minus}(\text{subtrahend} : \text{Integer}()) : \text{Integer}(); \]
\[ \quad \text{through}(\text{upper} : \text{Integer}(), \text{includeLower} : \text{Boolean}(), \text{includeUpper} : \text{Boolean}()) : \text{Iterable}(!\text{Integer}()); \]
\[ \quad \text{onwards}(\text{inclusive} : \text{Boolean}()) : \text{Iterable}(!\text{Integer}()); \]
\[ \quad \text{lessThan}(\text{that} : \text{Integer}(), \text{strict} : \text{Boolean}()) : \text{Boolean}(); \]
\[ \quad \text{equals}(\text{that} : \text{Integer}()) : \text{Boolean}(); \]
\[ \}, \quad \text{class } \text{Character}() \text{ extends } T \{ \]
\[ \quad \text{unicode}() : \text{Integer}(); \]
\[ \quad \text{equals}(\text{that} : \text{Character}()) : \text{Boolean}(); \]
\[ \}, \quad \text{class } \text{String}() \text{ extends } \text{Iterable}(!\text{Character}()) \{ \text{equals}(\text{that} : \text{String}()) : \text{Boolean}(); \} \]

\[ \Delta_0 = \quad \text{character}(\text{unicode} : \text{Integer}()) : \text{Character}(), \text{string}()\{\text{characters} : \text{Iterable}(!\text{Character})\} : \text{String}() \]

\[ \Gamma_0 = \quad \text{input} : \text{Iterable}(!\text{String}) \]

\[ \Psi_0 \mid \Delta_0 \mid \Gamma_0 \vdash p \]

\[ \vdash p \]

Figure 11: Program Checking