Semantic Analysis of Pict in Java

Darius Antia and Franck van Breugel

Technical Report CS-2003-10

October 2003

Department of Computer Science

4700 Keele Street, Toronto, Ontario M3J 1P3, Canada

Semantic Analysis of Pict in Java

Darius Antia and Franck van Breugel

York University, Department of Computer Science
4700 Keele Street, Toronto, Canada M3J 1P3

dariusa@cs.yorku.ca and franck@cs.yorku.ca

October 2003

Abstract

We focus on mechanically generating object oriented code for semantic analysis. As an example, we consider scope checking of a compiler of which the source language is Pict and of which the target and implementation language is Java.

1 Introduction

A compiler generator mechanically generates a compiler from a series of specifications. This requires developing small specification languages for each major compilation phase. The purpose of such languages is to allow a compiler writer to specify each phase in a manner that is lucid and precise. Accompanying each such specification language is a tool that can mechanically convert a specification into code. The ability to transform high level specifications into implementation level code eliminates a large amount of tedious manual coding. This consequently enhances maintainability since changes to the definition of the language only need to be reflected in the easily modifiable specifications. This approach is common practice to generate lexers and parsers. Here, we focus on the semantic analysis phase. For a more detailed discussion about compiler generators, we refer the reader to, for example, Tofte's text [Tof90].

In the development of our compiler we want that both the code we write by hand, and the code we mechanically generate be highly object oriented, that is, we try to leverage core object oriented concepts including encapsulation, inheritance and polymorphism as much we can. In several object oriented compiler frameworks, these concepts are not considered in the interest of programmer convenience. In our work we aim to steadfastly adhere to the object oriented paradigm, while at the same time retain, or possibly even enhance, programmer convenience. It is a well documented fact that object oriented code is easily maintainable (see, for example, [GHJV94]). However, the main advantage of the mechanically generated code being object oriented is robustness. Encapsulation, for example, limits the scope of what the generated code is capable of doing. Often times, errors in the generation process, which would otherwise have gone unnoticed, show up as encapsulation violations in the resulting object oriented code. Polymorphism eliminates the need for case analysis, contributing not only to robustness, but also to the overall ease of generating the code.

Using Java as the target and implementation language allows us to experiment with object oriented compiler generation techniques. Another reason for choosing Java as our implementation language is that our compiler is capable of running on all major platforms. Additionally, the current popularity of Java makes the inner workings of our compiler accessible to a larger audience. Considering that our compiler can run on numerous different platforms, it behooves us to have our compiled code run on those platforms too. We therefore chose the target language of our compiler to be Java as well. The widespread availability of Java makes our compiler and the code it produces potentially platform independent.

As an example, we consider part of the semantic analysis of Pict. This language was introduced by Pierce and Turner. For an overview of the language we refer the reader to [PT98b, PT00]. Pict was built on top of the π-calculus. This calculus extends the Calculus of Communicating Systems, also known as CCS, with mobility. For a detailed account of the π-calculus we refer the reader to [Mil99]. The main reasons for choosing Pict are the facts that the semantic analysis of Pict is far from trivial and that most parts of the semantic analysis are defined formally (see [PT98a] for more details). In particular, we concentrate on scope checking, that is, checking the scoping constraints. These constraints are formally defined in terms of a proof system. We present some of the scoping rules in Section 3.1. As we will see in Section 3.4, these high-level rules can be mechanically translated into low-level Java code.

To carry out semantic analyses like scope checking, a program is usually represented as a syntax tree. In Section 2 we discuss different ways of implementing syntax trees in Java. We consider two different contemporary approaches to syntax tree representation. In particular, we compare the approaches taken by ANTLR [Mag] and JavaCC [MS] with those taken by SableCC [Gag98, GH98] and Zephyr ASDL [WAKS97]. Roughly, in the former approach all the nodes of a syntax tree are instances of one and the same class, whereas the latter approach utilizes a different class for every kind of node of a syntax tree. Clearly, in the latter approach a large number of classes need to be introduced. A number of tools have been developed to mechanically generate these classes which form large inheritance hierarchies. In our compiler, we often deal with more than 200 such classes at the same time. Clearly, in such a situation type conversions are frequent and may lead to runtime failures. We try to minimize the use of downcasting, which is supported by our implementation language. For example, our code for scope checking does not use downcasting at all. In other phases of our compiler where we are forced to use downcasting, we are able to ensure that such casts never fail at runtime.

Semantic analysis entails traversing the syntax tree, that is, visiting the nodes of the tree in a suitable order. Different analyses may well visit the nodes in different orders. This is, for example, the case for scope checking and kinding (checking if type expressions are well-formed) of Pict programs. In Section 3, we present different ways of implementing syntax tree traversals in Java. In particular, we consider implementing scope checking of Pict using the visitor design pattern in Section 3.2, SableCC's tree walker in Section 3.3, and pure object oriented techniques in Section 3.4.

2 Constructing Syntax Trees

When developing a representation of syntax trees in Java we have to address the following two fundamental questions. Should the nodes of the syntax tree be typed based on the semantic constructs they represent or should we use one and the same type for all nodes? What kind of methods should the nodes of the syntax tree provide?

2.1 Homogeneous versus Heterogeneous Trees

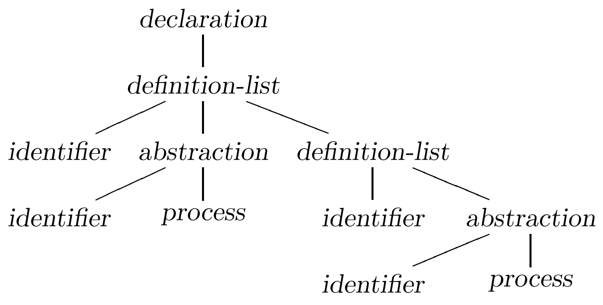
On the one hand, in a *homogeneous* syntax tree all the nodes are instance of one and the same class, say `Node`. On the other hand, a *heterogeneous* syntax tree utilizes a different type for every kind of node that it contains. This terminology is used in [Par]. Consider, for instance, the following fragment of the grammar of Pict.

$$\begin{array}{rcl} \textit{declaration} & \rightarrow & \texttt{def}\ \textit{definition-list} \\ \textit{definition-list} & \rightarrow & \textit{identifier abstraction} \\ & & \textit{identifier abstraction}\ \texttt{and}\ \textit{definition-list} \\ \textit{abstraction} & \rightarrow & \textit{identifier} = \textit{process} \end{array}$$

The parse tree of the Pict program

```
def x a = () and y b = (),
```

where `a`, `b`, `x` and `y` are *identifier*s and `()` is a *process*, would look like



On the one hand, if the above tree was represented by a homogeneous syntax tree, each node of the above tree would be represented by an instance of the class `Node`. On the other hand, if the tree was represented by a heterogeneous syntax tree, nodes of different kinds would be represented by instances of different classes. For example, the *identifier* nodes are represented by instances of the class `Id` and *abstraction* nodes by instances of the class `Abs`.

The main advantage of heterogeneous syntax trees over homogeneous syntax trees is the fact that the former are more robust than the latter. First of all, the type system of the implementation language can be exploited to constrain the shape of these syntax trees. For example, in the heterogeneous representation it is impossible to have an *abstraction* node with three children, simply because the class `Abs` would be defined to have only two instance variables, one of type `Id` and one of type `Proc`. Secondly, in the heterogeneous setting it is impossible to create a syntax tree that does not correspond to a parse tree. This is clearly not the case in the homogeneous setting. Thirdly, not every node implements the same interface. That is, different kinds of nodes may allow different kinds of operations to be performed on them. For example, an *input-channel* node can be checked to be well-kinded whereas an *abstraction* node cannot. Hence, the method `kindCheck` should be applicable to an object representing an *input-channel* node but not to an object representing an *abstraction* node. As we will see below, this can easily be accommodated in the heterogeneous approach. Finally, when traversing a syntax tree different kinds of nodes may need to be treated differently. More details about traversing heterogeneous syntax trees will be provided in Section 3.

In the heterogeneous setting we introduce a class for every kind of node. This large number of classes which need to be introduced—in the homogeneous setting we only need to introduce the class `Node`—may, at first, be viewed as a disadvantage of the heterogeneous approach. The syntax trees for our target language Pict, which is a relatively small language, utilize in excess of 100 classes. A typical implementation for Java would utilize over 500 such classes. However, as we will see in the next section, these classes can be generated mechanically.

We believe that heterogeneous syntax trees provide a far superior alternative to the homogeneous variant. SableCC [Gag98, GH98] and Zephyr ASDL [WAKS97] also use heterogeneous syntax trees. In JJTree, which is part of JavaCC [MS], homogeneous trees are the norm. ANTLR [Mag] is quite flexible on the matter and makes provisions for either kind of syntax tree.

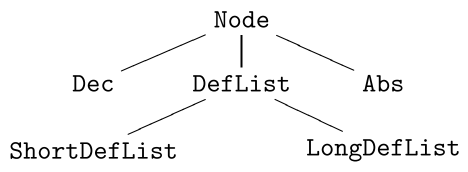
2.2 Mechanically Generating Classes

The classes of heterogeneous syntax trees can be mechanically generated from the specification of its syntax. Part of our specification of the syntax corresponding to the grammar presented in the previous section is given below.

```
  % Declaration %
Dec ::=  % defList : list of definitions %
        DefList
  % List of definitions %
DefList ::=  % List of one definition %
```

```
            ShortDefList
             % id : name of abstraction,
               abs : body of abstraction %
            Id Abs
          |  % List of more than one definition %
            LongDefList
             % id : name of first abstraction,
               abs : body of first abstraction,
               tail : tail of the list %
            Id Abs DefList
 % Abstraction %
Abs ::=  % id : name of parameter,
           proc : process %
        Id Proc
```

From the above specification the classes `Dec`, `Abs`, `DefList`, `ShortDefList` and `LongDefList` can be mechanically extracted. These classes are part of the following inheritance hierarchy.



The class `Node` is a superclass of all the generated classes. This class contains information general to all the classes like the `position` within the original source file of the characters from which this `Node` of the syntax tree is derived (see, for example, [App98, page 101]). Note that the classes `Id` and `Proc` have not been defined in the above specification nor have they been included in the above inheritance diagram. A generated class is abstract if and only if it is not a leaf in the inheritance diagram (see, for example, [WB99, Section 4.4.1]). These classes correspond to the nonterminals in the grammar. By making these classes abstract—recall that abstract classes cannot be instantiated—we disallow creation of syntax trees corresponding to incomplete parse trees. The class `DefList` is abstract.

```
/**
   List of definitions.

   @see Node
*/
abstract class DefList extends Node {}
```

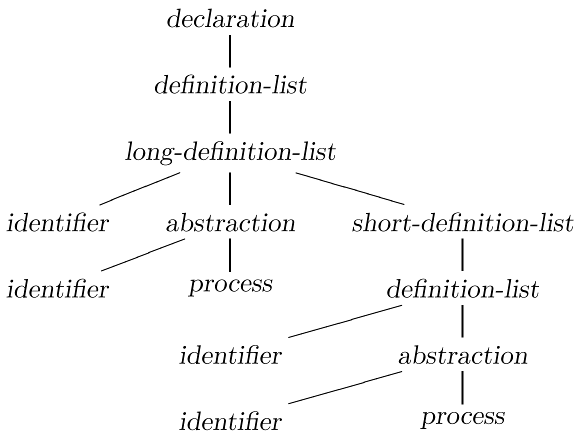
This abstract class has concrete subclasses `ShortDefList` and `LongDefList`.

```
/**
   List of more than one definition

   @see DefList
*/
class LongDefList extends DefList
{
  private Id id;
  private Abs abs;
  private DefList tail;
  /**
     List of more than one definition

     @param pos position within the source file of the characters from which this node is derived
     @param id name of first abstraction
     @param abs body of first abstraction
     @param tail tail of the list
  */
  LongDefList(int pos, Id id, Abs abs, DefList tail)
  {
    super(pos);
    this.id = id;
    this.abs = abs;
    this.tail = tail;
  }
}
```

Given a fragment of the Pict grammar, we constructed a syntax specification from which we mechanically extracted a hierarchy of Java classes. Not every grammar immediately gives rise to such an inheritance hierarchy. However, as we will discuss next, each grammar can be transformed into a grammar which does. A context-free grammar is in *inheritance normal form* if whenever there are multiple productions with the same left-hand-side then the right-hand-sides of those productions consist of a single nonterminal or terminal. Hence, given a nonterminal A of the grammar, we either have one production of the form $A \rightarrow \alpha_1 \ldots \alpha_n$ or we have one or more productions $A \rightarrow \alpha_1, \ldots, A \rightarrow \alpha_n$, where α_i is either a nonterminal or a terminal. In the former case, the class corresponding to A has instance variables of types corresponding to $\alpha_1, \ldots, \alpha_n$. The latter case, the class corresponding to A has subclasses corresponding to $\alpha_1, \ldots, \alpha_n$. Note that the grammar presented in the previous section is not in this format. However, each grammar can be transformed into a grammar in inheritance normal form. If we encounter multiple productions of the form $A \rightarrow \alpha_1 \ldots \alpha_n$ with the same left-hand-side, we introduce for each such production a fresh nonterminal, say A', and we replace the production $A \rightarrow \alpha_1 \ldots \alpha_n$ with the productions $A \rightarrow A'$ and $A' \rightarrow \alpha_1 \ldots \alpha_n$. When transforming the grammar of the previous section we introduce the nonterminals *short-definition-list* and *long-definition-list*. An important property of the transformation is that the derivation trees obtained from the grammar before transformation are similar to those obtained after transformation. A collapse of the latter gives rise to the former. For example, the derivation tree



can be collapsed to the one presented in Section 2.1.

It may be argued that the inheritance normal form is merely a special case of Chomsky normal form, and hence our transformation is redundant. However, a transformation to Chomsky normal form could easily result in a grammar whose derivation trees bear no resemblance to those in the original grammar. The notion of similar derivation trees is very important to us since a grammar is structured the way it is for a purpose, and significantly altering its structure defeats that purpose.

Since Java does not support multiple inheritance, we may need to apply another transformation to the grammar. A context-free grammar is in *single inheritance normal form* if no nonterminal appears all by itself as the right-hand-side of more than one production. Each grammar can be transformed into a grammar in single inheritance normal form. This transformation is similar to the one presented in the previous paragraph. The details of both transformations can be found in the first author's thesis [Ant].

Java does support interfaces, which can be used to roughly emulate multiple inheritance. It is therefore possible to represent a grammar in inheritance normal form using interfaces. However, we chose not to use interfaces, since they do not allow for as much code sharing as abstract superclasses do.

SableCC and Zephyr asdlGen also produce hierarchies of Java classes very similar to the ones we produce. Our syntax specification is more verbose than the ones used by SableCC and Zephyr asdlGen. The additional nomenclature and annotations present in our specification allows us to generate code with meaningful identifier names and JavaDoc comments. We agree with Wang et al. [WAKS97] that generated code should not only be understood by other tools but also by programmers.

2.3 Methods of Nodes of Syntax Trees

Next we address the following question. What kind of methods should the nodes of the syntax tree provide? There are two opposing views on this topic. The predominant view is that a syntax tree is simply a structural representation of a program, and hence does not need to contain any methods that are unrelated to structural information. Appel [App98, page 99] refers to this view as *syntax separate from interpretation*. Such syntax trees are devoid of any operations that interpret the tree (for example, perform semantic analysis). Therefore, a node in such a syntax tree may contain only methods such as `getChild`, but would not contain methods such as `scopeCheck`.

Contrasting with the above view is a more *object oriented* view, in which a syntax tree is viewed as an encapsulated and self contained entity. Its nodes therefore would be equipped with methods such as `scopeCheck`, but not with methods such as `getChild`. The rational being that we wish to hide structural details of the object and instead emphasize operations that manipulate it.

As we have seen, a syntax tree contains different kinds of nodes, such as *identifier* and *abstraction*. Furthermore, for each node we have different kinds of interpretations such as scope checking and type checking. On the one hand, the syntax separate from interpretation approach allows the different kinds of interpretations to be handled in a modular way. If we were to add for example type checking, we would simply add a class `TypeChecker` which handles type checking for all different kinds of nodes. In the object oriented approach we would need to add a method `typeCheck` to each class. On the other hand, the object oriented approach can handle the different kinds of nodes in a modular way. If we were to add for example *input-channel* nodes, we would simply add a class `InChan`. In the syntax separate from interpretation approach we would need to add code to handle this new node to all the interpretation classes. Note that the two different directions of modularity are orthogonal (see also [App98, pages 99–101]).

ANTLR, JavaCC and SableCC adopt the syntax separate from interpretation view. Since these tools are exclusively utilized during the syntactic phase of compilation, it would be infeasible for them to place methods for semantic analysis in the code that they generate. These tools have no knowledge of which interpretation methods to provide or how to code them. They could of course, in an attempt to be object oriented, place abstract methods such as `scopeCheck` in the classes they create, but that would mean that users of the syntax trees would have to override a vast number of such methods — possibly several thousand in the case of a Java compiler. It is therefore reasonable for these tools to generate classes that violate object oriented principles in the interest of usability.

We have chosen to remain true to the object oriented paradigm in our compiler. As we will see in the next section, the main reason that we can afford to do so, is that the semantic analysis phase of our compiler is defined in terms of a number of high-level specifications. These specifications are then subsequently converted into low-level Java code. The specifications provide modularity with respect to different kinds of interpretations. Our framework supports both views: each specification file contains an interpretation for each kind of node, and each generated Java class contains all interpretations for a given kind of node.

3 Traversing Syntax Trees

Syntax tree traversal seems to present a design dilemma. If encapsulation is to be preserved the syntax tree must provide its own traversal method. However, this would of course mean that users of the syntax tree are limited to only those traversal schemes made available by the implementation. On the other hand, if the structure of the syntax tree is exposed, its users can perform any kind of traversal they desire. It may seem that a trade off needs to be made between encapsulation and flexibility. Can these seemingly conflicting alternatives be combined? We address this question in the context of scope checking of the fragment of Pict we introduced above.

In our compiler, we have chosen to throw an exception if scope checking fails. In this way, we can exploit the exception handling capabilities of Java to conveniently produce a witness to that failure. This design decision is orthogonal to all that follows. To simplify the presentation, we will leave out all the exception handling code from the examples below.

3.1 Scope Checking of Pict

We present part of the scope system of Pict. The complete system can be found in the Pict definition [PT98a]. In the scoping rules presented below, we use Γ and Δ to denote environments. Since we neither consider type definitions nor type checking in this paper, we can simplify environments to just being lists of identifiers. In the rules we encounter two types of scope resolution judgments. $\Gamma \vdash \mathit{con}$ expresses that the Pict construct *con* is well-scoped in environment Γ. We use $\Gamma \vdash \mathit{con} \triangleright \Delta$ to denote that the Pict construct *con* is well-scoped in environment Γ and introduces identifiers Δ.

$$(\textsc{Dec})\ \frac{\Gamma \vdash \mathit{def\text{-}list} \triangleright \Delta}{\Gamma \vdash \texttt{def}\ \mathit{def\text{-}list} \triangleright \Delta}$$

$$(\textsc{Def})\ \frac{\Gamma, \mathit{id}_1, \ldots, \mathit{id}_n \vdash \mathit{abs}_i \quad \text{for each } i}{\Gamma \vdash \mathit{id}_1\ \mathit{abs}_1\ \texttt{and} \cdots \texttt{and}\ \mathit{id}_n\ \mathit{abs}_n \triangleright \mathit{id}_1, \ldots, \mathit{id}_n}$$

$$(\textsc{Abs})\ \frac{\Gamma, \mathit{id} \vdash \mathit{proc}}{\Gamma \vdash \mathit{id} = \mathit{proc}}$$

The recursive definition $\texttt{def}\ \mathit{id}_1\ \mathit{abs}_1\ \texttt{and} \cdots \texttt{and}\ \mathit{id}_n\ \mathit{abs}_n$ introduces the identifiers $\mathit{id}_1, \ldots, \mathit{id}_n$. Each abstraction abs_i is scoped in the environment Γ extended with the identifiers $\mathit{id}_1, \ldots, \mathit{id}_n$, thereby allowing recursive references to $\mathit{id}_1, \ldots, \mathit{id}_n$. In an abstraction $\mathit{id} = \mathit{proc}$, the identifier *id* has scope *proc*.

To formalize the $\cdots$ in rule (Def), we replace this rule with the following two rules.

$$(\textsc{Def}_1)\ \frac{\Gamma, \mathit{id} \vdash \mathit{abs}}{\Gamma \vdash \mathit{id}\ \mathit{abs} \triangleright \mathit{id}}$$

$$(\textsc{Def}_2)\ \frac{\Gamma, \mathit{id} \vdash \mathit{tail} \triangleright \Delta \qquad \Gamma, \mathit{id}, \Delta \vdash \mathit{abs}}{\Gamma \vdash \mathit{id}\ \mathit{abs}\ \texttt{and}\ \mathit{tail} \triangleright \mathit{id}, \Delta}$$

One can easily verify that the rule (Def) is equivalent to the rules ($\textsc{Def}_1$) and ($\textsc{Def}_2$). That is, anything we can prove using the rule (Def) and the other rules of the scope system can also be proved using the rules ($\textsc{Def}_1$) and ($\textsc{Def}_2$) and the other rules of the scope system, and vice versa.

3.2 The Visitor Design Pattern

The visitor design pattern is quite commonly utilized for the purpose of traversing syntax trees. Below we present the use of this design pattern, as described by Watt and Brown in [WB99, Section 5.3], applied to scope checking of the fragment of Pict presented above. A similar use of the visitor design pattern can be found, for example, in Barat [BS98] and classgen [Klea].

The idea behind the visitor design pattern is quite simple. For a comprehensive discussion we refer the reader to Gamma et al. [GHJV94, pages 331–350]. The basic idea is that the traversal code needed to do scope checking is not stored within the syntax tree itself, but rather is extracted and placed into a `Visitor` object instead.

First, we introduce a visitor method for each concrete class. These methods are collected in an interface.

```
interface Visitor
{
  Object visitDec(Dec dec, Object arg);
  Object visitShortDefList(ShortDefList list, Object arg);
  Object visitLongDefList(LongDefList list, Object arg);
  Object visitAbs(Abs abs, Object arg);
}
```

Each of these visitor methods has an argument that is the root of the subtree to be traversed. It also has an `Object` argument which allows us to pass additional information if needed. In the case of scope checking, we pass an environment, that is, an instance of the class `Env`. Instances of this class can more or less be viewed as lists of identifiers. The instantiation `new Env(env, id)` extends the environment `env` with the identifier `id`, and the creation `new Env(id)` constructs an environment solely consisting of the identifier `id`. Furthermore, each method has an `Object` result which allows us to return information if needed. In the case of scope checking, we sometimes return an environment. Note that this interface is very general. It cannot only be used for scope checking but also for type checking or any other traversal of syntax trees.

Next, we implement this interface to do scope checking.

```
class ScopeChecker implements Visitor
{
  Object visitDec(Dec dec, Object arg);
  {
    DefList defList = dec.getDefList();
    return defList.visit(this, arg);
  }

  Object visitShortDefList(ShortDefList list, Object arg)
  {
    Env env = (Env) arg;
    Id id = list.getId();
    visitAbs(list.getAbs(), new Env(env, id));
    return new Env(id);
  }

  ...
}
```

Note that we need to add accessor methods, like `getDefList`, to the classes representing the syntax tree. As a consequence, the visitor design pattern exposes the internal data of the syntax tree. Hence, data encapsulation is violated. Also notice that the method `visitShortDefList` uses downcasting, and hence has the risk of runtime failure.

Finally, we add `visit` methods to the classes representing the syntax tree. To the class `Node` we add the abstract method

```
abstract Object visit(Visitor vis, Object arg);
```

For each concrete class we implement the `visit` method by simply calling the appropriate visitor method. For example, for the concrete class `Dec` this amounts to

```
Object visit(Visitor vis, Object arg)
{
  return vis.visitDec(this, arg);
}
```

To do type checking, we only have to write a class `TypeChecker` which implements the interface `Visitor`. This interface and the classes representing the syntax tree stay the same.

Essentially the visitor design pattern exposes internal data of syntax trees and increases the risks of runtime failures in order to be able to provide a very general and flexible means for traversing it. This approach complements quite well with the syntax separate from interpretation approach, where too the structure of the syntax tree is exposed.

3.3 SableCC's Tree Walker

One of the main features of SableCC is the separation of the code which specifies how to traverse the syntax tree from the code which defines what to do in each node of the tree. This is achieved through the use of a tree walker. Conceptually a tree walker is rather similar to a visitor, but with two major differences. First of all, the tree walker code is mechanically generated by SableCC. Secondly, the tree walker object contains only traversal code. Instead of the code to be executed at the nodes it contains methods with an empty body which the user needs to override. SableCC takes as input a syntax specification similar to the one we presented in Section 2.2 (without the comments). For our scope checking example the mechanically generated code by SableCC would roughly looks as follows.

```
class DepthFirstAdapter extends AnalysisAdapter
{
  void visitDec(Dec node)
  {
    inDec(node);
    node.getDefList().visit(this);
    outDec(node);
  }
  void inDec(Dec node) {}
  void outDec(Dec node) {}

  void visitShortDefList(ShortDefList node)
  {
    inShortDefList(node);
    node.getId().visit(this);
    node.getAbs().visit(this);
    outShortDefList(node);
  }
  void inShortDefList(ShortDefList node) {}
  void outShortDefList(ShortDefList node) {}

  ...
}
```

Note that we still need accessor methods. `DepthFirstAdapter` provides a very general traversal scheme which cannot only be used for scope checking, but also for type checking, code generation, etc. Next, we add the user specified code by extending the class `DepthFirstAdapter` and overriding the methods with an empty body such as `inDec`.

```
class ScopeChecker extends DepthFirstAdapter
{
  void inDec(Dec node)
  {
    setIn(node.getDefList(), getIn(node));
  }

  void outDec(Dec node)
  {
    setOut(node, getOut(node.getDefList()));
  }

  void inShortDefList(ShortDefList node)
  {
    setIn(node.getAbs(), new Env((Env) getIn(node), node.getId()));
  }

  void outShortDefList(ShortDefList node)
  {
    setOut(node, new Env(node.getId());
  }

  ...
}
```

The class `ScopeChecker` inherits the methods `setIn`, `setOut`, `getIn` and `getOut`. The methods `setIn` and `setOut` are used to store information in the global hash tables `in` and `out`. To retrieve information from those hash tables the methods `getIn` and `getOut` are used. The use of these methods that manipulate the global instance variables `in` and `out` violates the object oriented encapsulation law. Furthermore, the specified user code is error prone. For example, if we accidently write `node` instead of `node.getDefList()` in `outDec`, it still gives rise to code that compiles successfully. Since the hash tables `in` and `out` store `Object`s, downcasting is needed in the user specified code, and hence the risk of runtime failure exists.

The kind of traversals that can be performed is limited to the kind of tree walkers available. Currently, only preorder and postorder traversals are supported. There are however situations where the syntax tree needs to be traversed in an idiosyncratic manner. For instance, consider the scope checking rule (Def$_2$) which corresponds to the class `LongDefList`. This class has the instance variables `id`, `abs` and `tail`. The class `DepthFirstAdapter`, which implements a preorder traversal, would first visit `id`, then `abs` and finally `tail`. The postorder traversal is implemented by the class `ReversedDepthFirstAdapter` first visits `tail`, then `abs` and finally `id`. However, according to the scoping rule, we should first visit `id`, then `tail` and finally `abs`. Since the subtree related to `id` only consists of a single node, this case can still be handled by the class `ReversedDepthFirstAdapter`. However, in Pict one is also allowed to use patterns in place of identifiers in recursive definitions. Because the subtree related to a pattern may consist of multiple nodes, this case can neither be handled by the class `ReversedDepthFirstAdapter` nor by the class `DepthFirstAdapter`. Idiosyncratic walking may also give rise to a more efficient traversal. Typically when checking for Pict's kinding rules, we only need to examine subtrees that are related to typing information. For instance, nodes corresponding to the fragment of Pict studied in this paper can be safely ignored. However, SableCC's predefined tree walkers visit every node of the tree.

Both the limitations described in the previous paragraph can be dealt with by overriding some of the generated traversal code like the method `visitLongDefList`. In scope checking of Pict approximately 15% of the methods would need to be overridden. However, overriding these methods reduces the benefits of mechanically generating the traversal code and also introduces the disadvantages of the visitor design pattern discussed in the previous section.

In essence, SableCC's tree walker provides an elegant way of separating mechanically generated traversal code from the rest of the code. The SableCC framework provides generality at the cost of exposing the internal data of the syntax tree and the risk of runtime failures.

3.4 Our Approach

In our compiler, we mechanically generate all the code for scope checking from a high-level specification. Our decision to do so was based primarily on the following facts. First of all, the number of scoping rules is considerable. Coding all these rules by hand is a tedious and error prone task. Secondly, there is no assurance (other than rigorous testing) to ensure that scope checking has been implemented in a manner consistent with the scope system. Thirdly, the Pict definition [PT98a] presents the scoping rules in a cogent and programmable manner (see also Section 3.1). The same also applies to the kinding and typing rules. Mechanically generating code for these rules is merely a matter of translating a high-level specification language to the implementation language Java. Finally, one of our primary goals was to develop a maintainable compiler. Thus generating scope checking code from a high-level specification language is a positive step in that direction.

A high-level specification is usually easier and faster to write than Java code. Furthermore, such a specification is often smaller. Also, the specification is usually easier to read and easier to maintain than Java code. In the design of our specification language we have made a trade off. On the one hand, we want to stay close to the original scoping rules. On the other hand, the specification should be easily translatable into Java code and should not contain redundant information. Below we present the specification corresponding to the scoping rules given in Section 3.1.

```
A declaration is well scoped if its constituent list of definitions is well scoped.

G |- defList > D
----------------
  G |- Dec > D

A recursive definition such as def id abs introduces the binding id. The abstraction abs
is scoped in the context G extended with the binding id.

     G, id |- abs
----------------------
G |- ShortDefList > id

A recursive definition such as def id abs tail introduces the bindings id plus the bindings
introduced by tail. The abstraction abs is scoped in the context G extended with id as
well as the bindings introduced by tail.

G, id |- tail > D    G, id, D |- abs
------------------------------------
      G |- LongDefList > id, D

In an abstraction such as id = proc, the process proc is scoped in the context G extended
with the binding id. An abstraction itself does not yield any bindings.

G, id |- proc
-------------
  G |- Abs
```

Each rule is preceded by a comment. This comment will be included in the generated Java code. Clearly, the rules are very similar to the rules presented in Section 3.1. However, there are also a few differences. First of all, the conclusion of a rule merely contains the name of the class of a syntax tree node corresponding to the Pict construct in the original rule. For example, instead of `def` *def-list* we use `Dec` in the specification. Secondly, in the premises of a rule we use the instance variables of the class used in the conclusion of that rule. For example, the class `Dec` only has the instance variable `defList` which is used in the premise of the rule.

We use JFlex [Kleb] to lexically analyze the above specification. For example, the JFlex snippet

```
^[ \t]*---+[ \t]*$  { return new Token(IMPLIES); }
\|\-                { return new Token(ENTAILS); }
```

recognizes `-----` and `|-` and returns the tokens `IMPLIES` and `ENTAILS`, respectively. The scoping rules of the above specification can be generated by the following grammar.

$$\begin{array}{rcl} \mathit{rule} & \rightarrow & \mathit{premises}\ \texttt{IMPLIES}\ \mathit{conclusion} \\ \mathit{premises} & \rightarrow & \mathit{premise} \\ & \rightarrow & \mathit{premise}\ \mathit{premises} \\ \mathit{premise} & \rightarrow & \mathit{environment}\ \texttt{ENTAILS}\ \mathit{construct} \end{array}$$

All of the productions for *conclusion*, *environment* and *construct* and some of the productions for *premise* have been left out. We use CUP [Hud] to parse a sequence of tokens produced by JFlex and to generate the corresponding Java code. The above productions translate into the following CUP snippet.

```
Rule     ::= Premises IMPLIES Conclusion
Premises ::= Premise
           | Premise Premises
Premise  ::= Environment:env ENTAILS Construct:con
```

In the above snippet, the action code has been left out. The action code for the last production amounts to something like

```
{: RESULT = formatPremise(env, con); :}
```

where the method `formatPremise` is defined as

```
String formatPremise(String env, String con)
{
  return con + ".scopeCheck(" + env + ");"
}
```

The lexer and parser, generated from the above described JFlex and CUP specification, translate each scoping rule into a `scopeCheck` method. This generated method is preceded by documentation, which is extracted from the specification, augmented with `@param` and `@return` tags if applicable. The documented method is subsequently mechanically inserted into the corresponding class of the syntax tree node. To the class `Dec` the following documented method is added.

```
/**
   A declaration is well scoped if its constituent list of definitions is well scoped.

   @param env context in which to scope check.
   @return newly introduced bindings.
*/
Env scopeCheck(Env env)
{
  return defList.scopeCheck(env);
}
```

Next, we present the generated `scopeCheck` method of the class `ShortDefList`.

```
/**
   A recursive definition such as def id abs introduces the binding id. The abstraction abs
   is scoped in the context env extended with the binding id.

   @param env context in which to scope check.
   @return newly introduced bindings.
*/
Env scopeCheck(Env env)
{
  abs.scopeCheck(new Env(env, id));
  return new Env(id);
}
```

Our code is object oriented and does not violate the encapsulation law. Furthermore, there is no need for downcasting in our approach. We believe that our code is more readable than the code presented in the previous sections and it is also well documented. However, these advantages come at the cost of writing specification files for JFlex and CUP for each phase of the semantic analysis. For example, to do type checking we have to specify the typing rules. Furthermore, we have to write JFlex and CUP specifications. From these specifications we can mechanically extract a Java program that translates typing rules into `typeCheck` methods.

Reusing the specification and the JFlex and CUP code, we also mechanically extract a LaTeX representation of the scoping rules. We only need to provide different implementations of methods like

```
String formatPremise(String env, String con)
{
  return env + " \\vdash " + con;
}
```

In this way, we can easily keep our scope system and our scope checking code synchronized. Palsberg [Pal92] uses LaTeX in a similar way in his Cantor system.

To generate all scope checking code for Pict, we wrote 1748 bytes of specification, 971 bytes of JFlex code, 2340 bytes of CUP code and 1858 bytes of Java code (methods like `formatPremise`). From this data we mechanically generated 12931 bytes of scope checking code. To generate 8096 bytes of LaTeX code (approximately 3 pages of formatted scoping rules) we only need 1524 bytes of Java code (the specification and the JFlex and CUP code can be reused).

Although scope checking only takes a small fraction of the time needed to compile a program, we compared the running times of scope checking implemented by means of the visitor design pattern to running times of our implementation. Both gave rise to very similar running times. In fact our implementation is slightly faster. We did not have a look at the running times of the implementation of scope checking using the SableCC tree walker, but we expect them to be very similar to ours as well. The number of lines of Java code needed to implement the syntax tree traversal for scope checking in the visitor design pattern approach and in the SableCC tree walker approach is more than twice the number of lines of Java code needed in our approach.

It may be argued that the additional layer of JFlex and CUP code may lead to a loss of robustness. However, we have not found this to be the case. Firstly, as we have seen above, this additional layer of code is small in size and relatively simple. It is therefore not likely to be a source of subtle errors. Secondly, the code that we generate is highly object oriented and lacks downcasting. In our experience, most errors in the generation process manifest themselves as compile time type errors or encapsulation violations in the generated code. Lastly, the extra layer of code can prove advantageous when it comes time to formally verify the correctness of our compiler. In this scenario, it would suffice to only verify the correctness of our code generation program, rather than the correctness of an entire phase of compilation.

There is a similarity between our approach and the approach of using attribute grammars (see, for example, [Paa95]). In both approaches one creates an association between a grammar's productions and the semantic actions that apply to them. However, when using attribute grammars this association is traditionally achieved by embedding implementation level code into a high level syntax specification (see, for example [Paa95, page 206]). In our approach, high level semantic rules are specified separate from the syntax description. By virtue of this difference, attribute grammars are certainly a more general way in which to specify semantic rules. However, in general attribute grammars tend to be less high level, and consequently less readable and possibly less maintainable. For an example of the use of attribute grammars, we refer the reader to [ESL89].

4 Conclusion

Appel [App98], Gagnon [Gag98, GH98], Watt and Brown [WB99] and others have presented representations of syntax trees in Java which are very similar to the one we present in Section 2. Although it may well already have been known that every context-free grammar can be represented in such a way in Java, we have not found this fact in the literature. In Section 2.2 we formalized that this approach to representing syntax trees is generally applicable.

As we have seen in Section 3, there are several different ways to implement syntax tree traversals in Java. We considered two different approaches from the literature, one using the visitor design pattern [WB99] and another based on SableCC's tree walker [Gag98, GH98]. We also proposed our own way of implementing traversals of syntax trees in Java. We made a detailed comparison of the different approaches pointing out their strengths and weaknesses. We believe that our proposal is best suited for the development of our Pict compiler. However, we do not claim that our approach is always superior. One of the other approaches may well be better when applied in a different setting. We hope that our detailed comparison of the three approaches will help implementors pick the one which suits their needs best.

Although we focused on the implementation of syntax trees and syntax tree traversals in Java for the semantic analysis of Pict, many of the ideas presented in this paper are also applicable to the implementation of languages different from Pict in an object oriented implementation language different from Java.

Besides scope checking, we have implemented a transformation from concrete to abstract syntax trees as well as kinding, using the approach described in this paper. For more details, we refer the reader to the first author's thesis [Ant]. In the future, we intend to see if our approach can also be applied to the type checking, the type inference and possibly even the code generation phase of our Pict compiler.

We are also interested to see if we can come up with a more general specification language (and a corresponding tool) in which we can express scope checking, kinding, type checking and type inference. Ideally, this generic specification language would not only be applicable to the semantic analysis of Pict but to that of other languages as well.

4.1 Acknowledgements

The authors would like to thanks Laura Apostoloiu for her development of the syntax of the specification presented in Section 2.2 and for many fruitful discussions.

References

[Ant] D. Antia. Semantic analysis of Pict in Java. Master's thesis, York University, Toronto. In preparation.

[App98] A.W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[BS98] B. Bokowski and A. Spiegel. Barat – a front end for Java. Technical Report B-98-09, Free University Berlin, 1998.

[ESL89] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG – a generator for efficient back ends. *ACM SIGPLAN Notices*, 24(7):227–237, July 1989.

[Gag98] E. Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, McGill University, Montreal, 1998.

[GH98] E. Gagnon and L.J. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of the 26th International Conference on Technology of Object-Oriented Languages and Systems*, pages 140–154, Santa Barbara, 1998. IEEE.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hud] S.E. Hudson. CUP, LALR parser generator for Java. `www.cs.princeton.edu/~appel/modern/java/CUP`.

[Klea] G. Klein. classgen. `www.doclsf.de/classgen`.

[Kleb] G. Klein. JFlex, the fast scanner generator for Java. `www.jflex.de`.

[Mag] MageLang Institute. ANTLR. `www.antlr.org`.

[Mil99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[MS] Metamata and Sun Microsystems. JavaCC. `www.webgain.com/products/java_cc`.

[Paa95] J. Paakki. Attribute grammar paradigms: A high level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Pal92] J. Palsberg. A provably correct compiler generator. In B. Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434, Rennes, 1992. Springer-Verlag.

[Par] T. Parr. ANTLR reference manual. `www.antlr.org/doc`.

[PT98a] B.C. Pierce and D.N. Turner. *Pict Language Definition*, 1998. `www.cis.upenn.edu/~bcpierce/papers/pict`.

[PT98b] B.C. Pierce and D.N. Turner. *Programming in the Pi-Calculus: A Tutorial Introduction to Pict*, 1998. `www.cis.upenn.edu/~bcpierce/papers/pict`.

[PT00] B.C. Pierce and D.N. Turner. Pict: A programming language based on the pi-calculus. In G.D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. The MIT Press, 2000.

[Tof90] M. Tofte. *Compiler Generators: what they can do, what they might do and what they will probably never do*. Springer-Verlag, 1990.

[WAKS97] D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, 1997. USENIX.

[WB99] D.A. Watt and D.F. Brown. *Programming Language Processors in Java*. Prentice-Hall, 1999.