Semantic Analysis of Pict in Java

Darius Antia

A thesis submitted to the Faculty of Graduate Studies
in partial fulfilment of the requirements
for the degree of

Master of Science

Graduate Programme in Computer Science
York University
Toronto, Ontario
February 2004

Abstract

Our objective is to examine semantic analysis as it applies to a compiler for Pict. We show how syntax tree construction, syntax tree transformation, scope checking, and kind checking can be automated. This requires developing small specification languages for each of these tasks. 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.

In the development of our Pict 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. 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.

Acknowledgements

I am most grateful to my supervisor, Franck van Breugel, for his invaluable guidance throughout the course of this thesis. Without his untiring efforts this thesis would not have reached completion.

I am also grateful to Jonathan Ostroff for his assistance with Eiffel, Eshrat Arjomandi for meticulously proof reading the final draft of this document, and Richard Paige for his help with early versions of this thesis.

I am also indebted to York University's Computer Science department for providing me with the necessary support to complete this thesis.

Table of Contents

Chapter 1

Introduction

The π-calculus [16] provides a theory of communicating and mobile systems. Pict [23] is a high-level language built on top of the π-calculus. As such, the concurrent language Pict is related to the π-calculus much like functional languages such as ML are related to the λ-calculus.

Compilers for Pict should be designed to accommodate frequent changes to the language. This is because Pict is used as a basis for experimenting with variations on the π-calculus, as well as various kinds of type systems. Consequently, its definition and semantics are volatile in nature, and compilers unable to cope with such changes can soon become defunct.

1.1 *Hello World* in Pict

A program in Pict essentially consists of concurrent processes communicating over channels. The process `print!"hello, world"`, demonstrates the concept of communicating over channels. In it, we transmit the string `hello, world` over the channel `print`. Communication over channels is akin to function calls in conventional languages, and as such is an easily understood concept.

A more interesting aspect of the π-calculus, and consequently Pict, is that, not only can channels carry ordinary data like the string `hello, world`, but they can also carry other channels. This is refered to as mobility. In the following mobile version of a *hello world* program,

```
 1  new request:  ^[]
 2  new response: ^^String
 3
 4  ({-- Server process --}
 5   request?[] = (new x:^String
 6                 (response!x
 7                  | x?str = print!str))
 8
 9   {-- Client process --}
10   | (request![]
11      | response?pr = pr!"Hello World")))
```

the channel `response`, in Line 6, is used to carry the channel `x`. In this program, the server process, upon receiving a signal over channel `request`, creates a new channel called `x` (Line 5), which it then transmits over channel `response` (Line 6). It then prints out whatever it receives over channel `x` (Line 7). The client process on the other hand, sends a signal along the channel `request` (Line 10), and then reads from channel `response`. What it reads is an actual channel (`pr`), along which it transmits the string `hello, world` (Line 11).

The Pict Tutorial [21] further elaborates on the topics of communication and mobility. It also discusses other important concepts such as concurrency, synchronization, and typing. We refer the interested reader to this document.

1.2 Research Goals

The overall goal of our research is to develop a Pict compiler that will serve as a convenient basis for further research and development of (a) languages based on the π-calculus, (b) type systems for concurrency, and (c) Pict itself. Additionally we wish to explore the applicability of object oriented techniques in compiler development.

1.2.1 Platform Independence

In the interest of widespread utilization we have decided that our compiler should be capable of running on all major platforms. To this end we chose to develop our compiler in Java [10], specifically the Java Development Kit (JDK). 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 several different platforms, it behooves us to have our compiled code run on those platforms too. We therefore chose to make our compiler's target language executable on the Java virtual machine (JVM) [14]. The widespread availability of the JVM and the JDK makes our compiler and compiled code potentially platform independent.

1.2.2 Object Oriented Construction

We wish to examine how applicable object oriented techniques are in the realm of compiler development. We observed that in several object oriented language processors (or toolkits), core object oriented concepts are violated 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 even enhance) programmer convenience. Using Java as an implementation language allows us to experiment with object oriented compiler construction techniques.

1.2.3 Attention to Static Typing

Static typing is in our opinion crucial to safe programming. However, since it is limited in our implementation language (Java), it makes our compiler susceptible to runtime type violations.

In our work we attempt to utilize Java's type system in a manner that reduces the risk of runtime type mismatches. Such an approach is particularly important when a large number of classes is involved. In our compiler we often deal with over two hundred classes at the same time.

1.2.4 High Level Specification Languages

One of our design goals is to be able to view our compiler as 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 semantic actions in a manner that is lucid, terse and precise.

Accompanying each such specification language is a tool that can convert it into highly object oriented Java code. This generated code is intended to be understood by humans, and is thus very readable and well documented.

The ability to transform high level specifications into implementation level code also eliminates a large amount of tedious manual coding. This consequently enhances maintainability since changes to the definition and semantics of Pict only need to be reflected in the (easily modifiable) specification files.

1.3 Previous Work

A considerable amount of work with goals closely related to ours has already been carried out. First of all, there already exists a Pict compiler. Furthermore, considering the current popularity of object oriented programming, a large variety of techniques to develop object oriented software have been developed. Several

of these are applicable to compiler construction. Also, a large variety of tools to automate various compilation tasks, several of them producing Java code, have already been built.

1.3.1 Pict Compiler

Previous work on a Pict compiler has been performed by Pierce and Turner [23]. This includes a working Pict compiler as well as a precise definition of the language's semantics. However, this compiler is implemented in Objective CAML [6] — a language supported on most major platforms, but little known outside of the functional programming community. Consequently, further development on this compiler is limited to functional programmers.

1.3.2 Object Oriented Compiler Design

Modern Compiler Implementation in Java [4], is a well recognized text for compiler construction. It describes how to develop a compiler using an object oriented implementation language. Some of the techniques described in this book however, depart from fundamental object oriented principals.

Design Patterns [9] describes a wide range of object oriented techniques in general. Some of these techniques, for example visitor design, are quite applicable to compiler development.

1.3.3 Compiler Toolkits for Java

There exist several compiler development toolkits for Java. These include *JLex* [5] and *JFlex* [12] for generating scanners, *CUP* [11] and *JavaCC* [15] for generating parsers, and *JJTree* [15] for generating syntax trees. There also exist more general purpose toolkits such as *SableCC* [8] and *ANTLR* [19], which handle everything from scanning to syntax tree generation. These toolkits also make provisions for traversing the parse trees that they generate. We discuss some of the pros and cons of such toolkits in Chapter 8.

1.3.4 Semantics-Directed Compiler Construction

There exist several systems which are able to construct a working compiler given an input specification. An important differentiating point among these systems is the nature of the input that they accept. *ACTRESS* [18], is one such system that generates a compiler from an input based on action semantics [17]. *PERLUETTE* [7] is another such system, but uses algebraic semantics as input instead.

1.4 Our Contributions

Our main contributions are to (a) formally define the syntax of Pict, (b) develop a framework for generating syntax trees, (c) using high level specification languages to perform semantic analysis, and (d) evaluate contemporary object oriented approaches to compiler development.

1.4.1 Formal Language Definition

The original Pict definition is a rather informal document. Several aspects of its lexical definition are ambiguous, and its grammar is missing some productions. It should be noted that this document is only intended to give a flavour of the language's syntax rather than rigorously define it.

We begin our work by formally defining a core fragment of the Pict language. In our definition we correct the errors found in the original definition, as well as incorporate Unicode support into the language.

1.4.2 AST Generation Framework

An abstract syntax tree (AST) is a core data structure in any multi-pass compiler. Conceptually it is a data structure that represents the essential structure of an input program. It typically serves as the interface between the syntactic phase of compilation, in which it gets generated, and the semantic phases of compilation, in which it gets utilized.

It is important that an AST implementation be both robust and easily maintainable. Upon an AST rests the entire semantic analysis phase of a compiler, so it is crucial that it be robustly implemented. Additionally, in order to cope with an evolving language definition, ASTs also need to be maintainable enough to facilitate frequent modifications. We have developed a framework for generating ASTs that meets both of the above criteria. Our ASTs mainly rely on encapsulation and static typing to achieve their robustness. Furthermore, since they are automatically generated, they are inherently maintainable.

1.4.2.1 Representing Syntax Trees

A parse tree for a string s and a context free grammar G represents the way in which s can be decomposed as per the rules of G. A syntax tree — as defined for the purpose of this discussion — is a data structure capable of representing all parse trees for G.

We propose a method by which any context free grammar G can be converted into a syntax tree S. We then prove that there is a one-to-one correspondence between the set of all parse trees for G and the set of all instances of S.

The syntax tree data structure that is produced by our transformation is object oriented in nature. In fact, the output of the transformation is an inheritance hierarchy of Java classes.

1.4.2.2 Specifying and Transforming Syntax Trees

We first introduce a simple language for specifying tree-like (i.e., hierarchical) data structures. Our main intent for doing so is to be able to specify the inheritance hierarchy mentioned above.

We also introduce a language for mapping one kind of syntax tree into another. This language permits for a high level specification to indicate how the nodes of one syntax tree should be transformed into the nodes of another. Normally, this a tedious task to code in a language like Java, so the ability to only have to specify it provides a great degree of convenience.

For both of the above languages, we develop tools that transform specifications written in these languages into implementation level Java code. Like all our other mechanically generated code, this code too is object oriented, readable and well documented. Additionally, we ensure that our tree transformation code is type safe.

1.4.3 Specifying Semantics Analysis

Semantic analysis is of course the process by which a compiler examines a syntactically correct input for conformance to the language's semantics. It is however a rather tedious process to perform. Firstly, there are a lot of checks that need to be performed. In a language like Pict, where there are about eighty semantic constructs and five separate semantic phases (e.g., scope checking, type checking, etc.), there would be about four hundred individual cases that would need to be checked. Secondly, there is no assurance (other than rigorous testing) to ensure that the semantic analyzer has been implemented in a manner consistent with that of the formal semantic definition.

In our work we address both of the aforementioned issues. We propose a framework in which a compiler writer only be required to provide a high level specification of the language semantics. This definition, which is both terse and precise, should be considered as an authoritative source of the language's semantics. A program then converts this definition into implementation level code. Our experience suggests that it is well worth the effort to create a framework such as the one described above. A compiler writer now only

needs to maintain a high level semantic description (and its corresponding simple transformation program), rather than a voluminous quantity of implementation level code. Additionally, proving the correctness of a semantic analyzer generated in this manner, reduces to the task of proving the correctness of a relatively simple transformation program.

The transformation programs that we develop have several notable properties. Firstly, they produce Java code that is both human readable and documented. Secondly, the code generated is not only highly object oriented, it is also statically typed. Lastly, in addition to just transforming a specification into Java code, our program is also capable of converting it into a LaTeX document, suitable for publication.

1.5 Overview

The remainder of this thesis can be subdivided into three broad categories — syntactic analysis, description of data structures, and semantics analysis. In Chapter 2 to 5 we discuss syntactic analysis. In Chapter 6 to 10 we discuss those data structures that are crucial to our implementations. And, in Chapter 11 to 12 we discuss semantic analysis.

In discussing syntactic analysis we describe how we go from program text to parse tree. Our main focus here is to precisely define the syntax of Pict, propose object oriented approaches to lexical analysis, and minimize the amount of user defined code used for parsing.

Our discussion on data structures concerns syntax trees and symbol tables. When discussing syntax trees we are mostly concerned with object oriented design, static typing, and overcoming limitations of Java's type system. Later, we describe how we implement persistent symbol tables, and how they are integrated into our abstract syntax trees.

Scope checking and kind checking are the major topics in our discussion on semantic analysis. Here our emphasis is on describing a mechanism for automatically converting high level semantic definitions into implementation level Java code.

We conclude by comparing our object oriented approach with a potential functional approach.

Chapter 2

Lexical Analysis

Lexical analysis is the process of identifying and categorizing tokens in an input program. Conceptually this is a fairly simple process to implement, but in the case of a language like Pict, care must be taken to keep it maintainable. This is because Pict is still in an experimental stage, and thus its lexical definition is still volatile.

Our lexical analyzer (lexer) is developed in a very object oriented manner. Our goal was to implement a lexer capable of coping with changes in Pict's lexical definition. To this end, we used several object oriented design techniques when implementing our lexer. Although the techniques we utilized are not new, we have hitherto not seen them used in the context of lexical analysis.

2.1 Mechanically Generating a Lexer

In Chapter 3, we will define the tokens that constitute a Pict program. We would like to implement a lexer capable of identifying these tokens from a stream of input. It is of course a simple matter to hand code a program capable of doing so, but using a *lexer generator* is a far more convenient approach. A lexer generator is a tool that uses a high level specification of a language's lexical structure to generate a lexer for it. The idea of mechanically generating a lexer is very well known and documented in [1, Chapter 2].

Since our compiler is being implemented in Java, it is desirable to utilize a lexer that too is coded in Java. Fortunately, we have at our disposal several tools that meet this requirement. *JLex* [5] and *JFlex* [12] are two popular stand alone lexer generators. *JavaCC* [15], *SableCC* [8], and *ANTLR* [19] are popular Java based compiler toolkits, part of which include a facility for lexer generation.

Two of the main factors dictating our choice of lexer generator were: (a) The overall readability of the specification file, and (b) the object orientedness of the generated code. Together, these criteria determine the overall maintainability of the lexical phase of our compiler. Since all of the aforementioned tools generate adequately fast code, performance was not a deciding factor. For reasons we shall explain below, we settled upon using *JFlex* as our lexer generator.

2.1.1 Readability of Specification

Most of the effort involved in creating a mechanically generated lexer lies in writing a specification for it. It therefore behooves us to utilize a lexer generator that supports a rich and uncluttered specification syntax.

For the most part, the input to a lexer generator is an encoding of generalized finite automaton (GFA). A GFA is merely a finite automaton whose transitions are labeled by regular expressions, rather than just individual characters. The richer the set of regular expressions supported by a lexer generator, the fewer will be the states in the underlying GFA, and consequently the more high level the resulting specification.

JFlex supports a rich set of regular expressions in its specification syntax. It for instance supports the concept of a minimal match (i.e., the `~` operator). This allows us for example to easily specify a string literal using the expression `"~"` (assuming for the sake of simplicity that a string literal itself does not contain a `"` character). Using some of the other lexer generators we would need to specify a string literal as:

```
State DEFAULT:
      "     goto state STRING
State STRING:
      .     save this character as being part of the string
      "     goto state DEFAULT
```

The convenience of such extended regular expression operators greatly contributes to the overall succinctness of the lexer specification. For this reason it is advantageous to use *JFlex* as a lexer generator instead of *SableCC* or *JavaCC*.

ANTLR too has a rich set of operations available for specifying tokens. In fact, whereas all the other lexer generators we looked at only allow regular expressions to be utilized, *ANTLR* actually allows us to use context free grammars! The additional expressiveness of a context free grammar can be put to good use in a lexical specification. Pict for example allows comments to be nested, as in `{ a { b } }`. Using *ANTLR*, such a comment can be easily identified by the expression, `COMMENT: '{' ( COMMENT | ~'}' )* '}'`. For exact detail about this expression we refer the reader to the *ANTLR* documentation [19], but the main point to note is that a `COMMENT` is defined in terms of itself (i.e., recursively). Any of the other lexer generators would require us to introduce an explicit `COMMENT` state, and also keep track of the nesting level ourselves.

Pict programs are written in Unicode, so we require a lexer that can work with a steam of Unicode input. At the time of this writing *JLex* does not support Unicode at all, and *ANTLR* only supports it poorly. While it is still possible to perform lexical analysis on a Unicode input stream using a lexer that only has a concept of one byte characters, it is however inconvenient to do so. For instance, an integer can be specified as `[0-9]+` using a lexer generator that supports Unicode, but without the benefit of Unicode support it would have to be specified as `(\000[0-9])+`. For this reason we opt to choose *JFlex*, which has good Unicode support, over *ANTLR* or *JLex*.

2.1.2 Object Orientedness of Generated Code

A good guideline to writing a maintainable lexer specification is to minimize the amount of user defined code it contains. Not only does this aid in readability, but it also separates the task of identifying tokens from the task of interpreting (processing) them.

The separation between identification and interpretation can be very naturally modeled in an object oriented manner. As we shall explain in Section 2.2, a mechanically generated lexer can be viewed as an *abstract* class that knows how to identify tokens, but does not define the methods required to process them. We then *extend* such a lexer with the methods required for interpretation.

In order to formally define and enforce such a separation, we need to utilize a lexer generator that supports the notion of an abstract lexer. *JFlex* supports such a notion, and does so in a very flexible manner. Toolkits such as *SableCC*, *ANTLR*, and *JavaCC* intend for the lexer that they generate to be used in conjunction with the rest of tools they provide, and so do not necessarily support features required for implementing a flexible stand alone lexer.

2.2 Writing a Lexer Specification

Despite the rich set of regular expressions provided by *JFlex*, a *JFlex* specification can easily get cluttered and unmaintainable if not done properly.

2.2.1 Embedding User Defined Code

There is a tendency to squeeze a lot of code into a lexer definition file. This is typically done in the interest of efficiency and possibly convenience, but at the cost of maintainability. For example, the *JFlex* distribution provides a sample Java specification that suffers from this malady. In this specification an integer literal is

handled as follows: (a) It is first identified, (b) it is converted to a mathematical integer, (c) error checking is performed (for example checking for overflow), and (d) any potential errors are reported.

The underlying problem with action code in the lexer specification is that the generated lexer processes its input in addition to identifying it. In such a scenario, if some implementation of Pict decided to utilize 64 bit integers rather than 32 bit, the overflow checking code would need to be modified. But this modification would have to be made in the lexical specification, where clearly it does not belong. Consequently, this specification cannot be shared with the 32 bit version of Pict, even though the lexical structure of both is identical.

2.2.2 Distributing User Defined Code

An alternative to embedding code in the lexical specification is to place it in some other file instead. A seemingly obvious object oriented approach would be to create a separate class of each kind of token known to the lexer. In Pict for example we would have an `IntegerToken` class, a `StringToken` class, and a `CharToken` class, to name a few. These classes would then be equipped with the methods required to perform conversions and error checking. For instance, we would no longer error check an integer for overflow in the lexical specification, but instead do so somewhere in the `IntegerToken` class.

This approach certainly minimizes the clutter in the specification file, but still contains some flaws. The main problem with this approach is that policy decisions get distributed into classes, when in fact such decisions are best centralized. For instance, it is now up to the `IntegerToken` class to decide whether out of bound integers are an error or a warning. Furthermore, there are some sections of code that simply do not belong in any of the token classes (e.g., the code to convert escapes sequences to characters).

While such problems can be remedied through the use of additional classes that define policy and conversion rules, the overall solution gets needlessly complicated. As indicated earlier in Section 2.1.2, we can achieve a rather elegant solution using an object oriented approach that utilizes inheritance.

2.2.3 Inheriting User Defined Code

Writing a lexical specification is similar to writing an event driven program. An event in this case is the identification of a token, and the action is the task of interpreting it. For any given lexical specification the set of events remain constant. In a Pict lexer for example, some of these events would be `foundInteger`, `foundString`, etcetera. However, there can be a variety of interpretations of these events. For example, a 32 bit implementation of Pict would handle a `foundInteger` event differently from a 64 bit implementation. For maximum flexibility we need to come up with a scheme that allows us to associate with a set of events, an arbitrary set of actions (or event handlers).

Object oriented design techniques allow us to conveniently create a flexible interface between events and event handlers. One obvious approach is to formally define an *interface* of events that an event handler object must *implement*. Then any object that implements this interface can be utilized as an event handler. This of course gives us the ability to handle any event in an arbitrary number of ways. When applying this approach to the task of lexical analysis, we would initialize our lexer object, with an event handler that implements our choice of behavior. Thus the behaviour of our lexer will be governed by the arguments used to initialize it.

An alternative approach is to use an abstract base class rather than an interface to specify events. Doing so, allows us to introduce the concept of an abstract lexer — an object that knows how to recognize tokens, but does not know how to process them. For instance, a specification such as,

```
"~"     { foundString( matching-text );  }
[0-9]+  { foundInteger( matching-text ); }
```

results in a lexer such as,

```
abstract class Lexer {
```

```
    Private methods for identifying tokens and triggering actions.
    ⋮
    abstract foundInteger (String s);
    abstract foundString (String s);
}
```

Note the conceptual similarities between the lexical specification and the abstract object. Both of them are only concerned with identifying tokens, and neither of them describe how to process them.

Using inheritance, we can now morph the abstract lexer object into a concrete lexer object of our choosing. For instance, if we wanted to create a 32 bit implementation of a Pict lexer, we would simply extend the `Lexer` as follows

```
class Lexer32 extends Lexer {
    foundInteger (String s) { ... ; x = Integer.parseInt(s); ... }
    ⋮
},
```

and to create a 64 bit implementation we can extend `Lexer32` like,

```
class Lexer64 extends Lexer32 {
    foundInteger (String s) { ... ; x = Long.parseLong(s); ... }
    ⋮
}
```

We believe that for practical as well as esthetic reason, using inheritance to define a lexer's behaviour, is a better approach than initializing a lexer with an appropriate event handler.

- From a design point of view we can now say that a `Lexer32` *is a* `Lexer`, rather than a `Lexer` *has an* event handler for 32 bit integers. We feel that in this case an *is a* relationship is more natural than a *has a* relationship.
- It is also more object oriented, as the behaviour of an object is transformed using inheritance and polymorphism, rather than special initialization parameters.
- Finally, this approach is also more practical since the automatically generated lexer does not make provisions for working with an event handler parameter.

2.2.3.1 Limitations

The concept of inheriting an abstract lexer ensures that the lexical specification is immune from implementation level changes. However, changes in the lexical structure of the language still need to be reflected in the lexer specification. This of course could result in having to define new event handlers, and consequently require us to update all subclasses of the abstract `Lexer` class.

An abstract lexer only declares abstract method for events (i.e., patterns) that occur in the lexical specification. Therefore, using the same abstract lexer, in different applications is not necessarily feasible. For instance, our lexer declares methods such as `foundInteger`, `foundString`, etc., but does not define a method such as `foundComment`. Consequently, an application such as a pretty printer, which requires the presence of a `foundComment` method will not be able to reuse our abstract lexer. Its implementation would require us to create a different (albeit very similar) lexical specification, and derive a new abstract lexer from it.

2.3 Handling Error Conditions

It is of course important to be able to report errors in the lexical structure of an input program. It is also important to be able to continue on, in spite of such errors. At first it may seem appropriate to use Java's exception handling mechanism to handle error conditions. Exceptions are designed to allows us to report errors and potentially even recover from them. We however have found this not to be the case in a lexer implementation.

The problem with using exceptions in a lexer is that an exception can leave a lexer in an undefined state. Recall that a lexer is essentially a large finite state automaton. But, when an exception is thrown, a lexer is forced to enter a state that is not part of its underlying finite automaton. Worse still, there is no obvious way in which to restore the lexer to a known state. Consequently, throwing an exception hinders the possibility of error recovery.

How then should errors in a lexer be dealt with? We believe that in a lexer, an error condition should be the norm and not the exception. That is to say, our lexer should handle error events, using exactly the same mechanism it uses to handle normal events (such as `foundInteger` or `foundString`). An example of an error event handler would be `extraBackslash`. Typically, this event handler would be called when an extraneous \ character is detected. When such an event occurs it is up to the event handler code to decide whether to abort the processing, or emit a warning and continue. Throwing an exception named `ExtraBackslash` would leave little room for recovery.

Chapter 3

Lexical Definition

The purpose of a language's lexical definition is to clearly describe the textual elements of that language. Examples of textual elements would include string literals, comment delimiters, escape conventions, etc. Naturally, it is only after these components have been described that it is possible to identify them. This description is therefore the basis of the lexical analysis process that we described in Chapter 2.

The original Pict language definition [22, Chapter 3] is both brief and informal in its treatment of lexical rules. Consequently, it suffers from certain minor omissions and errors. In our work we rigorously define a revised set of lexical rules that addresses these shortcomings. Additionally, our lexical rules permit for a greater degree of internationalization.

3.1 Unicode Support

Programs are written in Unicode and are allowed to contain Unicode escapes anywhere within their text.

Unicode is an evolving standard, the latest information about which is available at unicode.org. Informally, the Unicode character set can be viewed as a "wide" version of ASCII. It utilizes 16 bits to encode its characters (as opposed to ASCII's 7 bits). The Unicode Standard [2] is a mapping that provides a unique number for every character.

The most obvious advantage of allowing a program to be written in Unicode is that its tokens are not restricted to basic Latin characters. We feel that for identifiers within a program such a feature provides little advantage. Furthermore, it is liable to result in cryptic identifier names such as ∪ instead of `union`. However, for string and character literals it provides a considerable degree of useful flexibility. For instance, a string may now trivially contain a `£` character. Additionally, by virtue of the consistency of the Unicode standard, a precise lexicographic order exists amongst Unicode characters (e.g., a `$` always precedes a `£`).

3.2 Escape Conventions

Typically a character is represented by a single symbol. For example, the digit one is represented by the symbol `1`. An escaped character however is represented by multiple symbols. For instance, in the C language, the digit one can be represented by the escape sequence `\061`. The need for escape sequences arises because there exist more characters than we have symbolic representations. For example a US keyboard lacks a symbolic representation (i.e., key) for the `£` character.

Pict supports three different kinds of escape conventions: octal number escapes, character escapes, and Unicode escapes. Escape sequences such as `\061` are referred to as octal number escapes. Specifically, an octal number escape is a sequence of the form *ooo*, where *ooo* is a three digit octal number. Such an escape sequence represents the character at location *ooo* in the ASCII table. Escape sequences such as *c* are referred to as character escapes (where *c* is one of `n`, `t`, `"`, `'`, or \). These can be viewed as predefined abbreviations for octal number escapes. For instance, `\n` is an abbreviation for `\012`. Octal number escapes

and character escapes are commonly found in programming languages, and hence should be familiar to most programmers.

3.2.1 Unicode Escapes

Unicode escapes deserve some additional attention. Since the Unicode sequence consists of many thousand characters, and a keyboard consist of only a few hundred symbols, most Unicode characters would need to be represented by escape sequences. The Unicode standard defines an escape sequence for every Unicode character. Furthermore, every Unicode escape sequence can be represented using only ASCII characters. This fact makes it possible to represent any Unicode character using only a standard keyboard.

A Unicode escape looks like, `\u`*hhhh*, where *hhhh* is a four digit hexadecimal number. This escape sequence represents the character at location *hhhh* in the Unicode standard. For example, the Unicode escape for £ is `\u00a3`, and location 163 (i.e., A3 in base 16) of the Unicode standard is the £ symbol. Unicode escapes are case insensitive, so `\u00a3` is the same as `\U00A3`, or `\u00A3`.

Unicode escapes make octal number escapes and character escapes redundant. However, as we will argue in Section 3.4, it is still desirable to maintain these older escape conventions.

3.3 Lexical Structure

A Pict program can be viewed as being consisted of tokens, whitespaces, and comments. However, only tokens play a role in any further compilation stages. Whitespace and comments are discarded. Tokens can be subdivided into five categories: identifiers, keywords, string literals, character literals, and integer literals.

3.3.1 Identifiers

There are two classes of identifiers: alphanumeric and symbolic. Alphanumeric identifiers begin with one of the following symbols:

```
A...Z a...z _
```

Subsequent symbols may contain the following characters in addition to those mentioned above:

```
0...9 '
```

Upper and lower case letters are different. Symbolic identifiers only consist of one or more of the following symbols:

```
~ * % \ + - < > = & | @ $ , '
```

The maximum allowable length of an identifier is implementation dependent. In our compiler we limit it to 256 characters.

3.3.2 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise.

```
and     Bool    Char  Con  def   else  false  if
import  inline  Int   Neg  new   now   Pos    rec
run     String  then  Top  true  Type  type   val
where   with    !     #    (     )     ->     .
/       :       ;     <    =     >     ?      @
[       \       ]     ^    _     {     |      }
```

3.3.3 String Literals

A string literal is a sequence of the form "x", where x can be a finite sequence of zero or more Unicode characters, Unicode escapes, octal escapes or one of the escape sequences shown below:

newline	NL	`\n`
carriage return	LF	`\r`
double quote	`"`	`\"`
horizontal tab	HT	`\t`

However x cannot contain the Unicode character (or the Unicode escape, or the octal escape) for `"` unless it is immediately preceded by a `\`. String literals cannot contain the character (or the Unicode escape, or the octal escape) that represents a line break. The `\n` or `\r` escape must be used instead.

The reason for disallowing line breaks is that different host systems encode line breaks differently. Unicode does not attempt to standardize the encoding of a line break, hence the behavior of a program would depend on the host system on which it was compiled.

Examples of valid strings include `"hello\n"`, `""` and `"\u00a3 10"`. Examples of incorrect string literals include `"""` and `"\u0022"` (Unicode for `"`).

3.3.4 Character Literals

A character literal is a sequence of the form 'x', where x is either a Unicode character, a Unicode escape, an octal escape, or one of the escape sequences shown below:

newline	NL	`\n`
carriage return	LF	`\r`
single quote	`'`	`\'`
horizontal tab	HT	`\t`

However x cannot be the Unicode character (or the Unicode escape, or the octal escape) for `'`. The only way in which to create a character literal for `'` is to use the special escape sequence `\'` as defined above.

Also x cannot be the Unicode character (or the Unicode escape, or the octal escape) that represents a line break (newline or carriage return). The only way in which to create a character literal to represent a line break is to use the special escape sequence `\n` or `\r`.

Examples of correct character literals include: `'a'`, `'\''` and `'\n'`. Examples of incorrect character literals include: `'aa'`, `'''` and `'\u000a'` (Unicode escape for a newline).

3.3.5 Integer Literals

An integer literal is a finite sequence of one or more digits optionally prefixed by a `-` sign. A `-` sign indicates a negative integer while the lack of a sign indicates a non-negative integer. Integer literals attempting to represent the value zero should not be prefixed by a sign. Examples of valid integer literals include `10`, `-10`, `01` and `-001`.

All integer literals are a decimal encoding of the integer value they represent. The largest and smallest such values that can be represented are implementation dependent. In our implementation integers range from -2^{31} to $2^{31} - 1$.

3.3.6 Whitespace

White space is mostly ignored during lexical analysis, but is required in some scenarios to separate otherwise adjacent tokens, as illustrated by the following examples:

Input	Tokens
`[false 7]`	$\texttt{[}_{key} \cdot \texttt{false}_{key} \cdot 7_{int} \cdot \texttt{]}_{key}$
`[false7]`	$\texttt{[}_{key} \cdot \texttt{false7}_{iden} \cdot \texttt{]}_{key}$
`[7false]`	$\texttt{[}_{key} \cdot 7_{int} \cdot \texttt{false}_{key} \cdot \texttt{]}_{key}$
`x!y`	$\texttt{x}_{iden} \cdot \texttt{!}_{key} \cdot \texttt{y}_{key}$
`->*-#?`	$\texttt{->*-}_{iden} \cdot \texttt{\#}_{key} \cdot \texttt{?}_{key}$
`a'b'`	$\texttt{a'b'}_{iden}$
`a 'b'`	$\texttt{a}_{iden} \cdot \texttt{'b'}_{char}$
`a' b'`	$\texttt{a'}_{iden} \cdot \texttt{b'}_{iden}$

3.3.7 Comments

A comment is introduced by the characters `{-` and terminated by the characters `-}`. Comments can be nested and can be as long as desired. However, comments cannot occur within a string or character literal. Examples of some comments include:

```
{- comment -}
{-
    x!y       {- comment -}
-}
```

3.3.8 Line Termination

Lines can be terminated using the newline convention of the host system.

3.4 Conventional Escape Sequences

Unicode support would seem to imply that all conventional escapes such as `\n`, `\\`, `\`*ooo* etc. can be replaced with their Unicode counterparts. For example, `\\` can be replaced by `\u005c`. Obviously this is possible, since we are merely substituting one set of symbols for another. However there are good reasons for not doing so.

The first such reason is to do with the fact that Unicode escapes and conventional ASCII escapes convey different meaning. For example, the ASCII escape `\"` conveys the fact that "this double quote does not terminate the string", whereas normally the Unicode escape sequence `\u0022` indicates that "this is a double quote". Treating `\"` as equivalent to `\u0022` overloads the meaning of the Unicode escape. It now conveys two pieces of lexical information: (a) this is character 0022 (hex) in the Unicode chart, and (b) do not mistake this for a string terminator. The additional complexity that arises from this overloading does not justify the need for having it. For this reason the `\"` escape is retained and `\u0022` is not given any special treatment. Similar arguments can be made for retaining `\'`, `\n`, `\r` and `\\`. Also since `\"` is more intuitive than `\u0022` to represent a `"` it would likely be the preferred choice for programmers.

Another reason for not blindly substituting escapes such as `\t` with its corresponding Unicode escape `\u0009` is that the interpretation of `\t` is context dependent. Within a string literal it represents a horizontal tab, but it may also appear outside a string literal as in the following legal Pict statement for process abstraction:

```
\t=()
```

In the latter case it does not mean a horizontal tab. The complexity involved with selectively replacing a `\t` is significantly greater than treating `\t` as a special case within string and character literals. For this reason it too is retained.

Escape sequences such as *ooo* are in fact quite easily replaceable by Unicode escapes. In our compiler a preprocessing pass is made over the input to convert every sequence of the form *ooo* to `\u`*hhhh*.

Converting octal escapes to Unicode escapes, rather than actual Unicode characters has 2 advantages. Firstly, there already exists a mechanism in the second pass of the compiler to handle Unicode escapes. Secondly, converting to Unicode escapes has the advantage of not creating any additional (and unwanted) Unicode escapes for the second pass. For instance consider two cases of how the string `\134u005c` (octal backslash - u - 0 - 0 - 5 - c) could be preprocessed. In the first case, pass 1 of the compiler replaces octal escape sequences with Unicode escape sequences (as implemented in our compiler), while in the second case, octal escapes are replaced with actual Unicode characters.

	Case 1: (Unicode escape)	Case 2: (Unicode character)
Input	`\134u005c`	`\134u005c`
After pass 1	`\u005c005c`	`\u005c`
After pass 2	`\005c`	`\`

As evident from the above example case 1 yields a more reasonable result.

3.5 Differences

The original Pict language definition [22] captures the essence of the language reasonably well but is somewhat lacking in detail. Our language definition remedies that situation as well as introduces Unicode support. Some minor differences between our definition and the original one include:

- Allow octal escapes *ooo* (the original allows decimal escapes).
- Allow leading underscore in identifier names.
- Disallow integers such as `-0`, `-00`, etc.
- Disallow `\"` escape in character literals.
- Disallow `\'` escape in string literals.
- Allow `\r` in string and character literals.

Chapter 4

Parser

Parsing, for the purposes of our discussion, is a process that transforms an input program into some implementation dependent output format. This format, in the case of a multi-pass compiler is a parse tree, while in the case of a single-pass compiler is the corresponding target language code. A *parser* is a program that performs this transformation. It should be noted that the input to a parser first undergoes lexical analysis, as described in Chapter 2.

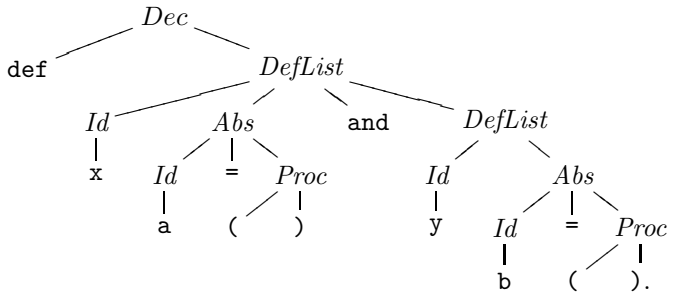
In this chapter we discuss issues such as maintainability, robustness, and programmer convenience within the context of implementing parsers.

4.1 Tree Generation

There are a variety of ways in which a parser can transform its input. In the case of a multi-pass compiler however, we need to transform the input in a manner that is conducive to making multiple passes over it. A parse tree data structure is a good candidate for fulfilling this requirement. It not only allows us to make multiple passes over the input, but also preserves the essential structure of the input program. For example, consider the context free grammar (CFG)

$$
\begin{array}{rcl}
\mathit{Dec} & \longrightarrow & \texttt{def}\ \mathit{DefList} \\
\mathit{DefList} & \longrightarrow & \mathit{Id}\ \ \mathit{Abs}\ \mid\ \mathit{Id}\ \ \mathit{Abs}\ \texttt{and}\ \ \mathit{DefList} \\
\mathit{Abs} & \longrightarrow & \mathit{Id}\ \texttt{=}\ \mathit{Proc} \\
\mathit{Proc} & \longrightarrow & \texttt{( )} \\
\mathit{Id} & \longrightarrow & \texttt{a}\ \mid\ \texttt{b}\ \mid\ \texttt{x}\ \mid\ \texttt{y}.
\end{array}
$$

This CFG is a simplified version of the Pict CFG defined in Chapter 5. As per standard convention, keywords are displayed using a `typewriter` font, and non-terminals are displayed using an *italic* font. The start symbol is *Dec*. Using this grammar, the parse tree for the Pict program `def x a = () and y b = ()`, would be



Notice how this parse tree preserves the syntactic structure of the input program.

4.2 Parser Generation

Parsers can be very easily generated from a CFG. Most texts on automata theory, such as Sipser's book [24], will formally define this process. Tools known as *parser generators* will implement it. We, for example, generate our parser from the CFG shown in Chapter 5, using the *CUP* [11] parser generator.

There are many advantages to using a parser generator. Parsers created by a parser generator typically run faster than what can be reasonably expected from a hand coded equivalent. Furthermore, the effort involved in using a parser generator is insignificant compared to the effort involved in writing a well optimized parser. For these reasons, parser generators are now the de facto means of obtaining a parser.

Compiler writers have at their disposal a wide variety of parser generators. In fact, we know of at least six such programs for generating parsers in Java. These include *JavaCC* [15], *SableCC* [8], *ANTLR* [19], *Zephyr* [26], *Corretto* [3] and *CUP*. These tools can all generate a parser which would meet our needs. However, they differ in the amount of user defined *action code* that they require us to provide.

4.2.1 Explicit Action Code

Action code is a code fragment that is associated with a rule of a CFG. Every time a parser performs a reduction on this rule (i.e., collapse of the parse stack), the corresponding action code gets triggered. The result from that action code is subsequently placed on the parse stack. Parser generators such as *YACC* [13], *CUP*, *ANTLR*, and *JavaCC* support this mechanism.

We can use action code to build a parse tree. Consider for example, the CFG from Section 4.1. A simplified *CUP* specification corresponding to this CFG would be,

```
Dec      ::=  DefList:dl                  {:  new Dec(dl);                :}
DefList  ::=  Id:i Abs:a                  {:  new ShortDefList(i, a);     :}
           |  Id:i Abs:a DefList:dl       {:  new LongDefList(i, a, dl);  :}
Abs      ::=  Id:i Proc:p                 {:  new Abs(i, p);              :}
Proc     ::=  ϵ                           {:  new Proc();                 :}.
```

Note that in the interest of clarity we have omitted keywords such as `def`, `and`, `(`, etc. Nevertheless, this specification largely resembles the CFG, except that it contains labels and action code. For example, in the case of rule

```
Abs  ::=  Id:i Proc:p  {:  new Abs(i, p);  :},
```

the label `i` refers to the *Id* component of the rule, and the label `p` refers to the *Proc* component. The action code, which is defined within `{:` and `:}`, creates a node whose left child is the subtree *Id* and right child is the subtree *Proc*. Essentially, as the parser performs reductions, it creates corresponding nodes of the parse tree. Eventually, it will create the root node of the parse tree and place it on the parse stack.

Action code is a valuable mechanism to employ when we wish to transform our input in an arbitrary manner. However, as we shall describe next, when we only need to transform our input into a parse tree, we can do so without using any action code.

4.2.2 Implicit Action Code

A closer look at the above *CUP* specification will reveal an obvious correspondence between a rule and its action code. Essentially, the left hand side of a rule becomes a new node of the parse tree, and the individual components on the right hand side, become the children of that node. Tools such as *Corretto* and *SableCC*, exploit this fact to relieve the user from writing any action code at all.

There are many advantages to using a parser generator such as *Corretto*. In our experience, about half the data in a *CUP* specification is action code, and thus being able to eliminate it, greatly contributes to readability. Secondly, by virtue of not having to write action code, the specification is inherently more

maintainable. Lastly, by eliminating the human factor entirely from the parsing process, we can go from program text to parse tree without fear of committing any errors.

In our work we used *CUP* as our parser generator. However, motivated by the inconveniences we experienced owing to this choice, and the fact that eliminating action code is entirely possible, the tool *Corretto* was subsequently (and separately) developed.

4.2.3 The Corretto Parser Generator

Corretto is designed to mechanically build a *CUP* specification from an annotated CFG. For this reason, it is really a wrapper around *CUP*, rather than a full fledged parser generator. The main advantages that *Corretto* offers over directly using *CUP* are (a) it allows us to write a more readable specification, and (b) it does not require us to write *any* action code. The following is a *Corretto* specification corresponding to the CFG in Section 4.1:

```
% Declaration %
Dec := % dl: list of definitions %
       DefList

% List of definitions %
DefList := % Single element definition list %
           ShortDefList
           % i: name of abstraction,
             a: body of abstraction  %
           Id Abs

         | % Multiple element definition list %
           LongDefList
           % i:  name of abstractions,
             a:  body of abstraction,
             dl: remainder of list %
           Id Abs DefList

% Abstraction %
Abs ::=  % i: name of parameter,
           p: process %
           Id Proc

% Process %
Proc.
```

Note that here too, in the interest of clarity, we have omitted terminals. *Corretto*, like *CUP*, requires components of a production to be labeled. Unlike *CUP* however, labels are not specified in-line, but rather are specified separately between `%` delimiters. Furthermore, labels can be augmented with descriptive comments. For example in the case of the *Corretto* fragment

```
Abs ::=  % i: name of parameter,
           p: process %
           Id Proc,
```

`i` and `p` are labels that refer to components *Id* and *Proc* with respect to the production $Abs \rightarrow Id$ `=` $Proc$.

Upon feeding this input to *Corretto*, we would get an output that very much resembles the *CUP* specification shown in Section 4.2.1. *Corretto* will of course have generated all the action code that is present in this output. The large quantity of comments that are present in a *Corretto* specification would at first glance

seem superfluous. However, there is an additional component of *Corretto*, called *Grappa*, which makes good use of these seemingly superfluous comments. We will discuss *Grappa* in Chapter 8.

Chapter 5

Concrete Syntax

A grammar describes a language's syntactic structure. In this chapter we provide a precise grammar for our implementation of Pict. We henceforth refer to this grammar as the *concrete syntax* of Pict. This should avoid confusing it with any other grammar that we utilize.

Our concrete syntax borrows heavily from that of the original Pict compiler. A notable difference between the two is that ours is considerably smaller than the original one, as we only implement a core subset of Pict. The original definition defines what is core and what is derived. Additionally the original contains some omissions, that we have addressed.

5.1 Definition

The concrete syntax defined below is in LALR form. It is therefore suitable for using with parser generators like *CUP* [11]. In our definition we adopt the following conventions:

- Keywords are written in a `typewriter` font, while non-terminals are written in *italized* font. The symbol "=" separates the left hand side of a production from its right hand side. The terminal symbol ϵ represents the empty string.
- The symbols *Id*, *String*, *Char*, and *Int* are not defined in this chapter. Instead, they are defined in Chapter 3, which concerns itself with lexical structure.
- Productions that only consist of a right hand side, are assumed to have the same left hand side of the production above them. For example,

  Consyn = *Dec* *DecList*
  Dec

  is equivalent to,

  Consyn = *Dec* *DecList*
  Consyn = *Dec*

- The rightmost column optionally contains a comment. Its presence or absence has no bearing on the grammar itself. Often these comments employ the adjectives *long* and *short* to describe list like structures, as in

DefList	=	*Id* *Abs*	Short definition list
		Id *Abs* `and` *DefList*	Long definition list

What follows is the Pict concrete syntax.

5.1.1 Compilation units

Consyn	=	*Dec DecList*	Long declaration list
		Dec	Short declaration list

5.1.2 Declarations

Dec	=	`new` *Id* `:` *Type*	Channel creation
		`def` *DefList*	Recursive definition
DefList	=	*Id Abs*	Short definition list
		Id Abs `and` *DefList*	Long definition list

5.1.3 Abstraction

Abs	=	*Pat* `=` *Proc*	Process abstraction

5.1.4 Patterns

Pat	=	*Id RType*	Variable pattern
		`[` *FieldPatList* `]`	Record pattern
		`(` `rec` *RType Pat* `)`	Recursive type pattern
		`_` *RType*	Wild card pattern
		Id RType `@` *Pat*	Layered pattern
FieldPat	=	*Pat*	Value field
		`#` *Id* `<` *Type*	Subtype constraint
		`#` *Id* `:` *Kind*	Kinding constraint
		`#` *Id* `=` *Type*	Equality constraint
FieldPatList	=	*Id* `=` *FieldPat FieldPatList*	Named record field
		FieldPat FieldPatList	Anonymous record field
		ϵ	End of record

5.1.5 Process

Proc	=	*Val* `!` *Val*	Output atom
		Val `?` *Abs*	Input prefix
		`(` `)`	Null process
		`(` *ProcList* `)`	Parallel composition
		`(` *DecList Proc* `)`	Local declarations
		`if` *Val* `then` *Proc* `else` *Proc*	Conditional

ProcList	=	*Proc* `|` *ProcList*	Long process list
		Proc `|` *Proc*	Short process list
DecList	=	*Dec* *DecList*	Long declaration list
		Dec	Short declaration list

5.1.6 Values

Val	=	*Constant*	Constant
		Path	Path (projection)
		`[` *FieldValList* `]`	Record
		`(` `rec` *RType* *Val* `)`	Recursive type value
FieldValList	=	*Id* `=` *FieldVal* *FieldValList*	Named record field
		FieldVal *FieldValList*	Anonymous record field
		ϵ	End of record
FieldVal	=	*Val*	Value field
		`#` *Type*	Type field
Constant	=	*String*	String constant
		Char	Character constant
		Int	Integer constant
		`true`	Boolean constant
		`false`	Boolean constant
Path	=	*Id*	Variable
		Path `.` *Id*	Record field pattern

5.1.7 Types

Type	=	`Top`	Top type
		Id	Type identifier
		`^` *Type*	Input/output channel
		`!` *Type*	Output channel
		`/` *Type*	Responsive output channel
		`?` *Type*	Input channel
		`Int`	Integer type
		`Char`	Character type
		`Bool`	Boolean type
		`String`	String type
		`[` *FieldTypeList* `]`	Record type
		`\` *KindedPolarityIdLs* `=` *Type*	Type operator
		`(` *Type* *TypeLs* `)`	Type application
		`(` `rec` *Id* *Kind* `=` *Type* `)`	Recursive type

TypeLs	=	*Type* *TypeLs*	Long type list
		Type	Short type list
FieldType	=	*Type*	Value field
		`#` *Id* `<` *Type*	Type field with subtype constraint
		`#` *Id* `:` *Kind*	Type field with kinding constraint
		`#` *Id* `=` *Type*	Type field with equality constraint
FieldTypeList	=	*Id* `=` *FieldType* *FieldTypeList*	Named record field
		FieldType *FieldTypeList*	Anonymous record field
		ϵ	End of record
RType	=	ϵ	Omitted type annotation
		`:` *Type*	Explicit type annotation

5.1.8 Polarities

Polarity	=	ϵ	Mixed polarity
		`Pos`	Positive polarity
		`Neg`	Negative polarity
		`Con`	Constant polarity

5.1.9 Kinds

Kind	=	`(` *PolKindPairLs* `->` *Kind* `)`	Operator kind
		`Type`	Type kind
PolKindPairLs	=	*Polarity* *Kind*	Short polarity-kind list
		Polarity *Kind* *PolKindPairLs*	Long polarity-kind list
KindedPolarityId	=	*Id* `:` *Polarity* *Kind*	Explicitly-kinded identifier
KindedPolarityIdLs	=	*Id* `:` *Polarity* *Kind*	Short kinded-polarity-id list
		Id `:` *Polarity* *Kind* *KindedPolarityIdLs*	Long kinded-polarity-id list
OptKind	=	`:` *Kind*	
		ϵ	

The above concrete syntax definition was automatically derived from the CUP specification. It is therefore not as compact as it can be.

Chapter 6

Syntax Tree Construction

A syntax tree, as defined for the purposes of this discussion, is a data structure that is capable of representing all parse trees of a given context free grammar (CFG). Syntax trees play a vital role in the implementation of a modern compiler. While their exact utilization varies between implementations, their essential role remains to encode a program in a manner that is convenient for subsequent analysis and transformation.

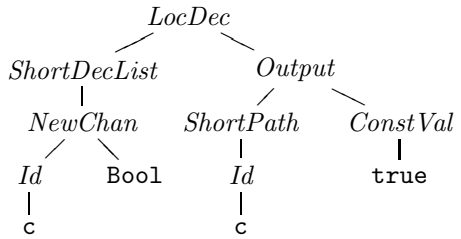
As one could reasonably expect, there exists a close relationship between a CFG and its corresponding syntax tree. In our work we formally define what this relationship is, and consequently develop an algorithm to transform a grammar into a syntax tree (and vice-versa). On the basis of this algorithm, as we describe later in Chapter 9, we mechanically generate the code required to implement a syntax tree.

6.1 Encoding a Program

As described in Chapter 4, the parser transforms an input program into a concrete syntax tree (CST) — the structure of which closely corresponds to that of the input program. Consider for instance, the following trivial Pict program:

```
def a [] = (new c: Bool  c ! true)
```

Here we define a process wherein, first, a boolean channel named `c` is created, and then the value `true` is transmitted along it. As per the syntax of Pict, this process is the right hand side of an abstraction named `a`. The CST for even this small program is eight levels deep and hence too large to elaborate upon here. Thus, in the interest of brevity, we shall confine our discussion to the incomplete program `(new c: Bool c ! true)`. Furthermore, we shall ignore the keywords `new`, `:`, and `!`. Now, our simplified CST would be as follows:



This CST contains seven different kinds of internal nodes: *LocDec* — a local declaration, *ShortDecList* — a single element declaration list, *NewChan* — a new channel declaration, *Id* — an identifier, *Output* — an output process, *ShortPath* — a single element path, and *ConstVal* — a constant value.

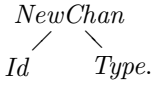
6.2 Implementing a Syntax Tree

When implementing syntax trees in Java, we are faced with fundamental design 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? That is, should syntax trees be *heterogeneous* or should they be *homogeneous*? 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.

6.2.1 Heterogeneous Syntax Trees

In terms of robustness, heterogeneous syntax trees offer a significant advantage over homogeneous syntax trees. In the heterogeneous setting, the type system of the implementation language can be exploited to constrain the shape of a node. Here for example, a *NewChan* node can only be of the form,



The implementation language's type system will prohibit us from having a *NewChan* node with, say three children. This is because our implementation would have a class called `NewChan`, whose constructor would only accept input of type $\langle$`Id`, `Type`$\rangle$. By contrast, in a homogeneous setting, since we would only have a class called `Node`, its constructor would not be able to impose any such constraints. As a result, we can write valid code that generates a nonsensical syntax trees.

In our work we utilize a heterogeneous syntax tree, as outlined by Appel [4, Chapter 4]. The code for this data structure is statically typed, and does not require on any type casting. The main point about this implementation is that, different kinds of nodes are instances of different classes. For example, a *NewChan* node would be an instance of class `NewChan`. This class would have a constructor of the form,

```
NewChan(Id iden, Type ty).
```

Naturally, this enforces the constraint that a *NewChan* node must only consist of an *Id* node and a *Type* node.

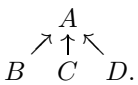
At first glance there may appear to be a discrepancy in our above discussion. Notice that, in the figure shown in Section 6.1, the right child of node *NewChan* is node `Bool`, and not node *Type*, as the above constructor would suggest. Nevertheless this is correct, since in our definition `Bool` is a subclass of `Type`. As we will show next, such subclass relationships are not arbitrarily imposed, but in fact arise quite naturally.

6.3 Deriving an Inheritance Hierarchy

The subclass relationship between classes is of course defined by the inheritance hierarchy of the syntax tree. The inheritance hierarchy however is designed to closely parallel the production rules of the underlying grammar. For instance a grammar fragment such as

$$A \longrightarrow B \mid C \mid D$$

would result in the inheritance hierarchy



Observe that according to the grammar, B, C or D can be used wherever A is expected, and according to the inheritance hierarchy, instances of class B, class C or class D can be used wherever an instance of class A is expected.

Any CFG can be modeled in terms of an inheritance hierarchy by rewriting it such that all its productions are of the form

$$A \longrightarrow \alpha \mid \beta \mid \ldots \mid \omega$$

or

$$A \longrightarrow \alpha\ \beta\ \ldots\ \omega$$

where A is a nonterminal and $\alpha, \ldots, \omega$ are either nonterminals or terminals. We henceforth refer to a grammar in this format as being in *inheritance normal form* (INF).

As a rule of thumb, in the former case, class A would be a direct superclass of classes $\alpha, \ldots, \omega$ (i.e., there is an *is a* relationship between classes $\alpha, \ldots, \omega$ and class A). In the latter case, class A would have member variables of types $\alpha, \ldots, \omega$ (i.e., there is a *has a* relationship between class A and classes $\alpha, \ldots, \omega$).

In the special case, where we have a rule of the form $A \longrightarrow \alpha$, and there is no other rule which has A on its right-hand side, we have two options available to us. We can either make α a subclass of A or make α an instance variable in A. Despite the fact that the former option seems more natural, we choose to adopt the latter one, for reasons we describe in Section 6.3.3.

6.3.1 Transforming a Grammar to INF

It is trivial to rewrite a grammar so that it is in INF. Informally, an algorithm to do this would involve converting rules of the form

$$A \longrightarrow \alpha \ldots \omega \mid \ldots$$

to

$$\begin{array}{lcl} A & \longrightarrow & A' \mid \ldots \\ A' & \longrightarrow & \alpha \ldots \omega \end{array}$$

where A' is a newly introduced fresh non-terminal. In order to formally define our transformation to INF we adopt the following conventions when discussing grammars:

- Letters A, B, and C denote nonterminals. Letters a, b, and c denote terminals. Letters α, β and γ denote either terminals or nonterminals.
- A sequence of the form $\alpha_1 \ldots \alpha_n$ has length at least 1. Likewise the sequence $\alpha_1\alpha_2 \ldots \alpha_n$ has length at least 2.
- Grammars are represented by the tuple $\langle N, T, P, S\rangle$, where N is a set of non-terminals, T is a set of terminals, P is a set of productions, and S is the start symbol.

Definition 6.1 *A nonterminal A is complex if there are distinct production rules of the form $A \longrightarrow \alpha_1\alpha_2 \ldots \alpha_n$, and $A \longrightarrow \beta_1 \ldots \beta_m$.*

We now define a single step of our transformation, wherein we replace a production that violates INF with a pair of productions that satisfy it.

Definition 6.2 *A transformation relation $\rightarrow$ on grammars labeled by nonterminals is defined by*

$$\langle N, T, P, S\rangle \xrightarrow{A'} \langle N', T, P', S\rangle$$

where

$$\begin{array}{lcl} N' & = & N \cup \{A'\} \\ P' & = & (P \setminus \{A \longrightarrow \alpha_1\alpha_2 \ldots \alpha_n\}) \cup \{A \longrightarrow A', A' \longrightarrow \alpha_1\alpha_2 \ldots \alpha_n\} \end{array}$$

if

- $A \longrightarrow \alpha_1\alpha_2 \ldots \alpha_n \in P$,

- *A is complex and*
- $A' \notin N$.

Definition 6.3 *The transformation relation* $\Rightarrow$ *on grammars labeled by sets of nonterminals is defined by*

$$\langle N, T, P, S\rangle \overset{\{A_1,\ldots,A_n\}}{\Longrightarrow} \langle N', T, P', S\rangle$$

if

$$\langle N, T, P, S\rangle \xrightarrow{A_1} \cdots \xrightarrow{A_n} \langle N', T, P', S\rangle \not\rightarrow .$$

The notation $\not\rightarrow$ implies that no additional $\rightarrow$ transformation are possible. Note that according to this definition the transformation $G \overset{\mathcal{A}}{\Longrightarrow} G'$ introduces the set of fresh nonterminals $\mathcal{A}$ in G'.

Proposition 6.1

1. *For every grammar G there exists a grammar G′ such that* $G \Rightarrow G'$.
2. *If* $G \Rightarrow G'$ *then G′ is in INF.*

Proof Both follow from the observation that each single transformation step

- removes a production which causes the grammar not to be in INF and
- does not introduce any productions which would cause the grammar not to be in INF.

□

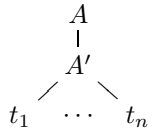
6.3.2 Preservation of Structure

An important property of the transformation relations $\rightarrow$ and $\Rightarrow$ is that derivation trees obtained from the grammar before transformation are *similar* to those obtained after transformation. By *similar* we mean that the transformation operation does not alter the intended meaning of the grammar.

It can be argued that the INF is merely a special case of Chomsky Normal Form (CNF), and hence our transformations are redundant. However, a transformation to CNF 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.

6.3.2.1 Collapsing a Derivation Tree

Note that if a transformation introduces a nonterminal A' then a derivation tree containing A' has a subtree of the form



where the nonterminal A was already present in the original grammar. Therefore a derivation tree of the transformed grammar can be collapsed to a derivation tree of the original grammar. This collapse is formalized as follows.

Definition 6.4 *The* collapse $col_{\mathcal{A}}(t)$ *of derivation tree* t *with respect to the set of nonterminals* $\mathcal{A}$ *is defined by*

$$col_{\mathcal{A}}(a) = a$$

$$col_{\mathcal{A}}\left(\begin{array}{c} A \\ | \\ A' \\ t_1 \quad \cdots \quad t_n \end{array}\right) = \begin{array}{c} A \\ col_{\mathcal{A}}(t_1) \quad \cdots \quad col_{\mathcal{A}}(t_n) \end{array}$$

$$col_{\mathcal{A}}\left(\begin{array}{c} A \\ t_1 \quad \cdots \quad t_n \end{array}\right) = \begin{array}{c} A \\ col_{\mathcal{A}}(t_1) \quad \cdots \quad col_{\mathcal{A}}(t_n) \end{array}$$

where $A' \in \mathcal{A}$ *and* $A \notin \mathcal{A}$.

First, we show that each transformation step preserves most of the structure of the derivation trees. In the following proposition we use $\mathcal{T}_\alpha(G)$ to denote the collection of derivation trees of grammar G with root α.

Proposition 6.2 *Let* $G \xrightarrow{A'} G'$.

1. *For all* $t \in \mathcal{T}_\alpha(G)$ *there exists a* $t' \in \mathcal{T}_\alpha(G')$ *such that* $col_{\{A'\}}(t') = t$.
2. *For all* $t' \in \mathcal{T}_\alpha(G')$ *there exists a* $t \in \mathcal{T}_\alpha(G)$ *such that* $col_{\{A'\}}(t') = t$.

Proof We only prove the first part. The second part can be dealt with similarly. The proof of the first part is by structural induction on t. The case where the tree t consists of a single terminal is trivial. Now assume that the tree t is of the form

$$\begin{array}{c} A \\ t_1 \quad \cdots \quad t_n \end{array}$$

where the subtrees $t_1, \ldots, t_n$ have roots $\alpha_1, \ldots, \alpha_n$, respectively. By induction, there exist trees $t'_i \in \mathcal{T}_{\alpha_i}(G')$ such that $col_{\{A'\}}(t'_i) = t_i$ for $i = 1, \ldots, n$. Clearly the production $A \to \alpha_1 \ldots \alpha_n \in P$. We distinguish the following two cases.

- If the production $A \to \alpha_1 \ldots \alpha_n$ is removed in the transformation then $A \to A', A' \to \alpha_1 \ldots \alpha_n \in P'$. Hence,

  $$t' = \begin{array}{c} A \\ | \\ A' \\ t'_1 \quad \cdots \quad t'_n \end{array} \in \mathcal{T}_A(G')$$

  and $col_{\{A'\}}(t') = t$.
- If the production $A \to \alpha_1 \ldots \alpha_n$ is not removed in the transformation then

  $$t' = \begin{array}{c} A \\ t'_1 \quad \cdots \quad t'_n \end{array} \in \mathcal{T}_A(G')$$

  and $col_{\{A'\}}(t') = t$.

□

Next, we show that our transformation also has this property. Below, we write $\mathcal{T}(G)$ to denote the collection $\mathcal{T}_\alpha(G)$ where α is any nonterminal in grammar G.

Proposition 6.3 *Let* $G \stackrel{\mathcal{A}}{\Longrightarrow} G'$.

1. *For all* $t \in \mathcal{T}(G)$ *there exists a* $t' \in \mathcal{T}(G')$ *such that* $col_{\mathcal{A}}(t') = t$.
2. *For all* $t' \in \mathcal{T}(G')$ *there exists a* $t \in \mathcal{T}(G)$ *such that* $col_{\mathcal{A}}(t') = t$.

PROOF To prove the first part we show the above for $G \xrightarrow{A_1} \cdots \xrightarrow{A_n} G'$ by induction on n. The case $n = 0$ is trivial. Now assume that $G \xrightarrow{A_1} G'' \xrightarrow{A_2} \cdots \xrightarrow{A_n} G'$. Let $t \in \mathcal{T}(G)$. According to Proposition 6.2 there exists a $t'' \in \mathcal{T}(G'')$ such that $col_{\{A_1\}}(t'') = t$. By induction there exists a $t' \in \mathcal{T}(G')$ such that $col_{\{A_2,\ldots,A_n\}}(t') = t''$. Hence,

$$\begin{aligned} & col_{\{A_1,\ldots,A_n\}}(t') \\ &= col_{\{A_1\}}(col_{\{A_2,\ldots,A_n\}}(t')) \\ &= col_{\{A_1\}}(t'') \\ &= t. \end{aligned}$$

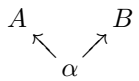
The second part can be proved similarly. □

6.3.3 Handling Multiple Inheritance

While constructing an inheritance hierarchy for a grammar it is quite possible for a class to have more than one superclass. For instance, in the case of the grammar fragment,

$$\begin{aligned} A &\longrightarrow \alpha \mid \ldots \\ B &\longrightarrow \alpha \mid \ldots \end{aligned}$$

(which is already in INF), the resulting inheritance hierarchy is,



That is, class α is a direct subclass of both class A and class B. Of course, this relationship can be easily modeled using multiple inheritance. However, some object oriented languages, such as our implementation language (Java), do not support multiple inheritance. We therefore need to devise a way in which to cope with this limitation.

Our approach to dealing with multiple inheritance is to further normalize an INF grammar in such a way that the need for multiple inheritance simply does not arise. For example, the above grammar would be rewritten as

$$\begin{aligned} A &\longrightarrow M \mid \ldots \\ B &\longrightarrow N \mid \ldots \\ M &\longrightarrow \alpha \\ N &\longrightarrow \alpha \end{aligned}$$

where M and N are fresh nonterminals.

Now according to our rule of thumb, class M would be a subclass of class A and class N would be a subclass of class B. Note however that the productions for M and N are instances of the special case we mentioned earlier. As a consequence class α would be a member variable of class M and of class N. Had we chosen the other option of making class α a subclass of class M and of class N, multiple inheritance would not have been resolved. We refer to INF grammars that have undergone this transformation as being in *single inheritance normal form* (SINF).

6.3.3.1 Transformation to SINF

We now formally define how an INF grammar can be transformed to an SINF grammar. This process is rather similar to the transformation to INF, so we omit parts of it for brevity.

Definition 6.5 *A terminal or nonterminal α is multiparented if there are distinct production rules of the form $A \longrightarrow \alpha$ and $B \longrightarrow \alpha$.*

We redefine Definition 6.2 for "$\rightarrow$" as follows:

Definition 6.6 *A transformation relation $\rightarrow$ on grammars labeled by nonterminals is defined by*

$$\langle N, T, P, S\rangle \xrightarrow{A'} \langle N', T, P', S\rangle$$

where

$$\begin{aligned} N' &= N \cup \{A'\} \\ P' &= (P \setminus \{A \longrightarrow \alpha\}) \cup \{A \longrightarrow A', A' \longrightarrow \alpha\} \end{aligned}$$

if

- $A \longrightarrow \alpha \in P$,
- α *is multiparented and*
- $A' \notin N$.

Clearly Definition 6.3 remains unchanged. Proposition 6.1 can now be restated as:

Proposition 6.4

1. *For every grammar G in INF there exists a grammar G' such that $G \Rightarrow G'$.*
2. *If $G \Rightarrow G'$ then G' is in SINF.*

Proof Similar to the proof of Proposition 6.1. □

6.3.3.2 Preservation of Structure

Similar to what we showed in Section 6.3.2, we now show that going from INF to SINF also preserves the structure of the derivation trees.

Note that if a transformation introduces a nonterminal A' then a derivation tree containing A' has a subtree of the form

$$\begin{array}{c} A \\ | \\ A' \\ | \\ t \end{array}$$

where the nonterminal A was already present in the original grammar. Therefore a derivation tree of the transformed grammar can be collapsed to a derivation tree of the original grammar.

Definition 6.7 *The* collapse $col_{\mathcal{A}}(t)$ *of derivation tree* t *with respect to the set of nonterminals* $\mathcal{A}$ *is defined by*

$$
\begin{array}{rcl}
col_{\mathcal{A}}(a) & = & a \\
col_{\mathcal{A}}\left(\begin{array}{c} A \\ \vert \\ A' \\ \vert \\ t \end{array}\right) & = & \begin{array}{c} A \\ \vert \\ col_{\mathcal{A}}(t_1) \end{array} \\
col_{\mathcal{A}}\left(\begin{array}{c} A \\ t_1 \quad \cdots \quad t_n \end{array}\right) & = & \begin{array}{c} A \\ col_{\mathcal{A}}(t_1) \quad \cdots \quad col_{\mathcal{A}}(t_n) \end{array}
\end{array}
$$

where $A' \in \mathcal{A}$ *and* $A \notin \mathcal{A}$.

Based on the above definition we can prove that the SINF transformation preserves the structure of the derivation trees, like we showed for the INF transformation.

6.3.4 Multiple Inheritance Via Interfaces

Java does support the concept of *interfaces*, which can be used to roughly emulate multiple inheritance. It is therefore possible to model an INF grammar using interfaces rather than inheritance. For instance, the hierarchy

$$
\begin{array}{ccc}
A & & B \\
& \nwarrow \quad \nearrow & \\
& C &
\end{array}
$$

can be coded in Java as `class C implements A, B { ... }`.

We however, choose not to use this feature since it does not allow for code sharing. Instead we use abstract super classes.

As a consequence we are constrained to an implementation which does not support multiple inheritance. However, since the need for multiple inheritance seldom arises while modeling the Pict abstract syntax this fact is of little consequence.

Chapter 7

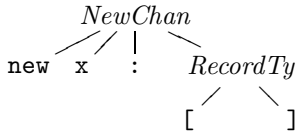
Abstract Syntax Design

To proceed with the task of semantic analysis, we need a suitable data structure to encode our input. The concrete syntax tree (CST) that we constructed upon parsing the input, (see Chapter 4), can be looked at as a possible candidate for this task. However, as we will shortly explain, it is not necessarily the most convenient one. Having finished with the syntactic phase of compilation, we seek a data structure that is abstracted from a program's syntactic minutiae. The structure that we use for this purpose is called an abstract syntax tree (AST), named so because, like a CST it too is a syntax tree, and the information it contains is more abstract than that contained in a CST.

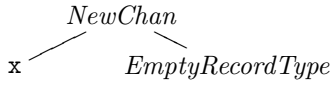
Designing an abstract syntax, and consequently an AST, is largely a subjective matter. Our overall goal is to roughly achieve a one-to-one correspondence between the productions of the abstract syntax and semantic rules of the language. In this chapter we outline some techniques used to achieve this. Of course, what we present is not an algorithm but merely some rules of thumb. The later part of this chapter defines the abstract grammar used in our compiler.

7.1 Removal of Keywords

One of the problems with using a CST to represent a program is that it contains a lot of extraneous information. Since a CST is a parse tree based on the concrete grammar, it will contain nodes for all terminals, including keywords and punctuation. For example, the CST for the Pict program `new x:[]` is



Clearly, if we seek an abstract description of this program, all we need to do is capture the fact that we are declaring an empty `record` type named `x`. Hence, a suitable abstract encoding for this program could simply be



Notice the absence of keywords and punctuation in this encoding. From a semantic point of view keywords and punctuation are of little importance. They are mainly hints given to a parser to enable it to uniquely associate a portion of a program text with a production in the grammar. For instance, without keywords and punctuation the above program would simply be `x`, and consequently nonsensical to parse. However, since an abstract syntax is not intended for parsing, it need not contain all the keywords and punctuation found in a concrete syntax.

What then should the terminals of an abstract syntax be? After all, keywords such as `true`, `false`, and `Int`, for example, convey useful semantic information and cannot just be discarded. A good rule for determining which terminals to preserve, and which to discard is:

> Terminals for which there exists a production whose right-hand side solely consists of the that terminal, should be retained. The rest can be discarded.

For example, in the case of the Pict grammar fragment,

$$\mathit{Type} \longrightarrow \texttt{Int} \mid \texttt{!}\ \mathit{Type},$$

we would retain the terminal `Int` (since it is the sole terminal on the right-hand side), but not the terminal `!`.

7.2 Rearrangement of Productions

One of the steps in obtaining an abstract syntax is to rearrange the rules of the concrete syntax. The purpose of the rearrangement is to have a syntax with production rules that capture those language constructs that are meaningful during semantic analysis.

Consider for example the following grammar for a language that performs logical operations on bits:

$$\begin{aligned} S &\longrightarrow B\ O\ B \\ B &\longrightarrow \texttt{0} \mid \texttt{1} \\ O &\longrightarrow \texttt{and} \mid \texttt{or} \end{aligned}$$

From a naturally occurring semantic point of view there are two main constructs in this language, conjunction (C) and disjunction (D). We would therefore like our abstract grammar to explicitly contain production rules for C and D. The following is one such grammar:

$$\begin{aligned} S &\longrightarrow C \mid D \\ C &\longrightarrow B\ B \\ D &\longrightarrow B\ B \\ B &\longrightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

The syntax tree for this grammar, as we explained in Chapter 6, will contain distinct nodes for C and D. In an object oriented setting classes `C` and `D` will be equipped with the methods required to process them, and class S would contain abstract declarations of these methods. Say one such method is called `compile`, and we are given a node `s` of type `S` to compile. We could very elegantly achieve this by the code fragment `s.compile()`. If instead we did not perform this rearrangement, `S` would not be a superclass, and its `compile` method would roughly be implemented as follows:

```
if isConjunction() then ...     elif isDisjunction() then ...     else RUN-TIME-ERROR
```

The advantages of the rearrangement should now be obvious. Not only is the resulting code far more elegant, it is also much safer, since in a language like Java, it is guaranteed to be free from runtime failure.

7.3 Representation of Lists

Often a grammar contains rules to represent a list like structure. For instance consider the grammar,

$$P \longrightarrow \texttt{x}\ P \mid \texttt{x},$$

which produces a list of xs. Even though the above concrete syntax makes no reference to an *end marker*, we find it useful to introduce such a construct in the abstract syntax. With this in mind our abstract syntax would be,

$$P \longrightarrow \texttt{x}\ P \mid \epsilon.$$

Introducing the end marker (i.e., ϵ) in the abstract syntax simplifies the semantic analysis phase. Firstly, it gives us a way to represent an empty list. Secondly, it simplifies defining recursive functions on lists, because we now have a consistent base case (i.e., the empty list).

In passing, note that the abstract grammar allows an empty sequence of xs, while the concrete syntax does not. However this fact is irrelevant, since it is the concrete syntax that is used for parsing.

7.4 The Pict Abstract Syntax

Shown below is the abstract syntax we utilize in our compiler. We stress that this definition is our personal interpretation of how a Pict program ought to be abstractly viewed. It is meant to document the internal workings of our compiler, and is in no way a definition of any aspect of the Pict language.

The grammar shown here is in inheritance normal form (INF). Details about INF can be found in Chapter 6. Here, the start symbol is *Absyn*, nonterminals are shown using *slanted* text, and terminals are shown in `typewriter` text. The terminal `Key` represents an arbitrary identifier, such as the name of a variable.

Absyn	$\longrightarrow$	*Dec* | *Abs* | *Pat* | *FldPat* | *Constr* | *Proc* | *DecLs*
		| *Val* | *Path* | *FldVal* | *Const* | *Type* | *OptKind*
		| *KindPolIdLs* | *TypeLs* | *FldTy* | *RType*
		| *Polarity* | *Kind* | *PolKindPairLs* | *Label*
		| *KindPolId*
Dec	$\longrightarrow$	*NewChan* | *RecDef*
NewChan	$\longrightarrow$	`Key` *Type*
RecDef	$\longrightarrow$	*RecDefElem* | `RecDefEnd`
RecDefElem	$\longrightarrow$	`Key` *Abs* *RecDef*
Abs	$\longrightarrow$	*ProcAbs*
ProcAbs	$\longrightarrow$	*Pat* *Proc*
Pat	$\longrightarrow$	*Var* | *RecordPat* | *RecurPat* | *Wild* | *Layer*
Var	$\longrightarrow$	`Key` *RType*
RecordPat	$\longrightarrow$	*RecordPatElem* | `RecordPatEnd`
RecordPatElem	$\longrightarrow$	*RecordPatElemWithLabel*
		| *RecordPatElemWithoutLabel*
RecordPatElemWithLabel	$\longrightarrow$	`Key` *FldPat* *RecordPat*
RecordPatElemWithoutLabel	$\longrightarrow$	*FldPat* *RecordPat*
RecurPat	$\longrightarrow$	*RType* *Pat*
Wild	$\longrightarrow$	*RType*
Layer	$\longrightarrow$	`Key` *RType* *Pat*
FldPat	$\longrightarrow$	*ValFldPat* | *TypeFldPatKindConstr*
		| *TypeFldPatSubtypeConstr*
		| *TypeFldPatEqConstr*
ValFldPat	$\longrightarrow$	*Pat*
TypeFldPatSubtypeConstr	$\longrightarrow$	`Key` *SubtypeConstr*
TypeFldPatKindConstr	$\longrightarrow$	`Key` *KindConstr*
TypeFldPatEqConstr	$\longrightarrow$	`Key` *EqConstr*
Constr	$\longrightarrow$	*TypeConstr* | *KindConstr*
TypeConstr	$\longrightarrow$	*SubtypeConstr* | *EqConstr*

SubtypeConstr ⟶ *Type*
EqConstr ⟶ *Type*
KindConstr ⟶ *Kind*
Proc ⟶ *Output* | *Input* | `NullProc` | *LocDec*
| *IfThen* | *ParProc*
Output ⟶ *Val* *Val*
Input ⟶ *Val* *Abs*
LocDec ⟶ *DecLs* *Proc*
IfThen ⟶ *Val* *Proc* *Proc*
ParProc ⟶ *ParProcElem* | `ParProcEnd`
ParProcElem ⟶ *Proc* *ParProc*
DecLs ⟶ *DecLsElem* | `DecLsEnd`
DecLsElem ⟶ *Dec* *DecLs*
Val ⟶ *ConstVal* | *ValPath* | *RecordVal*
| *RecurVal*
ConstVal ⟶ *Const*
ValPath ⟶ *Path*
RecordVal ⟶ *RecordValElem* | `RecordValEnd`
RecordValElem ⟶ *Label* *FldVal* *RecordVal*
RecurVal ⟶ *RType* *Val*
Path ⟶ *PathElem* | `PathEnd`
PathElem ⟶ *Path* `Key`
FldVal ⟶ *ValFldVal* | *TypeFldVal*
ValFldVal ⟶ *Val*
TypeFldVal ⟶ *Type*
Const ⟶ *StrConst* | *CharConst* | *IntConst* | `True`
| `False`
StrConst ⟶ `String`
CharConst ⟶ `Character`
IntConst ⟶ `Integer`
Type ⟶ `TopTy` | *TyIden* | *InOutTy* | *OutTy* | *InTy*
| *RespOutTy* | `IntTy` | `CharTy` | `BoolTy` | *TyOp*
| `StrTy` | *RecordTy* | *TyApp* | *RecTy*
TyIden ⟶ `Key`
InOutTy ⟶ *Type*
OutTy ⟶ *Type*
RespOutTy ⟶ *Type*
InTy ⟶ *Type*
RecordTy ⟶ *RecordTyElem* | `RecordTyEnd`
RecordTyElem ⟶ *Label* *FldTy* *RecordTy*
TyOp ⟶ *Type* *KindPolIdLs*
TyApp ⟶ *Type* *TypeLs*
RecTy ⟶ `Key` *Kind* *Type*
OptKind ⟶ *ExpKind* | `OmitKind`
ExpKind ⟶ *Kind*
KindPolIdLs ⟶ *KindPolIdLsElem* | `KindPolIdLsEnd`
KindPolIdLsElem ⟶ `Key` *Polarity* *Kind* *KindPolIdLs*
TypeLs ⟶ *TypeLsElem* | `TypeLsEnd`
TypeLsElem ⟶ *Type* *TypeLs*
FldTy ⟶ *ValFldTy* | *TypeFldTySubtypeConstr*
| *TypeFldTyKindConstr*

		| *TypeFldTyEqConstr*
ValFldTy	⟶	*Type*
TypeFldTySubtypeConstr	⟶	`Key` *SubtypeConstr*
TypeFldTyKindConstr	⟶	`Key` *KindConstr*
TypeFldTyEqConstr	⟶	`Key` *EqConstr*
RType	⟶	`OmitTy` | *ExpTy*
ExpTy	⟶	*Type*
Polarity	⟶	`MixPol` | `PosPol` | `NegPol` | `ConstPol`
Kind	⟶	*OpKind* | `TyKind`
OpKind	⟶	*PolKindPairLs* *Kind*
PolKindPairLs	⟶	*PolKindPairLsElem*
		| `PolKindPairLsEnd`
PolKindPairLsElem	⟶	*Polarity* *Kind* *PolKindPairLs*
KindPolId	⟶	*ExpKindPolId*
ExpKindPolId	⟶	`Key` *Polarity* *Kind*
Label	⟶	`AnonLabel` | *ExpLabel*
ExpLabel	⟶	`Key`

Chapter 8

Syntax Tree Implementation

There are several tools that can mechanically generate the code for a syntax tree data structure. *JavaCC* [15] and *SableCC* [8] are two of the better known tools to perform this task. We however do not use either of these tools, preferring instead to use a simple syntax tree generation tool, called *Grappa*, that we developed ourselves.

We have found that in several popular object oriented compiler construction methodologies (and toolkits) syntax trees are neither robust nor object oriented. Robustness, for example the prevention of runtime type mismatches, is often sacrificed in favour of reduced code size. Fundamental object oriented principles, such as encapsulation for example, are compromised in the interest of programmer convenience. The syntax tree data structure we utilize however is free from such problems while at the same time is very usable.

8.1 The Need for Automation

A typical heterogeneous syntax tree implementation requires a significant number of classes. While the actual code itself is not particularly complicated, it is voluminous enough to consider automating. For instance, a toy language like Tiger [4, Chapter 3], as implemented by Appel, requires in excess of 35 different classes. Our implementation of Pict, which is relatively small and developed along similar lines, requires in excess of 110 classes. In general, the number of classes required will roughly equal to the number of production rules in the grammar. Hence, in the case of a large language like Java, we would require over 270 classes!

A syntax tree is intrinsically linked with a language's grammar, and therefore, updates to the grammar must be reflected in the syntax tree implementation. Given the large number of classes involved, and the interdependency between them, this task is particularly cumbersome to perform manually. Furthermore, changes to the grammar (such as resolving an ambiguity) are quite common during development, and hence the frequent burden of updating the syntax tree implementation is undesirable.

8.1.1 JavaCC

JavaCC is a commonly used toolkit for constructing compilers in Java. *JJTree* is a component of it which can be used to generate code for syntax trees. While *JJTree* may be a viable option for some, we however do not use it because it offers us neither flexibility nor robustness nor readability.

- Using *JJTree* requires that *JavaCC* be used for lexical analysis and parsing. However, as described in Chapter 2 and Chapter 4, we find *JFlex* and *CUP* are a superior alternative for the syntactic phase of a compiler.
- The syntax tree generated by *JJTree* is a homogeneous tree, whose nodes are indistinguishable to the type system. This makes our implementation inherently fragile, as per our discussion in Section 6.2.1.
- The *JJTree* specification is rather unwieldy. It is essentially a union of the parser specification, the syntax tree specification, and the user defined action code. Consequently it is difficult to read.

Considering our strong preference for robust and readable code, we find *JJTree* to be a poor candidate for generating syntax trees.

8.1.2 SableCC

SableCC is another compiler construction toolkit for Java. Like *JavaCC*, it too can mechanically generate syntax trees from a context free grammar. In many ways it is superior to *JavaCC*. Not only is its input specification very readable, the syntax trees that it generates are also heterogeneous in nature.

Our main problem with *SableCC* is its tree traversal mechanism. In it, *SableCC* creates a *tree walker* class that knows how to traverse a syntax tree. We then extend this class in a manner that allows us to process the tree as its nodes are encountered. For a simplified discussion on tree walkers consider the grammar from Section 4.1, restated below:

$$
\begin{array}{rcl}
\textit{Dec} & \longrightarrow & \texttt{def}\ \textit{DefList} \\
\textit{DefList} & \longrightarrow & \textit{Id}\ \textit{Abs} \mid \textit{Id}\ \textit{Abs}\ \texttt{and}\ \textit{DefList} \\
\textit{Abs} & \longrightarrow & \textit{Id}\ \texttt{=}\ \textit{Proc} \\
\textit{Proc} & \longrightarrow & \texttt{( )} \\
\textit{Id} & \longrightarrow & \texttt{a} \mid \texttt{b} \mid \texttt{x} \mid \texttt{y}.
\end{array}
$$

Assuming that we had a *SableCC* specification for it, we could generate the following tree walker object:

```
class TreeWalker {
  ... internal traversal code ...

  void inDec(Dec node) {};
  void outDec(Dec node) {};
  ⋮
  void inProc(Proc node) {};
  void outProc(Proc node) {};

}
```

The methods `in`*Node* and `out`*Node* get called every time node *Node* is entered or exited. For instance, the `inDec` method is called when entering a `Dec` node, and the `outDec` method is called when exiting it. The intent is for us to override these methods, so that our custom code gets to run at entry and exit points instead. For instance, if we wished to perform scope checking, we would write a `ScopeChecker` class such as the following:

```
class ScopeChecker extends TreeWalker {
  void inDec(Dec node) {  ... scope checking code ...  };
  ⋮
}
```

While this appears to be an elegant solution, we find it to be non object oriented, inflexible, and fragile.

Our first objection to this approach is that it is non object oriented. The core tenants of object oriented design dictate that a `Dec` object should know how to scope check itself. Yet here, we do the exact opposite, placing the scope checking code outside the `Dec` object.

Secondly, a tree walker is quite inflexible in the kinds of traversals it can perform. We often need the ability to perform arbitrary traversals. For instance, scope checking would require us to traverse every node in the tree, but kind checking would only require us to traverse nodes pertaining to type information. More importantly, in the case of a language like Pict (that supports mutually recursive definitions), we need at any given point, the ability to revisit an arbitrary node of the syntax tree. Currently *SableCC* only knows

how to traverse a tree in either a preorder or a postorder fashion. Consequently, it does not provide us the flexibility that we need.

Finally, based on the signatures of methods such as `inDec`, there is no provision to pass extra parameters (such as a symbol table for instance), nor is there a way to return a result. The option that *SableCC* provides, involves using global variables, and requires type casting. This approach, is in our opinion fraught with danger, and quite fragile.

In hindsight however, we could have used *SableCC* to just construct our syntax trees, and ignored its tree walker mechanism. However, in the interest of maximum flexibility we developed a simple tool to generate our syntax tree.

8.1.3 Grappa

Grappa is a tool to generate a syntax tree from a CFG. In its original incarnation (that we developed), it was a simple Perl script, but it was subsequently reimplemented by Apostoloiu [3], and packaged with the *Corretto* parser generator. *Corretto* is described in Chapter 4.

The input to *Grappa* is not a CFG, but is instead a *Grappa* specification. It is the job of *Corretto* to generate this specification. Considering that a *Grappa* specification is seldom written by hand, we omit discussing it here (we refer the interested reader to [3]), and focus instead on *Grappa*'s output. The *Grappa* output corresponding to the *Corretto* fragment,

```
DefList := % Single element definition list %
           ShortDefList
           % i: name of abstraction,
             a: body of abstraction  %
           Id Abs.
```

from Section 4.2.3 is,

```
/**
 * Single element definition list.
 */
class ShortDefList extends DefList {
  private i;
  private a;

  /**
   * Single element definition list.
   *
   * @left position in input of left side of token represented by the node.
   * @right position in input of right side of token represented by the node.
   * @param i name of abstraction.
   * @param a body of abstraction.
   */
  ShortDecList(int left, int right, Id i, Abs a) {
    super(left, right);
    this.i = i;
    this.a = a;
  }
}.
```

While conceptually, *Grappa*'s output is similar to *SableCC*'s, it differs in the quantity of meaningful documentation. *Grappa*'s output is extensively documented using *JavaDoc* comments, while *SableCC*'s is not. This may seem frivolous to document mechanically generated code, but in the case of a syntax tree,

this documentation is particularly useful. Such documentation is frequently consulted when implementing later phases of the compiler. We, for instance, heavily rely on this documentation when writing our scope checking (Chapter 11) and kind checking (Chapter 12) rules.

Finally, notice how a single production results in over 30 lines of Java code. In our opinion, this fact alone justifies the need to automate the generation of a syntax tree data structure.

Chapter 9

Syntax Tree Transformation

In Chapter 6 we addressed the issue of how to construct a concrete syntax tree (CST). In this chapter we look at how to transform a CST into an abstract syntax tree (AST). It is only after we have an AST, that we can begin the task of semantic analysis.

Manually implementing a transformation from a CST to an AST is a tedious process. This is mainly so because a typical CST, as we described in Chapter 6, consists of a large number of different classes of nodes. In order to transform it, each class of nodes needs to be considered individually. Consequently there are a large number of cases to consider. In our implementation of Pict there are in excess of a 100 such cases, but for a large language like Java there will be many more. Furthermore, since the structure of an AST is subject to frequent modifications, so too are such transformation functions.

In our work we develop a high level language to specify a transformation between two syntax trees. Additionally, we develop a program that converts a specification written in this language into Java code that can implement it. By doing so we greatly reduce the tedium associated with creating and maintaining tree transformation functions. Additionally, in keeping with the design goals of our compiler, the emitted code is documented, object oriented, and free from potential type mismatches at runtime.

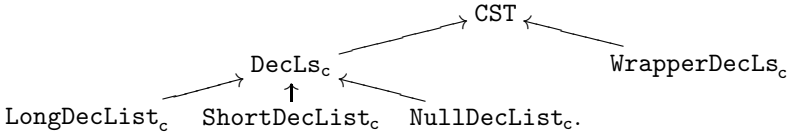
9.1 A Sample Transformation

The following is a sample transformation of a CST to an AST. The grammars used in this example are similar to what is used in our actual implementation. However, it should be borne in mind that for the purpose of illustration, these grammar contain some superfluous productions.

Consider the following fragment of concrete grammar:

$$
\begin{array}{rcl}
\mathit{DecLs}_c & \rightarrow & \mathit{LongDecList}_c \mid \mathit{ShortDecList}_c \mid \mathit{NullDecList}_c \\
\mathit{LongDecList}_c & \rightarrow & \mathit{Dec}_c : \mathbf{dec}_c \;\; \mathit{DecLs}_c : \mathbf{decList}_c \\
\mathit{ShortDecList}_c & \rightarrow & \mathit{Dec}_c : \mathbf{dec}_c \\
\mathit{NullDecList}_c & \rightarrow & \epsilon \\
\mathit{WrapperDecLs}_c & \rightarrow & \mathit{DecLs}_c : \mathbf{decList}_c
\end{array}
$$

We adopt the convention that **boldface** entries in the grammar are labels. These labels only provide a means to refer to individual components of a production. A similar notation is utilized in parser generators such as *CUP* [11] and *ANTLR* [19]. Also, for the sake of brevity we omit listing the productions for the nonterminal Dec_c. According to our discussion on heterogeneous syntax trees in Section 6.2.1, the inheritance class hierarchy for the corresponding CST would be,



For reasons we explained in Chapter 7 our goal is to transform parse trees corresponding to above concrete grammar, into parse trees corresponding to the following abstract grammar:

$$\begin{array}{rcl} DecLs_a & \rightarrow & DecLsEnd_a \mid DecLsElem_a \\ DecLsElem_a & \rightarrow & Dec_a \;\; DecLs_a \\ DecLsEnd_a & \rightarrow & \epsilon \end{array}$$

We would for instance like to transform trees having root $LongDecList_c$ into trees having root $DecLsElem_a$. To achieve this we need to define a CST to AST transformation function $\mathcal{T}$, such as the one shown below:

Domain (CST)		Range (AST)
$LongDecList_c$ $Dec_c \quad DecLs_c$	$\Longrightarrow$	$DecLsElem_a$ $\mathcal{T}(Dec_c) \quad \mathcal{T}(DecLs_c)$
$ShortDecList_c$ Dec_c	$\Longrightarrow$	$DecLsElem_a$ $\mathcal{T}(Dec_c) \quad DecLsEnd_a$
$NullDecList_c$	$\Longrightarrow$	$DecListEnd_a$
$WrapperDecList_c$ $DecLs_c$	$\Longrightarrow$	$\mathcal{T}(DecLs_c)$

We utilize a similar function in our compiler. However, it is specified using a special language that we developed for this purpose. In this language, the above transformation can be very succinctly specified as

```
LongDecList_c   ->  DecLsElem_a(dec_c, decList_c)
ShortDecList_c  ->  DecLsElem_a(dec_c, DecLsEnd_a())
NullDecList_c   ->  DecLsEnd_a()
WrapperDecLs_c  ->  decList_c.
```

Note that in our specification shown above we apply the transformation function $\mathcal{T}$ to the node $DecLs_c$, yet do not specify how to transform $DecLs_c$. In fact we only specify how the leaf nodes of a CST hierarchy should be transformed. Notice that the left hand side of every rule in the specification is indeed a leaf node in the CST hierarchy shown in Section 9.1. However, we will in Section 9.3.3 describe how non leaf nodes of a class hierarchy can be transformed.

9.1.1 Conversion to Java Code

As per our inheritance hierarchy above, we have classes such as `LongDecList_c`, `Dec_c`, and `DecLsElem_a`, to name a few. Since it is instances of these classes, that we wish to transform, it makes sense to equip these classes with a `transform` method. Therefore, in the case of class `LongDecList_c`, we would have a transform method such as,

```
public Absyn transform () {
    return new DecLsElem_a(dec_c.transform(), decList_c.transform());
}.
```

The correspondence between this method, and the rule

```
LongDecList_c -> DecLsElem_a(dec_c, decList_c)
```

should be obvious. In our work we have automated the task of converting such rules into Java code.

9.2 Generalized Return Type

We are faced with an implementation dilemma regarding the return type of a `transform` method. For reasons we will elaborate on shortly, all `transform` methods, regardless of the class to which they belong, have a return type of `Absyn`, where `Absyn` is the root node of the AST hierarchy. For all practical purposes this means that `transform` maps its entire domain to a single point (i.e., `Absyn`). Clearly this is not what $\mathcal{T}$ is defined to do. For instance, according to the definition of $\mathcal{T}$, a $LongDecList_c$ node maps to a $DecLsElem_a$ node, but according to the signature of $\texttt{LongDecList}_c\texttt{::transform}$, class $\texttt{LongDecList}_c$ maps to class `Absyn`.

9.2.1 Limitation of Java's Type System

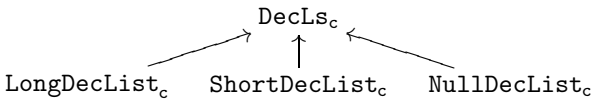
A seemingly obvious resolution to the above dilemma is to assign the appropriate return type to each `transform` method. This would seem trivial to do, as the name of this type is already used in the `new` statement. Assuming such a scenario, the `transform` method for $\texttt{LongDecList}_c$ would be,

```
public DecLsElem_a transform () {
  return new DecLsElem_a(dec_c.transform(), decList_c.transform());
},
```

and that for $\texttt{NullDecList}_c$ would be,

```
public DecLsEnd_a transform () {
  return new DecLsEnd_a();
}
```

Recall however, that the inheritance hierarchy for above classes is,



Of course, each of $\texttt{LongDecList}_c$, $\texttt{ShortDecList}_c$, and $\texttt{NullDecList}_c$ have a `transform` method. However, in order to support a call like $\texttt{decList}_c\texttt{.transform()}$ (where $\texttt{decList}_c$ is of type $\texttt{DecLs}_c$), class $\texttt{DecLs}_c$ needs a transform method too. What then would the return type of this `transform` method be?

An unfortunate limitation of Java is that in a given inheritance hierarchy one method can only override another method if the return types of both methods are identical. In our case this would mean that the `transform` methods of $\texttt{DecLs}_c$, $\texttt{LongDecList}_c$, $\texttt{ShortDecList}_c$, and $\texttt{NullDecList}_c$ all must have the exact same return type. Clearly, since the `transform` methods for $\texttt{LongDecList}_c$ returns $\texttt{DecLsElem}_a$, and that of $\texttt{NullDecList}_c$ returns $\texttt{DecLsEnd}_a$ this requirement of the Java type system cannot be fulfilled.

9.2.2 Possible Remedies

One possible remedy to this conundrum is for all `transform` methods to have a return type of class `Object`. Such a solution is typical of what is found in the Java standard libraries. However, we believe that this solution is dangerously general. In such a scenario the following clearly incorrect `transform` method would successfully compile:

```
public Object transform () { return "nonsense"; }
```

Since our `transform` methods are automatically generated we would very much like for our implementation language's type system to detect such obvious errors. The more constraints that a type system is able to provide, the better we are able to verify the correctness of our generated code.

Working within the limitation of Java's type system the best we can seem to do is to use class `Absyn` as the return type of our `transform` function. Since `Absyn` is the root of the AST class hierarchy, it is general enough to utilize as the return type of any `transform` method. While not perfect, it would prevent nonsensical code, such as the above example, from compiling. However, the unpleasant fact remains that the return type of a `transform` method is more general than it needs to be, and consequently the signature of a `transform` method is not consistent with the definition of $\mathcal{T}$.

9.3 Type Mismatches

Having settled for an imperfect definition of `transform`, we now address the type mismatches that arise as a result of this decision.

Recall from our discussion in Section 6.2.1 on heterogeneous syntax trees, that in order to construct a node N, it must be supplied with nodes $N_1, \ldots, N_n$, such that $N \longrightarrow N_1 \ldots N_n$ is a production in the corresponding syntax. For instance, in the case of our abstract syntax specified in Section 9.1, there exists a production of the form $DecLsElem_a \longrightarrow Dec_a\ DecLs_a$. Since a $DecLsElem_a$ node can only be formed given a Dec_a node and a $DecLs_a$ node, the constructor for class $\texttt{DecLsElem}_\texttt{a}$ will have a signature of the form,

$\texttt{DecLsElem}_\texttt{a}\texttt{(Dec}_\texttt{a}\texttt{, DecLs}_\texttt{a}\texttt{)}.$

However, the code fragment shown in Section 9.1, invokes this constructor as

$\texttt{new DecLsElem}_\texttt{a}\texttt{(dec}_\texttt{c}\texttt{.transform(), decList}_\texttt{c}\texttt{.transform())}.$

This in fact is a violation of Java's type system. Since all `transform` methods have a return type of `Absyn`, we are attempting to create a $\texttt{DecLsElem}_\texttt{a}$ object from a pair of $\langle \texttt{Absyn}, \texttt{Absyn} \rangle$, rather than a pair of $\langle \texttt{Dec}_\texttt{a}, \texttt{DecLs}_\texttt{a} \rangle$. To correct this we have to resort to type casting. However, as we describe next, we achieve this in a manner that assures us freedom from any runtime type mismatches.

9.3.1 Unsafe Type Conversion

A naive manner in which to perform casting is to perform it solely with the intent of satisfying the type system. In this approach, whenever the type system is expecting an object of type `X`, and we instead have an object of type `Y`, we would simply cast the `Y` object to an `X` object.

Utilizing such an approach in our case, we would notice that the type system expects a tuple of type $\langle \texttt{Dec}_\texttt{a}, \texttt{DecLs}_\texttt{a} \rangle$, and we instead have a tuple of type $\langle \texttt{Absyn}, \texttt{Absyn} \rangle$. So to satisfy the type system we would modify our code to the following form:

$\texttt{new DecLsElem}_\texttt{a}\texttt{((Dec}_\texttt{a}\texttt{)(dec}_\texttt{c}\texttt{.transform()), (DecLs}_\texttt{a}\texttt{)(decList}_\texttt{c}\texttt{.transform()))}.$

Even though such an approach is simple, and guaranteed to fix all compile time type mismatches, it is quite unsafe. The fundamental flaw with it is that it relaxes the constraints of the type system with the sole intent of suppressing compile time errors, whether they be legitimate or not. For instance, when utilizing such an approach, the following meaningless piece of code will compile just as well:

$\texttt{new DecLsElem}_\texttt{a}\texttt{((Dec}_\texttt{a}\texttt{)(new Object()), (DecLs}_\texttt{a}\texttt{)(new Object()))}$

In the interest of good programming practice we reject such an approach.

9.3.2 User Defined Type Conversions

Another possible solution to the casting problem is to simply augment the transformation specification with casting information. For instance, the transformation for $\texttt{LongDecList}_\texttt{c}$ can be specified as,

$\texttt{LongDecList}_\texttt{c}$ -> $\texttt{DecLsElem}_\texttt{a}$((Dec$_\texttt{a}$)dec$_\texttt{c}$, (DecLs$_\texttt{a}$)decList$_\texttt{c}$).

Such an approach places the entire responsibility of type conversion upon the user. A careful user, it can be argued, is less likely to make incorrect type conversion decision, and hence less likely to encounter runtime failure.

Besides its visual unattractiveness, this approach suffers from several fundamental problems. Firstly, it introduces into the specification, information that is only meaningful to the eventual implementation language. Recall, that the reason we are performing type conversion is to compensate for the inadequacies of Java's type system. Hence, if we were to use an implementation language with a more advanced type system, the need for explicit type conversion may not even arise. Furthermore, our goal is to keep our specification, as close as possible to its mathematical definition, and since the mathematical definition makes no mention of type conversion nor should we.

Secondly, determining the correct type to convert to is necessarily not a trivial matter. For example, consider the statement (DecLs$_\texttt{a}$)(decList$_\texttt{c}$.transform()), that we have used above. The specification itself makes no mention of how to convert a $DecLs_c$ node. Hence, there is no immediate reason to believe that the object yielded upon transforming $\texttt{decList}_\texttt{c}$ (of type $\texttt{DecLs}_\texttt{c}$), can be type converted to $\texttt{DecLs}_\texttt{a}$. This is information that needs to be indirectly inferred from the specification, as we describe next. Thus it is unreasonable to expect a user to annotate the specification with type conversion information.

9.3.3 Automatically Inferred Type Conversions

Having realized that blind type conversion is too dangerous, and user defined type conversion too impractical, we develop a simple inference algorithm that suffers from neither of these defects. This algorithm, uses the transformation specification, the CST hierarchy, and the AST hierarchy, to determine the AST node that is yielded as a result of transforming a given CST node.

9.3.3.1 Terminal Nodes

Recall that we mentioned in Section 9.1, that our transformation specification only deals with terminal nodes of the CST hierarchy. Considering that the transformation of terminal nodes is explicitly specified, the task of determining $\mathcal{T}(n)$, where n is a terminal node, is almost trivial. For instance, since we have the rule, $\texttt{LongDecList}_\texttt{c}$ -> $\texttt{DecLsElem}_\texttt{a}$(...), it trivially follows that $\mathcal{T}(LongDecList_c)$ yields $DecLsElem_a$.

A slightly more complicated case arises with rules such as $\texttt{WrapperDecLs}_\texttt{c}$ -> $\texttt{decList}_\texttt{c}$. The difficulty here is that the right hand side of these rules specifies the name of a variable, rather than the name of a class. We refer to such rules as *indirect* rules. They are indirect in the sense that we first have to obtain the class name corresponding to the variable, and then apply $\mathcal{T}$ on that class name.

To handle such a rule we first define a function *typeOf* that when given a variable name returns its corresponding class name. For example, $typeOf(\texttt{decList}_\texttt{c})$ yields $\texttt{DecLs}_\texttt{c}$.

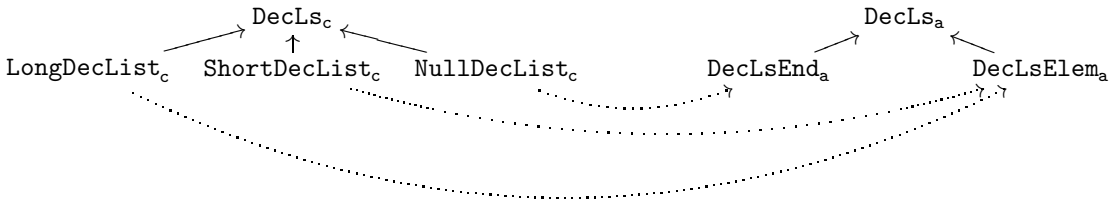
Therefore,

$$\mathcal{T}(\texttt{WrapperDecLs}_\texttt{c}) = \mathcal{T}(typeOf(\texttt{decList}_\texttt{c})) = \mathcal{T}(\texttt{DecLs}_\texttt{c})$$

9.3.3.2 Non Terminal Nodes

We begin by considering the simple task of determining $\mathcal{T}(DecLs_c)$. As we mentioned in Section 9.1, $DecLs_c$ is a non leaf node in the CST hierarchy, and lacks an explicitly specified transformation rule. The key to determining $\mathcal{T}(DecLs_c)$, is to first determine $\mathcal{T}$ for each of the subclasses of $DecLs_c$, and then find a class

common to them all. The following diagram shows the mapping (indicated by dotted arrows) between CST nodes and AST nodes. Note that this mapping information is directly available from the transformation specification.



Intuitively speaking, $\texttt{DecLs}_\texttt{c}$ is a generalized version of $\texttt{LongDecList}_\texttt{c}$, $\texttt{ShortDecList}_\texttt{c}$, and $\texttt{NullDecList}_\texttt{c}$. Hence it is only logical to conclude that class $\mathcal{T}(\texttt{DecLs}_\texttt{c})$ will be a generalized version of $\mathcal{T}(\texttt{LongDecList}_\texttt{c})$, $\mathcal{T}(\texttt{ShortDecList}_\texttt{c})$, and $\mathcal{T}(\texttt{NullDecList}_\texttt{c})$. Determining $\mathcal{T}(\texttt{DecLs}_\texttt{c})$ therefore involves following the dotted arrows, and then ascending up to a common point in the AST hierarchy, namely class $\texttt{DecLs}_\texttt{a}$.

To formalize the above process we begin by introducing the following notational preliminaries:

- The function $leaves(n)$ returns the set of leaf nodes that are reachable from node n. For example, in the case of the CST hierarchy shown in Section 9.3.1

  $$leaves(\texttt{DecLs}_\texttt{c}) = \{\texttt{LongDecList}_\texttt{c},\ \texttt{ShortDecList}_\texttt{c},\ \texttt{NullDecList}_\texttt{c}\}.$$

  Note that if n is a leaf node itself then $leaves(n) = \{n\}$. We often use $l(n)$ as an abbreviation for $leaves(n)$.

- The predicate $ancestor(n_1, n_2)$ yields true if node n_1 is an ancestor of node n_2. Note that we allow a node to be its own ancestor.

- The common ancestors (ca) for a non empty set of nodes N are those nodes from which all nodes in N are reachable. That is,

  $$ca(N) = \{x \mid \forall n \in N :\ ancestor(x, n)\}$$

- The most common ancestor (mca), for a non empty set of nodes N, is that common ancestor which has the greatest depth. That is,

  $$mca(N) \in ca(N) \ \wedge\ \forall x \in ca(N) :\ ancestor(x, mca(N))$$

Based on these definition, $\mathcal{T}$ for a non terminal node n, can be defined as

$$\mathcal{T}(n) = mca\left(\bigcup_{x \in l(n)} \mathcal{T}(x)\right).$$

Chapter 10

Symbol Table Design

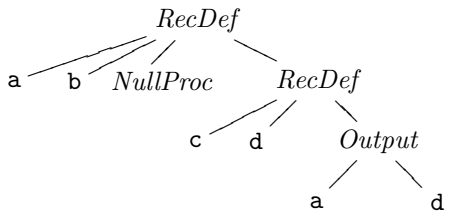
A symbol table is essentially a data structure that allows us to associate with a given symbolic identifier, a set of properties. For example, in the case of the Pict program `new s: String`, we would like to use a symbol table to store the fact that symbol `s`, has been declared to be of type `String`. Any data structure that is capable of storing key-value pairs is a candidate for implementing a symbol table. For a general discussion on symbol table design and implementation we refer the reader to Appel [4], Aho [1] and Watt [27].

In this chapter we discuss some of the core issues pertaining to symbol table design in the context of a multi pass compiler. In particular we emphasis the benefits of using a symbol table based on a persistent data structure.

10.1 Avoiding a Symbol Table

In a single pass compiler the need for a symbol table is quite obvious. Since such a compiler only gets to visit its input once, it must store information about the symbols it encounters, when it encounters them. For instance, when given the input `def a b = () and c d = a!d`, a single pass compiler for Pict would create symbol table entries for `a`, `b`, `c`, and `d` just as soon as it encountered them. Not doing so would result in having no contextual information when processing the expression `a!d`.

In a multi pass compiler however, the need for a symbol table is somewhat debatable. Since such a compiler first converts its input into an abstract syntax tree (AST), every symbol that occurs in the program will also occur in the AST. For instance, the AST for the above program would roughly resemble the following:



When performing semantic analysis on the expression `a!d`, (i.e., the *Output* node), the compiler has at its disposal the entire AST, and consequently every symbol that occurs in the program. For this reason, it can be argued that the AST can itself be used for maintaining symbol information. For instance, the scope checking phase, when examining the expression `a!d`, would require us to verify that symbol `a` has indeed been defined. In the absence of a symbol table we can traverse the AST upwards from the node *Output* until we find a *RecDef* node for symbol `a`.

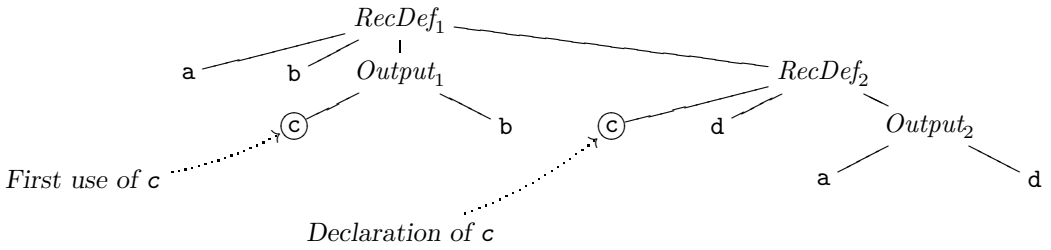
To counter the claim, that repeatedly traversing the AST is inefficient, we point out that the traversal is in an upward direction, and will on average have a cost of $O(\log(n))$, where n is the number of nodes in the AST. To further improve efficiency we can store a pointer from the occurrence of a symbol to its declaration,

thereby making all future lookups instantaneous. However, as we will describe next, there are some serious flaws with this approach.

10.2 Need for a Symbol Table

The approach of using an AST in place of a symbol table is not problem free. We often require certain symbols to be predefined and globally visible throughout a program. The `print` channel is one such symbol in a Pict, as illustrated in the program `def a b = print!b`. The difficulty with handling such a scenario using only an AST is that, there is no node in the AST corresponding to the declaration of symbol `print`. Consequently it is not possible to set or retrieve information about the symbol `print`. Using a symbol table however, such a scenario can be easily handled by initializing the symbol table with the symbol `print`. In general, using an AST for maintaining symbols results in the inability to import symbols into a name space.

An even greater flaw with using only an AST is that it is not always trivial to locate the node in which a symbol was declared. In the examples given above we suggested that looking for the declaration of a node merely involves an upward traversal. Depending on the language's scoping rules this may not always be the case. Consider for instance the valid Pict program `def a b = c!b and c d = a!d`, the AST for which is



This program is valid since Pict allows for mutually recursive definitions. Hence it is possible to use the symbol `c`, in the definition `a`, despite the fact that `c` has yet to be declared. As shown in the above AST, the symbol `c` is used before it is declared. Furthermore, there is no upward path from the node in which it is used, to the node in which it is declared. Hence, when processing the expression `c!b`, it is non trivial to locate the node corresponding to the declaration of symbol `c`. More generally, locating symbols used in mutually recursive definitions could involve us having to search the entire AST. The needless complexity and expense associated with such an operation makes a good case for using a symbol table.

10.3 Conformance to Scoping Rules

Scoping rules, which we shall elaborate on later in Chapter 11, define the visibility of symbols with respect to their location in a program. For example, in the case of the Pict program `def A b = () and C d = ()`, the symbol `b` is only visible in definition `A`. We would therefore like our symbol table to be designed such that symbol `b` is only accessible when we are examining definition `A`. Not doing so may cause an incorrect usage of symbol `b` to go undetected, as in the program `def A b = () and C d = print!b`. Here `b` is not in the scope of definition `C`, but unless after examining definition `A` it is purged from the symbol table, it will remain accessible even when examining definition `C`.

In the case of a single pass compiler it is simple to keep a symbol table consistent with the underlying scoping rules. Such a compiler does not revisit its input, and hence only needs to maintain the scope of the program fragment it is currently examining. Consequently, we are only interested in the current state of the symbol table, and thus are free to add and delete symbols regardless of side effects. That is to say, the symbol table can be *destructively* updated. The ability of performing destructive updates makes it relatively simple to maintain a consistent symbol table. For example, in the case of the program fragment `def a b = () and c d = ()`, a single pass compiler would add the symbols `a` and `b` upon encountering the keyword `def`, and remove the symbol `b` upon encountering the keyword `and`. Considering, that by design, keywords

are intended to trigger such additions and deletions, updating a symbol table in a consistent manner can be quite a routine task.

In the case of a multi pass compiler however, maintaining a consistent symbol table is more challenging. Recall that a multi pass compiler constructs an AST with the intent of repeatedly traversing it (often in an arbitrary manner). Consequently, it is crucial that updates made to the symbol table be retained between traversals. For instance, if in the case of the program `def a b = () and c d = ()`, symbol `b` is removed after the scope checking phase finishes examining definition `a`, it will be erroneously missing when the type checking phase proceeded to examine the definition `a`. This observation quite clearly rules out the possibility of destructively updating the symbol table. What is needed instead is for every node of the AST to have a copy of the symbols it can access. In Section 10.4 we discuss how this can be achieved.

10.3.1 Symbol Name Reuse

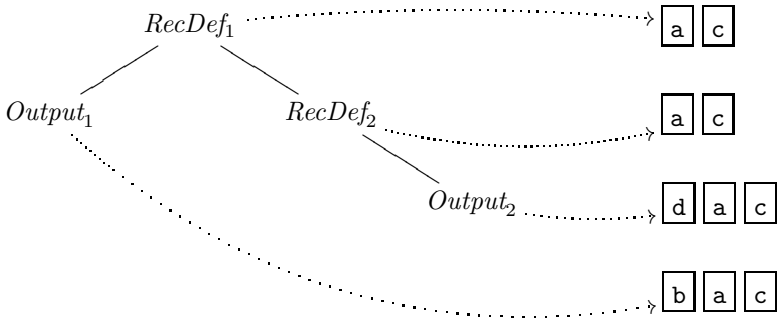
In most programming languages, identifier names do not have to be unique. It is possible therefore to have two distinct identifiers with the same symbolic name. For instance, in the program `def a b = () and c a = print!a and d e = a!2`, there are two distinct identifiers named `a`. The first one is the name of a definition, while the second one is an input parameter. Therefore, when examining definition `c` it is important that symbol `a` be considered as a parameter, but when examining definition `d`, symbol `a` ought to be considered as a definition.

10.4 Symbol Table Data Structures

In Section 10.3 we mentioned that each node of an AST ought to have a list of symbols that it can access. We now consider how this can be achieved. The key to successfully achieving this involves using a persistent data structure.

10.4.1 Per Node Name Spaces

A seemingly obvious manner in which to implement an association between an AST node and its name space is to store within the AST node a pointer to its name space. For instance, in the case of the previous AST (for program `def a b = c!b and c d = a!d`), this association would be implemented as depicted below:



Here a name space is represented by a sequence of boxed symbols, and a name space pointer by a dotted arrow. Note that in the interest of brevity only the non-leaf nodes of the original AST have been shown.

Introducing name space pointers into an AST node is a fairly easy task to accomplish. In an object oriented environment it is also quite natural to do, as we are merely modeling the fact that a node *has a* name space pointer. In the case of a heterogeneous AST implementation such as ours (as described in Section 6.2.1), we would essentially introduce a `namespace` member variable into the top class of the AST hierarchy.

This approach, despite its simplicity, contains several major flaws. Firstly, it is quite space intensive. Since in a given program the same symbol can appear in several scopes, it will therefore need to be repeated

in several different name spaces. In the case of the above example, the symbol `a` is repeated in four different name spaces. In a worst case scenario, all symbols would be visible in all nodes. Since in the case of an n node AST there are $O(n)$ symbols, the worst case space requirement for maintaining name spaces is $O(n^2)$. Note that the worst case scenario is not just a theoretical possibility, and will occur in the case of a language that only supports a global scope. Furthermore, n can be a large number, often in excess of 2000, so an $O(n^2)$ data structure is quite impractical.

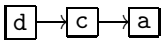
Another problem with this approach is related to updating information about symbols. Say at some point we wished to associate the signature `Int` $\mapsto$ `void` with the symbol `a`. To do so every occurrence of symbol `a` would need to be updated. Such an operation would have a time complexity of $\Omega(n)$, since at the very least every node of the AST would need to be visited. This problem of slow updates can however be alleviated if the name spaces were to store pointers to symbols, rather than the symbols themselves. However, this is not an option we choose to adopt as (a) the additional level of indirection adds an additional level of complexity, and (b) the $O(n^2)$ space requirement still remains.

On the whole, this approach is simple, but is neither space efficient nor time efficient. The duplication of symbols not only wastes memory, but also complicates update operations. To remedy the situation we require a data structure where symbols (or pointers to symbols) are not duplicated.

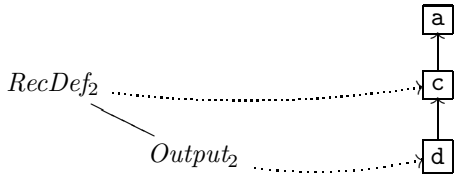
10.4.2 Persistent Symbol Tables

Observe that in an AST, for the most part, the name space of a child node is just an extension of the name space of its parent node. If therefore, we could extend the name space of the parent, without actually altering it, all symbols common to both parent and child would be shared, and hence only have a single occurrence. For instance, the name space of node $RecDef_2$ is {`a`, `c`} and that of its child node $Output_2$ is {`a`, `c`, `d`}.

Assuming that name spaces are implemented using linked lists we could create a list such as,



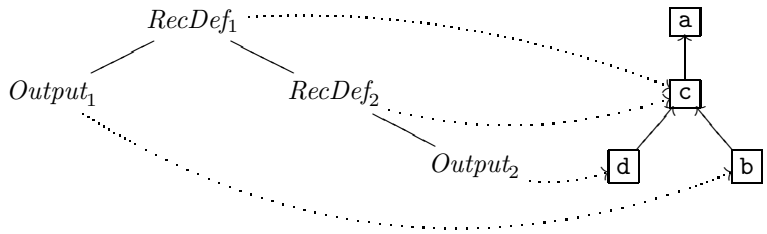
and have node $RecDef_2$ point to symbol `c` and node $Output_2$ point to symbol `d`, as depicted below:



Note that despite the fact that the name space is implemented as a single list, node $RecDef_2$ can only access symbols in its name space (for example it cannot access symbol `d`). Furthermore, there is only one occurrence of symbols `c` and `a`, despite the fact that they occur in two different name spaces.

The most important detail about such a data structure is that the lists are created in a non destructive manner. For example, concatenating symbol `d` with list `[c, a]`, does not alter the state of list `[c, a]`. Hence, any nodes that are pointing to list `[c, a]` can safely continue to do so. Functional programmers would no doubt recognize this concept as being the basis of the `cons` function in LISP, or the `::` operator in SML.

Extending the concept of a persistent symbol table to the entire AST for program `def a b = c!b and c d = a!d`, we would end up with a structure such as the following:



Note that the data structure described above is a persistent list. Consequently when using such a symbol table, there is an $O(1)$ cost for adding a symbol, and an $O(n)$ cost for finding a symbol. Since searching for a symbol is a very frequently performed operation, a data structure with better search capability may be desirable. Appel [4, Chapter 4] recommends that a symbol table be implemented using a persistent AVL tree. This data structure has an $O(\log n)$ time complexity for searching, but is more complicated to implement. In the interest of simplicity we opted to use a list based symbol table instead.

Chapter 11

Scope Checking

Scope checking involves analyzing a program to see if conforms to its language's scoping constraints. The task of scope checking itself is conceptually quite simple. However, some of the design decisions that get made during this process have far reaching consequences.

In our implementation we have attempted to remain true to the object oriented paradigm. We have found that while the utilization of object oriented principles improves code organization, it also creates a considerable amount of programmer overhead, especially when there are a large number of classes involved. In our implementation we have minimized this disadvantage by automating most programmer intensive tasks.

There are two noteworthy points about our scope checking implementation. Firstly, the code to perform scope checking is mechanically generated from a high level specification syntax. Secondly, the design of the scope checking code is more object oriented than what is found in conventional object oriented compiler construction.

11.1 Separate Scope Checking Phase

Traditionally, the task of scope checking is intertwined with the task of type checking. That is, the judgment about *where* a variable gets used is performed in conjunction with *how* it gets used. The primary advantages to such an approach are (a) efficiency — since two tasks are performed in the same phase, and (b) programmer convenience — since separate subroutines for scope checking and type checking need not be coded. This also happens to be the approach taken in the original Pict compiler.

In our compiler however, we decided to clearly differentiate between the task of scope checking and that of type checking. Our decision to do so is primarily based upon the following facts:

- In Pict, both the scope checking rules and the type checking rules are quite complex. Thus combining the tasks of scope checking with that of type checking would hinder maintainability.
- It is not entirely clear how much more efficient the task of semantic analysis becomes when scope checking is combined with type checking. It can be argued to the contrary that by separating the two tasks, and thus detecting scoping errors before beginning type checking, the compilation of incorrectly scoped programs can be aborted sooner.
- The programmer inconvenience of coding separate subroutines for scope checking and type checking is inconsequential. As we describe later, in our compiler the code required to perform these tasks is mechanically generated.

11.2 Code Placement

In an object oriented language there are two logical places where scope checking code can be placed — either it can be internal to the AST, or it can be external. In the former case, scope checking would become a

feature of the AST, while in the latter it would be an operation that gets performed on the AST. From an object oriented design point of view it is much better to make scope checking a feature of the AST. Not only does this improve organization of the code, it also serves to encapsulate it.

Placing code scope checking code inside an AST is quite a straightforward task. It merely involves placing the scoping rule for a given clause into the AST class that represents it. It is however a rather tedious task to do manually. There are after all a large number of scoping rules and AST classes. As we will explain later, in our compiler we have automated the generation and placement of scoping rules, therefore the programmer effort involved with the object oriented approach is inconsequential.

For compiler implementation where programmer effort is an issue, scope checking code (and semantic analysis code in general) is often placed outside the AST. Below we explain some of the more popular alternatives we initially considered.

11.2.1 Syntax Separate From Interpretation

Appel [4] clearly separates the code to build an AST (i.e., syntax related code) from the code that analyzes it (i.e., its interpretation). In this *syntax separate from interpretation* approach, the AST code is kept very simple, consisting of nothing more than member variables and constructors (as described in [4, Chapter 4]). Any analysis that is performed on the AST is done though external functions.

Consider for instance the Pict abstract grammar from Section 7.4. An external function to perform semantic analysis on it would be as follows:

```
function scopecheck (Node: n, SymbolTable: s): Boolean
   if n is Abs
   then
      s1 = ...
      s2 = ...
      return scopecheck(n.pattern, s1) and scopecheck(n.process, s2)
   elif n is IfThen
   then
      s1 = ...
      s2 = ...
      return scopecheck(n.if, s1) and scopecheck(n.then, s2)
   ⋮
   else
      Error: Case not handled
```

This function would of course not be part of the AST. Notice that in this approach there would be one function for every phase of the compiler. Therefore adding a new phase to a compiler is merely a matter of coding a function for that new phase. Like all the other "interpretation" functions it too would do case analysis on each of the AST's nodes.

Appel argues that in a typical language the abstract syntax is relatively static. Therefore, the case analysis structure of the interpretation functions is unlikely to change. However, the compilation phases are liable to change. For example, each machine architecture would have a different code generation phase. It is therefore necessary to be able to add and replace phases as a whole. Since a syntax separate from interpretation approach allows this to happen it provides for a maintainable compiler.

While the above argument itself is correct there are still several disadvantages to this approach.

- The idea of having to do case analysis in an object oriented language is quite undesirable. It is in sharp contrast to the object oriented paradigm, and hence questions the purpose of employing an object oriented language.

- There is no compile time guarantee that each function is able to handle every kind of node in the AST. Observe that in the above code there is always the possibility of reaching the last `else` statement, and hence having a runtime failure.
- In the case of our language Pict, the syntax is quite likely to change. This is contrary to Appel's conjecture that the syntax is relatively stable. We for instance are currently only implementing a core subset of the language, and therefore implementing the entire language would result in significant additions to the AST. Furthermore, Pict itself is a rather new language and hence liable to syntactic changes. This would of course mean that every "interpretation" function would have to be modified in order to accommodate the new clauses.

11.2.2 Visitor Design Pattern

Visitor design [9, Chapter 5] is a variation of the syntax separate from interpretation approach. It too separates the AST code from the semantic analysis code. However it differs from the syntax separate from interpretation approach in the following respects.

- Rather than having a single function that does case analysis on each clause, each clause is placed in a separate function. For instance, the *Abs* clause is moved to function *scopecheckAbs*, and the *IfThen* clause is moved to function *scopecheckIfThen*. These methods are placed in a *Visitor* class, as shown below.

```
Object Visitor
   function scopecheckAbs (Abs: a, SymbolTable: s)
      s1 = ...
      s2 = ...
      return a.pattern.scopecheck(s1) and a.process.scopecheck(s2)

   function scopecheckIfThen (IfThen: i, SymbolTable: s)
      s1 = ...
      s2 = ...
      return i.if.scopecheck(s1) and i.then.scopecheck(s2)

   ⋮
```

- The AST nodes are not completely devoid of methods. Each node will contain a stub method for each compilation phase. For instance the `Abs` class will contain methods called `scopecheck`, `typecheck`, `codegen`, etc. as will the `IfThen` class.

The arguments to the stub methods are the visitor class and a symbol table. Whenever a stub method gets called it immediately invokes its corresponding method from the visitor class, sending as arguments itself and the symbol table. For instance, the scope checking stub method for the `Abs` class would be,

```
function scopecheck (Visitor: v, SymbolTable: s): Boolean
   return v.scopeCheckAbs(this, s).
```

The visitor design pattern offers some major advantages over the plain syntax separate from interpretation method. One of them is that it is possible to ensure that every clause has a semantic method associated with it. Another is that it is possible to override select methods in the Visitor class and hence get greater reusability.

Even though the visitor design pattern utilizes several object oriented features to its advantage, it violates the notion of encapsulation. As it stands, the visitor class needs to know the structure of every class of the AST.

11.3 Generation of Scope Checking Code

In our compiler we automated the generation of code that does scope checking. Our decision to do so was based primarily on the following facts:

- The Pict Language Definition [22] presents the scoping rules in a very cogent and programmable manner. Mechanically generating code for these rules is merely a matter of translating a specification language into an implementation language.
- The number of scoping rules involved is large — there are rules for both value variables and type variables. Coding these rules by hand is a tedious and error prone task.
- One of the primary goals of this project was to develop a maintainable compiler for a still evolving language. Thus generating scope checking code from a high level specification language is a positive step in that direction.

11.3.1 Scope Specification Language

The process of scope checking is performed on an AST. It is essentially a recursive process such that an AST is well-scoped if its subtrees are well-scoped. In the Pict Language Definition, scope checking is specified in term of axioms. For instance the process abstraction scope checking axiom is,

$$\frac{\Gamma \vdash \texttt{p} \triangleright \Delta \qquad \Gamma, \Delta \vdash \texttt{e}}{\Gamma \vdash \texttt{p = e}}\ .$$

That is, a process abstraction consisting of the pattern `p` and an expression `e`, is well-scoped, if both `p` and `e` are well-scoped. The items on the right-hand side of the $\vdash$ symbol, represent the context (i.e., symbol table) that should be used for scope checking. Specifically, it states here that, `p` should be scope checked in the same context as `p = e`, but `e` should be scope checked in a context that includes the symbols introduced by `p`. Clearly, the axiomatic definition is more precise than the corresponding prose definition.

In our work we adopt the axiomatic specification found in the original Pict Language Definition, but with the following minor modifications:

- The ASCII character set is used to specify a rule. Therefore, to compensate for the lack of Greek characters we adopt the convention that uppercase `G`, `H` and `I` represents environments.
- The conclusion of an axiom (i.e., its bottom line) merely contains the name of an AST node that it corresponds to, rather than its contents. So in the case of our process abstraction axiom, we would use `procAbs` rather than `p = e`.
- The premises in the top line would utilize names of instance variables found in the corresponding AST node. For example, since the class `procAbs` contains instance variables `pat` and `proc`, we would use these instead of `p` and `e`.

With these modifications in place, the above axiom for process abstraction would be specified as,

```
  G |- pat > H      G + H |- proc
-----------------------------------.
          G |- ProcAbs
```

Most of the effort involved in writing scoping rules is concentrated in the premises. Informally, premises can be though of as the tests which collectively determine the scope correctness of some clause. One of the most fundamental test is to check for the existence of a symbol. Since Pict has the concept of both value variables

and type variables, we require tests to check for each. The `exist_v` and `exits_t` functions serve these roles. The following is a list of premises that can occur in a scope checking specification:

`G |-` *c*	Clause *c* is well scoped in environment `G`.
`G |-` *c* `> H`	Clause *c* is well scoped in environment `G` and yields bindings `H`.
`uniq(x, xs)`	Label `x` does not occur in list `xs`.
`exist_v(x, G)`	Symbol `x` exists as a value variable in environment `G`.
`exist_t(x, G)`	Symbol `x` exists as a type variable in environment `G`.

Symbol table manipulation is another common task performed in the premises. The following is a list of symbol table manipulations that can occur in a scope checking specification:

`[]`	Create an empty type or value binding table.
`[s]`	Create a one element type or value binding table.
`[]:[]`	Create an empty environment.
`G:[]`	Create an environment consisting only of the value bindings found in `G`.
`[]:G`	Create an environment consisting only of the type bindings found in `G`.
`G + H`	Create an environment by merging `H` with `G`.

The following grammar defines the language of the scope checking rules:

RuleList = *Rule* *RuleList*
ϵ

Rule = *PremiseList* `----...----` *Conclusion*

PremiseList = *Premise* *PremiseList*
ϵ

Conclusion = *RecvParam* `|-` *iden* `>` *Yield*
RecvParam `|-` *iden*

RecvParam = *iden*
iden `,` *iden*

Premise = *SendParam* `|-` *iden* `>` *iden*
SendParam `|-` *iden*
`uniq (` *iden* `,` *UniqList* `) >` *iden*
`exist_v (` *iden* `,` *iden* `)`
`exist_t (` *iden* `,` *iden* `)`
`*`

Yield = *EnvironExp*

UniqList = *iden*
`[ ]`

ValTab = *iden*
`[ ]`
`[` *iden* `]`

TypeTab = *iden*
`[ ]`

[*iden*]

SendParam = *EnvironExp* , *UniqList*
EnvironExp
ϵ

Environ = *TypeTab* : *ValTab*
iden

EnvironExp = *Environ* + *EnvironExp*
Environ

11.4 Subset of Raw Scope Rules

Below we present some scope checking rules using the conventions defined above.

```
                    G |- type
                  ------------------------
                  G |- NewChan > []:[iden]


  G + []:[iden] |- defls > F      G + []:[iden] + F |- abs
  ------------------------------------------------------
              G |- RecDefElem > []:[iden] + F

                         *
                   ----------------------
                   G |- RecDefEnd > []:[]


                  G |- pat > H      G + H |- proc
                  -------------------------------
                            G |- ProcAbs

                             G |- rtype
                          --------------------
                          G |- Var > []:[iden]

uniq(label, ul) > newul     G |- fldpat > H      G + H:[], newul |- next > I
-------------------------------------------------------------------------------
                   G, ul |- RecordPatElemWithLabel > H + I


            G |- fldpat > H      G + H:[], ul |- next > I
            ---------------------------------------------
              G, ul |- RecordPatElemWithoutLabel > H + I


                                 *
```

```
-----------------------------
G, ul |- RecordPatEnd > []:[]
```

11.5 Formatted Scope Rules

From the scope checking rules we not only mechanically generate the Java code, but also a LaTeXrepresentation of the rules. Below we give the formatted scoping rules that correspond to the rules given above.

$$\frac{\mathcal{G} \vdash \mathit{type}}{\mathcal{G} \vdash \texttt{NewChan} \triangleright \mathit{iden}_v}$$

$$\frac{\mathcal{G} \cdot \mathit{iden}_v \vdash \mathit{defls} \triangleright \mathcal{F} \qquad \mathcal{G} \cdot \mathit{iden}_v \cdot \mathcal{F} \vdash \mathit{abs}}{\mathcal{G} \vdash \texttt{RecDefElem} \triangleright \mathit{iden}_v \cdot \mathcal{F}}$$

$$\frac{\cdot}{\mathcal{G} \vdash \texttt{RecDefEnd} \triangleright \bullet}$$

$$\frac{\mathcal{G} \vdash \mathit{pat} \triangleright \mathcal{H} \qquad \mathcal{G} \cdot \mathcal{H} \vdash \mathit{proc}}{\mathcal{G} \vdash \texttt{ProcAbs}}$$

$$\frac{\mathcal{G} \vdash \mathit{rtype}}{\mathcal{G} \vdash \texttt{Var} \triangleright \mathit{iden}_v}$$

$$\frac{\mathit{distinct}(\mathit{label}, \overline{\mathit{ul}}) \triangleright \overline{\mathit{newul}} \qquad \mathcal{G} \vdash \mathit{fldpat} \triangleright \mathcal{H} \qquad \mathcal{G} \cdot \mathcal{H}_t, \overline{\mathit{newul}} \vdash \mathit{next} \triangleright \mathcal{I}}{\mathcal{G}, \overline{\mathit{ul}} \vdash \texttt{RecordPatElemWithLabel} \triangleright \mathcal{H} \cdot \mathcal{I}}$$

$$\frac{\mathcal{G} \vdash \mathit{fldpat} \triangleright \mathcal{H} \qquad \mathcal{G} \cdot \mathcal{H}_t, \overline{\mathit{ul}} \vdash \mathit{next} \triangleright \mathcal{I}}{\mathcal{G}, \overline{\mathit{ul}} \vdash \texttt{RecordPatElemWithoutLabel} \triangleright \mathcal{H} \cdot \mathcal{I}}$$

$$\frac{\cdot}{\mathcal{G}, \overline{\mathit{ul}} \vdash \texttt{RecordPatEnd} \triangleright \bullet}$$

Chapter 12

Kind Checking

Kinding is the process of checking a program's type expressions for well-formedness. In our implementation of kinding we make design decisions similar to those we made in our implementation of scope checking, as described in the previous chapter. In particular, we continue to remain true to the object oriented paradigm. Furthermore, our Java code for kinding is also mechanically generated from a high level specification. As we will see, special attention has to be paid to return types of kind checking methods.

12.1 Type Expressions and Kinds

In a language such as Java, which lacks parameterized types, type expressions are solely categorized as *proper types*. Examples of proper types include, `String` and `Int[]`. These type expressions are proper in the sense that they completely describe the type that they represent. On the other hand, in a language such as Pict, which supports parameterized types, type expressions can also be categorized as *type operators*. An example of a type operator, would be `\X = ^X`. This type expression is an operator in the sense that it maps an arbitrary type `X` into an input channel for type `X`.

A *kind* is a means of classifying type expressions. Proper type expressions have kind `Type`. Operator type expressions have kind $K_1 \ldots K_n \rightarrow K$, where each K_i and K is itself a kind. Informally we can view it as a mapping from a tuple of kinds $K_1 * \ldots * K_n$ to a kind K. For example, the kind of the type expression `\X = ^X` is `Type` $\rightarrow$ `Type` (note that unless specified otherwise a type parameter `X` is assumed to be a proper type, and hence have kind `Type`). Another example would be the type expression `\X Y = X`, which has kind `Type Type` $\rightarrow$ `Type`, or the type expression `\X Y:(Type->Type) = ^X` which has kind `Type (Type->Type)` $\rightarrow$ `Type`.

The process of kind checking can be summarized as comparing the kind of a type expression for conformance to the language's kinding rules. It is similar to the more familiar concept of type checking, where value expressions are compared for conformance to the language's typing rules.

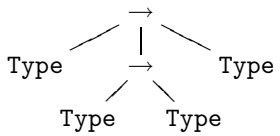
12.2 Representing Kinds

The need to represent a kind exists not only for the purpose of kind checking but also for the purpose of constructing an abstract syntax tree. The Pict grammar defines a kind as follows:

Kind $\longrightarrow$ `(` *Polarity* $Kind_1$ ... *Polarity* $Kind_n$ `->` *Kind* `)` | `Type`

For the purpose of this discussion assume that *Polarity* is always ϵ. Considering the recursive nature of this definition it is reasonable to use a tree data structure to represent it. The leaves of such a tree would correspond to `Type` and its internal nodes would correspond to `->`. For instance, the kind `Type (Type` $\rightarrow$

`Type)` → `Type` would be represented by the tree:

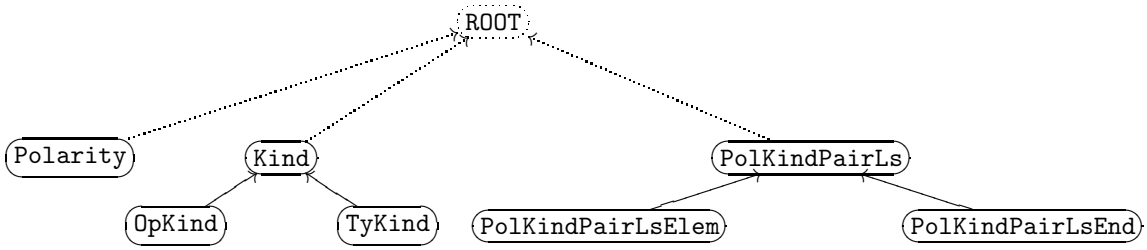


The rightmost branch of each → node would correspond to its return value, while the other branches would correspond to its input parameters.

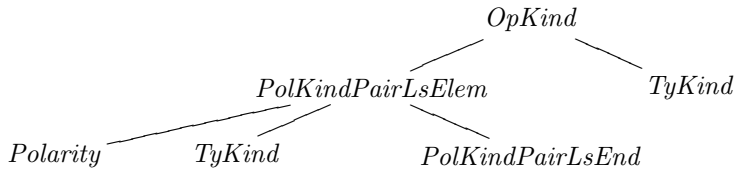
However, as discussed in Chapter 6, the use of a homogeneous tree for representing such a structure is undesirable. In keeping with our discussion on representing abstract syntax trees (ASTs), we first normalize the above grammatical definition into the following form:

$$
\begin{array}{rcl}
\mathit{Kind} & \longrightarrow & \mathit{OpKind} \quad | \quad \mathit{TyKind} \\
\mathit{TyKind} & \longrightarrow & \texttt{Type} \\
\mathit{OpKind} & \longrightarrow & \mathit{PolKindPairLs}\ \texttt{->}\ \mathit{Kind} \\
\mathit{PolKindPairLs} & \longrightarrow & \mathit{PolKindPairLsElem} \quad | \quad \mathit{PolKindPairLsEnd} \\
\mathit{PolKindPairLsElem} & \longrightarrow & \mathit{Polarity}\ \ \mathit{Kind}\ \ \mathit{PolKindPairLs} \\
\mathit{PolKindPairLsEnd} & \longrightarrow & \epsilon \\
\mathit{Polarity} & \longrightarrow & \epsilon
\end{array}
$$

Note that we intend to use this grammar as the basis of our abstract syntax representation of kinds. Therefore, the fact that it does not represent the exact same language as the previous grammar is irrelevant. Keywords such as `Type` and `->` are only shown here in the interest of clarity. According to our previous discussion in Chapter 6, on converting grammars into inheritance hierarchies, we end up the following hierarchy of classes.



In this scheme of things, the kind `Type` → `Type` is represented by the following structure:



Each node of this tree is labeled by the type of the instance variable that it represents.

12.3 Kinding Rules

A language's kinding rules define what the kind of a given type expression should be. For instance, in Pict, the kinding rule for an input channel type expression `^T` is that, the kind of `^T` is `Type` if the kind of `T` is also `Type`.

12.3.1 Formally Defining Kinding Rules

It should be evident that prose definitions of kinding rules are generally informal. For instance, the above description for `^T` fails to mention the context in which kind checking should take place. Consequently such descriptions are open to alternate interpretations.

Fortunately, the original Pict definition defines the language's kinding rules in a very succinct manner. In it, kinding is defined in terms of a proof system. For instance, the kinding rule for `^T` is stated in terms of the following proof rule:

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{Type}}{\Gamma \vdash \mathtt{\hat{}T} \in \mathtt{Type}}$$

The notation $\Gamma \vdash T \in \mathtt{Type}$ simply means that in context Γ type expression T is a proper type (i.e., has kind `Type`). As per this definition we can conclude $\Gamma \vdash \mathtt{\hat{}T} \in \mathtt{Type}$ if we can prove $\Gamma \vdash \mathtt{T} \in \mathtt{Type}$.

12.3.2 Normalizing Kinding Rules

Several kinding rules in the original Pict definition are defined with a human audience in mind. Almost always, rules dealing with list like structures are described with "...", rather than recursively defined. While there is nothing imprecise about such rules, we prefer to redefine these rules in a recursive fashion. Our purpose for doing so is related to the fact these kinding rules are later mechanically transformed into executable code. Having them recursively defined considerably simplifies the transformation process. In many respects this task is akin to transforming an EBNF grammar to an equivalent BNF grammar.

Consider for example the following kinding rule for type applications:

$$\frac{\Gamma \vdash \mathtt{S} \in (\mathtt{K_1}\ \ldots\ \mathtt{K_n} \rightarrow \mathtt{K}) \quad \Gamma \vdash \mathtt{T_1} \in \mathtt{K_1} \quad \ldots \quad \Gamma \vdash \mathtt{T_n} \in \mathtt{K_n}}{\Gamma \vdash (\mathtt{S}\ \mathtt{T_1}\ \ldots\ \mathtt{T_n}) \in \mathtt{K}}$$

This rule states that the type application $(\mathtt{S}\ \mathtt{T_1}\ \ldots\ \mathtt{T_n})$ has kind `K`, if each of its arguments $\mathtt{T}_i$ has kind $\mathtt{K}_i$. We begin the transformation by eliminating the "..." from between the premises. This is achieved simply by rewriting this rule as:

$$\frac{\Gamma \vdash \mathtt{S} \in (\mathtt{K_1} \ldots \mathtt{K_n} \rightarrow \mathtt{K}) \quad \Gamma \vdash \mathtt{T_1} \ldots \mathtt{T_n} \in \mathtt{K_1} \ldots \mathtt{K_n}}{\Gamma \vdash (\mathtt{S}\ \mathtt{T_1} \ldots \mathtt{T_n}) \in \mathtt{K}}$$

Notice that we have now introduced the notion of kind checking a list of types $\mathtt{T}_1 \ldots \mathtt{T}_n$. Next, we eliminate the "..." within the list structures. To do so we first introduce an alternate notation for representing lists. The empty list is represented by `[]` and nonempty lists are represented by `T::TS` (where `T` forms the head and `TS` forms the tail). The type application rule can then be restated as:

$$\frac{\Gamma \vdash \mathtt{S} \in (\mathtt{KS} \rightarrow \mathtt{K}) \quad \Gamma \vdash \mathtt{TS} \in \mathtt{KS}}{\Gamma \vdash (\mathtt{S}\ \mathtt{TS}) \in \mathtt{K}} \tag{12.1}$$

However, to complete the proof system we need to introduce kinding rules for lists of types. To do so, we first define the rule

$$\frac{\mathit{true}}{\Gamma \vdash [\,] \in [\,]}$$

for empty lists, and the rule

$$\frac{\Gamma \vdash \mathtt{T} \in \mathtt{K} \quad \Gamma \vdash \mathtt{TS} \in \mathtt{KS}}{\Gamma \vdash \mathtt{T} :: \mathtt{TS} \in \mathtt{K} :: \mathtt{KS}} \tag{12.2}$$

for non-empty lists.

12.3.3 Representing Kinding Rules

As we will describe shortly, we mechanically translate the above rules into Java code. However, before we can do so, we make some syntactic modifications to these rules. Specifically,

- Each rule is named to match its corresponding node in the abstract syntax class hierarchy. For instance, since rule 12.1 corresponds to type applications, we will name it `TyApp`.
- Variables intended to represent the structure of a node, are replaced with the variables defined in the corresponding abstract syntax node. So in the case of rule 12.1, we will use `type` and `typels` instead of `S` and `TS`.
- Symbols such as $\vdash$, $\in$, $\rightarrow$, and Γ are replaced by `|-`, `>`, `->`, and `G` respectively.

Applying these modifications, rule 12.1 would be written as,

```
G |- type > KS -> K       G |- typels = KS
------------------------------------------ .
              G |- TyApp > K
```
(12.3)

It should be evident that these modifications have no impact on the semantics of the rule.

12.4 Translating Rules

One of the main features of our compiler implementation is that we automatically generate the code for kind checking from the aforementioned kinding rules. To do so we need to consider implementation issues such as where the code should be placed and what it should look like.

12.4.1 Code Generation

Converting a kinding rule into Java code is a relatively straightforward process. For each rule we create a `kindCheck` method. The conclusion of a rule determines the signature of the corresponding `kindCheck` method. For example, in the case of rule 12.3 the signature of the corresponding Java function would be

```
public Kind kindCheck (Env G)
```

Each premise in the rule transforms into an invocation of method `kindCheck`. So in the case of rule 12.3, we have

```
Kind          tmp1 = type.kindCheck(G);
PolKindPairLs tmp2 = typels.kindCheck(G);
```

Based on whether the right hand side of a premise is a `>` or an `=`, we will translate it into a corresponding assignment or comparison. In the case of our current example, we would generate the code

```
PolKindPairLs KS = tmp1.lhs();
Kind          K  = tmp1.rhs();

if (! tmp2.equals(KS))
    throw new KindingException();
```

Finally, the right hand side of the conclusion will determine the return value of the function. Therefore, in our case we would have,

```
return K;
```

Applying transformations such as these, every kinding rule can be translated into equivalent Java code. These transformations are fairly straightforward, and writing a tool to perform them is a simple task. The astute reader will notice that the return type of the `kindCheck` method seems to vary. This is indeed the case, and we shall elaborate on it in Section 12.4.3.

12.4.2 Code Placement

Having figured out how to generate Java code for kind checking, we need to decide where this code should be placed. We decided to place the `kindCheck` methods in the nodes of the AST. Our rational behind this decision is based on the following:

- We know with certainty that for every kinding rule there should exist a corresponding node in the AST (otherwise we would have a broken kind system). So by placing a `kindCheck` method in the nodes themselves, we can exploit Java's type system to detect cases of missing `kindCheck` methods, and hence detect an incomplete kind system.
- From a design point of view, each node should be equipped to *kind check* itself, and hence should have a method to do so.
- Our scope checking methods too are placed within the nodes of an AST, so it would be unreasonable to place some semantic checking methods within the nodes, and others elsewhere.

12.4.3 Return Types

Although a `kindCheck` method always takes a value of type `Env` as its input argument, its return type is not always the same. For instance, the `kindCheck` method for node `TyApp` (rule 12.3) returns a value of type `Kind`, but the `kindCheck` method for node `TypeLsEnd` (rule 12.2) returns a value of type `PolKindPairLs` (i.e., a list of kinds). Therefore, we need to take a closer look at how we derive the signature of a `kindCheck` method.

An examination of the entire kind system reveals that there are four different return types that a `kindCheck` methods can have. In addition to the types `Kind` and `PolKindPairLs` that we have just seen, there also exists the type `Env`, as in the case of rule

```
G |- type = Type
----------------- ,
G |- ValFldTy > G
```

and the type `Type`, as the the case of rule

```
        *
----------------- .
G |- IntTy > Type
```

The mapping between nodes and the return type of their `kindCheck` methods is actually quite straightforward. Based on which subtree a given node is rooted in we can determine the return type of its `kindCheck` method by using the following function:

$$f(node) = \begin{cases} \texttt{Kind} & \text{if } node \text{ is a } \texttt{Type} \\ \texttt{Env} & \text{if } node \text{ is a } \texttt{KindPolIdLs} \\ \texttt{Env} & \text{if } node \text{ is a } \texttt{FldTy} \\ \texttt{PolKindPairLs} & \text{if } node \text{ is a } \texttt{TypeLs} \\ \texttt{Type} & \text{if } node \text{ is a } \texttt{TypeConstr} \end{cases}$$

Chapter 13

Conclusion

One of our primary goals was to examine the applicability of object oriented techniques to compiler construction. To this end we have implemented the concurrent language Pict in the object oriented language Java. In particular, we have implemented lexical analysis, parsing, syntax tree construction, syntax tree transformation, scope checking, and kind checking, all in a very object oriented manner.

Good code maintainability was another of our key goals. We initially believed that a good object oriented design would be the key to maintainability. However, we found that often, the very features that contribute to object orientedness, also detract from maintainability. In order to preserve maintainability and object orientedness, we wrote portions of our code in a very maintainable specification language, which we then translated into very object oriented Java code.

Robust code (by which we mean code that is resilient to runtime failure) has consistently been a driving force in all our design decisions. We have concluded that using Java (in its current incarnation) is not conducive to robustness. This is mainly due to its support for type casting instead of type polymorphism. Despite the fact that type casting is the norm in Java, we performed our work with a minimal use of castings.

13.1 Implementation Language Paradigms

The underlying implementation of our compiler significantly differs from that of the original Pict compiler [22]. For one, not only are the implementation languages disparate, but so are the central design paradigms. Our compiler is written in Java, and employs object oriented design techniques. The original Pict compiler instead, is implemented in Objective Caml [6], and is very functional in its design. Based on our experience with both compilers, we have concluded that, while modern functional languages are particularly well suited to compiler implementation, object oriented languages provide a compelling alternative.

Modern functional languages such as Standard ML [20], Objective Caml, and Haskell [25], support several features that are very applicable to implementing multi-pass compilers. The two most notable such features are recursive data types, and exhaustive pattern matching. These, as we will describe shortly, provide a very simple but powerful mechanism for manipulating syntax trees. Additionally, features such as strong static typing, complete type inference, and higher order functions allow for robust, yet elegant expression of ideas.

At first glance it may not be apparent how a typical object oriented language can be readily exploited for the task of compiler development. Compiler development largely involves manipulating syntax trees, and object oriented languages, unlike functional languages, are not particularly biased towards such operations. However, during the course of our work, we have realized that object oriented features such as inheritance and polymorphism can be profitably used for manipulating syntax tree.

The fundamental differences between a functional approach and an object oriented approach become quite apparent when commenting upon how we define and traverse syntax trees.

13.1.1 Defining Syntax Trees

A syntax tree is a data structure for representing parse trees. We can achieve an equally robust implementation of a syntax tree regardless of whether we adopt a functional approach or an object oriented approach. On the one hand, when it comes to simplicity, a functional implementation is clearly superior to an object oriented one. One reason why this is the case is that functional languages are by design directed towards trees and lists. There is a lot of syntactic sugar available for this. On the other hand, data encapsulation is better supported by object oriented languages.

13.1.1.1 Recursive Data Types

A syntax tree can be readily implemented in a programming language that permits data structures to be recursively defined. Most general purpose languages meet this requirement. However, programming languages that explicitly support a *recursive data type*, such as Standard ML, Objective Caml, and Haskell, allow for a particularly elegant implementation of syntax trees. For instance, in Standard ML, a syntax tree for the CFG fragment

Proc	$\longrightarrow$	*Val Val*	(1) output
	|	*Val Abs*	(2) input
	|	*Val Proc Proc*	(3) conditional

would simply be implemented as

```
datatype Proc = Out  of Val * Val
              | In   of Val * Abs
              | Cond of Val * Proc * Proc
```

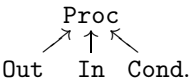
For the sake of brevity we have omitted the rules and data types for the non-terminals *Val* and *Abs*.

While there are several implementations possible, we consider the above one to be particularly advantageous. Firstly it is extremely simple. Despite the fact that the above code appears only to be a definition, it is in fact a working implementation. Secondly, by virtue of the obvious similarity between the CFG and the `datatype`, changes made to one can be trivially reflected in the other. Finally, Standard ML's strong type system will statically ensure that no matter how the syntax tree is instantiated, it will always be consistent with the `datatype`. Clearly, a functional approach such as the one above will result in a simple, maintainable, and robust implementation.

It should be quite apparent that the above functional implementation completely lacks data encapsulation. For instance, the structure of the `Out`, `In`, and `Cond` nodes is completely exposed. While this is not necessarily a bad design, it is still a violation of one of the most basic principles of good design. Object oriented programmers would be justified in finding fault with it.

13.1.1.2 Inheritance Hierarchies

We can also adopt an object oriented approach to designing syntax trees. In this case, the key to obtaining a robust solution lies in creating an inheritance hierarchy that mimics a CFG. For instance, in the case of the above CFG, we would create the inheritance hierarchy,



Based on the above inheritance hierarchy, a Java implementation of a syntax tree would resemble

```
abstract class Proc { ... }
class Out  extends  Proc { Val v1; Val v2; ... }
class In   extends  Proc { Val v;  Abs a;  ... }
class Cond extends  Proc { Val v;  Proc p1; Proc p2; ... }
```

For details on how to convert an arbitrary CFG into an inheritance hierarchy we refer the reader to our work in Chapter 6.

From an object oriented point of view, this implementation is quite good. Here, classes are effectively used to isolate name spaces, and inheritance formally captures the relation between these classes. Constructor methods (which are not shown here) will ensure that any instantiation of a syntax tree will be true to its definition.

The biggest problem with this implementation however, is its large size. While the incomplete code snippet shown above is quite small, an actual working implementation would be about five times as large. Therefore, the programmer effort required to write such code is significant. We estimate that coding a syntax tree for a large programming language would involve tens of thousands of words and span hundreds of files (one per class). In our compiler for instance, the syntax tree implementation uses in excess of 6000 words, and is spread amongst more than 100 files. By the contrast, the original Pict compiler is able to do the same in under 600 words of code. However, note that our syntax tree implementation was mechanically extracted from a specification of only 1200 words.

Of course, it is possible to ignore object oriented constructs such as constructors, protected name spaces etc., in order to obtain a terse implementation. However, there is little sense in using an object oriented implementation language, while simultaneously disregarding object oriented design principles.

13.1.2 Tree Traversals

Syntax tree traversal is an operation that is frequently performed by a multi-pass compiler. While the concept of traversing a syntax tree is elementary, the design challenges associated with it are numerous. We required that our traversal mechanism simultaneously be simple, robust, and object oriented.

Deciding between a functional approach and object oriented one, comes down to deciding whether we want to preserve encapsulation or discard it. In a functional approach encapsulation is traded in favour of simplicity, whereas in a purely object oriented approach the reverse holds true.

13.1.2.1 Pattern Matching

Pattern matching is a feature particularly well suited to traversing syntax trees. Most modern functional languages, including Standard ML, support this feature very well. Readers unfamiliar with functional languages can consider pattern matching to be an enhanced kind of a case expression. For example, in the following Standard ML program,

```
fn Out(_,_) = "out";
 | In(_,_)  = "in";
```

we use pattern matching to map `Out` nodes to the string `out`, and `In` nodes to the string `in`. Specifically, we first attempt to match our input against the pattern `Out(_,_)`, and failing that, attempt to match it against the pattern `In(_,_)`. To a casual observer this may appear as just some syntactic sugar for an `if-then-else` construct. However, pattern matching is more than just syntactic sugar. Its greatest benefit lies in the compile time consistency checks it is able to perform. For instance, in the case of the above program, the compiler will notice that there is no clause for `Cond` nodes. In fact, regardless of how complex the patterns are, we are guaranteed to be able to statically detect missing or redundant clauses. We believe therefore, that pattern matching greatly aids writing robust code.

Had we written our compiler in a functional manner we would have made extensive use of pattern matching during semantic analysis. For example, consider the following subset of Pict's simplified scoping rules:

$$\frac{\vdash v_1 \quad \vdash v_2}{\vdash \texttt{Out}(v_1,\ v_2)}\ , \qquad \frac{\vdash v \quad \vdash a}{\vdash \texttt{In}(v,\ a)}\ , \qquad \frac{\vdash v \quad \vdash p_1 \quad \vdash p_2}{\vdash \texttt{Cond}(v,\ p_1,\ p_2)}\ .$$

Based on these rules, a Standard ML implementation of scope checking would resemble,

```
fun scopeCheck Out(v1,v2)    = scopeCheck(v1); scopeCheck(v2)
  | scopeCheck In(v,a)       = scopeCheck(v) ; scopeCheck(a)
  | scopeCheck Cond(v,p1,p2) = scopeCheck(v) ; scopeCheck(p1); scopeCheck(p2)
```

The clarity and robustness of such an implementation is quite admirable. Not only is there an obvious similarity between the definition and the implementation, but there is also the guarantee that we are not missing any clause.

13.1.2.2 Polymorphism

Object oriented programmers would be correct in observing that the benefits offered by pattern matching can also be realized through the use of inheritance and polymorphism. In fact, not only can polymorphism be used to the same effect as pattern matching, it can even do a better job. There is however, a considerable amount of tedium associated with this approach.

The Standard ML implementation shown above can be easily converted to a robust Java implementation. Essentially, each clause of the `scopeCheck` function would be moved into a class corresponding to that clause. For example, the clause `Out(v1,v2)` would be moved into the class `Out`, as in,

```
class Out extends Proc {
  ⋮
  void scopeCheck () { v1.scopeCheck(); v2.scopeCheck(); }
}.
```

The inheritance hierarchy apparent in this example is as per our discussion in Section 13.1.1.2. Our need however, is to be able to invoke the `scopeCheck` method on a `Proc` object, as in the case of the code,

```
Proc p = ... ;
p.scopeCheck();
```

This necessitates that class `Proc` must have a `scopeCheck` method. We make this `scopeCheck` method an abstract method, since after all scope checking a `Proc` object is an abstract idea. After we do so, the class `Proc` will resemble,

```
class Proc {
  ⋮
  abstract void scopeCheck ();
}.
```

This object oriented implementation now has all the robustness of the corresponding functional implementation. Here, Java's type system will ensure that it is impossible to omit a `scopeCheck` method in any descendant of class `Proc`. This is much like Standard ML's type system ensuring that the `scopeCheck` function has a clauses for all variations of datatype `Proc`. In fact, Java is better in this regard, since any such omission results in a fatal error, whereas Standard ML only generates a warning.

Based on our description above, the object oriented code looks fairly terse. However, this is only because we have omitted everything but the bare essentials. In reality it would be larger by a factor of four. Furthermore, each `scopeCheck` method will need to be inserted into a separate class. As per Java's coding conventions, these classes will be in individual files. Considering the 100 plus classes in our implementation this is no doubt a tedious task.

13.1.3 Choosing a Paradigm

Deciding whether we want an object oriented solution or a functional solution, is really a matter of deciding whether we want encapsulated code, or whether we want simple (and high-level) code. Modern functional languages are designed for constructing and traversing heterogeneous syntax trees. Features such as exhaustive pattern matching and strong static tying, if judiciously used, can result in a simple and robust solution. However, these benefits typically come at the cost of violating the cherished principle of data encapsulation. Object oriented languages are also well equipped for handling heterogeneous syntax trees. They can be used to attain a solution that is not only robust, but also encapsulated. However, achieving such a solution is not simple, for it involves a large quantity of code, that typically is distributed amongst hundreds of files.

In our work, we have adopted a somewhat hybrid approach. Our implementation code is indeed object oriented, well encapsulated, and quite voluminous, but none of it is manually written. This code is actually mechanically generated from a series of high level (almost functional) specifications. Our effort lay in writing these specifications, and the programs that translate them.

13.2 Java

Java, our implementation language, though adequate for our task, lacks certain features, which make it inconvenient and fragile. A considerable amount of our design effort was devoted to avoiding what we perceived to be the shortcomings of Java.

One shortcoming is the lack of multiple inheritance. Recall in Chapter 6 we had to compensate for it by introducing SINF. While interfaces compensate of this shortcoming, they are a poor substitute in our case.

Another hindrance is the inability to overload return types. This was a problem in our tree transformation program as it forced us to cast, despite there being no apparent ambiguity as to the type in question. We felt this to be a completely unnecessary restriction, since there is no reason why parameters can be overloaded, but not return types.

Perhaps the biggest flaw with the current version of Java is the absence of polymorphic types. This creates an unhealthy dependence on casting. Java may well be type-safe, but that is of little reassurance to us as casting makes it dynamically typed.

13.3 Helper languages

We often used helper languages such as Perl for the purpose of prototyping and testing. For instance, the lexer and parser have been tested with the help of Perl scripts. Furthermore, the AST generation program has been implemented in Perl. We found it convenient to work with such helper languages. They are easy to develop in, and what they lack in robustness, they compensate for in convenience. Once fine tuned, these tools can be reimplemented. In our case, the AST generation program was reimplemented in Java.

13.4 Specification Languages

A somewhat unique part of our semantic analysis is to create specification languages that are used to describe the various semantic phases of our compiler. Using specifications written in these languages, we mechanically generate working implementations for each phase. As can be expected, there is some initial overhead associated with this approach. Not only do we have to design a specification language, but we also have to write a translator for it. This overhead however is quite small, and the benefits it provides outweighs its cost.

The specification languages we use are actually very simple. They largely mimic the syntax of proof systems (which incidentally are already used to describe semantic analysis). For instance, one of the rules

in Pict's scope checking proof system is

$$\frac{\Gamma \vdash \texttt{p} \triangleright \Delta \quad \Gamma, \Delta \vdash \texttt{e}}{\Gamma \vdash \texttt{p = e}},$$

which in our scope checking language would be specified as

```
  G |- p > H      G + H |- e
------------------------------ .
         G |- ProcAbs
```

Differences between the two specifications are largely cosmetic. For example, in our language we are restricted to only using ASCII characters. Furthermore, we use identifiers such as `ProcAbs` in place of descriptions such as `p = e`. Clearly, designing the syntax of such a language is not difficult. It should be noted however, that as per our discussion in Section 12.3.2, we occasionally have to normalize certain rules in order to accommodate them into our language.

Implementing a translator for these languages is not difficult either. Recall from our discussion in Section 12.4 that there is a fairly obvious mapping from our specification language to our implementation language. Given the widespread availability of lexer generators such as *JFlex*, and parser generators such as *CUP*, a translator to Java can be easily implemented. Using *JFlex* and *CUP* we implemented our scope checking and kind checking translators in about 500 lines of code each.

The correctness of our approach has been a contentious point. It has been correctly argued that subtle errors in the translating process can lead to hard to find errors in the final implementation. Our main defense against this problem is to ensure that the implementation code produced by our translators is statically typed. In our experience, once we get past Java's type system, the implementation almost immediately works. However, if one wishes to formally prove the correctness of a compiler, then our approach definitely has its merits. If for instance we wished to formally verify the correctness of our scope checking and kind checking phases, then in our approach we would only need to examine about 1000 lines of relatively high level translator code. The alternative to this would be to examine over 12,000 lines of implementation code.

Using specification languages also affords us the flexibility to frequently modify our semantics without worrying about the underlying implementation. Considering that Pict is an evolving language, such modification are not uncommon. The benefits of only having to alter high level specification code rather than low level implementation code are obvious.

Finally, having a high level specification also helps with formally defining our language's semantics. By virtue of being lucid and terse, the specification itself can be made a part of the official language definition. In fact, we can do even better. Our translator programs not only produce Java code, but also produce LaTeX code. See for example Section 11.5 for such a formatted listing of our scope checking rules. This has the added benefit of ensuring that our implementation remains consistent with the language's semantic definition.

13.5 Future Work

There are three promising areas for future work. The first is to simply complete the semantic phases that we left unfinished. The second is to unify the specifications used in different semantic phases. And, the third is to add editor support for writing specification languages.

In our work we have only completed the scope checking and kind checking phases. Still pending, are the type checking and code generation phases. We believe that it would be particularly appropriate to tackle the type checking phase next. It is a large phase that encompasses sub-typing and type inference. It should therefore serve as a good proving ground for our approach.

An increasingly apparent flaw with our approach is the use of multiple specification languages. Currently, each phase has a separate specification language, even though these languages are largely similar. For

instance, the scope checking rule, and the kind checking rule for node `InOutTy` are,

```
 G |- type
------------
G |- InOutTy
```

and

```
  G |- type = Type
-------------------
G |- InOutTy > Type
```

respectively. Syntactically, these rules are similar, but they are written in two different specification languages. Unifying these languages is a logical next step. Doing so should make our approach more palatable to new users. It may in fact even encourage language designers to write their language's semantics in this unified specification language. This will no doubt help with clarity and documentation, but will also be a big step in the direction of automatic compiler generation. The tree transformation language we described in Section 9.1 should also be considered for this unification.

Editor support would be a great finishing touch to this project. An interactive editor, where proof rules rather than characters, are the primitives, would be an asset for language designers. Having the ability to work at the level of semantic proof rules, while ignoring implementation issues, will provide a very desirable level of abstraction. It is likely that such an editor would be a front-end to a unified specification language.

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Priciples, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] J. Aliprand and J. Allen. *The Unicode Standard, Version 3.0*. Addison-Wesley, 1991.

[3] L. Apostoloiu. Corretto: a CUP of Java with Grappa. Master's project report, York University, November 2002. www.cs.yorku.ca/~franck/students/laura.ps.gz.

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

[5] E. Berk. *JLex: A lexical analyzer generator for Java*, September 2000. www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html.

[6] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, April 2000.

[7] P. Deschamp. PERLUETTE: a compilers producing system using abstract data types. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 63–77, Torino, April 1982. Springer-Verlag.

[8] E. Gagnon. Sable, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, March 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, MA, 1996.

[11] S.E. Hudson. *CUP User's Manual*. Georgia Institute of Technology, July 1999. www.cs.princeton.edu/~appel/modern/java/CUP.

[12] G. Klein. *JFlex User's Manual*, October 2001. www.jflex.de/manual.html.

[13] J.R. Levine. *Lex & Yacc*. O'Reilly & Associates, 1992.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading, MA, 1999.

[15] Metamata and Sun Microsystems. *Java Compiler Compiler*, October 2000. www.webgain.com/products/metamata/zip_files/javaccdocs.zip.

[16] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[17] P.D. Mosses and D.A. Watt. The use of action semantics. In *Proceedings of the IFIP TC2 Working Conference on Formal Description of Programming Concepts III*, pages 135–166, Gl. Avernæs, 1986. North-Holland.

[18] H. Moura and D.A. Watt. Action transformations in the ACTRESS compiler generator. In *Proceedings of the International Conference on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1994.

[19] T. Parr. *ANTLR Reference Manual*, October 2000. www.antlr.org/doc.

[20] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[21] B.C. Pierce. Programming in the pi-calculus. www.cis.upenn.edu/~bcpierce/papers/pict/pict-4.1/Doc/tutorial.ps.gz.

[22] B.C. Pierce and D.N. Turner. Pict language definition. www.cis.upenn.edu/~bcpierce/papers/pict/pict-4.1/Doc/defn.ps.gz.

[23] B.C. Pierce and D.N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing, chapter 15, pages 455–494. The MIT Press, 2000.

[24] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[25] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.

[26] 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*, pages 213–228, Berkeley, October 1997. USENIX Association.

[27] D.A. Watt and D.F. Brown. *Programming Language Processors in Java*. Prentice-Hall, 2000.