Attribute Grammar-based Language Extensions for Java*

Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin

Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455, USA
evw,krishnan,bodin,schwerdf@cs.umn.edu

Abstract. This paper describes the Java Language Extender framework, a tool that allows one to create new domain-adapted languages by importing domain-specific language extensions into an extensible implementation of Java 1.4. Language extensions may define the syntax, semantic analysis, and optimizations of new language constructs. Java and the language extensions are specified as higher-order attribute grammars. We describe several language extensions and their implementation in the framework. For example, one embeds the SQL database query language into Java and statically checks for syntax and type errors in SQL queries. The tool supports the modular specification of composable language extensions so that programmers can import into Java the unique set of extensions that they desire. When extensions follow certain restrictions, they can be composed without requiring any implementation-level knowledge of the language extensions. The tools automatically compose the selected extensions and the Java host language specification.

1 Introduction

One impediment in developing software is the wide semantic gap between the programmer's high-level (often domain specific) understanding of a problem's solution and the relatively low-level language in which the solution must be encoded. General purpose languages provide features such as classes, generics/parametric polymorphism, and higher-order functions that programmers can use to specify abstractions for a given problem or problem domain, but these provide only the functionality of the desired abstractions. Domain-specific languages (DSLs) can be employed to provide this functionality, but they can also provide domain-specific language constructs (new syntax) for the abstractions. These constructs raise the level of abstraction of the language to that of the specific domain and thus help to reduce the semantic gap. As importantly, domain specific languages also provide domain-specific optimizations and analyses that are either impossible or quite difficult to specify for programs written in general purpose languages. But problems often cross multiple domains and no language will contain all of the general-purpose and domain-specific features needed to address all of the problem's aspects, thus the fundamental problem remains — programmers cannot "say what they mean" but must encode their solution ideas as programming idioms at a lower level of abstraction. This process is time-consuming and can be the source of errors.

* This work is partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

The Java Language Extender: The Java Language Extender (JLE) is a language processing tool that addresses this fundamental problem. It supports the creation of extended, domain-adapted variants of Java by adding domain-specific language extensions to an extensible implementation of Java 1.4. An extended language defined by this process has features that raise the level of abstraction to that of a particular problem. These features may be new language constructs, semantic analyses, or optimizing program transformations, and are packaged as *modular language extensions*. Language extensions can be as simple as a the Java 1.5 *for-each* loop or the more sophisticated set of SQL language constructs that statically check for syntax and type errors in SQL queries. We have also developed domain-specific extensions to support the development of efficient and robust computational geometry program and extensions that introduce condition tables (useful for understanding complex boolean expressions) from the modeling language RSML^{-e}. Extensions can also be general purpose in nature; we have defined extensions that add algebraic datatypes and pattern matching from Pizza [18], add concrete syntax for lists and hashmaps, and others that add the automatic boxing and unboxing of Java primitive types.

JLE is an attribute grammar-based extensible language framework in which an extensible host language is specified as a complete attribute grammar (AG) and the language extensions are specified as attribute grammar fragments. These are written in Silver, an attribute grammar specification language developed to support this process. The attribute grammars define semantics of the host language and the language extensions. Silver also supports the specification of concrete syntax that is utilized by parser and scanner generators. The Silver extensible compiler tools combine the AG specifications of the host language and the programmer selected language extension to create an AG specification for the custom extended language desired by the programmer. An attribute grammar evaluator for this grammar implements the compiler for the extended language. Concrete syntax specifications are similarly composed.

It is important that constructs implemented as language extensions have the same "look-and-feel" of host language constructs. By this we mean that their syntax should fit naturally with the host language, error messages should be reported in terms of the extension constructs used by the programmer, not in terms of their translations to some implementation as is the case with macros. Also, extension constructs should be efficient and should thus generate efficient translations. Forwarding [24], an extension to higher-order attribute grammars [30], facilitates the modular definition of languages and supports the implementation of constructs that satisfy this look-and-feel criteria. It allows one to implicitly specify the semantics of new constructs by translation to semantically equivalent constructs in the host Java 1.4 language. For example, an SQL query translates to a calls to the JDBC library. But it also allows the explicit specification of semantics since attributes can be defined on productions defining language constructs to, for example, check for errors are the extension level. This is the case with the SQL extension.

In Silver, attribute grammars are package as modules defining either a host language or a language extension. Module names, like Java packages, are based on Internet domains to avoid name clashes. The module `edu:umn:cs:melt:java14` defines Java 1.4 and defines the concrete syntax of the language, the abstract syntax and the semantic analyses required to do most type checking analyses and to do package/type/expression name disambiguation. The grammar defines most aspects of a Java compiler but it does not specify byte-code generation. Language extensions add new constructs and their translation to Java 1.4 code; a traditional Java compiler then converts this to byte-codes for execution. The static analysis we perform is to support analysis of extensions and to ensure that any statically detectable errors (such as type errors and access violations) in the extended Java language are caught so that erroneous code is not generated. Programmers should not be expected to look at the generated Java code; errors should be reported on the code that they write.

Of particular interest are language extensions designed to be *composable* with other extensions, possibly developed by different parties. Such extensions may be imported by a programmer into Java without requiring any implementation level knowledge of the extensions. Thus, we make a distinction between two activities: (*i*) implementing a language extension, which is performed by a domain-expert feature designer and (*ii*) selecting the language extensions that will be imported into an extensible language specification in order to create an extended language. When extensions are composable, this activity can be performed by a programmer. This distinction is similar to the distinction between library writers and library users. Thus a programmer, facing a geometric problem in which the geometric data is stored in a relational database may import into Java both the SQL extension and the computational geometry extension to create a unique domain-adapted version of Java that has features to support both of these aspects of her problem.

The goal of composability does restrict the kind of features that can be added to a language in a composable language extension. The primary determinant is the type of transformations (global or local) that are used to translate the high-level extension constructs into the implementation language of Java 1.4. If global transformations are required in translating to the host language then the order in which these reductions are made may matter and selecting this ordering would require that the programmer has some implementation level knowledge of the implementation of the extensions being imported. Thus, composable extensions (which include those listed above) use primarily local transformations.

Contributions The paper shows how general purpose languages, in this case Java 1.4, can be implemented in such a way that rich domain-specific and general purpose language features can be imported to create new extended language adapted to particular problems and domains. Two key characteristics of the extensions presented here are that they perform semantic analysis at the language extension level and that they are composable. Thus, it is feasible that programmers, with no implementation-level knowledge of the extensions, can import the set of extensions that address a particular problem. Note however that JLE can also be used to define non-composable extensions introduce features that require fundamental alteration or replacement of host language constructs. Section 2 shows several sample language extensions, Section 3 describes the attribute grammar based implementation of Java 1.4 and some selected extensions. Section 4 shows how extensions are composed, discusses analyses of extension specifications, and briefly described an integrated parser/scanner that support composition of concrete syntax specifications. Section 5 describes related and future work and concludes.

2 Sample Language Extensions

Several composable extensions have been specified and implemented for the host Java 1.4 language and we describe several of them here. Samples uses and their translations to the host language are given as well as short descriptions of the unique characteristics of each extension. Others are described during the description of the Java 1.4 host language specification.

SQL: Our first extension embeds the database query language SQL into Java allowing queries to be written in a natural syntax. The extension also statically detects syntax and type errors in embedded queries. A previous workshop paper [23] reports the details of this language extension but we discuss it here and in Section 3.3 to describe how it interacts with and extends the environment (symbol-table) defined in the Java attribute grammar. Figure 1 shows an example of code written using the SQL extension to Java 1.4. The `import table` construct defines the table `person` and its columns along with their types. For example, the column `person_id` has SQL type `INTEGER`. This information is used to statically type check the SQL query in the `using ... query` construct. Thus,

```
public class Demo {
  public static void main(String args) {
    import table person [ person_id INTEGER, first_name VARCHAR,
                          last_name VARCHAR, age INTEGER ] ;
    int limit = 25 ;
    connection c = "jdbc:/db/testdb;";
    ResultSet rs = using c query
          {SELECT last_name FROM person WHERE age > {limit} } ;  } }
```

Fig. 1. Code using the SQL extension to Java 1.4

if the column `age` had type `VARCHAR` (an SQL string type) instead of `INTEGER` the extension would report the type error on the > expression at compile time instead of at run-time as is done in the JDBC [32] library-based approach. The extension translates SQL constructs to pure Java 1.4 constructs that use the JDBC library as shown in Figure 2. Here the SQL query is passed as a Java String to the database server for execution. When queries are written this way, statically checking for syntax or type errors requires extracting queries from the Java strings and reconstructing them. This is much more difficult than when the query constructs can be examined directly as is possible in the SQL extension.

```
public class Demo {
  public static void main(String args) {
    int limit = 25 ;
    Connection c = DriverManager.getConnection("jdbc:/db/testdb;");
    ResultSet rs = c.createStatement().executeQuery((
       "SELECT "+" last_name "+" FROM "+" person "+" WHERE "+" age "+
       " > " + (limit) )) ;    } }
```

Fig. 2. Equivalent code in Java 1.4

Complex Numbers: This extension adds complex numbers as a new primitive type to Java 1.4. This extension specifies the new type, means for writing complex number literals and subtype relations with existing host language types, for example that the new type is a super-type of the host language `double` type. It also specifies productions to coerce `double`-valued types into the new type. Thus, in the code fragment `double d; complex c; d = 1.4; c = d + 1.7; c = complex(1.2,2.3);` written in a version of Java extended with the complex number type the first assignment to `c` first translates to `c = complex(d + 1.7,0.0);` after which the complex literals translate to calls to constructors for the Java class `Complex` constructor that implements the complex number type. Finally, this extension also overloads the arithmetic operators, such as + and *. It is further described in Section 3.2.

C++ allows operator overloading as well. But in the attribute grammar-based extensible language approach proposed here, there are mechanisms for optimizing new numeric types that do not exist in C++. Thus, the production for complex add may optimize the operands before performing the addition.

Computational Geometry: More interesting opportunities for optimization are prevalent in the domain of computational geometry and we have specified numerical types for unbound-precision integers in a language extension [25] for this domain. This extension was developed for a small C like language and takes advantage of domain knowledge that unavailable in general purpose language and library based implementations, to perform optimizations that generated C language code 3 to 20 times faster than the equivalent code utilizing the CGAL geometric library - typically regarded as the best C++ template library for the domain.

Algebraic Datatypes: JLE can also be used to specify general purpose language extensions. We have written an extension that extends Java 1.4 with Pizza-style [18] algebraic datatypes. Figure 3 shows code in this extended version of Java inspired by the examples in [18]. The extension translates to this pure Java 1.4 code the adds new `Nil` and `Cons` subclasses of `List`. The subclasses have a tag field used to identify them as nil or cons objects and fields on `Cons` for the `char` and `List` parameters to the `Cons` constructor in Figure 3 The translation of the pattern matching `switch` statement is a nested `if-then-else` construct that uses the tag field to determine the constructor in place of pattern matching.

```
algebraic class List {
  case Nil;
  case Cons (char, List);
  public List append (List ys) {
    switch (this) {
      case Nil:
        return ys;
      case Cons (x, xs):
        return new Cons (x, xs.append (ys));
    }
    return null;    }
}
```

Fig. 3. Code using a version of Java extended with Pizza-style algebraic datatypes

3 Extensible Java and Java Extension Specifications

In this section we describe the Silver AG specifications that define the host language Java 1.4 and describe several composable language extensions. We have simplified some minor features of the grammar to aid presentation and to focus on aspects that are important while specifying extensions. Thus the interaction of the extensions with the type checking and environment (symbol-table) specifications are highlighted.

In addition to the traditional AG constructs introduced by Knuth [15], Silver supports higher-order attributes [30], forwarding [24], collection attributes [3] and various general-purpose constructs such as pattern matching and type-safe polymorphic lists. Silver also has mechanisms for specifying the concrete syntax of language constructs. These specifications are used to generate a parser and scanner for the specified language. Language extensions are specified as AG fragments, also written in Silver. New production define new language constructs, new attribute declarations and definitions on productions define semantic analyses such as type checking and the construction of optimized translations of constructs (in higher-order attributes).

Silver's module system combines the host language specification and the selected language extension specifications to create the AG specification defining the new extended language. This process is described in Section 4.1. A Silver module names a directory, not a file, and the module consists of all Silver files (with extension `.sv`) in that directory. The scope of a declaration includes all files in the module.

Section 3.1 describes part of the Java 1.4 host language AG specification. Section 3.2 describes how extensions interact with the typing and subtyping information collected and processed by the host language AG. Section 3.3 shows how extensions interact with and extend the environment (symbol-table) defined by the host language AG.

Due to space considerations, these specifications are necessarily brief and have been simplified so that the main concepts are not obscured by the details required in building real languages.

3.1 Java attribute grammar specification

A small simplified portion of the Java 1.4 host-language AG specification is shown in Figure 4. A Silver file consists of the grammar name, grammar import statements (there are none here) and an unordered sequence of AG declarations. The specification first defines a collection of non-terminal symbols; the non-terminal `CompilationUnit` is the start symbol of the grammar and represents a `.java` file. Nonterminals `Stmt`, `Expr`, and `Type` represent Java statements, expressions, and type expressions respectively. Other nonterminals are declared (but not shown) for class and member declarations. The nonterminal `TypeRep` is used by abstract productions to represent types. Next is the declaration for the terminal symbol `Id` that matches identifiers with an associated regular expression . The terminal declaration for the while-loop keyword follows and provides the fixed string that it matches (indicated single quotes).

Concrete productions and the incident nonterminals and terminals are used to generate a parser and scanner. The parser/scanner system is further discussed in Section 4.3. In the specification here attribute evaluation is done on the concrete syntax as this simplifies the presentation. In the actual AG the concrete syntax productions are used only to generate the abstract syntax trees over which attribute evaluation is actually performed. Aspect productions, which are used later, allow new attribute definitions to be added to existing concrete or abstract productions from a different file or grammar module.

Several synthesized attributes are also defined. A pretty-print `String` attribute `pp` decorates (`occurs on`) the nonterminals `Expr`, `Stmt`, and others. The `errors` attribute is a list of `String`s and occurs on nearly all nonterminals. The `typerep` attribute is the representation of the type used internally for type-checking. It is a data structure (implemented as a tree) of type `TypeRep` and decorates `Type` and `Expr` nonterminals. For each nonterminal `NT` in the host language there is a synthesized attribute `hostNT` of type `NT`. On a node of type `NT` it holds that node's translation to the host language. These attributes are used to extract the host language Java 1.4 tree from the tree of an extended Java language program. This is discussed further in the context of extensions below.

Following these are two productions, one defining the while loop, one defining local variable declarations. The environment attributes `defs` and `env` implement a symbol table and are described further in Section 3.3. Silver productions name the left hand side nonterminal and right hand side parameters (terminals, nonterminals, or other types) and borrow the Haskell "has type" syntax `::` to specify, for example, that the third parameter to the while production is named `cond` and is a tree of type nonterminal `Expr`. Attribute definitions (between curly braces) follow the production's signature. For the while loop the definition of the `errors` attribute uses pattern matching to check that condition is of type `Boolean`. Pattern matching in Silver is similar to pattern matching in ML and Haskell. In Silver however, nonterminals play the role of algebraic types and productions play the role of value constructors.

Type representations (trees of type `TypeRep`) are constructed by the productions `doubleTypeRep` and `arrayTypeRep` and used to create type representations on `Type` nodes, as in `doubleType`. One attribute on a typerep is `superTypes` which is a list of super types of that particular type along the means for converting to the super type if runtime conversion is required. This attribute is a *collection* attribute (similar to those defined by Boyland [3]). It allows aspect productions to contribute additional elements to the attribute value. In the `doubleTypeRep` production, the `superTypes` attribute is given an initial value (using the distinct assignment operator `:=`) of the empty list since it has no super types in Java 1.4. The complex number extension will specify an aspect production on `doubleTypeRep` and contribute to this list of super types to indicate that the complex number type is a super type of double. This is described in Section 3.2. We have elided some details here since the manner in which type representations are implemented is not as important as how extensions can interact with the JavaAG to define new types.

3.2 Types and subtyping in Java AG and its extensions

Types and subtyping are important aspects of Java and therefore in extensions to Java. Thus the host language specification must provide the mechanisms for examining the types of expressions, defining new types, specifying new subtype relationships, and checking if one type is the subtype of another. For example, if a new type for complex number is specified in an extension how can this new type interact with existing host language types? How can we specify that the Java `double` type is a subtype of the new complex number type? How can this be done in a composable manner? This section shows the Java AG for examining and creating types through the specification of two composable extensions: the enhanced-for loop in Java 1.5 and a complex number type.

Enhanced for-loop extension: Here, we show how to add the enhanced `for` loop from Java 1.5 as a composable extension to the Java 1.4 AG specification. This extension is quite simple and not very compelling since the construct now exists in Java 1.5. However, we discuss it because its semantics are well understood and it illustrates several concepts in writing extensions and using parts of the type checking infrastructure of the Java AG. The construct allows the programmer to specify loops that iterate over the members of any array

```
grammar edu:umn:cs:melt:java14;

start nonterminal CompilationUnit ;
nonterminal Expr, Stmt, Type, TypeRep ;
terminal Id / [a-zA-Z][a-zA-Z0-9_]* / lexical precedence = 5 ;
terminal While_t 'while'  lexical precedence = 10 ;

synthesized attribute pp :: String occurs on Expr, Stmt, Type ...;
synthesized attribute errors :: [ String ] occurs on Expr, ... ;
synthesized attribute typerep :: TypeRep occurs on Type, Expr ;
synthesized attribute hostStmt :: Stmt occurs on Stmt ;
synthesized attribute hostExpr :: Expr occurs on Expr ;
synthesized attribute hostType :: Type occurs on Type ;

concrete production while
s::Stmt ::= 'while' '(' cond::Expr ')' body::Stmt
{ s.pp = "while (" ++ cond.pp ++ ") \n" ++ body.pp ;
  cond.env = s.env ;    body.env = s.env ;
  s.errors = case cond.typerep of
               booleanTypeRep() => [ ]
             | _ => [ "Error: condition must be boolean"]
             end  ++ cond.errors ++ body.errors ;
 s.hostStmt = while(cond.hostExpr,body.hostStmt);   }

concrete production local_var_dcl  s::Stmt ::= t::Type id::Id ';'
{ s.pp = t.pp ++ " " ++ id.lexeme ++ ";"
 s.defs = [ varBinding (id.lexeme, t.typerep) ] ;
 s.hostStmt = local_var_dcl(t.hostType,id); }

concrete production idRef  e::Expr ::= id::Id
{ e.typerep = ... extacted from e.env ... ;
  e.errors = ... ;    e.hostExpr = idRef(id); }

synthesized attribute superTypes :: [ SubTypeRes ] collect with ++ ;
attribute supreTypes occurs on TypeRep ;

concrete production doubleType  dt::Type ::= 'double' ;
{ dt.pp = "double";  dt.typerep = doubleTypeRep() ;   }

abstract production doubleTypeRep   dtr::TypeRep ::=
{ tr.name = "double"  ; tr.superTypes := [ ] ;    }

abstract production arrayTypeRep   atr::TypeRep ::= elem::TypeRep
{ tr.name = "array"  ; tr.superTypes := [ ] ; }
```

Fig. 4. Simplified Java host language Silver specification.


```
ArrayList herd = ... ;
for (Cow c: herd) {
  c.milk ();
}
```

(*a*)

```
ArrayList herd = ... ;
for (Iterator _it_0 = herd.iterator();
     _it_0.hasNext();)  {
 Cow c = (Cow) _it_0.next();  c.milk();    }
```

(*b*)

Fig. 5. Use of enhanced-for statement (*a*), and its translation to Java 1.4 (*b*).

or expression of type `Collection`[1] that implement the `Iterator`, `hasNext` and `next` methods. Figure 5(a) shows a fragment of code that uses the enhanced `for` construct. It uses forwarding to translate this code into the Java 1.4 code in Figure 5(b) that makes explicit calls to methods in the `Collection` interface.

The grammar module `edu:umn:cs:melt:java14:exts:foreach` contains the specification of the enhanced-for extension and defines a new production and attribute definitions. Figure 6 shows part of the specification. This grammar imports the host Java language grammar since it utilizes constructs and attribute defined in that grammar. The concrete production `enhanced_for` defines the concrete syntax of the new construct and provides explicit definitions for the `pp` and `errors` attributes. The extension does not declare any new terminal, nonterminals or attributes but uses those defined in the host language. The definition of the `pp` attribute is straightforward and is used in the generated error message if a type-error exists. This occurs if the type of the expression `coll` is not a subtype of the `Collection` interface or is not an array type. This check is realized by examining the node (stored in local attribute `st_res` of type `SubTypeRes`) returned by the `subTypeCheck` function and pattern matching the type against the `arrayTypeRep(_)` tree. The `SubTypeRes` node is decorated with a boolean attribute `isSubType` that specifies if `coll.typerep` is a subtype of the `TypeRep` for Java Collection interface returned by the helper function `getTypeRep`.

The `subTypeCheck` function, the `SubTypeRes` nonterminal, and productions for building trees of this type are part of the Java host language AG framework that extension writers uses to access the typing information maintained by the host language. This use of semantic information at the language extension level distinguishes AG based extensions from macros.

The enhanced `for` production also uses *forwarding* [24], an enhancement to higher-order AGs, to specify the Java 1.4 tree that the node constructed by the `enhanced_for` production translates to. Pattern matching and `st_res` are used again here in determining whether the enhanced `for` translates to a `for` loop with iterators (stored in the local attribute `for_with_iterators`) or a `for` loop that increments an integer index to march across an array (stored in the attribute `for_over_array`). The definitions of these attributes are elided here but use host language productions and the trees `t`, `id`, `coll`, and `body` to construct the appropriate tree as expected. The value of `for_with_iterators`

[1] Java 1.5 introduces the `Iterable` type for use in the enhanced `for` loop. Since `Collection` is the similar type in Java 1.4, we use it here.

```
grammar edu:umn:cs:melt:java14:exts:foreach;
import edu:umn:cs:melt:java14;

concrete production enhanced_for
f::Stmt ::= 'for' '(' t::Type id::Id ':' coll::Expr ')' body::Stmt {
 f.pp = ...
 f.errors = if st_res.isSubType || match(e.typerep,arrayTypeRep(_)) then [ ]
  else [ "Enhanced for " ++ f.pp ++ " must iterate over Collections or arrays."];
 forwards to if st_res.isSubType then for_with_iterators
             else case e.typerep of
                     arrayTypeRep(_) => for_over_array
                   | _               => skip()        end ;
  local attribute st_res :: SubTypeRes ;
  st_res = subtypeCheck(coll.typerep, getTypeRep("Collection" ) ;
  local attribute for_with_iterators :: Stmt ;
  for_with_iterators = ... ;
  local attribute for_over_array :: Stmt ;
  for_over_array = ... ;
}
```

Fig. 6. Silver code that implements the enhanced `for` statement

for the enhanced `for` in Figure 5(a) can be seen in Figure 5(b).

Forwarding provides an implicit definition for all synthesized attributes not explicitly defined by the production. When an enhanced-for node is queried for its `host_Stmt` attribute (its representation as a host language construct) it forwards that query to the pure Java 1.4 construct defined in the `forwards to` clause. That tree then returns its translation. The Java 1.4 tree does not simply return a copy of itself since its children, `body` for example, may contain instances of extension constructs that must be "translated away" in a similar fashion.

Complex numbers Adding a new complex number type as a language extension requires specifying new type expressions and adding new subtype relationships – in this case, specifying that the Java `double` type is a subtype of the introduced complex type.

So that complex numbers types can be present in, for example, local variable declarations, the complex number extension grammar defines the terminal symbol `Complex_t` which matches the single string "complex" and the production `complexType` seen in Figure 7. The complex type constructs are translated via forwarding to references to a Java class `Complex` that implements complex numbers. This is packaged with the language extension. The type representation `typerep` of complex numbers is constructed by the production `complexTypeRep`. The production `complexLiteral` specifies complex number literals. It defines such expressions to have the type (`typerep`) of Complex numbers (`complexTypeRep()`) and checks that `r` and `i` have the correct type. The


```
grammar edu:umn:cs:melt:java14:exts:complex ;
import  edu:umn:cs:melt:java14;
terminal Complex_t 'complex' ;
concrete production complexType  t::Type ::= 'complex'
 { t.typerep = complexTypeRep();  forwards to ``Complex number class'' }
concrete production complex_literal
c::Expr ::= 'complex' '(' r::Expr ',' i::Expr ')'
 { c.pp = "complex (" ++ r.pp ++ "," ++ i.pp ++ ")" ;
   c.errors = ... check that r and i have type double ... ;
   forwards to ``new Complex( r, i)'' ;   }

abstract production complexTypeRep  tr::TypeRep ::=
 { d.superTypes := [ mkComplexToComplex() ] ;  }
aspect production doubleTypeRep   d::TypeRep ::=
 { d.superTypes <- [ mkDoubleToComplex() ] ;  }
abstract production mkDoubleToComplex  t::SubTypeRes ::=
 { t.isSubType = true ;   t.supertype = complexTypeRep() ;
   t.converted = complex_literal(``complex'', ``('', t.toConvert,
                                 ``,'', ``0.0'', ``)'');  }
```

Fig. 7. Portion of complex number language extension specification.

definition of the `errors` attribute is omitted here but it uses pattern matching as was done earlier in the `while` loop in Figure 4.

Adding new subtype relationships: To achieve a natural, close integration of the extension and the host language, the host language AG provides mechanisms for specifying new subtype relations and means for run-time conversion. Host language productions like the assignment production utilize this information to implement the complex number translation steps illustrated in Section 2.

The aspect production `doubleTypeRep` in Figure 7 adds a new super type to Java doubles. This is realized by contributing a new *super type* element (of type `SubTypeRes`) (constructed by production `mkDoubleToComplex`) to the collection attribute `superTypes` that decorate `TypeRep`s. The information in each *super type* in `doubleTypeRep`'s `superTypes` is used by the Java AG function `subTypeCheck` to determine if `doubleTypeRep` is a subtype of another specified type. If so, the function returns the super type (`SubTypeRes`) tree, which includes functionality for run-time conversion, if needed. The `mkDoubleToComplex` production builds a tree that sets the `isSubType` attribute to *true* and the `supertype` attributes to the complex number `TypeRep`. It plays the role of a function that creates the `Expr` tree that performs the run-time conversion of a yet-to-be-specified input `double`-typed `Expr`. The inherited attribute `toConvert` plays the role of the function input and the synthesized attribute `converted` plays the role of the output. It is set to the complex literal constructed from the input double expression and the literal `0.0`. The terminal symbols and literal

0.0 are shown as strings in the stylized definition of converted. In the actual specification they need to properly typed terminals and Expr nonterminals.

```
grammar edu:umn:cs:melt:java14 ;
nonterminal SubTypeRes ;
synthesized attribute superType :: TypeRep ;
synthesized attribute converted :: Expr ;
inherited attribute toConvert :: Expr ;

concrete production assign
a::StmtExpr ::= lhs::Expr '=' rhs::Expr ';'  {
 a.pp = lhs.pp ++ " = " ++ rhs.pp ++ ";"  ;
 local attribute res :: SubTypeRes ;
 res = subTypeCheck(lhs.typerep,rhs.typerep);
 res.toConvert = rhs ;

 a.errors = if length(a.transformed) == 1 then [ ]
       else if length(a.transformed) == 0 then [...type error...]
       else [... internal error, multiple translations ....];
 local attribute transformed :: [ StmtExpr ] collect with ++ ;
 a.transformed := if ! res.isSubType  then [ ]
                  else [ converted_assign(lhs, res.converted) ] ;
 forwards to if length (a.transformed) != 1 then skip()
             else head (a.transformed) ;
}
abstract production converted_assign  a::StmtExpr ::= l::Expr r::Expr {...}
```

Fig. 8. Portion of Java host language specification.

To see how this information is used, consider the host language assignment production assign specified in Figure 8. It calls subTypeCheck to check if the type of the expression on the right hand side (rhs.typerep) is a subtype of the type of the left hand side expression (lhs.typerep). The synthesized attribute converted and inherited attribute toConvert, both of type Expr act as the function described above that creates the expression tree whose type is lhs.typerep and performs any run-time conversion on the rhs to give it the type of lhs. For Java 1.4 classes, res.converted is the same tree as res.toConvert since no source-level conversion are needed. This provide the extension point or "hook" that language extensions that introduce new types will use.

The assign production forwards to an assignment in which the run-time conversion is applied to the rhs tree. This assignment is created by the converted_assign production. The transformed attribute is a collection attribute that holds the trees to which the initial assignment assign will forward. If lhs is a subtype of rhs then the new assignment is added to the list, otherwise it is initially empty. If transformed has exactly one element, then no errors are generated. If it has zero elements, then the lhs and rhs are not compatible. It is possible

for aspect productions on assign to add new elements to transformed (as the autoboxing/unboxing extension does below) so that if there is more than one element, then an error is raised since a decision cannot be made as to which one to forward to. If transformed has exactly one element, then that is what

```
aspect production assign  a::StmtExpr ::= lhs::Expr '=' rhs::Expr ';'
{ a.transformed <- if match(lhs.typerep, getTypeRep("Integer") &&
                      match (rhs.typerep, intTypeRep())
       then [ converted_assign ( l, ``new Integer ( rhs )'') ] else [ ]; }
```

Fig. 9. Portion of the autoboxing/unboxing extension specification.

assign forwards to – otherwise and error will have been raised and it forwards to the skip statement.

Complex numbers can be implemented as reference types (so that references instead of values are copied) or as primitive types. The extension designer makes this decision in implementing mkComplexToComplex. The converted attribute implement either option. Similar productions for other "copy" operations (parameter passing) exist in the Java AG and can be similarly extended.

The definition of the host language assignment is not trivial - but the complexity arises because it is designed to be extended later and we want the definitions of the extensions to be the simpler ones to write. If there is no discipline in adding new elements to the subtype relationship, then extension composition could introduce non-trivial circular subtype relations. This can be avoided if extension writers adhere to the guideline of not adding a set of subtype relations in which a host type as both the subtype and the supertype.

Autoboxing/unboxing Java 1.5 autoboxing and unboxing can also be added as an extension to the Java 1.4 host language. This is done by adding to the transformed collection attribute on the assign production in Figure 8. The aspect production in Figure 9 adds the boxing of primitive type int to class Integer by checking that lhs and rhs have the appropriate types. When they do, the transformed attribute in assign will contain only the assignment constructed by the converted_assign production and the use of the Integer class constructor shown in the contribution to the attribute. (The <- operator adds the value of the following expression to the collection attribute, folding up all such values using the specified collect with operator, ++ in the case of transformed.) Note that this production is not adding to the subtype relationship, but simply overloading the assignment operator when lhs and rhs have the specified types. Overloading of other operators is accomplished in a similar manner.

```
nonterminal EnvItem ;
synthesized attribute env_items :: [ EnvItem ];
nonterminal Scope with scope_type, env_items;
synthesized attribute defs :: [ EnvItem ] ;
inherited attribute env :: [ Scope ];
abstract production varBinding
e::EnvItem ::= name::String  dcl::TypeRep {...}

function addLocalScope
[Scope] ::= items::[EnvItem] enclosing_env::[Scope] {...}

-- lookUp returns typereps of name from nearest matching enclosing scope
function lookUpVar [ TypeRep ] ::= name::String env::[ Scope ] {...}
```

Fig. 10. The API of the JLE environment.

3.3 Using and extending the Java environment

Semantic analysis performed on a node in the syntax tree often requires information from another node. For example type-checking a variable expression requires information from the syntax tree node where the variable was declared. Similarly, access to an object might depend on its permission level, again set on its declaration. Attribute grammars pass such information around the abstract syntax tree using a combination of synthesized and inherited attributes.

In JLE, declarations for the environment are collected using the synthesized attribute `defs`, a list of environment items (`EnvItem`). These are passed up to the tree until a production that defines a new local scope, such as the enhanced-for production in Figure 6. It adds the definitions to the inherited environment attribute `env` that is passed down the tree for use by, for example, variable references. In the enhanced-for production, the following attribute definition performs this task:

```
body.env = addLocalScope([varBinding(id.lexeme,t.typerep)], f.env);
```

The `varBinding` production is used to create the data-structure binding the name of the identifier to its type representation. (In practice, more information than just the identifiers type is needed, but we've omitted those details here.) Figure 10 contains a partial specification of the nonterminals, productions, and functions used to pass declaration information to the parts of the syntax tree where it is used. It is important that language extensions, such as the enhanced-for, can contribute to this process.

The `env` attribute is a list of scopes; each stores bindings of various kinds for a particular scope in the object program as a list of `EnvItem`s in the attribute `env_items`, as seen in Figure 10. These bindings are represented by trees created by different productions, `varBinding` being one example. The environment contains scopes for top level declarations in the current file, declarations from other files in the same package, single-type named imports, and on-demand imports.

Other scopes may be created and added to when needed, for example within methods and inner class definitions.

While the enhanced-for loop only uses existing host language constructs for manipulating the environment, other extensions may want to add new kinds of information to the environment. The SQL extension extends the environment to contain the type representations of the tables and columns defined in the `import table` constructs shown above. As the environment is defined, the specifications in the enhanced-for loop and the SQL extension work together so that an SQL query enclosed in an enhanced-for loop can extract from the environment the definitions added by any `import table` constructs.

```
grammar edu:umn:cs:melt:java14:exts:sql ;
import  edu:umn:cs:melt:java14 ;

concrete production sqlImport
s::Stmt ::= 'import' 'table' t::Id '[' columns::SqlColTypes ']' ';' {
 s.defs = [ varBinding (t.lexeme, tableTypeRep (columns.defs)) ];   }
abstract production tableTypeRep   tr::TypeRep ::= col::[EnvItem]  { ... }
inherited attribute sql_env :: [EnvItem];
concrete production sqlQuery
e::Expr ::= 'using' c::Conn 'query' '{' q::SqlQuery '}' { ... }

concrete production sqlSelect
q::SqlQuery ::= 'SELECT' flds::SqlExprs 'FROM' t::Id  'WHERE' cond::SqlExpr {
 local attribute result :: [ TypeRep ] ;
 result = lookUp (table.lexeme, q.env);
 q.errors = flds.errors ++ cond.errors ++
  ...ensure that result has length 1 indicating precisely 1 decl for t... ;
 columns = if lenght(result) == 1
           then case (head(result)).typerep of
                tableTypeRep (cols) => cols
                | _                 => [ ]; -- error raised above.
           else [ ] ;
 flds.sql_env = columns;   cond.sql_env = columns;
 cond.env = q.env;          }
```

Fig. 11. Portion of Silver specification of the SQL extension

Figure 11 shows a small portion of the Silver specification of the SQL language extension. Of interest is that the import-table construct adds to the environment (via `defs`) a variable binding with a new kind of `TypeRep` tree constructed by the the production `tableTypeRep` defined in the SQL grammar. This information propagates up the syntax tree through host language defined productions to an enclosing scope-defining production that adds this information to an inherited `env` attribute. From here, it flows down the tree, through the `sqlQuery` construct to a `sqlSelect` construct which uses the `lookUpVar` function to get the list of type representations bound to `t` in the nearest enclosing scope that binds `t`. This list should have length 1, otherwise an error is generated and `columns` will be the empty list. The local attribute `column` is a simplified SQL environment `sql_env` built as a list of `EnvItem`s that are passed to the fields `flds` and condition `cond`.

The SQL extension uses pattern matching to extract the value for `columns` from the `TypeRep` bound to `t`. The use of the production name `tableTypeRep` which is defined in this grammar and is not visible to other extensions ensures that the value extracted is what the SQL extension added in the `sqlImport` production. Although other extensions may inappropriately remove bindings from an environment they may not subtly alter its contents in a manner that is undetected by the SQL extension. Similarly, since the production `tableTypeRep` is unknown to other extensions, any information contained in the type representation is not accessible to other extensions. This ensures that values added to the environment by one extension construct (SQL import-table) are passed correctly to another construct (SQL query) even if they pass through constructs defined in a different extension (enhanced-for). Extensions may incorrectly remove elements from the environment, but this type of mistake is dramatic and tends to arise in all uses of the offending extensions.

4 Composition of Language Extensions

4.1 Composing host language and extension specifications

The declarative syntax and specifications in Silver are easily composed to form the specification of a new extended language. In Figure 12 is the Silver specification for Java extended with the SQL and the computational geometry extension that implements the randomized linear perturbation (`rlp`) scheme for handling data degeneracies in geometric algorithms. What the `rlp` extension does specifically is not of interest here. This composed extended language has features to support both the domains of relational database queries and computational geometry. The `import` statements import the grammar specifications in the named Silver module. The `with syntax` clauses import the concrete syntax specifications from the named modules to build the parser for the extended language. The `main` production is similar in intent to the C `main` function; here it delegates to the main production `java_main` in the `java14` host specification. The `parse` value passed to `java_main` is the parser constructed from the concrete syntax specifications imported into the module. This is a boiler-plate Silver specification that can easily enough be generated from the names of the extension grammar modules which are to be imported into the Java 1.4 host language.

4.2 Issues in Composition

Even though the process of composing language specifications, for both syntax and AG-based semantics, is a straightforward union of the components and


```
grammar edu:umn:cs:melt:composed:java_sql_cg ;
import core ;

import edu:umn:cs:melt:java14 with syntax ;
import edu:umn:cs:melt:java14:exts:sql with syntax ;
import edu:umn:cs:melt:java14:exts:rlp with syntax ;

abstract production main top::Main ::= args::String
{ forwards to java_main(args, parse) ;  }
```

Fig. 12. Composed language Silver specification.

can performed automatically, there may be no guarantee that the resulting language specification is well-defined. For example, an undisciplined composition of concrete syntax productions may lead to ambiguous grammars or may introduce conflicts (shift-reduce or reduce-reduce) into the parse tables for LR-style parsers. In terms of semantics, combining AG specifications may result in circular AGs in which an attribute instance may be defined such that its value depends on itself. If programmers are going to compose extended languages from the composable, modular language extensions proposed here, then detecting or better yet preventing these sorts of problems becomes critical. There are some analyses that can be performed at different times that provides some assurance that the composed language is well-defined.

The table in Figure 13 identifies tools and techniques that perform some analysis and points in time in which they are performed. The first two analyses listed along the top relate to syntax specifications, the last two to semantics. Analysis can be performed by the extension designer during development, it can be performed when the extensions are composed (perhaps by a programmer) with the host language specification, or it can be performed dynamically – during parsing or AG evaluation. (We do not consider analysis during execution of a program written in an extended language.) These analyses may differ for different tools and techniques and are thus not precisely the same (or even applicable) for all approaches. The table is meant to provide a framework for comparing approaches and understanding the goal of composability in extensible languages.

If an analysis can be performed when the extension designer is implementing an extension (at design time), then information from that analysis may be used by the designer to fix any discovered problems. Alternatively, an analysis can be performed when the a set of extension are composed with the host language. These analyses may prevent a programmer from using an ill-defined language, but this analysis may be too late as the person directing the composition of the extension may not be the extension designer and may not be able to make use of the information to fix the problem.

A language recognizer, traditionally a scanner, exhibits *lexical determinism* if for non-erroneous input it returns exactly one token. A recognizer, traditionally a parser, is *syntactically deterministic* if for non-erroneous input it returns

analyses → points in time ↓	lexical determinism	syntactic determinism	AG non-circularity	termination of tree construction
Ext. design	IPS	IPS		
Ext. composition		Yacc	Knuth [15], Vogt [30]	
Parse/ AG Eval.	SGLR	SGLR	JastAddII	

Fig. 13. Analyses and points in time for analysis.

exactly one syntax tree [15, 30]. An AG specification is *non-circular* if for any syntactically valid syntax tree, no attribute instance has a circular definition. (In AGs with forwarding a similar analysis [24] that ensures that each attribute instance will have exactly one non-circular definition (defined either explicitly or implicitly via forwarding) is used, but the distinction is not made here and both considered as "non-circular" analyses.) A general termination analysis may ensure that a finite number of trees are created (via forwarding or in higher-order attributes in AG systems, or in term rewriting systems).

Visser's scannerless generalized LR (GLR) parsers [28] do not have a separate scanner but parse down to the character level. For any context free grammar (including ambiguous ones) a GLR parser can be created that will parse any valid phrase in the language of the grammar and return all syntax trees corresponding to the phrase – for ambiguous grammars more than one tree may be returned for some phrases. Thus, one does not know until parse time if a single tree will be generated for a correct input phrase. Disambiguation filters [22] can be written to filter out undesired trees on ambiguous parses but there is no static guarantee for the grammar designer that the specified filters ensure syntactic determinism. Yacc [12] is deterministic and an analysis at composition time exists, but LALR(1) parsers are somewhat brittle in that seemingly innocuous additions to a grammar may cause conflicts. This problem is exacerbated when multiple extensions that add new syntax are combined. Thus, Yacc-like tools may not be suitable for extensible languages. In Section 4.3 we describe an integrated parsing and scanning system (IPS) that places some restrictions on the type of concrete syntax that extension designers can specify. These ensure at extension design that no conflicts are created when the host language syntax and syntax from other extensions observing the restrictions are composed

For attribute evaluation circularity/definedness tests exist for standard AGs [15], higher-order AGs [30], and AGs with forwarding [24]. These can be performed when grammars are composed, but are in essence whole-program analyses that require examining the entire grammar. JastAddII [6] is a Java-based AG system that employs reference attributes [10]. These can be thought of as attributes that contain pointers (references) to other nodes in the syntax tree and have been shown to be especially useful in linking a variable reference node to its declaration. There is no static analysis to ensure that such grammars are not circular however and thus JastAddII uses a dynamic check. This system has also been used to build an extensible Java 1.4 compiler and the lack of a static check may be less critical than in the case of parsing. Adding new constructs/productions

tend to not add new types of attribute dependencies and thus AGs are much less "brittle" and more easily extended than LALR(1) concrete syntax grammars.

Also of interest are analyses that ensure that a finite number of trees are constructed via forwarding or as higher-order attributes. Such an analysis combined with the circularity/definedness analysis provides a guarantee of termination during attribute evaluation.

4.3 Parser-context based lexical disambiguation

Our integrated parser/scanner system uses a standard LR parsing algorithm that is slightly modified in the manner in which it calls the scanner: it passes to the scanner the set of terminal symbols that are "valid lookahead" for the current state of the parser, *viz.*, those that have non-error entries (shift, reduce, or accept) in the parse table for the current parser state. Simply put, the parser passes out what it *can* match and gets back what *does* match. The scanner's use of the parser state for disambiguation allows the parser to determine what terminal a certain string matches based on the context.

Crafting the concrete syntax specification of an extension that when combined with the concrete syntax specification of the host language results in a deterministic LR specification is not trivial but can be accomplished by the extension designer. Our goal, during composition of several extensions with the host language, is to *maintain* the deterministic nature of the grammar. This characteristic is maintained by restricting the type of syntax that can be added in the extension and depends on the integrated nature of the parser and scanner. The parser-context based lexical disambiguation is key to the process.

The analysis that an extension, say E_1, will not introduce shift-reduce or reduce-reduce conflicts in the parse table of a language composed from the host and extensions E_1 and others, say $E_2, ..., E_n$ is based on an examination of the parser table of the host language and the parse table of the host composed just with E_1. The details of this analysis will appear in an upcoming technical report [27], but the essence of the analysis is that the extension E_1 is allowed to (i) add new states in the parser table that are used solely in parsing extension constructs and (ii) add a restricted set of items to existing host states that only allow a shift-action to a new extension-added state. The *critical characteristic* of the composed parse table is that the states are partitioned so that every state is associated with exactly one grammar: either the host or one of the extensions.

Each production introduced by extension E that has a host nonterminal, say H, on the left side must have a right hand side that begins with what is called a "marking token" for that extension. The marking tokens are used to obtain the partitioning of the parser states as described above. These are not to be referenced elsewhere in the extension. For example, in the SQL extension the concrete production `sqlQuery` with signature `Expr ::= 'using' Conn 'query' '{' SqlQuery '}'` the marking token is `'using'`. These productions provide the shift-action, based on the marking token, mentioned above that take the parser from a "host state" to an "extensions state".

Additionally, restrictions are made on "back references" to host language nonterminals on the right hand side of extension productions. If they are not followed, the analysis is likely to fail. Each extension production with a host nonterminal H on the right-hand side should adhere to one of these two forms:

- $E \rightarrow \mu^L H \mu^R$, where μ^L is a host or extension terminal such that E does not derive $\mu^L X$ for any $X \neq H\mu^R$, and μ^R is a host terminal, preferably one that would be commonly found after an H-expression, such as a right parenthesis after a math expression.
- $E \rightarrow \mu^L H$, where μ^L is under the same restrictions as before, and the H is when derived invariably at the end of the extension expression — *e.g.*, the loop body in an extension defining a the enhanced-for loop.

The goal of these restrictions is to limit the ways that extensions can affect the host language parse table so that conflicts are not introduced. Because JLE provides mechanisms for overloading the operator symbols, the syntax specifications of most extensions (such as the SQL extension) meet these restrictions.

But due to the partitioning of the LALR(1) DFA described above, in a context-filtered system, most extension terminals will not show up in the same context as a host terminal; for example, in our SQL extension, as only host states have Java identifiers in their valid lookahead and only SQL extension states have the keyword `select`, the two tokens are never together in the valid lookahead set of any parser state. Hence, no disambiguation is needed between `select` and Java identifiers. Thus, "select" can still be used as a variable name in the Java code (but not as a column name in the SQL query.

It is worth noting that even if extensions do not pass these tests, the integrated parser/scanner approach has, in our experience, been much less brittle than standard LALR(1) approaches since it avoids many of the lexical ambiguities that would exist in traditional disjoint approaches. In fact the set of grammars that are deterministic in the integrated approach is strictly larger than those that are classified as LALR(1). Thus, performing the deterministic check at composition time is more likely to succeed. The enhanced-for loop production shown in Figure 6 does not meet the above restrictions because it does not introduce an extension defined marking token but instead reuses the host language defined `'for'` terminal. Yet in composing several different extensions this production has not introduced any conflicts into the parser table. Additional details are available in the technical report [27].

The key to checking for lexical determinism is the partitioning of parser states into states associated with the host or a single extension grammar. This restricts the type of lexical conflicts (terminals with overlapping defining regular expressions) to one of two types. The first type is a conflict between an extension defined terminal and a host language defined terminal or another terminal defined in the same extension. In this case, the extension writer can use several techniques to resolve the ambiguity. IPS provides several precedence setting constructs and a means for writing Silver expressions to perform the disambiguation at parse time. We will not go into these details here; the key point is that the extension writer can be made aware of the ambiguity at extension design time and fix it. This disambiguation is then maintained during composition.

The second possible type of conflict is between the marking tokens of two different extensions. These are unavoidable and must be resolved "on the fly" by the programmer. We introduce the notion of *transparent prefixes* to enable the programmer to do this without knowledge of the language grammars. Transparent prefixes allow a disambiguating prefix (typically based on the name of the extension) to precede the actual lexeme of tokens in the program, without being visible to the parser. This prefix then directs the scanner to recognize the following lexeme as coming from that grammar. This approach is motivated by the use of fully qualified names in Java in which classes with the same name from different packages are distinguished by specifying the package name.

5 Discussion

5.1 Related Work

There have been many efforts to build tools for extensible compilers for Java and other languages. Some of these, like the JLE, are attribute grammar-based. Others are based on traditional rewrite systems or pass-based processors.

Attribute Grammar Based Tools: Much previous work has investigated the use of attribute grammars as framework for the modular specifications of languages [8, 13, 7, 1, 20]. There are also several well-developed AG-based language specification tools: such as LRC [16], JastAddII [6], and Eli [9]. These systems implement transformations in different ways, some are functional while others are object-oriented. They do not support forwarding and thus the modularity and ease-of-composition of language features specified as AG fragments is achieved by writing attribute definitions that "glue" new fragments into the host language AG. JastAddII and Eli do not have the general purpose features of pattern matching and polymorphic lists in Silver and instead use a "back-door" to their implementation languages (Java and C) for general-purpose computations.

JastAddII [6] is based on rewritable reference attribute grammars and has been used to develop an extensible Java 1.5 compiler [5]. To the best of our knowledge Silver and JastAddII are the only AG systems that allow for the implicit specification of semantics by translation to a host language. JastAddII does so by the application of (destructive) rewrite rules. But since rewriting of a subtree takes place before attributes may be accessed from the that tree, one cannot both explicitly and implicitly specify a construct's semantics. For example, consider the enhanced-for loop extension described in Section 3.2. With JastAdd II, any semantic analysis is performed only after rewriting is done and the equivalent host language for-loop is generated. Thus all semantics are implicit (except for the attributes that are computed during rewriting that may be used to guide rewriting). With forwarding, rewriting is non-destructive. Extensions and forwards-to trees exist side-by-side allowing some semantics to be specified explicitly by the extension while others are specified implicitly via forwarding. On the other hand, the rewrite rules in JastAddII are more general than forwarding and can be used in a wider of variety of language processing applications, such as using rewrite rules to implement optimizing transformations. In Silver, such transformations must be encoded as definitions of higher-order attributes.

Other Approaches to Extensibility: Embedded domain-specific languages [11] and macro systems (traditional syntactic, hygienic and programmable [31]) allow the addition of new constructs to a language but lack an effective way to specify semantic analysis and report domain specific error messages. Meta-object protocol systems [14] provide limited opportunities to add new language constructs but can perform abstract syntax based optimizations and check for certain errors, as can some modern macro systems, e.g. [2]. Traditional pass-based approaches such as Polyglot [17] require an explicit specification of the order in which analysis and translation passes are performed on the syntax tree. Requiring this level of implementation level detail to compose extensions is what we hope to avoid.

JavaBorg is an extensible Java tool that uses MetaBorg [4], an embedding tool that allows one to extend a host language by adding concrete syntax for objects. It is based on the Stratego/XT rewriting system [29] that allows for the specification of conditional abstract syntax tree rewriting rules to process programs. In addition, it allows for the specification of composable rewrite strategies that allow the user to program the manner in which the rewrites are performed. These rewrites can be used to perform generative as well as optimizing transformations – both general purpose and domain-specific. Rules and strategies may be bundled into libraries and composed. MetaBorg is well-suited for performing transformative optimizations since its rewrites are destructive and performed in-place. However, specifying semantic analyses like error checking, even when using dynamically generated rules [19], is less straight forward than using attributes. It is also not clear that different extensions can be so easily combined. MetaBorg uses scannerless GLR parsers [28].

Intentional Programming originated in Microsoft Research and proposed forwarding in a non-attribute grammar setting. More recent work [21] still employs generative programming techniques but it is not clear if forwarding is still used.

5.2 Ongoing and Future Work

JLE aims to report all errors that the traditional Java compiler would report. As of this writing there are two types of errors that are not yet reported. The first is Java definite assignment errors. We have built an extension to Silver that constructs control flow graphs for imperative programs and performs data flow analysis by model checking these graphs [26]. We are currently adding the necessary Silver specifications that use this extension to construct an extensible data-flow analysis framework that extension writers can use in analysis of language extension constructs. Second, a relatively few productions in the Java AG specification only propagate the `errors` attribute up the syntax tree but do not add their own errors. These are currently being completed.

Although many new Java 1.5 constructs can be specified as modular language extensions to Java 1.4 we have not yet done this for generics. The effect of adding generics is felt throughout the design of Java 1.5 and it is unclear if they can be "translated away" to Java 1.4 constructs via primarily local transformations.

We are also investigating additional analyses on AG-based extension specifications to provide some level of assurance that composed language specifications are well-defined. To mention just one, we are investigating analysis to check for the termination of tree construction via forwarding as described in Section 4.2. One analysis extracts rewrite rules from the attribute grammar specification in such a way that if the rewrite rules can be shown to terminate, then forwarding will terminate. From a production p of the form $X \to X_1 ... X_n$ that forwards to a tree of the form $p'(X_1, ..., X_n)$ (where p' is a tree with "holes" for $X_1, ..., X_n$) we extract the rewrite rule $p(X_1, ..., X_n) \to p'(X_1, ..., X_n)$. We have used approaches that place orderings on term constructors (productions in AGs) to show that forwarding terminates for some simple AG specifications extending this to more general AG specifications. The important point is that AGs provide a high-level domain-specific "vocabulary" for these types of analyses - that the specifications are at a higher level of abstraction means they can be more easily analyzed than if they were written in a general purpose language like Java.

5.3 Conclusion

One reasons that libraries are a successful means for introducing new abstractions, either as new classes in object oriented languages or higher-order functions in functional languages, is that the programmer can freely import the set of libraries that address his or her particular problem at hand. It is our belief that for language extension techniques (either those proposed here or others described in Section 5.1) to have real-world impact they must be composable in a manner that is similar to libraries. In this paper we have shown that it is possible to implement languages and *composable* language extensions so that new, customized, domain-adapted languages can be created from the host language and selected language extensions with no implementation level knowledge of the extension.

An important area of future work centers on means for ensuring, either by analysis of specifications or by restricting the types of extensions that can be described, that language extensions that have the look-and-feel of the host language can be easily composed by the programmer. We have also proposed a few analyses that begin this exploration but there is much work to be done.

References

1. S. R. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, Department of Elec. and Comp. Sci., UK, 1993.
2. D. Batory, D. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proc. 5th Intl. Conf. on Software Reuse*. IEEE, 2–5 1998.
3. J. T. Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

4. M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proc. of OOPSLA '04 Conf.*, pages 365–383, 2004.
5. T. Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Lund, Sweeden, 2006.
6. T. Ekman and G. Hedin. Rewritable reference attributed grammars. In *Proc. of ECOOP '04 Conf.*, pages 144–169, 2004.
7. R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars. In *19th ACM SIGPLAN-SIGACT Symp. on Prin. of Prog. Lang.*, pages 223–234, 1992.
8. H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programing*, 3(3):223–278, 1983.
9. R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *CACM*, 35:121–131, 1992.
10. G. Hedin. Reference attribute grammars. *Informatica*, 24(3):301–317, 2000.
11. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), 1996.
12. S. Johnson. Yacc - yet another compiler compiler. Technical Report 32, Bell Laboratories, July 1975.
13. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.
14. G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the MetaObject Protocol*. MIT Press, Cambridge, MA. USA, 1991.
15. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971) pp. 95–96.
16. M. Kuiper and S. J. Lrc — a generator for incremental language-oriented tools. In *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, 1998.
17. N. Nystrom, M. R. Clarkson, and A. C. Myer. Polyglot: An extensible compiler framework for java. In *Proc. 12th International Conf. on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152. Springer-Verlag, 2003.
18. M. Odersky and P. Wadler. Pizza into java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–159. ACM Press, 1997.
19. K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *Proc. 14th Intl. Conf. on Compiler Construction*, volume 3443 of *LNCS*, pages 204–220. Springer-Verlag, 2005.
20. J. Saraiva and D. Swierstra. Generic Attribute Grammars. In *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204. INRIA rocquencourt, 1999.
21. C. Simonyi, M. Christerson, and S. Clifford. Intentional software. *SIGPLAN Not.*, 41(10):451–464, 2006.
22. M. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Computational Complexity*, pages 143–158, 2002.
23. E. Van Wyk, D. Bodin, and P. Huntington. Adding syntax and static analysis to libraries via extensible compilers and language extensions. In *Proc. of LCSD 2006, Library-Centric Software Design*, 2006.

24. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.
25. E. Van Wyk and E. Johnson. Composable language extensions for computational geometry: a case study. In *Proc. 40th Hawaii Intl' Conf. on System Sciences*, 2007.
26. E. Van Wyk and L. Krishnan. Using verified data-flow analysis-based optimizations in attribute grammars. In *Proc. Intl. Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, April 2006.
27. E. Van Wyk and A. Schwerdfeger. Parser context based lexical disambiguation. Technical report, Univ. of Minnesota, 2007. To appear. See `www.melt.cs.umn.edu/ips`.
28. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
29. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger-Verlag, June 2004.
30. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *ACM PLDI Conf.*, pages 131–145, 1990.
31. D. Weise and R. Crew. Programmable syntax macros. *ACM SIGPLAN Notices*, 28(6), 1993.
32. S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC API Tutorial and Reference: Universal Data Access for the Java 2 Platform*. Addison-Wesley Professional, 2nd edition, 1999.