April 2008

Concise Concrete Syntax

Stephen Tse
University of Pennsylvania

Stephan A. Zdancewic
University of Pennsylvania, stevez@cis.upenn.edu

Concise Concrete Syntax

Abstract
We introduce a notion of *ordered context-free grammars (OCFGs) with datatype tags* to concisely specify grammars of programming languages. Our work is an extension of syntax definition formalism (SDF) and concrete datatypes that automate scanning, parsing, and syntax tree construction. But OCFGs also capture associativity and precedence at the level of *production rules* instead of *lexical tokens* such that a concrete syntax grammar is succinct enough be an abstract syntax definition.

By *expanding* and *re-indexing* grammar symbols, OCFGs can be translated to grammars for standard **lex** and **yacc** such that existing and efficient parsing infrastructures can be reused.

We have implemented a Java 5 compiler frontend with OCFGs. The complete grammar for such a realistic language fits comfortably in two pages of this paper, showing the practicality of our formalism.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-11.

Concise Concrete Syntax

Stephen Tse Steve Zdancewic

University of Pennsylvania

Abstract

We introduce a notion of *ordered context-free grammars (OCFGs) with datatype tags* to concisely specify grammars of programming languages. Our work is an extension of syntax definition formalism (SDF) and concrete datatypes that automate scanning, parsing, and syntax tree construction. But OCFGs also capture associativity and precedence at the level of *production rules* instead of *lexical tokens* such that a concrete syntax grammar is succinct enough be an abstract syntax definition.

By *expanding* and *re-indexing* grammar symbols, OCFGs can be translated to grammars for standard `lex` and `yacc` such that existing and efficient parsing infrastructures can be reused.

We have implemented a Java 5 compiler frontend with OCFGs. The complete grammar for such a realistic language fits comfortably in two pages of this paper, showing the practicality of our formalism.

1 Introduction

Algebraic datatypes have been successfully used for formal specifications of abstract syntax and semantics of programming languages. The concrete syntax of a language, however, is often left out of the formal specification and relies on adhoc interfaces between the scanner, the parser, and the datatypes of the abstract syntax. This paper shows that the formal specification of concrete syntax can also be concisely integrated with the usual notation of datatypes.

As an example, consider the following *concrete syntax* (on the left) and *abstract syntax* (on the right, in OCaml [10]) of the lambda calculus [12]:

e	::=	$\lambda x.e$	functions	`type e = Fun of x * e`
	|	$e\ e$	applications	`| App of e * e`
	|	x	variables	`| Var of x`

We represent variables as strings x. Note that following details about the concrete syntax are missing above: (1) functions associate to the right, (2) applications associate to the left, (3) parentheses delimit terms, and (4) variables and parenthesized terms have the highest precedence. In fact, incorporating these constraints is the motivation behind our work.

```
e: e1 { $1 }

e1: e2 { $1 }
  | Kfun x_plus_comma Dot e1 { Fun ($2,$4) }

e2: e3 { $1 }
  | e2 e3 { App ($1,$2) }

e3: x { Var $1 }
  | Lparen e Rparen { Atom $2 }

x_plus_comma: x { [$1] }
  | x Comma x_plus_comma { $1 :: $3 }
```

```
type e =
  | Fun of x list * e
  | App of e * e
  | Var of x
  | Atom of e

rule =
  | "("   { Lparen }
  | ")"   { Rparen }
  | ","   { Comma }
  | "."   { Dot }
  | "fun" { Kfun }
```

Fig. 1. Translation for the OCFG for lambda calculus: `ocamlyacc` parser definition (left), OCaml type definition (top right), and `ocamllex` scanner definition (bottom right). Note the expansion and re-indexing of e in the parser definition.

Here we introduce *ordered context-free grammars with datatype tags*, or OCFG for short, by the example. The following is an OCFG for the lambda calculus (ASCII source code on the left and LaTeX rendering on the right):

```
e: > Fun: 'fun' x ++ ',' '.' e
   < App: e e
   = Var: x
   | Atom: '(' e ')'
```

$$
\begin{array}{lcll}
e & ::> & \texttt{fun}\ x^{+}_{,}\ \texttt{.}\ e & \text{(Fun)} \\
 & ::< & e\ e & \text{(App)} \\
 & ::= & x & \text{(Var)} \\
 & | & \texttt{(}\ e\ \texttt{)} & \text{(Atom)}
\end{array}
$$

The full notation of OCFGs will be formally introduced in Section 3.1. Here, the occurrence of more than one binding variable makes this example a bit more interesting. Incorporating built-in repetitions (similar to the extended BNF notation), like $[x^{+}_{,}]$ above, not only simplifies the notation but is also significant for the correctness of our translation (see Section 3.4).

We translate this extended notion of context-free grammars and datatypes to definitions for standard scanners, parsers, and ordinary datatypes. This allows us to reuse the existing and efficient parsing infrastructures.

Expansion and re-indexing Fig. 1 shows the translation for the lambda calculus. The translation is done in a functional style (in OCaml [10] notation); the main idea is to *expand* and *re-index* grammar symbols to impose associativity and precedence constraints. For example, expression e is expanded into e_1, e_2 and e_3; the case for functions $[\texttt{fun}\ x^{+}_{,}\ \texttt{.}\ e]$ is re-indexed to be $[\texttt{fun}\ x^{+}_{,}\ \texttt{.}\ e_1]$, and the case for applications $[e\ e]$ is re-indexed to $[e_2\ e_3]$, while the cases for variables x and atoms $[\texttt{(}\ e\ \texttt{)}]$ stay the same.

Programmers can easily understand the mapping from the OCFG above to the abstract syntax in the top right of Figure 1: ignore all literal tokens and treat repetitions as lists. The example above shows that the concrete syntax grammar of a language can be concise enough to be an abstract syntax

definition (except for some minor artifact like the extra `Atom` rule for grouping).

We will formally show the set of translation rules for expansion and re-indexing in Section 3.3. The rules are mostly straightforward but there are some special cases. Another challenge, which we address in Section 3.5, is code generation for error diagnosis such that parsing errors are reported uniformly in terms of OCFGs. Our tool also generates pretty-printers and other generics operations (such as maps and iterators), but we will not discuss them here.

Contributions Some ideas behind our notation and translation, such as scanner-less parsing in the syntax definition formalism (SDF) [15] and concrete datatypes [1], have been individually studied before (see Section 4 for more related work). Our new contributions are:

(i) for *compiler writers*, an extension to parsing grammars with *simple semantics* for connecting different compiler phases, including scanning, parsing, error diagnoses, and syntax tree constructions;

(ii) for *language designers*, an extension to datatypes with a *concise notation* for specifying both the concrete and abstract syntax of a language, allowing fast prototyping of compiler frontends.

Specifically, our notation allows associativity and precedence to be specified per *production rule* instead of per *lexical token*. We also pay special attention to the semantics of repetitions and error diagnosis, to avoid parser conflicts during re-indexing.

Our formalism allows language designers to focus on high-level concepts like associativity and precedence instead of symbol expansion and re-indexing. This greatly improves the readability of a grammar: a concrete syntax specification is now concise enough to be an abstract syntax definition.

We have implemented the translation for OCaml (using its LALR(1) parser `ocamlyacc`) and used OCFGs for a few interpreter projects [18]. Our formalism and the translation here are not specific to LR parsing or functional languages; we have also implemented a Java 5 frontend [17] using Generalized LR parsing [16,14,8] (see Section 2).

Outline The rest of the paper is organized as follows. Section 2 demonstrates the practicality of OCFG by specifying a full grammar for Java 5. Section 3 formally defines OCFG and shows their translations for scanning, parsing, type definitions, and error diagnosis. Section 4 discusses related work, and finally Section 5 concludes the paper.

2 Applications

Before diving into formal definitions and translation, this section demonstrates the practicality of OCFG by specifying a grammar for Java 5.

Fig. 2 and Fig. 3 show that the concrete syntax of such a realistic language

cunit ::= *package*? *import** *decl**
package ::= `package` *name* `;`
ty ::= *ty dimen*+ (Tarray)
::= *ty targs* (Tinst)
::= *name* (Tname)
| `boolean` (Tbool)
| `byte` (Tbyte)
| `char` (Tchar)
| `double` (Tdouble)
| `float` (Tfloat)
| `int` (Tint)
| `long` (Tlong)
| `short` (Tshort)
| `void` (Tvoid)
wty ::= *ty* (WildType)
| `?` (WildCard)
| `?` `extends` *ty* (WildExtends)
| `?` `super` *ty* (WildSuper)
tparams ::= `<` *tparam*+, `>`
tparam ::= *id extend*? *tinter**
tinter ::= `&` *ty*
targs ::= `<` *wty*+, `>`
ifelse ::= `else` *stmt*
block ::= `{` *stmt** `}`
extend ::= `extends` *ty*
extends ::= `extends` *ty*+,
bodies ::= `{` *body** `}`
dimen ::= `[ ]`
size ::= `[` *exp* `]`
default ::= `default` *exp*
arg ::= `(` *exp**, `)`
labelexp ::= *id* `=` *exp*
name ::= *id*+.
all ::= `. *`
vdecl ::= *id dimen** *vinit*?
vinit ::= `=` *exp*
exps ::> *exp* `,` *exps* (ExpsCons)
| *exp* (ExpsSome)
| (ExpsNil)
mod ::= *basicmod* (BasicMod)
| `@` *name* (AnnotMark)
| `@` *name* `(` *exp* `)` (AnnotSingle)
| `@` *name* `(` *labelexp*+, `)` (AnnotMore)

exp ::= `{` *exps* `}` (Array)
::> *exp assign exp* (Assign)
::> *exp* `?` *exp* `:` *exp* (Cond)
::< *exp* `||` *exp* (Cor)
::< *exp* `&&` *exp* (Cand)
::< *exp* `|` *exp* (Or)
::< *exp* `^` *exp* (Xor)
::< *exp* `&` *exp* (And)
::< *exp* `==` *exp* (Eq)
| *exp* `!=` *exp* (Ne)
::< *exp* `<` *exp* (Lt)
| *exp* `>` *exp* (Gt)
| *exp* `<=` *exp* (Le)
| *exp* `>=` *exp* (Ge)
| *exp* `instanceof` *ty* (Instof)
::< *exp* `<<` *exp* (Shl)
| *exp* `>>` *exp* (Shr)
| *exp* `>>>` *exp* (Ushr)
::< *exp* `+` *exp* (Add)
| *exp* `-` *exp* (Sub)
::< *exp* `*` *exp* (Mul)
| *exp* `/` *exp* (Div)
| *exp* `%` *exp* (Rem)
::> `(` *ty* `)` *exp* (Cast)
| `++` *exp* (PreIncr)
| `--` *exp* (PreDecr)
| `+` *exp* (Pos)
| `-` *exp* (Neg)
| `!` *exp* (Not)
| `~` *exp* (Inv)
::< *exp* `[` *exp* `]` (Offset)
| *exp* `.` *exp* (Dot)
| *ty* `.` `class` (ExpClass)
| *exp arg* (Invoke)
| *exp* `++` (PostIncr)
| *exp* `--` (PostDecr)
::= *targs exp* (Inst)
::= `this` (This)
| `super` (Super)
| *literal* (Literal)
| *id* (Ident)
| `(` *exp* `)` (Atom)
| `new` *ty size*+ *dimen** (NewArr1)
| `new` *ty* `{` *exps* `}` (NewArr2)
| `new` *name targs*? *arg bodies*? (NewObj)

Fig. 2. An OCFG for Java 5 (page 1/2): the only missing definitions are *literal* (*char*, *float*, *integer*, *string*), *basicmod* (`abstract`, `final`, `native`, `private`, `protected`, `public`, `static`, `strictf`, `synchronized`, `transient`, `volatile`), and *assign* (`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`).

can fit comfortably in two pages of space. (Appendix B contains the full ASCII source code of the grammar for our tool.) We have implemented a compiler frontend using this grammar and tested against *all* files in the JDK version 1.5, covering the new features like generics, enumerations, and annotations.

More importantly, the generated type definitions are concise enough to be used directly for later stages of the compiler such as typechecking. For example, there is only *one* kind of expression in our grammar, compared to 22 kinds of expressions in the Java Language Specification (JLS):

> *AdditiveExp*, *AndExp*, *ArrayCreationExp*, *AssignmentExp*, *CastExp*, *ClassInstanceCreationExp*, *ConditionalAndExp*, *ConditionalExp*, *ConditionalOrExp*, *EqualityExp*, *ExclusiveOrExp*, *Exp*, *InclusiveOrExp*, *MultiplicativeExp*, *PostDecrementExp*, *PostIncrementExp*,*PostfixExp*, *PreDecrementExp*, *PreIncrementExp*, *RelationalExp*, *ShiftExp*, *UnaryExp*

Essentially, the JLS grammar does expansion and re-indexing manually, while our translation does them automatically. This automation makes our OCFG, unlike the JLS grammar, resemble an abstract syntax definition.

Our implementation is based on Generalized LR parsing [16] (instead of LALR) and thus requires the following disambiguation filters [19]. The details of the parsing technique and the filter construction [1] are beyond the scope of this paper, but the fact that these filters are specified in term of the datatype tags of the OCFG further justifies our formalism.

```
1    If(_,If(_,_,None),Some _)
2    If(_,If(_,_,Some(If(_,_,None))),Some _)
3    If(_,If(_,_,Some(If(_,_,Some(If(_,_,None))))),Some _)
4    Invoke(Atom _,[_])                (* (C)(x); *)
5    Cast(Tname _,Pos _)               (* (int)(x) + y; *)
6    Cast(Tname _,Neg _)               (* (int)(x) - y; *)
7    Offset(NewArr1 _,_)               (* new int[0][0]; *)
8    Assign(Gt _,_,_)                  (* C<D> x = y; *)
9    Assign(Lt _,_,_)                  (* C<D<E>> x = y; *)
10   Exp(Gt _)                         (* C<D> x; *)
11   Exp(Lt _)                         (* C<D<E>> x; *)
12   ForExp(Gt _::_,_,_,_)             (* for (C<D> x;;); *)
13   ForExp(Lt _::_,_,_,_)             (* for (C<D<E>> x;;); *)
```

Our implementation provides predefined rules for common lexical classes in programming languages: whitespace, newlines, identifiers x, literals for

[1] Each filter is an OCaml pattern, matching against the abstract syntax tree. Filters 1-3 are for the dangling-else statements. For example, the source [`if (e1) if (e2) s1 else s2;`] can be parsed as `If(e1,If(e2,s1,Some s2),None)` or `If(e1,If(e2,s1,None),Some s2)`, where `None` and `Some` are datatype constructors for the optional datatype. Filter 1 will discard the second tree. Dangling-else statements require an infinite number of filters; our tool supports only finite filters but it works well in practice. Filters 4-6 disambiguate casting from parenthesized expressions, while Filter 7 will discard expressions like `new int[0][0]` parsed as `(new int[0])[0]`. The rest are for instantiations of generic classes.

decl	::=	*mod** `class` *id tparams*$^?$ *extend*$^?$ *implements*$^?$ *bodies*	(Class)
	|	*mod** `interface` *id tparams*$^?$ *extends*$^?$ *bodies*	(Iface)
	|	*mod** `enum` *id implements*$^?$ `{` *enumbody* `}`	(Enum)
	|	*mod** `@ interface` *id bodies*	(Annot)
	|	`;`	(Empty)
body	::=	`static`$^?$ *block*	(Init)
	|	*decl*	(Inner)
	|	*mod** *tparams*$^?$ *id signat block*	(Ctor)
	|	*mod** *tparams*$^?$ *ty id signat* `;`	(MethodNone)
	|	*mod** *tparams*$^?$ *ty id signat block*	(MethodSome)
	|	*mod** *ty vdecl*$^+_,$ `;`	(Field)
enumbody	::=	*enumconst*$^+_,$ `,`$^?$ `;`$^?$	(EnumNone)
	|	*enumconst*$^+_,$ `,`$^?$ `;` *body*$^+$	(EnumSome)
enumconst	::=	*mod** *id arg*$^?$ *bodies*$^?$	
import	::=	`import static`$^?$ *name all*$^?$ `;`	
implements	::=	`implements` *ty*$^+_,$	
signat	::=	`(` *formals* `)` *default*$^?$ *dimen** *throws*$^?$	
stmt	::>	`for (` *mod** *ty vdecl*$^+_,$ `;` *exp*$^?$ `;` *exp*$^*_,$ `)` *stmt*	(ForDecl)
	|	`for (` *mod** *ty id* `:` *exp* `)` *stmt*	(ForEach)
	|	`for (` *exp*$^*_,$ `;` *exp*$^?$ `;` *exp*$^*_,$ `)` *stmt*	(ForExp)
	|	`while (` *exp* `)` *stmt*	(While)
	|	`if (` *exp* `)` *stmt ifelse*$^?$	(If)
	|	*id* `:` *stmt*	(Label)
	::=	`assert` *exp* `;`	(AssertNone)
	|	`assert` *exp* `:` *exp* `;`	(AssertSome)
	|	*block*	(Block)
	|	`break` *id*$^?$ `;`	(Break)
	|	`case` *exp* `:`	(Case)
	|	`continue` *id*$^?$ `;`	(Continue)
	|	*mod** *ty vdecl*$^+_,$ `;`	(Vdecl)
	|	`default :`	(Default)
	|	`do` *stmt* `while (` *exp* `) ;`	(Do)
	|	*exp* `;`	(Exp)
	|	*decl*	(Local)
	|	`return` *exp*$^?$ `;`	(Return)
	|	`switch (` *exp* `)` *block*	(Switch)
	|	`synchronized (` *exp* `)` *block*	(Sync)
	|	`throw` *exp* `;`	(Throw)
	|	`try` *block catch** *finally*$^?$	(Try)
catch	::=	`catch (` *mod** *ty id* `)` *block*	
finally	::=	`finally` *block*	
throws	::=	`throws` *name*$^+_,$	
formals	::>	*mod** *ty id dimen** `,` *formals*	(ArgMore)
	::=	*mod** *ty id dimen**	(ArgFix)
	|	*mod** *ty* `...` *id dimen**	(ArgVar)
	|		(ArgNone)

Fig. 3. An OCFG for Java 5 (page 2/2). Appendix B contains the ASCII source.

$$
\begin{array}{rl}
\text{Rules} & r ::= \epsilon \mid r,\ x = g \mid r,\ x = u \\
\text{Groups} & g ::= \epsilon \mid g\ c^{=} \mid g\ c^{>} \mid g\ c^{<} \\
\text{Cases} & c ::= \epsilon \mid c,\ X : u \\
\text{Items} & u ::= \epsilon \mid u\ v \\
\text{Item} & v ::= q \mid x \mid q^{?} \mid x^{?} \mid x^{*} \mid x^{+} \mid x_q^{*} \mid x_q^{+}
\end{array}
$$

Fig. 4. Formal grammar of OCFGs.

characters (*char*), integers (*int*), floating-point numbers (*float*), and strings (*string*). These predefined rules are sufficient for common languages, but in the future we plan to add a general mechanism [15] for defining lexical classes.

We have also simplified the grammars by accepting larger languages in the lexical stage: for example, (1) `case`, `break`, `catch`, and `finally` statements can occur syntactically outside `switch`, `for`, `while`, `do`, and `try`; (2) `super`, `this` and array initializers are general expressions; and, (3) any modifiers can be applied to declarations for classes, interfaces, methods and fields.

These fine-grained restrictions can easily be enforced during later stages of the compiler such as typechecking. Traditionally, concrete syntax has been specified in obscure ways; we argue that any possible measure should be taken to keep these specifications elegant and comprehensible to programmers.

3 Ordered context-free grammars with datatype tags

This section formally defines OCFGs and shows their translations for scanning, parsing, type definitions, and error diagnosis. Besides associativity and precedence, another feature of OCFGs are a restricted form of repetitions for options and lists. We will show that having repetitions not only simplifies the notation but is also significant in the semantics.

3.1 Definitions and notations

An *ordinary datatype* consists of a list of product types with constructor names. An *OCFG* supplies also information about the concrete syntax so that lexical analysis can be automated. The intuition is to order the list of datatype alternatives into groups by precedence and to annotate such groups with associativity. Fig. 4 shows the grammar of OCFGs.

A *rule* r is a list of precedence groups $x = g$, or an alias of items $x = u$. A *group* of cases is annotated with its associativity: left-associative $<$, right-associative $>$, or non-associative $=$. Groups inside a rule are ordered with increasing precedence (following `yacc`'s convention), while cases inside a group are unordered and have the same precedence. Both $x = g$ and $x = u$ bind constructs to identifiers in a mutually recursive fashion (again following `yacc`'s convention). The first rule is taken be the starting rule for parsing.

A *case* c is tagged with a constructor name X and a list of *items* u. We write ϵ for an empty list, x for lower-case identifiers, X for upper-case identifiers,

and q for quoted literals. We typeset literals in `teletype font` for readability.

For example, the lambda calculus introduced in Section 1 can be written in the following notation; this notation is designed mainly to facilitate the translation definitions in this section.

$$e \quad = \quad (\text{Fun} : \texttt{fun}\ x^{+}_{\texttt{,}}\ \texttt{.}\ e)^{>} \quad (\text{App} : e\ e)^{<} \quad (\text{Var} : x, \quad \text{Atom} : \texttt{(}\ e\ \texttt{)})^{=}$$

The keyword `fun`, the comma and the parentheses are quoted literals in this example. The object-level identifier x is a predefined rule representing the set of identifiers in the object-level grammar. Note that we slightly abuse the notation by writing x for both meta-level and object-level identifiers.

Compared to an ordinary datatype, which has a tag with a list of types in the definition (such as [`Fun of x * e`]), an OCFG has a tag with a list of *concrete tokens* and *rule identifiers* (such as [Fun : $\texttt{fun}\ x^{+}_{\texttt{,}}\ \texttt{.}\ e$]). The grammar also supports these forms of repetitions (similar to the extended BNF notation):

(i) item $q^{?}$ for zero or one occurrence of q,
(ii) item $x^{?}$ for zero or one occurrence of x,
(iii) item x^{*} for zero or more occurrences of x,
(iv) item x^{+} for one or more occurrences of x,
(v) item x^{*}_{q} for zero or more occurrences of x separated by q, and,
(vi) item x^{+}_{q} for one or more occurrences of x separated by q.

Instead of allowing repetitions for general items such as $v^{?}$, v^{*}, v^{+}, v^{*}_{v} and v^{+}_{v}, we restrict these forms to only literals and identifiers. We argue that this restricted form already captures most syntax idioms of programming languages. More importantly, this restriction substantially simplifies the translation (to be shown in the next subsections) and improves the readability of datatype definitions (see Fig. 2 and Fig. 3).

3.2 Translation for types and scanning

The semantics of OCFGs is translated to standard constructs provided by the `lex` scanner, the `yacc` parser, and the ML type definitions, by families of translation functions. The first two families of functions are:

(i) $[\![r]\!]_{\tau}$, $[\![g]\!]_{\tau}$, $[\![c]\!]_{\tau}$, $[\![u]\!]_{\tau}$, $[\![v]\!]_{\tau}$ denote the translations for *type* definitions,
(ii) $[\![r]\!]_{\sigma}$, $[\![g]\!]_{\sigma}$, $[\![c]\!]_{\sigma}$, $[\![u]\!]_{\sigma}$, $[\![v]\!]_{\sigma}$ denote the translations for *scanner* definitions.

The translation for parsing and error diagnosis are more involved and will be discussed in the following subsections. The translations for type and scanner definitions are straightforward. Ignoring associativity and precedence, the functions $[\![\cdot]\!]_{\tau}$ and $[\![\cdot]\!]_{\sigma}$ simply recurse and construct type definitions according

to the items in the rule. For brevity, only $[\![v]\!]_\tau$ is presented here:

$$\begin{array}{ll} [\![q]\!]_\tau = \epsilon & [\![x]\!]_\tau = x \\ [\![q^?]\!]_\tau = \texttt{bool} & [\![x^?]\!]_\tau = x\ \texttt{option} \\ [\![x^*]\!]_\tau = x\ \texttt{list} & [\![x^+]\!]_\tau = x\ \texttt{list} \\ [\![x^*_q]\!]_\tau = x\ \texttt{list} & [\![x^+_q]\!]_\tau = x\ \texttt{list} \end{array}$$

Following the practice in ML, the constructed types do not distinguish non-empty lists (x^+ for one or more occurrences) from possibly-empty lists (x^* for zero or more occurrence). A literal q can be recovered directly from the OCFG and hence requires no representation in the abstract datatype. Similarly, the representation of a marker $q^?$ requires only a Boolean bit of information.

The translation for scanner definitions, on the other hand, collects the set of all literals in an OCFG and assigns a name to each literal in order to establish a proper interface between the `lex` scanner and the `yacc` parser. We model such an interface by a *naming function* σ. For instance, the lambda calculus has the following naming of literals:

$$\sigma_{,} = \texttt{comma} \qquad \sigma_{(} = \texttt{lparen} \qquad \sigma_{)} = \texttt{rparen} \qquad \sigma_{.} = \texttt{dot} \qquad \sigma_{\texttt{fun}} = \texttt{kfun}$$

Given a naming of all symbolic *characters*, a naming function can be automatically constructed by concatenation for all symbolic *strings*. For example, in Java, given $\sigma_{>} = \texttt{rangle}$ and $\sigma_{=} = \texttt{eq}$, we can construct $\sigma_{>>=} = \texttt{rangle\_rangle\_eq}$. Note that keywords are prefixed with the character `k` to avoid name collisions.

3.3 Translation of associativity and precedence

The main part of the translation deals with associativity and precedence. The basic idea behind translating precedence of OCFGs is to expand and re-index rules with their precedence levels. However, doing so and handling associativity as well as repetitions at the same time turns out to be nontrivial. We will first explain the basic translation here and then discuss how to handle repetitions in the next subsection.

To impose the *precedence* constraint of an OCFG, each group inside a rule is stratified into an auxiliary rule indexed by its precedence level such that cases in the group can recurse only into groups of *equal or higher* precedence. For example, the OCFG for the lambda calculus has three precedence groups e_1, e_2, and e_3. Fig. 1 shows how the singleton cases $e_1 : e_2$ and $e_2 : e_3$ allow recursion into groups of higher precedence.

On the other hand, imposing the *associativity* constraint of an OCFG requires special handling of the leftmost and rightmost items. Here is where *re-indexing* happens. For a left-associative case such as $(\text{App} : e\ e)^<$, the leftmost item must be the same identifier as the rule being defined; in this case e. To favor left recursion over right recursion, the leftmost identifier is re-indexed to be the current group level, while the rightmost identifier is re-indexed to the *next* group level. Hence Fig. 1 has the case $e_2 : e_2\ e_3$.

The reasoning for the right-associative case is symmetric. Hence (Fun :

$$
\begin{aligned}
[\![r, x = c_1^{\alpha_1}, \ldots, c_n^{\alpha_n}, c_{n+1}^{\alpha_{n+1}}]\!]_\pi = {} & [\![r]\!]_\pi \\
& x \quad : x_1 \quad \texttt{\{ \$1 \}} \\
& x_1 \quad : x_2 \quad \texttt{\{ \$1 \}} \quad \texttt{|} \quad [\![c_1]\!]^{\alpha_1}_{1,x} \\
& \vdots \\
& x_n \quad : x_{n+1} \quad \texttt{\{ \$1 \}} \quad \texttt{|} \quad [\![c_n]\!]^{\alpha_n}_{n,x} \\
& x_{n+1} \quad : \qquad\qquad\quad [\![c_{n+1}]\!]^{\alpha_{n+1}}_{n+1,x}
\end{aligned}
$$

$$
\begin{aligned}
[\![r, x = v_1 \ldots v_n]\!]_\pi = {} & [\![r]\!]_\pi \\
& [\![v_1]\!]_\pi \ \ldots \ [\![v_n]\!]_\pi \ \texttt{\{ (\$}i \mid v_i \neq q, i = 1, \ldots, n\texttt{) \}}
\end{aligned}
$$

$$
\begin{aligned}
& [\![c, X\texttt{:}\ v_1\ v_2 \ldots v_n\ v_{n+1}]\!]^{<}_{i,x} \\
= {} & [\![c]\!]^{<}_{i,x} \quad \texttt{|} \quad [\![v_1]\!]^0_{i,x}\ [\![v_2]\!]_\pi \ \ldots \ [\![v_{n-1}]\!]_\pi\ [\![v_n]\!]^1_{i,x}\ \texttt{\{}\ X\ \texttt{(\$}i \mid v_i \neq q, i = 1, \ldots, n\texttt{)\ \}}
\end{aligned}
$$

$$
\begin{aligned}
& [\![c, X\texttt{:}\ v_1\ v_2 \ldots v_n\ v_{n+1}]\!]^{>}_{i,x} \\
= {} & [\![c]\!]^{>}_{i,x} \quad \texttt{|} \quad [\![v_1]\!]^1_{i,x}\ [\![v_2]\!]_\pi \ \ldots \ [\![v_{n-1}]\!]_\pi\ [\![v_n]\!]^0_{i,x}\ \texttt{\{}\ X\ \texttt{(\$}i \mid v_i \neq q, i = 1, \ldots, n\texttt{)\ \}}
\end{aligned}
$$

$$
\begin{aligned}
& [\![c, X\texttt{:}\ v_1\ v_2 \ldots v_n\ v_{n+1}]\!]^{=}_{i,x} \\
= {} & [\![c]\!]^{=}_{i,x} \quad \texttt{|} \quad [\![v_1]\!]_\pi\ [\![v_2]\!]_\pi \ \ldots \ [\![v_{n-1}]\!]_\pi\ [\![v_n]\!]_\pi\ \texttt{\{}\ X\ \texttt{(\$}i \mid v_i \neq q, i = 1, \ldots, n\texttt{)\ \}}
\end{aligned}
$$

$$
\begin{aligned}
[\![q]\!]_\pi &= \sigma_q^{\uparrow} & \qquad [\![x]\!]_\pi &= x \\
[\![q^?]\!]_\pi &= \sigma_q\texttt{\_marker} & \qquad [\![x^?]\!]_\pi &= x\texttt{\_option} \\
[\![x^*]\!]_\pi &= x\texttt{\_star} & \qquad [\![x^+]\!]_\pi &= x\texttt{\_plus} \\
[\![x^*_q]\!]_\pi &= x\texttt{\_star\_}\sigma_q & \qquad [\![x^+_q]\!]_\pi &= x\texttt{\_plus\_}\sigma_q
\end{aligned}
$$

$$
\begin{aligned}
[\![q]\!]^k_{i,x} &= \sigma_q^{\uparrow} & \qquad [\![y]\!]^k_{i,x} &= (x = y)\ ?\ x_{i+k} : y \\
[\![q^?]\!]^k_{i,x} &= \sigma_q\texttt{\_marker} & \qquad [\![y^?]\!]^k_{i,x} &= [\![y]\!]^k_{i,x}\texttt{\_option} \\
[\![y^*]\!]^k_{i,x} &= [\![y]\!]^k_{i,x}\texttt{\_star} & \qquad [\![y^+]\!]^k_{i,x} &= [\![y]\!]^1_{i,x}\texttt{\_plus} \\
[\![y^*_q]\!]^k_{i,x} &= [\![y]\!]^k_{i,x}\texttt{\_star\_}\sigma_q & \qquad [\![y^+_q]\!]^k_{i,x} &= [\![y]\!]^1_{i,x}\texttt{\_plus\_}\sigma_q
\end{aligned}
$$

Fig. 5. Translation $[\![r]\!]_\pi$, $[\![c]\!]^\alpha_{i,x}$, $[\![v]\!]_\pi$, $[\![v]\!]^k_{i,x}$ for parser definitions.

$\texttt{fun}\ x^+_{;}\ \texttt{.}\ e)^>$ is translated to be `Kfun x_plus_comma Dot` e_1. (The translation of repetitions $x^+_{;}$ will be explained in the next subsection.) For any associativity, there will be no re-indexing if the leftmost or the rightmost items are not the same identifier as the rule being defined. This reasoning covers the non-associative case where the leftmost and the rightmost items must not be this rule. Hence $(\text{Var} : x,\ \text{Atom} : \texttt{(}\ e\ \texttt{)})^=$ is translated to be $e_3 : x$ `|` `Lparen` e `Rparen` in Fig. 1.

Fig. 5 formalizes the ideas above with these translation functions:

(i) $[\![r]\!]_\pi$, $[\![v]\!]_\pi$ denote the translations for *parser* definitions of rules and items,

(ii) $[\![c]\!]^\alpha_{i,x}$ denotes the translation for *parser* definitions of a group of cases c with associativity α and group index i inside the rule x,

(iii) $[\![v]\!]^k_{i,x}$ denotes the translation for *parser* definitions of an item v with

increment k and group index i inside the rule x.

The function $[\![r]\!]_\pi$ simply stratifies each group of cases and sets up its recursion. Cases are translated respectively by $[\![c]\!]^{<}_{i,x}$, $[\![c]\!]^{>}_{i,x}$, and $[\![c]\!]^{=}_{i,x}$ for left, right, and non-associative groups. The leftmost and the rightmost items are translated specially with an *increment* k of zero or one for left or right recursions in $[\![v]\!]^{k}_{i,x}$. Throughout the paper, we omit the straightforward translations for the empty list $[\![\epsilon]\!]$.

The critical definition is $[\![y]\!]^{k}_{i,x}$, which re-indexes the identifier y to x_{i+k} if $x = y$, or simply outputs y otherwise. Following the convention of OCaml `yacc`, the constructor names of literal items are capitalized using the naming function $\sigma^{\uparrow}_{q}$, which can easily be derived from the original naming function σ_q.

Trees of the appropriate abstract datatype are automatically constructed during parsing. Similar to the translation for types $[\![v]\!]_\tau$ in Section 3.2, we simply skip literals and put all items together with the constructor name: X `($$i` $\mid v_i \neq q, i = 1, \ldots, n$`)`. The semantic actions inside the parse rules in Fig. 1 show an example of such syntax tree constructions. In `yacc`, the special symbol `$$i` refers to the i-th item of the case. For an alias $[\![r, x = v_1 \ldots v_n]\!]_\pi$ there is no associativity or constructor name; its translation simply recurses with $[\![v]\!]_\pi$ and constructs a product of non-literal items as its syntax tree.

3.4 Translation of repetitions

Now we translate repetitions and explain why re-indexing becomes tricky. We write $[\![r]\!]_\rho$, $[\![g]\!]_\rho$, $[\![c]\!]_\rho$, $[\![u]\!]_\rho$ and $[\![v]\!]_\rho$ to denote the translations of *repetitions*. Similar to the translations for the scanner, the translations $[\![\cdot]\!]_\rho$ collect the set of all repetitions $q^?$, $x^?$, x^*, x^+, x^*_q and x^+_q in an OCFG and apply the following translation of *repetitions* $[\![v]\!]^{\rho}_{\pi}$ for *parser* definitions:

$$
\begin{array}{rlll}
[\![q^?]\!]^{\rho}_{\pi} = \sigma_q\texttt{\_marker:} & \epsilon\ \texttt{\{ false \}} & \mid\ \sigma^{\uparrow}_{q} & \texttt{\{ true \}} \\
[\![x^?]\!]^{\rho}_{\pi} = x\texttt{\_option:} & \epsilon\ \texttt{\{ None \}} & \mid\ x & \texttt{\{ Some \$1 \}} \\
[\![x^*]\!]^{\rho}_{\pi} = x\texttt{\_star:} & \epsilon\ \texttt{\{ [] \}} & \mid\ x\texttt{\_plus} & \texttt{\{ \$1 \}} \\
[\![x^+]\!]^{\rho}_{\pi} = x\texttt{\_plus:} & x\ \texttt{\{ [\$1] \}} & \mid\ x\ \ x\texttt{\_plus} & \texttt{\{ \$1 :: \$2 \}} \\
[\![x^*_q]\!]^{\rho}_{\pi} = x\texttt{\_star\_}\sigma_q\texttt{:} & \epsilon\ \texttt{\{ [] \}} & \mid\ x\texttt{\_plus\_}\sigma_q & \texttt{\{ \$1 \}} \\
[\![x^+_q]\!]^{\rho}_{\pi} = x\texttt{\_plus\_}\sigma_q\texttt{:} & x\ \texttt{\{ [\$1] \}} & \mid\ x\ \sigma^{\uparrow}_{q}\ x\texttt{\_plus\_}\sigma_q & \texttt{\{ \$1 :: \$3 \}}
\end{array}
$$

Note that x^* is defined in terms of x^+ and thus, when x^* is added to the set of repetitions, x^+ must be added too. The same applies for x^*_q and x^+_q.[2] These translations are consistent with the following equations:

$$v^? = \epsilon \mid v \qquad v^+ = v \mid v\ v^+ \qquad v^* = \epsilon \mid v^+$$

We translate x^*, x^+, x^*_q and x^+_q using right recursion because ML lists are right-recursive: `type 'a list = Nil | Cons of 'a * 'a list`. For efficient parsers with constant stack space, left recursion can be used instead,

[2] Another way to define x^* is $x^* = \epsilon \mid x\ x^*$, which is independent of x^+. For x^*_q, however, the analogous definition $x^*_q = \epsilon \mid x\ q\ x^*_q$ is wrong (as it wrongly accepts the string $x\ q$).

at the expense of longer definitions and performing list reversals during construction. For example, here is a left-recursive translation of x^+:

$$[\![x^+]\!]^{\rho}_{\pi} = \begin{array}{ll} x\texttt{\_plus:} & x\texttt{\_plus\_rev \{ rev \$1 \}} \\ x\texttt{\_plus\_rev:} & x\ \texttt{\{ [\$1] \}}\ \ \texttt{|}\ \ x\texttt{\_plus\_rev}\ x\ \texttt{\{ \$2 :: \$1 \}} \end{array}$$

The following is the reason why re-indexing in Section 3.3 must be specially adapted to repetition handling. Consider, for example the identifier term e, which has the same prefix as its repetition term e^*. If repetitions are user-defined, more complex computations such as *nullable sets* and *next-token sets* [4] are necessary for re-indexing. The built-in repetitions, however, already capture most syntax idioms; OCFGs can thus be treated *syntactically* by assuming that all rules are prefix independent.

For nullable items like $x^?$, x^* and x^*_q, the lookahead sets are the same as the identifier item x; hence, these items have the same increment k as the identifier item x (see Fig. 5). Non-empty lists like x^+ and x^+_q, however, require looking ahead at the first token of the next element in the list or the separator of the list in order to decide whether to shift or reduce. Hence, these items always have increment $k = 1$, recursing only into the next precedence group.

Since re-indexing may introduce new repetitions like e_2`_bar_plus`, the overall order of translations must be as follows: first re-index with $[\![\cdot]\!]_{\pi}$, then collect repetition items with $[\![\cdot]\!]_{\rho}$, and last append the parser definitions of repetitions with $[\![v]\!]^{\rho}_{\pi}$.

One caveat about correctness: the re-indexing algorithm assumes that the leftmost or the rightmost items are not nullable immediately adjacent to the recursive identifier. For example, we disallow rules like $e : (\texttt{begin}^?\ e\ \texttt{end})^<$ or $e : (\texttt{begin}\ e\ \texttt{end}^?)^>$. Appendix A further discusses this problem.

3.5 Translation for error diagnosis

The last translation here deals with a practical issue of lexical analysis: generating *high-level* and *precise* error diagnosis. We will show how this can be done automatically with OCFGs and `yacc`'s `error` token.

We observe that, for any rule, parsing can be aborted *immediately after a literal* if the input string does not match the prefix of the rule. Hence we can insert the special `error` token provided by the `yacc` parser in place of the rest of each rule, and give a high-level description for error diagnosis in terms of the prefix and the complete rule.

This strategy of placing the `error` token after a literal not only gives *precise diagnosis* but also ensures that we do not introduce additional conflicts in LALR parsing. Different rules, however, may have the same prefix and hence we must group rules for error diagnosis by their common prefixes.

Formally we introduce two new helper functions $[\![c]\!]_p$ and $[\![c]\!]_u$ and extend the translation $[\![r]\!]_{\pi}$ for parser definitions in Section 3.3 in Figure 6.

The function $[\![c]\!]_p$ collects the set of *literal-terminating prefixes* for cases c, while $[\![c]\!]_u$ computes the inverse (the set of cases with the prefix u). The

$$[\![c, X{:}\,u]\!]_p = [\![c]\!]_p \cup \{\ u_1\, q \mid u = u_1\, q\, u_2,\ u_2 \neq \epsilon\ \}$$

$$[\![c, X{:}\,u]\!]_u = [\![c]\!]_u, (u = u_1\, u_2)\ ?\ (X : u)\ :\ \epsilon$$

$$[\![r, x = c_1^{\alpha_1}, \ldots, c_n^{\alpha_n}]\!]_\pi = \begin{array}{l} \vdots \\ \mid\ u\ \texttt{error \{ abort}\ u\ [\![c_1]\!]_u\ \texttt{\}}\ \ \mid u \in [\![c_1]\!]_p \\ \vdots \\ \mid\ u\ \texttt{error \{ abort}\ u\ [\![c_n]\!]_u\ \texttt{\}}\ \ \mid u \in [\![c_n]\!]_p \end{array}$$

Fig. 6. Translation $[\![c]\!]_p$, $[\![c]\!]_u$, and extended $[\![r]\!]_\pi$ for error diagnosis.

translation $[\![r]\!]_\pi$ in Section 3.3 now additionally inserts u `error` for each prefix u of the rule c_i and makes use of an external function `abort` that quits the program with the current parse state (the input file name, the line number, and the start and end columns).

For example, consider the shift operators in Java (see Figure 2):

$$\begin{array}{rcll} e & ::> & \ldots & \\ & \mid & \mathit{exp}\ \texttt{>}\ \texttt{>}\ \mathit{exp} & \text{(Shr)} \\ & \mid & \mathit{exp}\ \texttt{>}\ \texttt{>}\ \texttt{>}\ \mathit{exp} & \text{(Ushr)} \end{array}$$

After re-indexing the expression to be the eleventh level according to the translation in Fig. 5, the following additional code is generated for error diagnosis:

```
exp11: ... | exp11 Rangle Rangle error {
  abort "exp '>' '>' " [
    "Shr: exp '>' '>' exp";
    "Ushr: exp '>' '>' '>' exp"
  ]}
```

If an expression such as [`x >>`] with a dangling right angle brackets occur on the first column of line 3 in file `foo.bar`, the parser will complain as follows:

```
foo.bar:03:01: Parse error: premature prefix of possible rules
    prefix :  exp '>' '>'
    rule   :  Shr: exp '>' '>' exp
    rule   :  Ushr: exp '>' '>' '>' exp
```

Such systematic and precise error messages make it easier for programmers to fix a problem locally. They can also use constructor names like *Shr* to look up the context of the rule in the OCFG.

4 Related work

The idea of using associativity and precedence for deterministic parsing dates back to the work by Aho, Johnson and Ullman in 1973 [5]. However, their technique applies to *lexical tokens*, while ours applies to *production rules*. Various tricks, such as context-dependent precedence in `yacc`, are necessary to

distinguish the *minus for negation* (unary minus) from the *minus for subtraction* (binary minus) in arithmetic rules. Some parse rules, such as lambda applications $e\ e$, do not even have operators to begin with.

Aasa's work [2] on user-defined syntax is most directly related to our work in terms of the translation. Her translation, unlike ours, can handle grammars with arbitrary combinations of precedence and associativity. Instead, our translation handles practical grammars for programming languages; for example, non-associative rules in such practical grammars always have the highest precedence. This assumption greatly simplifies our translation, compared to Aasa's. Her translation is also proven correct with respect to a notion of *precedence-correct* parse trees; our work lacks a formal proof of correctness.

Laski [9] proposes ordered context-free grammars with theoretical development and a toy implementation. His semantics is based on a modified LALR algorithm that uses *precedence* and *associativity vectors* to resolve conflicts. We borrow the term OCFG from his work and extend it with datatype tags in a real implementation. More importantly, our formalism is based on translation and can reuse existing and efficient parsing infrastructures.

Extending datatypes with concrete syntax has been well-explored [3,11,20], usually in the context of scanner-less parsers. Scanner-less parsers [15,6,19] allow a uniform and integrated specification of scanning and parsing, eliminating the cumbersome interface between these two lexical phases.

Automatic construction of syntax trees is supported by many modern parser generators [11,13]. Our approach is guided by the constructor names of datatypes and constructs trees in a functional way.

Degano and Priami [7] present a comprehensive comparison of error handling in LR parsers. Our approach of using `yacc`'s `error` token so that parse errors can be given in high-level descriptions of OCFGs seems novel.

5 Conclusion

We have presented the notion of *ordered context-free grammars (OCFGs) with datatype tags* to concisely specify grammars of programming languages. OCFGs capture associativity and precedence at the level of *production rules* instead of *lexical tokens* such that a concrete syntax grammar is succinct enough be an abstract syntax definition. We have also presented a complete OCFG grammar for Java 5, showing the practicality of our formalism.

References

[1] A. Aasa. Precedences for conctypes. In *Functional programming languages and computer architecture*, 1993.

[2] A. Aasa. Precedences in Specifications and Implementations of Programming Languages. *Theoretical Computer Science*, 1995.

[3] S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, 1991.

[4] A. Aho and J. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall, 1972.

[5] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic Parsing of Ambiguous Grammars. In *ACM Symposium on Principles of Programming Languages*, 1973.

[6] J. A. Bergstra, T. B. Dinesh, J. Field, and J. Heering. A Complete Transformational Toolkit for Compilers. In *European Symposium on Programming*, 1996.

[7] P. Degano and C. Priami. Comparison of Syntactic Error Handling in LR Parsers. *Software - Practice and Experience*, 1995.

[8] A. Johnstone, E. Scott, and G. Economopoulos. Generalised parsing: some costs. In *ACM Symposium on Compiler Construction*, 2004.

[9] Z. Laski. Ordered Context-Free Grammars. Technical Report 99-18, University of California, Irvine, 2000.

[10] X. Leroy. The OCaml Programming Language. http://caml.inria.fr.

[11] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software - Practice and Experience*, 1995.

[12] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] A. Ranta. The Labelled BNF Grammar Formalism. Technical report, Chalmers University of Technology, Sweden, 2003.

[14] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992, 1992.

[15] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. In *ACM Conference on Programming Language Design and Implementation*, 1989.

[16] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic, 1986.

[17] S. Tse. Functional Java Compiler. http://www.cis.upenn.edu/~stse/javac.

[18] S. Tse and S. Zdancewic. Designing a Security-typed Language with Certificate-based Declassification. In *European Symposium on Programming*, 2005.

[19] M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction*, 2002.

[20] E. Visser. *Syntax Denition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

A Problems with translation of repetitions

Example A.1 To illustrate why non-empty lists must recurse into the next precedence group, consider

$$e : \ (X_1 : e^{+} \ \texttt{\$})^{<} \quad (X_2 : x)^{=}$$

for expressions as a list of identifiers x with delimiter `$`. If we incorrectly translated e^{+} to e_1`_plus` (instead of e_2`_plus` with increment $k = 1$), there would be a LALR(1) shift-reduce conflict:

$$\begin{array}{rcl} e_1 & : & e_2 \ \texttt{|} \ e_1\texttt{\_plus Dollar} \\ e_2 & : & x \\ e_1\texttt{\_plus} & : & e_1 \ \texttt{|} \ e_1 \ e_1\texttt{\_plus} \end{array}$$

For instance, the strings `x$` and `x$$` have the same prefix `x$`. We need to reduce `x` as e_1 in the rule e_1`_plus` to accept `x$`, but we need to shift `x` to e_1 e_1`_plus` in the rule e_1`_plus` to accept `x$$`. □

Example A.2 To illustrate the problem of nullable items on the sides, consider

$$s : \ (X_1 : \texttt{for} \ s, \ \ X_2 : \texttt{if} \ s \ \texttt{else}^{?})^{>} \quad (X_3 : x)^{=}$$

for two kinds of *right*-associative statements: `for`-statements and `if`-statements with an optional marker `else` at the end. If we simply ignore markers, there will be a LALR(1) shift-reduce conflict:

$$\begin{array}{rcl} s_1 & : & s_2 \ \texttt{|} \ \texttt{Kfor} \ s_1 \ \texttt{|} \ \texttt{Kif} \ s_1 \ \texttt{kelse\_marker} \\ s_2 & : & x \\ \texttt{kelse\_marker} & : & \epsilon \ \ \texttt{|} \ \texttt{Kelse} \end{array}$$

But, because of the default action of shifting for the longest matches in `yacc`, the conflict is harmless, just like the standard shift-reduce conflict for dangling-`else` statements in C or Java. For instance, the string `for if if x else` will be parsed correctly as X_1 (X_2 (X_2 (X_3 x) `true`) `false`), indicating that the marker `else` is associated with the second `if`.

Consider, on the other hand, a rather contrived example:

$$s : \ (X_1 : s \ \texttt{for}, \ X_2 : \texttt{if}^{?} \ s \ \texttt{else})^{<} \quad (X_3 : x)^{=}$$

Here we have two kinds of *left*-associative statements s: `for`-statements, and `else`-statements with an optional literal `if` as marker at the beginning. Even if the `else`-statements recurse with the lowest group s_2, there is still a LALR(1) shift-reduce conflict:

$$\begin{array}{rcl} s_1 & : & s_2 \ \texttt{|} \ s_1 \ \texttt{Kfor} \ \texttt{|} \ \texttt{kif\_marker} \ s_2 \ \texttt{Kelse} \\ s_2 & : & x \\ \texttt{kif\_marker} & : & \epsilon \ \ \texttt{|} \ \texttt{Kif} \end{array}$$

This conflict *is* harmful and prevents accepting strings like `x else`: looking at x, the parser shifts to s_1 in the rule s_1 `Kfor`, instead of reducing as `kif_marker` in the rule `kif_marker` s_2 `Kelse`, and thus eventually rejects the string. □

B Java 5 grammar (source code)

```
cunit: package? import* decl*;
package:  'package' name ';';
ty:
= Tarray: ty dimen+
= Tinst: ty targs
= Tname: name
| Tbool: 'boolean'
| Tbyte: 'byte'
| Tchar: 'char'
| Tdouble: 'double'
| Tfloat: 'float'
| Tint: 'int'
| Tlong: 'long'
| Tshort: 'short'
| Tvoid: 'void';
wty:
= WildType: ty
| WildCard: '?'
| WildExtends: '?' 'extends' ty
| WildSuper: '?' 'super' ty;
tparams: '<' tparam ++ ',' '>';
tparam: id extend? tinter*;
tinter: '&' ty;
targs: '<' wty ++ ',' '>';
ifelse: 'else' stmt;
block: '{' stmt* '}';
extend: 'extends' ty;
extends: 'extends' ty ++ ',';
bodies: '{' body* '}';
dimen: '[' ']';
size: '[' exp ']';
default: 'default' exp;
arg: '(' exp ** ',' ')';
labelexp: id '=' exp;
name: id ++ '.';
all: '.' '*';
vdecl: id dimen* vinit?;
vinit: '=' exp;
exps:
> ExpsCons: exp ',' exps
| ExpsSome: exp
| ExpsNil:;
mod:
= BasicMod: basicmod;
| AnnotMark: '@' name
| AnnotSingle: '@' name '(' exp ')'
| AnnotMore: '@' name '(' labelexp ++ ',' ')'
exp:
= Array: '{' exps '}'
> Assign: exp assign exp
> Cond: exp '?' exp ':' exp
< Cor: exp '||' exp
< Cand: exp '&&' exp
< Or: exp '|' exp
< Xor: exp '^' exp
< And: exp '&' exp
< Eq: exp '==' exp
| Ne: exp '!=' exp
< Lt: exp '<' exp
| Gt: exp '>' exp
| Le: exp '<' '=' exp
| Ge: exp '>' '=' exp
| Instof: exp 'instanceof' ty
< Shl: exp '<' '<' exp
| Shr: exp '>' '>' exp
| Ushr: exp '>' '>' '>' exp
< Add: exp '+' exp
| Sub: exp '-' exp
< Mul: exp '*' exp
| Div: exp '/' exp
| Rem: exp '%' exp
> Cast: '(' ty ')' exp
| PreIncr: '++' exp
| PreDecr: '--' exp
| Pos: '+' exp
| Neg: '-' exp
| Not: '!' exp
| Inv: '~' exp
< Offset: exp '[' exp ']'
| Dot: exp '.' exp
| ExpClass: ty '.' 'class'
| Invoke: exp arg
| PostIncr: exp '++'
| PostDecr: exp '--'
= Inst: targs exp
= This: 'this'
| Super: 'super'
| Literal: literal
| Ident: id
| Atom: '(' exp ')'
| NewArr1: 'new' ty size+ dimen*
| NewArr2: 'new' ty '{' exps '}'
| NewObj: 'new' name targs? arg bodies?;
```

```
decl:
= Class: mod* 'class' id tparams? extend? implements? bodies
| Iface: mod* 'interface' id tparams? extends? bodies
| Enum: mod* 'enum' id implements? '{' enumbody '}'
| Annot: mod* '@' 'interface' id bodies
| Empty: ';';
body:
= Init: 'static'? block
| Inner: decl
| Ctor: mod* tparams? id signat block
| MethodNone: mod* tparams? ty id signat ';'
| MethodSome: mod* tparams? ty id signat block
| Field: mod* ty vdecl ++ ',' ';'
enumbody:
= EnumNone: enumconst ++ ',' ','? ';'?
| EnumSome: enumconst ++ ',' ','? ';' body+
enumconst: mod* id arg? bodies?;
import:  'import' 'static'? name all? ';';
implements: 'implements' ty ++ ',';
signat: '(' formals ')' default? dimen* throws?;
stmt:
> ForDecl: 'for' '(' mod* ty vdecl ++ ',' ';' exp? ';' exp ** ',' ')' stmt
| ForEach: 'for' '(' mod* ty id ':' exp ')' stmt
| ForExp: 'for' '(' exp ** ',' ';' exp? ';' exp ** ',' ')' stmt
| While: 'while' '(' exp ')' stmt
| If: 'if' '(' exp ')' stmt ifelse?
| Label: id ':' stmt
= AssertNone: 'assert' exp ';'
| AssertSome: 'assert' exp ':' exp ';'
| Block: block
| Break: 'break' id? ';'
| Case: 'case' exp ':'
| Continue: 'continue' id? ';'
| Vdecl: mod* ty vdecl ++ ',' ';'
| Default: 'default' ':'
| Do: 'do' stmt 'while' '(' exp ')' ';'
| Exp: exp ';'
| Local: decl
| Return: 'return' exp? ';'
| Switch: 'switch' '(' exp ')' block
| Sync: 'synchronized' '(' exp ')' block
| Throw: 'throw' exp ';'
| Try: 'try' block catch* finally?;
catch: 'catch' '(' mod* ty id ')' block;
finally: 'finally' block;
throws: 'throws' name ++ ',';
formals:
> ArgMore: mod* ty id dimen* ',' formals
= ArgFix: mod* ty id dimen*
| ArgVar: mod* ty '...' id dimen*
| ArgNone:;
```

```
literal:
| Char: char
| Long: long
| Int: int
| Float: float
| Str: string;

basicmod:
| Abstract: 'abstract'
| Final: 'final'
| Native: 'native'
| Private: 'private'
| Protected: 'protected'
| Public: 'public'
| Static: 'static'
| Strictfp: 'strictfp'
| Synchronized: 'synchronized'
| Transient: 'transient'
| Volatile: 'volatile';

assign:
= Set: '='
| AddSet: '+='
| SubSet: '-='
| MulSet: '*='
| DivSet: '/='
| RemSet: '%='
| AndSet: '&='
| OrSet: '|='
| XorSet: '^='
| ShlSet: '<' '<' '='
| ShrSet: '>' '>' '='
| UshrSet: '>' '>' '>' '=';

- stmt: 'If (_, _, If (_,_,_,None), Some _)';
- stmt: 'If (_, _, If (_,_,_,Some (If (_,_,_,None))), Some _)';
- stmt: 'If (_, _, If (_,_,_,Some (If (_,_,_, Some (If (_,_,_,None))))), Some _)';
- exp: 'Invoke (_, Atom _, [_])';
- exp: 'Cast (_, Tname _, Pos _)';
- exp: 'Cast (_, Tname _, Neg _)';
- exp: 'Offset (_, NewArr1 _, _)';
- exp: 'Assign (_, Gt _, _, _)';
- exp: 'Assign (_, Lt _, _, _)';
- stmt: 'Exp (_, Gt _)';
- stmt: 'Exp (_, Lt _)';
- stmt: 'ForExp (_, Gt _::_, _, _, _)';
- stmt: 'ForExp (_, Lt _::_, _, _, _)';
```