McGill University
308-400B - Technical Project and Report

Technical Project Report

Developing a JOOS Compiler
Using a SableCC Framework

Submitted **July 2000**
To Dr. Laurie Hendren, supervisor

Othman Alaoui (9734589)

Contents

INTRODUCTION

This document describes the outcome of a semester-long compiler development project, undertaken for the "Technical Project and Report" course offered at the School of Computer Science of McGill University. It was completed under the supervision of Dr. Laurie Hendren, head of the Sable Research Group on optimizing compilers, and instructor at the School of Computer Science.

The project consisted of developing an object-oriented compiler for the Java Object-Oriented Subset (JOOS) programming language; the compiler had to be written in Java and take full advantage of the SableCC compiler compiler. The primary objective of this development effort is to fill a void in the practical part of a Compiler Design course, taught by Dr. Hendren, which consists, in part, of developing a small compiler in one of two environments: C and the Flex/Bison combination, or Java and SableCC. An important point of the course is learning by example; up to now, this was achieved "by providing the students with the source code for an almost complete implementation of the JOOS compiler, ... studied, extended, and used as a template..."[1] Unfortunately, this implementation was only available for the first environment, and thus gave an unfair advantage to the teams who chose it over the second one. By providing an implementation that exploits the second environment to the fullest, we aim at putting the two alternatives, and the teams choosing each of them, on an equal footing.

The compiler we developed is also, to our knowledge, the largest and most comprehensive example yet of a compiler that uses the SableCC environment, as up to now, SableCC has been mostly used for prototyping and front-end (interpreters) development.

1 Project Description

1.1 Technical Background

1.1.1 The JOOS Language

JOOS was developed by Dr. Laurie Hendren and Dr. Michael I. Schwartzbach from Aarhus University in Denmark. It is a large and strict subset of the Java programming language. Thus every JOOS program is a Java program but not every Java program is a JOOS program. Like Java, JOOS is fully object-oriented, as stated by one of the design goals of its authors: "extract the essence of the object-oriented subset of Java"[2] It is the main hands-on teaching tool for the *Compiler Design* course taught at both Aarhus and McGill.

The main restrictions relative to Java are[3]:

- it does not support packages, interfaces, exceptions, mixed statements and declarations, and some control structures (e.g. switch statement);
- fields can only be `protected`, methods and constructors can only be `public`;
- methods cannot be *overloaded*, only constructors can;
- arrays are not supported (this can be made up for by using `Vector` objects);
- the `main` method is the only allowable `static` member in a class;
- primitive types for real numbers are not supported.

An important design goal for JOOS was to be able to leverage existing Java code (e.g. Java libraries). For this purpose, a notion similar to external declarations in C was introduced: *external classes*. Basically, when a JOOS program needs to link to a class for which the code is not available or

[1] See *Philosophy of the course*, at http://www.sable.mcgill.ca/~hendren/520/philosophy.html
[2] Compiler Design Course Notes, The JOOS Language, p. 7, Fall 1999.
[3] See Compiler Design Course Notes, The JOOS Language, Fall 1999.

consists of Java code rather than JOOS code, an external declaration of the class needs to be provided to the JOOS compiler. These external classes are declared in a JOOS-specific syntax and for this reason, it is strongly advised that they be contained in files with filename extension '.joos' rather than '.java'. Thus when compiling a JOOS program, one must feed the compiler not only the Java/JOOS source files constituting the program ('.java' files) but also files containing external declarations of all the external classes used in the program ('.joos' files). If these requirements are not met, compilation will fail.

JOOS also provides its own library of precompiled class files for interfacing to functionality supported only in Java (e.g. constants, static methods and static fields). To use these from a JOOS program, one only needs to compile the '.joos' file containing external declarations for the JOOS library along with the program.

1.1.2 The Reference JOOS Compiler

A compiler for JOOS was developed along with the language to serve as teaching tool for the compiler design course taught by its creators. As such, the compiler also serves as a reference for the language, as it is in and by itself the only definitive specification of the JOOS language.

This compiler, referred to hereinafter as "the reference compiler," is written in C. The scanner code and the parser code are automatically generated by the well-known flex and bison compiler compiler tools respectively. The first takes a lexical language specification (tokens and corresponding semantic actions), and the second a syntactical specification (grammar productions, each with its set of alternatives and corresponding semantic actions). A semantic action is the C code that is executed when a given token or production alternative has been respectively scanned or parsed. The abstract syntax tree corresponding to a given input JOOS program is indirectly built by the parser, using the semantic actions specified in the grammar file. Through semantic actions, the compiler writer has direct control over what goes into the syntax tree. This power is exploited in the reference compiler to build a truly abstract syntax tree, by getting rid of any artifacts that were only necessary to enforce the syntax of the language (e.g. punctuation) and remove parsing ambiguities (e.g. multi-level expression trees)[4]. Scanner semantic actions are used to extract semantic values of tokens (e.g. `int` from integer constants) and store them as such rather than as tokens in the corresponding AST leaves. The compiler then performs the usual passes on the AST, and each pass is explicitly and manually implemented as a recursive descent of the syntax tree.

The implementation is written in a very terse, elegant, self-documenting (and undocumented indeed...) style. Recursion is used extensively, especially for list structures: a list traversal function will typically invoke itself recursively on the remainder of the list (known as the *cdr* of the list in Lisp-like functional languages), and then only visit the head of the list (*car* of the list). The backward nature of the traversal is necessary to restore the original list order, as lists in the AST are all originally built backward.

The compiler also makes heavy use of an often-maligned C feature: *union types*. These are simply the C version of variant record types. Union types enable type polymorphism, at low space cost, since only as much space as required by the largest "subtype" involved will be occupied by a variable of that union type. This is useful for storing heterogeneous objects in a common structure: e.g. expression and statement nodes in an AST, field and formal symbols in a symbol table, instructions for arithmetic and relational operations in a code stream, etc. The main drawback of union types is that they are error-prone and hard to debug, because their polymorphic use is completely unchecked by the C compiler.

[4] See the JOOS grammar specification file for Bison, *joos.y*.

1.1.3 The SableCC Compiler Compiler

SableCC is a *compiler compiler* developed by Etienne Gagnon, member of the Sable Research Group. It is written in Java and produces frameworks for writing compilers, or interpreters, in Java. In the tradition of the flex/bison compiler compiler toolset, it takes a grammatical description of the source language as input, and generates a lexer and a parser for it. In addition, it also generates an AST constructor as part of the parser, and an object-oriented program analysis framework upon which you can build anything from a basic interpreter to a highly optimizing compiler.

In summary, a SableCC-based compiler gets the following functionality free:

- lexical and syntactical analysis;
- automatic AST construction during parsing;
- AST traversal adapters, or *walkers*, to perform custom *analyses* on the program tree representation;
- internal hashtables inheritable from the adapter classes to store information specific to the inheriting analysis.

The fact that the AST is built for you at parsing time goes a long way toward removing the need for semantic actions in the grammar specification, since, in a compiler context, these tend to be used exactly for that purpose: building the AST. Indeed there are no semantic actions in SableCC. This has the advantage of increasing grammar readability and simplifying the front-end development and debugging process. Automation and ease of use come at the expense of flexibility however, since you cannot customize the tree building process, for the purpose of creating a leaner, more abstract, syntax tree for example.

An important aspect of syntax trees built by SableCC-based compilers is that they are strongly typed. SableCC generates a class hierarchy for each *production* of the grammar: an abstract class for the production itself, and a child class for each *alternative* of the production. Each alternative class in turn provides write and read accessors for its *elements*. Likewise, a `Token` class descendant is generated for each token in the grammar. Thus each node of the AST is either an instance of a `Token` class child or an instance of an alternative class. While making manipulations on the AST much safer, this, combined with the fact that the AST built invariably matches its corresponding concrete syntax tree (CST), introduces some interesting flexibility problems not present in - and thus not addressed by - the reference JOOS compiler.

Finally, SableCC provides a utility package that implements the `Collections` framework found in the Java 2 platform and later versions, but not in JDK 1.1.x, used here. In particular, our compiler makes extensive use of the `LinkedList` and `Iterator` classes.

1.1.4 The Jasmin Bytecode Assembler

For each non-external class compiled, the JOOS compiler emits a textual description of the corresponding class file. This textual description is the equivalent of assembly for the Java Virtual Machine. The Jasmin assembler program is then used to "compile" each such file into a binary class file containing Java bytecode that can be executed with any Java interpreter.

1.2 Project Requirements

Within the overall objective of delivering a running SableCC-based JOOS compiler implementation, we can identify three broad categories of requirements: functional requirements, design requirements, and finally implementation requirements.

1.2.1 Functional Requirements

The functional requirements for our compiler are identical to those of the reference compiler: take as input a set of JOOS source files representing a JOOS program and, assuming no errors are found, and compile each non-external class in the program into a Jasmin file. The new compiler must be equivalent to the reference compiler: given the (correct) source code for a certain JOOS program, both compilers must emit identical Jasmin files and, by transitivity, identical bytecode; all errors found by the reference compiler in the input program should also be signaled by our compiler.

There are two versions of the reference compiler: A- and A+. A+ has a few more features than the A- version: at the front-end, it supports long C-style comments, it adds increment statement expressions (e.g. "i++;"), and for-loops as *syntactic sugar* (i.e. implemented in terms of existing A- constructs); at the back-end, it uses a more robust algorithm for stack limit computation used in bytecode generation, and it also implements a much wider range of peephole optimizations. Students of *Compiler Design* are handed the source code for the A- compiler and are asked to bring it up to the level of the A+ compiler. From this point onwards, "reference compiler" will be used to refer to the A+ version. Our own compiler must at least be able to compile the same programs as the reference compiler, which implies supporting at least the added front-end features of the A+ compiler.

1.2.2 Non-Functional Requirements

The crux of this project resided not as much in porting the JOOS compiler from a language to another, as in exploring and solving the issues involved in building a large, object-oriented compiler for a realistically complex language on top of a SableCC-generated framework.

2 Architectural Design

2.1 Porting Strategy

As mentioned above, the purpose of the reference JOOS compiler is primarily pedagogical. In other words, it is intended as a *canonical example of good compiler design and implementation*. Our compiler is an attempt to preserve that fundamental substrate while innovating in every other aspect. It is only logical then that understanding the reference source code represented an important activity in our development process.

An important element of that substrate is the architecture of the reference compiler as a multi-pass compiler. This modular approach relies on a shared *intermediate representation*: the *abstract syntax tree*. Compiling a JOOS program thus consists of parsing the program source and building a corresponding AST, on which a sequence of *phases* is then performed. Each of these phases consists of one or more *passes*, or *walks*, on the AST. Each phase, along with associated data holders (e.g. the symbol table for the symbol table construction analysis), defines a *module* of the compiler. At the back-end of the compiler, the AST is complemented by a low-level intermediate representation to facilitate code optimization. Our approach is very much the same, with a few modules added or removed to better accommodate our design.

We also made no attempt at redesigning the various "micro-algorithms" that constitute the crux of the compiler analyses, because those are in fact the fundamental operations that implement -- and, in the case of JOOS, define -- the semantics of the compiled language.

2.2 Framework-based Development

As aforementioned, a SableCC-based compiler sits atop an object-oriented framework generated by the SableCC tool from the grammar specification of our source language. This framework is an embodiment of the *extended visitor design pattern*, which is "an object-oriented way of implementing a *switch* on the type of an element,"[5] improved upon to allow easily extendable switches across class hierarchies. The generated framework consists of the following Java packages:

[5] *SableCC, an Object-Oriented Compiler Framework*, Etienne Gagnon, McGill University, 1998.

- **`lexer`**: provides an easy-to-use scanner that returns the last token scanned
- **`parser`**: provides a parser that drives the lexer and builds an AST
- **`node`**: contains all the `Node` subclasses whose instances are used as nodes of the AST. Each node class is a *switchable* entity, i.e. provides an `apply(Switch)` used to switch on this particular node type.
- **`analysis`**: this is where the *framework*, in the usual object-oriented sense, really lies. It provides an implementation of the "object-oriented switch" pattern discussed above for the AST structure implied by the input grammar. At the top of this framework is the `Analysis` interface, which specializes the abstract notion of a switch (`Switch` interface) to one on the AST node type. `Analysis` is further specialized by the `AnalysisAdapter` abstract class to the notion of a complete traversal of the syntax tree. Finally, and of most immediate interest to us, are *adapter* classes, subclasses of the `AnalysisAdapter` class. Each adapter implements a tree traversal in a certain predefined order, that can be directly reused through inheritance to write your own *tree-walkers*. The nice thing about adapters is that they already do most of the work for you, and all that remains to be done is to override the appropriate node *visitors* (the equivalent of case clauses of a procedural switch statement) and do your custom analysis work for the visited entity there. You can completely relinquish responsibility for controlling the order of the AST traversal, by merely overriding "pre-visitors" (invoked by the adapter before traversing the children of the corresponding node) and "post-visitors" (invoked after visiting the children of the node). Of course, you can also mix and match visitors, pre-visitors, and post-visitors as required by your analysis. Likewise, if your traversal order needs are particularly unusual, you can also write your own adapter by subclassing the `AnalysisAdapter` class.

In our compiler, every pass subsequent to parsing is a subclass of the `DepthFirstAdapter` adapter class, which performs a depth-first traversal of the AST.

2.3 Module Partitioning

Our compiler is partitioned according to the phases the compilation process goes through. These phases are:

1. given an input program, **build a syntax tree** representing the entire program: this requires scanning and parsing each source file involved (.joos and .java files), and assembling all the resulting syntax trees into "program syntax tree."
2. **fix** the syntax tree: this performs various "fix-ups," or *transformations*, on the AST to minimize the amount of work involved in its analysis, which benefits all subsequent phases.
3. **abstract** the syntax tree: this is used to represent some nodes in a more abstract way, so that they can be used *generically*, that is to say without knowledge of their type.
4. **weed** out bad trees: this rejects programs that do not respect structural language rules that cannot be absorbed into the grammar.
5. **symbol table** construction: this phase builds the class library, collects information about all the symbols used in the program (types and variables) and checks their uses, and builds the class hierarchy; all this information is later used by the type-checking and code generation phases.
6. **type checking**: this performs *static type-checking* on the program, i.e. checks that all the type rules of the language are respected, in statements and expressions in particular. Intermingled with type checking, *type inference* is used to assign a type to each expression. Also part of this phase is checking the class hierarchy for inheritance and other anomalies. This phase constitutes the end boundary of the *front-end* of our compiler.
7. **resource generation**: as the start boundary of the compiler's *back-end*, this phase is a preparatory phase to code generation, i.e. it generates all the resources needed for code generation to proceed.

8. **code IR generation**: this phase uses the analysis data collected by earlier phases and *code templates* to generate a list-like *intermediate representation* of the target bytecode matching the body of each execution unit (method or constructor).
9. **code emission**: this phase emits for each non-external class in the program a jasmin file (.j) containing a textual description of the corresponding class file, including a transcription of the intermediate representation of the bytecode associated with each execution unit.

Every phase after the first one is itself made up of one or more passes on the AST. Ensuring the serial execution and the above order of execution of the various passes involved is of utmost importance. This is because each compiler pass makes assumptions about the state of the AST and the availability of certain pieces of information about the program (*pre-conditions*).

Each module is implemented inside a Java *package*, which are to a program's implementation what modules are to its design: a higher level of abstraction that gives at least the illusion of manageability, and forces you to think in terms of interfaces and boundaries at the module level as well.

2.4 Multi-File Compilation

All the source files forming a JOOS program must be compiled simultaneously. This is mostly required by the symbol table construction and type checking phases, which need to resolve class references across source files. In the reference compiler a unique AST is constructed for the whole JOOS program, possible because of the full control the compiler has over the construction process. In our compiler on the other hand, an AST is automatically constructed by the SableCC-generated parser module for each input file.

Our solution was to simulate the "super-AST" by first building a list of **`ClassFile`** before any pass on any AST is performed. Each `ClassFile` encapsulates the name of the parsed file and the root of the AST built for the file. This list can thus be seen as a conceptual representation of a *program-wide AST*, or *program AST* for short. Each subsequent phase takes this program AST as input, and can iterate on it (since it is just a list) to recover each of the individual AST.

2.5 Analysis Data Management

Usually, one of the responsibilities of a compiler pass is to collect information about specific nodes in the program AST. These *AST annotations* may then used by subsequent passes for their own analysis. For example, type checking assigns a type to a variable use by fetching the symbol annotation of the AST node corresponding to that use (a symbol entity), and retrieving its type.

In the reference compiler, these annotations were made directly in the corresponding AST nodes, which were designed ahead to accommodate the annotations that would need to be made by each pass. SableCC, on the other hand, advocates a strict separation of AST nodes and analysis data storage, the rationale being that a compiler is really an evolving body of *loosely coupled* analyses (more analyses may be added over time: for example, new optimizations). In the annotations-on-AST scheme, every time an analysis is added, the structure of each affected AST node must be modified as well, thus creating a strong coupling throughout the compiler. Another major advantage of the SableCC approach is the ability to dispose of the annotations made by a particular analysis once it's no longer needed.

This separation is not only advocated but also required by SableCC, as all AST node classes generated are declared `final` (i.e. they cannot be subclassed for customization). As a partial solution, methods are made available to any walker (i.e. descendant of an `AnalysisAdapter`) for uniquely mapping annotations to AST nodes and retrieving them; these are `getIn`, `setIn`, `getOut`, and `setOut`. These rely on the fact that every node in the AST (in fact in the program AST) is unique. These methods however are mostly useful for storing *transient* analysis data, i.e. data that does not need to survive beyond the lifespan of the associated analysis. For non-transient analysis data, we simply use our own `Hashtable` object -- or a wrapper thereof (in fact any *map* object could do).

Communication of analysis data is delegated to an `AnalysisDataManager` object, which is just a *universal proxy* to the various mappings of analysis data to AST nodes. Having such an object tremendously simplifies the interface between modules by centralizing access to analysis data. Whenever a new analysis is added to the compiler, only this object needs to be updated to add the (non-transient) annotations made by this analysis to the common pool of analysis data.

An important consideration is the degree of granularity of the data stored in a given map. Since these maps are how passes communicate each other analysis data, it is important that the nature of the data stored by each map be, if not explicit, at least coherent. By that we mean that each map should be as *homogeneous* as possible: in particular, there is no requirement that all the data resulting from a given analysis be stored in one and only one map; an effective rule of thumb is to keep breaking down a map into multiple ones until you can assign each one a meaningful name that gives at least a clue as to the nature of each unit of data stored in the map[6].

Finally, when a unit of analysis data consists of a primitive type, we use the corresponding Java *wrapper* to store the object in the map. When it consists of more than one object or primitive type, we package this information into a custom class.

3 Module Design and Implementation

What follows is a detailed run-down of all the modules comprising our compiler, in order of execution of the corresponding phases.

3.1 Grammar Specification

In SableCC-based compilers, much care must be taken in devising a lexical and syntactic specification for the source language, even more than in those based on the Flex/Bison pair like the reference compiler. This is because the entire framework generated by SableCC is derived from this specification; in particular, SableCC uses names of grammar entities to derive names of public classes and class members in the framework. Since these name are our primary interface to the framework, and since this framework forms the backbone of a SableCC-based compiler, it is not advisable to change grammar entity names in the middle of development.

It is also important to know the mapping between each SableCC grammar construct and the structure of the corresponding AST subtree built as a result of parsing such a construct. Often there is more than one way to represent a given syntax in the grammar: being able to predict the long-term effect of each way allows one to make a more educated choice.

A SableCC grammar specification consists of three parts: *helpers*, *tokens* and *productions*. The token definition part is used to generate a lexer, and the productions part a parser. Helpers are useful for trimming the tokens part.

3.1.1 Lexer Specification

The token specification of our grammar is mostly a direct port of the corresponding Flex specification from the reference compiler. Tokens in SableCC must be named, so that the appropriate `Token` subclasses can be generated. Token names are often self-evident so that isn't an issue.

Helpers are used to great effect to simplify and organize the token specification.

A remarkable difference between token specification in SableCC and in Flex is that there are no *semantic actions* in SableCC. This is by design. Semantic actions in Flex are usually used to customize the semantic value assigned to each token. Most often the appropriate semantic value is just the text of the token itself, automatically extracted by the SableCC-generated lexer in our compiler. In those rare

[6] The reader may notice that this is not always respected in our implementation: indeed this is a conclusion we have progressively come to, as it was conflicting with the goal of minimizing the interface to each module and pass, no longer an issue with the addition of the `AnalysisDataManager` class.

cases where the semantic value of the token must be customized, we delay that customization until it is really needed. This is how we handle tokens for *literals* (constants) in our compiler: the code of the semantic actions for `charconst`, `intconst`, and others is only performed at the code generation phase, where those values are actually needed.

3.1.2 Parser Specification

As for tokens, all productions and their alternatives must be named, with the exception that one and only one alternative in each production may be anonymous. Also, each instance of an element that appears more than once in an alternative must have a different name.

As we will see there are two major differences between productions in SableCC grammars and productions in Bison grammars: SableCC grammars are written in an extended version of the *Backus Naur Form* (BNF) used in Bison grammars, and, unlike Bison, SableCC does not permit the execution of *semantic actions* associated with each production alternative.

3.1.2.1 Taking Advantage of EBNF

In SableCC, context-free grammar rules (productions) are written using a variant of the *Extended Backus-Naur Form* (EBNF) notation. EBNF provides special notations for writing *repetitive* and *optional* constructs, which are very common in programming languages. Such constructs are implicitly expressed through recursion in the BNF notation used in Bison, and often require dedicated productions or individual alternatives. Thus, by taking advantage of the EBNF notation supported by SableCC, one can considerably reduce the number of productions and alternatives in the grammar.

The special EBNF notation is used to represent zero or more repetitions (*Kleene star* **`*`**), one or more repetitions (*Kleene plus* **`+`**), and optionality (**`?`**). Another good reason to take advantage of the special notation for repetition is that such repetitive constructs are automatically parsed by the SableCC-generated framework into straight `LinkedList`[7] objects. In contrast to the recursively-defined backward lists generated by the reference compiler at parse time, these lists offer a faithful, direct representation of repetitive program constructs. Also, the use of a class that is part of the standard Java Collections framework reduces the learning curve and enforces a style of list traversal and manipulation that is much more natural to Java programmers than the *functional* recursive style that forms the backbone of list processing in the reference compiler.

We can summarize the uses of the three special EBNF notations discussed above as follows:

EBNF Notation	Original (BNF) Grammar	EBNF Grammar	Savings

`*` (0 or more repetitions)	`<<entity_user>> :`
	`   ... <entities> ...;`
	`<entities> :`
	`   /* empty */ |`
	`   <ne_entities>;`
	`<ne_entities> :`
	`   <entity> |`
	`   <ne_entities> <entity>;`
		`<entity_user> :`
		`   ... <entity>* ...;`
		(for each `<<entity_user>>`)
			`<entities>` and `<ne_entities>` productions

`+` (1 or more repetitions)	`<entity_user> :`
	`   ... <entities> ...;`
	`<entities> :`
	`   <entity> |`
	`   <entities> <entity>;`
		`<entity_user> :`
		`   ... <entity>+ ...;`
		(for each `<<entity_user>>`)
			`<entities>` production

`?` (optionality)	`<entity_user> :`
	`   ... <entity> ...;`
	`<entity> :`
	`   /* empty */ |`
	`   ...;`
		`<entity_user> :`
		`   ... <entity>? ...;`
		`<entity> :`
		`   ...;`
		(for each `<<entity_user>>`)
			Empty alternative of `<entity>` production

[7] As defined by the Java Collections framework.

3.1.2.2 Alternatives to Semantic Actions

We have seen that no semantic actions can be associated with production alternatives in SableCC grammars, unlike in Bison grammars. Bison semantic actions are used in the reference compiler to accomplish the following:

(a) build a syntax tree representation of the input program, in a bottom-up fashion;

(b) make the syntax tree as *abstract* as possible by sharing polymorphic node constructors among different but similar types of program constructs (e.g. class and extern class), among other things;

(c) check constraints that could seemingly have been enforced at the grammar level but were not for a variety of reasons (to avoid grammar ambiguity, or to differentiate the error message from regular syntax error messages);

(d) build a more general syntax tree representation of program constructs whose grammar representation was overly specialized solely for syntax enforcement purposes;

(e) add syntactic sugar, i.e. language constructs that do not add any expressive power to the language itself but add convenience to the language user (e.g. i++; relative to i = i + 1;); syntactic sugar is simply recast into existing AST node kinds.

(a) is automatically performed for us by the SableCC-generated framework, but the AST built is really a *concrete* syntax tree (CST), since no program syntax information is discarded at all (even blank space can be recovered!). Most nodes in a CST can be treated the same way as in the corresponding AST, by ignoring irrelevant concrete syntax (e.g. punctuation). On the other hand, a syntax tree is just an intermediate representation of the input program whose purpose is to facilitate further program analysis, not hinder it. In particular, some of the abstraction brought by (b) needs to be preserved in our compiler as well. Even though SableCC allows a limited degree of control over the parsing and AST construction procedures (through an overridable `filter` method in the `Parser` class), the preferred approach is to perform *any* manipulation or analysis on the AST within one or more post-parsing passes over the AST, using the facilities offered by the SableCC-generated framework. In our compiler, this abstracting responsibility is taken on by the AST fixer module and the abstracter module. In keeping with this approach, (c), which is really a basic form of weeding, is integrated into the weeder module, and (d) is implemented as AST transformations in the AST fixer module.

The syntactic sugar referred to in (e) is only present in the A+ version of the reference compiler, and is implemented on top of A- constructs. In the reference compiler, only the grammar (flex and bison files) needs to be changed to accomodate these extensions to the JOOS language, thus the "syntactic sugar" name. In our compiler, the approach discussed above dictates that these extensions be implemented in a post-parsing pass over the AST. We manage however to shield most of the post-parsing compiler modules from the new constructs by recasting them into existing (A-) constructs at the AST fixer phase, the first post-parsing phase of the compiler.

3.1.2.3 Hidden Alternatives

Beside the regular production alternatives used for parsing, one can also define *hidden alternatives* in a SableCC grammar. These alternatives are hidden in the sense that they are not part of the SableCC-generated parser, and thus have no say on the syntax of the language defined by the grammar. This is the only element that distinguishes them from regular alternatives. In particular, upon encountering a hidden alternative, the SableCC tool *will* generate the corresponding framework classes, as for any other alternative.

Hidden alternatives are mostly used for AST transformations. One would, for example, define a more abstract version of a given alternative in a hidden alternative (of the same production), and at compile time, replace all AST nodes of the first type (created at parse time) by new nodes of the second type, using the `replaceBy` method of the `Node` class. Of course AST transformations can also use regular alternatives for the replacing nodes.

Other uses include generating new framework classes for purposes other than AST transformations. We use this technique to generate type nodes in the type checking module.

3.2 AST Transformations

3.2.1 Overview

The goal of this phase is to "fix" the syntax tree automatically constructed by the SableCC-generated parser, i.e. perform whatever abstracting transformations are required to simplify program analysis. Another possible objective of transformations - not pursued here - is to reduce the memory footprint of syntax trees.

An important part of the design of AST transformations is the definition of corresponding hidden alternatives in the grammar, since most transformations replace nodes of an existing alternative type by nodes of a more abstract alternative type.

3.2.2 Grammar Modifications

Often, as new transformations are devised, new hidden alternatives will need to be added to the grammar. Sometimes the need for such a transformation will not become apparent until a later stage in development, despite one’s best effort to avoid all such unforeseen needs. In that case, one should carefully evaluate the effect of such a transformation on existing modules, and update them as necessary. A good way to ensure that all updates are indeed performed is to *obfuscate* the name of "replaced" alternatives, and give "replacing" (hidden) alternatives the (non obfuscated) name of the former, rather than assigning them new names.

The first motivation for this is that phases subsequent to the AST fixer phase should not have to deal with complicated alternative names because of the new transformations that were introduced. On the contrary, since the life span of replaced alternatives is much shorter than that of replacing alternatives, the former can afford to have intermediate short-lived names.

Second, it reduces the possibility of inadvertently omitting to update modules that used the replaced alternatives. Following our obfuscation scheme ensures that such code will most probably break at compilation time if it is not updated, instead of silently compiling and potentially resulting in strange runtime behavior.

Our obfuscating scheme consists in appending `tmp_` to the name of replaced alternatives and giving corresponding replacing alternatives the name of the former.

3.2.3 Algorithm

The transformations on the AST are performed in two passes: `AstFixer` and `AstFixerSimplestm`. This separation is dictated by the fact that the `simplestm` transformation performed in the second pass would have potentially clashed with the `stm_no_short_if` transformation performed in the first pass, if both were done during the same pass over the AST. Note that AstFixerSimplestm *must* be performed after AstFixer.

There are two main approaches to AST transformations, corresponding to the order in which affected AST nodes are visited: *bottom-up* and *top-down*. The two approaches can be mixed, even within the same pass.

A bottom-up transformation starts at the "bottom" layer and uses a `Hashtable` for temporarily holding the replacement nodes for the nodes currently being visited (to be replaced) and, once in the "parent layer", fetches these from the hashtable and uses them as the new children nodes in the replacement of the node at the "top" layer.

This is in contrast to a top-down transformation, which starts at the "parent" node, queries it for its children, create the new nodes to replace its children, based on its current children's data, and finally performs the replacement on the parent node by feeding it with the new children nodes.

The first approach is quite natural and is most useful for collapsing deep structures, like cascading expression trees. The second approach is best for transformations that involve few layers of the AST, like list flattening, and is also less costly than the former, being less systematic.

3.2.4 Nature of Transformations

3.2.4.1 Disambiguation Reversal

The structure of the AST directly mirrors that of the grammar, and, as a result, inherits some of its unwieldy features. While these are required in the grammar to resolve ambiguities, they become redundant in the AST and so can be discarded from it, resulting in a considerably streamlined AST.

3.2.4.1.1 Collapsing The Expression Precedence Cascades

Without loss of generality, we consider the case of a *not*-expression for the sake of demonstration. Before AST transformation, such an expression is represented by a long chain of intermediate nodes, starting at the `exp` production (entry point of all non statement expressions): `ADefaultExp`, `ADefaultOrExp`, `ADefaultAndExp`, `ADefaultEqExp`, `ADefaultRelExp`, `ADefaultAddExp`, `ADefaultMultExp`, `ADefaultUnaryExp`, finally leading to the desired `ANotUnaryExpNotMinus` not-expression node. This cascade-like deeply nested structure is required in the grammar to enforce operator precedence. In the AST however, precedence is inherently represented in the tree structure, and thus all expression types can just be alternatives of the `exp` production. By collapsing this chain to just one node, we considerably reduce the work required for bottom-up propagation of analysis data along expression trees (e.g. type inference).

This transformation first requires adding a hidden alternative for each expression type to the `exp` production. The transformation itself is a bottom-up traversal of `exp` subtrees. Once the transformation is complete, all non statement expressions are instances of PExp. Note however that some productions which could also have been integrated into the `exp` are instead left separate. This is because they are shared with the `stm_exp` production, whose set of alternatives is mostly a subset of `exp`'s (in other words, not all expressions are allowed as statement expressions). For our purposes, we will generally refer to all of these - `exp`, `stm_exp` and `assignment`, `methodinvocation` and `classinstancecreation` productions, plus the `receiver` production - as expressions, using the term 'exp' to refer to the `exp` production exclusively.

3.2.4.1.2 Reversing The Dangling Else Disambiguation

Consider the following grammar production for `if` and `if-else` statements in JOOS:

```
<stm> : ... |
        if (<exp>) <stm> else <stm> |
        if (<exp>) <stm> |
        ...;
```

This grammar suffers from the *dangling else problem* illustrated in the following code snippet:

```
if (<exp>)
   if (<exp>)
      <stm>;
   else
      <stm>;
```

Indentation masks an inherent ambiguity in this statement: does the `else` belong to the first or second `if`? Since both interpretations are possible based on the grammar above, the latter is ambiguous.

JOOS uses the *most closely nested* rule to resolve this ambiguity, and thus the indentation in the example above corresponds to the correct interpretation in JOOS. This rule is enforced in the grammar by blocking "short `if`" statements from the `then` clause of `if-else` statements:

```
<stm> : ... |
        if (<exp>) <no_short_if_stm> else <stm> |
        if (<exp>) <stm> |
        ...;
<no_short_if_stm> : ... |
                    if (<exp>) <no_short_if_stm> else <no_short_if_stm> |
                    ...;
```

Note that the <no_short_if_stm> production not only does not have an alternative for short `if` statements, but it also does not contain references to the <stm> production within its alternatives.

It is clear however that this distinction between <stm> and <no_short_if_stm> is for the sole purpose of avoiding ambiguity during parsing, and that it is entirely unnecessary in the AST itself. By transforming `PStmNoShortIf` nodes back into `PStm` nodes of the corresponding alternative types, we remove the need for dedicated analysis code for `PStmNoShortIf` nodes on the one hand and for `PStm` nodes on the other hand.

The transformations required here are not trivial and the order in which they are performed is of utmost importance. The objective is to purge the AST from all `PStmNoShortIf` nodes, replacing them by equivalent `PStm` nodes. Fortunately there is a unique non recursive entry point for `stm_no_short_if` in the grammar: the `if_else` alternative of the `stm` production, as one would expect. Because `stm_no_short_if` alternatives have elements of type `stm_no_short_if` rather than `stm` however, we need a two-step bottom-up process:

1. transform each `PStmNoShortIf` node into a `PStmNoShortIf` node whose statement children are all `PStm` nodes. This requires adding a hidden alternative to `stm_no_short_if` for each alternative that has `stm_no_short_if` elements, with each `stm_no_short_if` element replaced by a `stm` element. This is a recursive procedure, with base case `ASimpleStmNoShortIf` whose grammar alternative contains no `stm_no_short_if` element. When this transformation has been performed on all applicable nodes, there are no more *nested* `PStmNoShortIf` nodes.
2. transform each PStm node into a PStm whose statement children are all PStm nodes. As aforementioned, only the if-else alternative of `stm` contains a `no_short_if_stm` element. Hence all that is required is to add a hidden if-else alternative containing a `stm` element in the then-clause, instead of the `stm_no_short_if` element in the original (obfuscated) alternative. This transformation is feasible because step 1 has ensured that all remaining `PStmNoShortIf` nodes in the AST are readily convertible to equivalent `PStm` nodes.

3.2.4.2 List Flattening

While the SableCC-generated parser automatically builds `LinkedList` representations of *explicitly sequential* program structures (indicated by the EBNF `*` and `+` notations), other structures cannot be directly parsed into `LinkedList` objects even though, from an abstract point of view, they are sequential entities and are ideal candidates for typical `LinkedList` operations like iterative traversal.

A prime example of such structures are punctuation *separated* lists. Indeed, unlike *terminators* (e.g. semicolon), *separators* (e.g. comma) do not follow a strictly repeating pattern. The solution to allow `LinkedList` representation is simply to discard the separators. In other words, if the structure is a list of punctuation separated `<entity>` elements, we obfuscate the original alternative representing it and add a hidden alternative of the form `<entity>+`, and replace nodes of the first type by equivalent nodes of the second type in the AST. This transformation is performed on `PIdentifierList`, `PFormalList`, and `PArgumentList` nodes.

The second type of list flattening transformations deals with declarations. It transforms each possibly multivariable declaration into a `LinkedList` of univariable declarations, thus realizing a tight coupling between the declaration specifier and each declared name. This transformation not only provides for sequential traversal of the names declared in a given declaration but also makes it easier to uniquely map a symbol to its declaring entity in the AST in the symbol table phase.

This transformation is performed on `PField` nodes (field declarations) and `ADeclStm` nodes (local variable declarations). It requires adding the `onefield` and `onelocal` productions respectively, plus hidden alternatives `onefield+` and `onelocal+` in the `field` and `stm` productions respectively, as for the first type of list flattening transformations.

3.2.4.3 A+ Syntactic Sugar Implementation

The A+ syntax extensions to JOOS consist of the addition of the for-statement and the (post)increment *statement expression*.

All for statements are removed from the AST through transformation into while-statement based structures. Likewise for increment statement expressions, which are transformed into assignment based structures.

3.2.4.4 Other Transformations

Beside the transformations above, we also perform the following:

- transformation of `PSimplestm` nodes into `PStm` nodes. This requires adding a hidden alternative to the `stm` production for each `simplestm` alternative. Thanks to this transformation, simple statement handling becomes no different from other statement handling. This transformation is performed in the second pass.
- transform `PConstructor` nodes to integrate the parent constructor invocation statement into the list of statements forming the body of the constructor. The only reason this statement is explicitly specified in the original `constructor` production alternative is to enforce the appearance of such a statement as the first one in a constructor body. This transformation requires not only the addition of a hidden alternative to the `constructor` production to be used for constructor replacement nodes, but also the addition of a hidden `supercons` alternative to the `stm` production, to be used for parent constructor invocation statement nodes.
- make inheritance from the `Object` class explicit. In the absence of the explicit specification of a parent class, classes in JOOS are assumed to extend the `Object` class, thus resulting in a single-rooted inheritance hierarchy. The purpose of this transformation is to make this implicit assumption explicit in the AST, by adding an `Object` extension clause to `AClass` and `AExternClass` nodes that are different from `Object` and have no extension clause. In this case there is no need for hidden alternatives since existing alternatives already include an optional `extension` element. This transformation simplifies checking the class hierarchy.

3.3 Abstracter Module

3.3.1 Overview

This module is an extension of the syntax tree abstraction effort initiated by the AST fixer module. It solves a problem associated with program entities that can have one of multiple *forms*, i.e. whose grammar definition consists of a production with multiple alternatives or even of multiple productions, and that we sometime need to handle *generically*, i.e. without knowledge of their specific form. The scheme by which SableCC maps grammar productions and alternatives to Java classes enforces strong typing on AST nodes. While this adds safety to tree operations, it is an impediment to generic handling of polymorphic program entities.

Our solution is to define, for each such entity, a unique generic class that abstracts the various AST node types defining the different forms of the entity. The `AstAbstracter` pass annotates each instance (AST node) of these entities with an instance of the corresponding generic class. These annotations are then available for use by any analysis that needs generic access to these entities.

The program entities of concern here are classes, constructors and methods. The corresponding generic classes are `GenericClass`, `GenericConstructor`, and `GenericMethod`. Each of these entities is defined grammatically in a production corresponding to an *external* form and in a production corresponding to a regular form. On the other hand most analyses, and particular those that have to deal with the class hierarchy, do not need to make this distinction. The objective of the abstracter phase is thus to close the gap created by this dichotomy.

3.3.2 Algorithm

This phase consists of one pass, AstAbstracter, which visits each class, constructor and method AST node, and maps it to an instance of the corresponding generic class through the `astToGenericMap Hashtable`. The instance is constructed from relevant elements of the corresponding AST node.

3.3.3 Summary of AST Annotations

Here is a summary of the `astToGenericMap` annotations:

AST Node	Annotation
Class (AClass, AExternClass)	GenericClass
Constructor (AConstructor, AExternConstructor)	GenericConstructor
Method (PMethod and PExternMethod)	GenericMethod

3.4 Weeding Module

3.4.1 Overview

The purpose of this phase is to enforce all the structural JOOS constraints that were left out of the grammar because of their context-dependent nature.

3.4.2 Algorithm

This phase consists of a single pass: `Weeder`. It performs the following tasks:

- check that the unique formal of each main method is of type `String`. In the reference compiler, this was performed at parse time.
- check that the "caster" element of a cast expression is an identifier expression, and, if so, transform the cast expression so that the caster element becomes simply an identifier. This requires a hidden cast expression alternative. This transformation is not performed as part of the AST fixer phase because it is conditional on a weeding condition being met.
- reject abstract methods inside non abstract classes.
- reject `this` expressions and `super` receivers inside (static) main methods.

- reject any local declaration statement (`ADeclStm`) that is not at the top of a method, constructor or block statement body, and follows neither another local declaration statement nor a parent constructor invocation statement (which is always the first statement when present).
- check that non-void non-abstract methods belonging to non-extern classes have a return statement at the *end* of *each* execution path.

Each of the last two checks uses a method that recursively examines relevant nested scopes. In these cases, one cannot entirely rely on the automated traversal facility provided by the SableCC framework, because of the need to propagate status information up and down during the traversal.

3.5 Symbol Table Module

3.5.1 Overview

The symbol table phase is responsible for checking symbol definitions, checking symbol uses and mapping them to their definitions. The object-orientation of JOOS assigns it other responsibilities as well: building a library of classes and establishing their hierarchy.

As in most modern languages, symbols in a JOOS program have spatially bound availability in the program's source code, and this availability is governed by *static nested scope rules*. These rules are checked by the JOOS compiler using a *stack of symbol tables*, to represent the nested nature of symbol scopes.

The legacy of this phase consists of: a *class library*, a *class hierarchy*, and a set of AST annotations.

3.5.2 The Symbol Entity

A symbol is the atomic unit upon which this module operates. In JOOS the different kinds of symbols are: class, field, method, formal, and local. A symbol maps a name in the program to its "meaning." In the design of our symbol entity, we chose to be conservative and leave the interpretation of that meaning open.

Two alternatives were available for the design of the symbol entity:

1. use inheritance: have an abstract `Symbol` class, and a "concrete" subclass for each kind of symbol
2. one class for all: have a generic `Symbol` class with a kind field that can be used for any symbol kind

From an object-oriented point of view, the first alternative sounds the most appropriate. In our case however, taking into consideration our conservative stance that the meaning of a symbol should be left up to its users, the second alternative makes more sense. Indeed, except for the kind field, all symbol kinds encapsulate the same basic information: their name, and their meaning (or value).

As in the reference compiler, the *value* of a **`Symbol`** object is just a reference to the AST node defining the symbol, from which all relevant information (e.g. type for a local) can be easily extracted.

Segregating symbols by kind implies defining a constant for each symbol kind. These constants are used by external entities that need to adapt their action according to the symbol kind. Our design uses the *prototype constant design pattern* to accomplish the following:

- **proper containment and uncluttered namespace**: by using a `Constants` interface, nested inside the `Symbol` class
- **easy access constants**: the `Symbol` user only needs to implement the `Constants` interface, to avoid using the fully qualified constant names.

- **type-safety**: since to the inner class `SymbolKind`, defined inside the `Constants` interface, has no public constructor, the `kind` of a symbol (of type `Constants.SymbolKind`) is restricted to only one of the predefined constant kinds.

Here is a summary of the content of each symbol kind:

Kind of Symbol	Value of Symbol
Class	Class AST node
Method	Method AST node
Field	`AOnefield` AST node
Formal	`AFormal` AST node
Local	`AOnelocal` AST node

3.5.3 Design of Symbol Tables

As in the reference compiler, we use the same structure both as an *explicit symbol table* and as an *implicit stack* of symbol tables. The **`SymbolTable`** class uses a `Hashtable` internally to implement individual symbol table behavior. Our hash function is just the default `String.hashCode()` function. Stack behavior is implemented internally through a `next` pointer that points to the `SymbolTable` object below the top of the stack represented by the `SymbolTable` object under discussion.

`SymbolTable` offers standard methods for pushing and popping a symbol. The `scope` method buries the top-most symbol table below a new symbol table: this corresponds to a new nested scope. The `unscope` method returns the table just below the one pointed to by the reference upon which the method is invoked.

3.5.4 Class Hierarchy Design

Looking-up the class hierarchy for a given symbol is a vital operation in the compilation of the statically typed object-oriented language that is JOOS. A prime example is checking that a certain method is indeed defined in the class hierarchy of a certain receiver: if the check fails, the compiler must signal it as an error to the programmer.

Each program has its own class hierarchy. This hierarchy can be seen as a reversed tree of classes on which lookup operations must be performed. Thus it is ideally suited for encapsulation into its own class: **`ClassHierarchy`**. This class is implemented as a map from AST class nodes to **`HClass`** objects, each `HClass` object encapsulating a pointer to the associated class' parent and interface symbol table. Thus the parent or interface of a given class can be easily retrieved by querying the `ClassHierarchy` object for the corresponding `HClass` object.

`ClassHierarchy` also encapsulates class hierarchy exploration methods that were also present in the reference compiler: `subClass`, `lookupHierarchy` and `lookupHierarchyClass`. These methods hide the details of dealing with the underlying `HClass` objects directly.

The class hierarchy is built in two phases: a first pass sets the link to the interface of a class, and the second pass sets the parent link, thus completing the implicit tree structure.

3.5.5 Algorithm

The symbol table phase consists of 3 successive passes on the AST: `SymInterfaceWalker`, `SymInterfaceTypesWalker`, and `SymImplementationWalker`. This pass separation not only creates a better division of labor but is also partially required to be able to handle *use before declaration* and *mutual recursion*. The ordering represents more or less a progression from outer program scopes (classes) to inner scopes (execution unit bodies and blocks).

AST annotations in this phase are performed on a `Hashtable`: `symAnnotations`.

3.5.5.1 SymInterfaceWalker

The responsibilities of this pass are:

- build a library of class symbols: check class symbols and create a class symbol table (`classlib`);
- build a symbol table for each class interface: check each class's interface (field and method symbols) and, for each class, create a symbol table from its interface symbols;
- initialize the class hierarchy: for each class, create a `HClass` object and assign it the symbol table corresponding to that class' interface.

3.5.5.2 SymInterfaceTypesWalker

The responsibilities of this pass are:

- complete the class hierarchy by checking and setting a parent link for each class, in `HClass`;
- check reference types in the declaration of formals and method returns (if applicable), mapping each such type to the corresponding class through `symAnnotations`.

3.5.5.3 SymImplementationWalker

The responsibilities of this pass are:

- check and create variable symbols (fields, locals and formals);
- analyze variable symbol uses in "inner" scopes (methods, constructors and block statements), using a stack of symbol tables (starting at `classlib`), and mapping each variable use to the corresponding variable symbol through `symAnnotations`;
- analyze class symbol uses in "inner" scopes (methods, constructors and block statements), using the same stack of symbol tables as above, and mapping each class use to the corresponding class node through `symAnnotations`;
- check reference types in the declaration of fields and locals (if applicable), mapping each such type to the corresponding class node through `symAnnotations`.

3.5.6 Summary of AST Annotations

All AST annotations in this module are found in the `symAnnotations` `Hashtable`:

AST Node	Annotation	Annotator
Variable uses (`AIdExp`, `AAssignment`)	Variable Symbol	SymImplementationWalker
Class uses (`AInstanceofExp`, `AClassinstancecreation`, `ACastExp`)	Class AST node	SymImplementationWalker
AReferenceType (in formal declaration)	Class AST node	SymInterfaceTypesWalker
AReferenceType (in field or local declaration)	Class AST node	SymImplementationWalker

3.6 Type Checking

3.6.1 Overview

JOOS inherits its main type system characteristics from Java. It is a *strongly typed* language, which means that type compatibility is mostly checked at compile time. This *static* type checking is the main responsibility of this module. For cases where type compatibility can only be determined at run time, JOOS provides a well-defined, statically checked protocol: *casting*. The effect of casting is to locally waive the *static type correctness* requirement, leaving the type checking responsibility to the runtime environment.

The responsibilities of the type-checking phase are:

- check the class hierarchy for inheritance-related issues: to prevent things like *cyclic inheritance*, invalid method overriding, and so on.
- *statically type check* the input program, i.e. conservatively verify that all the type rules of JOOS are respected,
- perform *type inference* on the program, i.e. percolate types up the expression subtrees of the AST, assigning a type to each expression node. This goes hand-in-hand with static type checking, and requires a *type node* representation. *Type coercion* (implicit type conversion) may also be used as necessary.

As in other object-oriented languages, class hierarchy is integrated into the *type system* (set of types available for use in a certain program), through the addition of the concept of *subtype*: each class adds a type to the type system; if the class has a *parent*, its type is then the subtype of the type defined by its superclass. This addition affects *assignment-compatibility* rules among other things.

The legacy of this phase consists of: a *type tree* mapping expression AST nodes to type nodes (`typeAnnotations`), a map binding each invocation site to the corresponding method or constructor (`invokeBindingsMap`), and a map flagging the expressions to be coerced into a `String` (`coerceToStringMap`).

3.6.2 Type Node Representation

3.6.2.1 JOOS Types

In a strongly-typed language, every value, variable or constant, is of a certain type, i.e. belongs to a certain *domain*. Possible types in JOOS are:

- **primitive types**: these are the types that are built into the language (int, char, boolean);
- **reference types**: these are the only user-definable types in JOOS, and are defined through class declarations;
- **String type**: the String type, defined by the class `String`, is a reference type used as a primitive type. It special status originates from its use as the type of string constants, and from its special handling by the overloaded *plus* operator.
- **polynull type**: this type represents a domain with only one value: `null`; it is assignment compatible to all reference types.
- **void type***:* it is used to represent the empty set, for methods that do not return any value for example.

It is important to note that these types are more than just the types that can be used in variable declarations.

3.6.2.2 Representation

Our type node representation reuse the classes derived by SableCC from the `type` production of the JOOS grammar. A type node is therefore an instance of one of those classes. The only caveat to this approach was that only declared types were represented in the production; our solution was to add a hidden alternative for each of the JOOS types not represented (*void* and *polynull*), prompting SableCC to generate the corresponding class (after grammar recompilation of course). This scheme thus automatically provided us with a class hierarchy consisting of a base class (`PType`) and inheriting subclasses (`AIntType`, `ACharType`, `ABooleanType`, `AReferenceType`, `AVoidType` and `APolynullType`). Note that extracting type information from variable declarations becomes trivial with this solution: one just needs to retrieve a reference to the type node embedded in the declaration. If we had defined our own type nodes, we would have needed a mechanism to map AST type nodes to equivalent custom-made type nodes.

3.6.3 Type System Representation

Type inference requires the support of a type system which provides it with a "database" of types. Although, conceptually, the type system can be seen as a separate entity, type inference is its only user. As a result we have opted to embed into the pass that performs type inference. In an attempt to minimize the object creation overload, each non user-defined type is represented by a static singleton of the corresponding type class, that can be reused as often as desired without any added cost of memory space.

For new user-defined types, a class (or reference) type constructor is used to create a new reference type node. Again object creation is minimized by propagating existing reference type nodes whenever possible. This constructor also adds a mapping from the newly created type node to the corresponding class AST node in the `symAnnotations` map, where it belongs.

The String type is represented by a static singleton, like a non user-defined type, but is set up like a user-defined reference type, except that it is set up at the start of the type checking phase.

3.6.4 Algorithm

This phase consists of two independent passes: `TypeHierarchyWalker` and `TypeImplementationWalker`. The first checks the class hierarchy for inheritance anomalies, and the second performs the actual type checking and inference.

3.6.4.1 Checking the Class Hierarchy

This pass uses the class hierarchy built by the symbol table phase to perform the following checks:

- inheritance validity: check that no *cyclic inheritance* happens, and that the parent class is not `final`;
- field hiding validity: check that no inherited field is hidden;
- method overriding validity: check that the overridden symbol is indeed a method name, that the overridden method is not `final`, and that the overriding and overridden methods have the same signature.

3.6.4.2 Type Checking and Inference

Type checking is performed as type inference is proceeding, and serves in a sense to validate the inference. However, while type inference stops at the root of each expression tree, type checking continues the bottom-up traversal of the AST up to the statement level, where the rules governing the types of expressions used in statements are checked (e.g. check that the test expression of an if statement is of type boolean). The bottom-up nature of the traversal is enforced in the code through the exclusive use of *post-visitors* for statements and expressions.

3.6.4.2.1 Constructor and Method Invocation Binding

`TypeImplementationWalker` also takes on a few responsibilities that would usually belong to the symbol table phase, if it wasn't for their strong coupling to the type system. These are:

- select the constructor to be invoked at a given class instance creation site, using a "most-specific signature" algorithm;
- check the validity of a method invocation site vis-a-vis the type of the receiver.

- bind invocation sites to the corresponding method or constructor, for the purposes of code generation, through the `invokeBindingsMap` map.

3.6.4.2.2 Type Coercion

In JOOS, as in Java, the plus operator is overloaded to behave as an arithmetic addition operator in the presence of integer types and as a concatenation operator if one of the operands is of type String. In such a case, if the other operand is not of type String, it must first be *coerced* into a String. The coercion itself is performed at code generation, but it can only happen if the type checking phase has flagged the expressions that need to be coerced. All expressions are flagged as not needing coercion (with `Boolean.FALSE`), except when used as operands of the plus operator (`Boolean.TRUE`). The `Boolean` value is stored in the `coerceToString` map. "Expressions" is used here in its restricted sense of `PExp` AST nodes.

3.6.5 Summary of AST Annotations

AST Node	Map	Annotation	Annotator
Expressions (PExp, PAssignment, PClassinstancecreation, PMethodinvocation, PReceiver)	typeAnnotations	PType node	TypeImplementationWalker
AReferenceType type nodes (not in AST!)	symAnnotations	Class AST node	TypeImplementationWalker
PExp nodes	coerceToStringMap	Boolean flag object	TypeImplementationWalker
Constructor Invocations (ASuperconsStm, AClassinstancecreation)	invokeBindingsMap	Constructor AST node	TypeImplementationWalker
AMethodinvocation	invokeBindingsMap	Method AST node	TypeImplementationWalker

3.7 Resource Generation

3.7.1 Overview

Resource generation is the first module of what is usually known as the back-end of the compiler. Its responsibility is to lay the ground for code generation by collecting the resources necessary for the latter to proceed. The nature of the resources collected will of course vary with the target language. In our case, we are concerned with the resources required to generate the Jasmin representation of class files containing Java bytecode.

The resources generated consist in: *offsets* for locals and formals, *label* indices for control structures, *locals limits* indicating the size of constructor's and method's stack frames, and finally label indices for expressions to be *string-coerced*.

The legacy of this phase consists of two sets of AST annotations: `resourcesMap` and `toStringResourcesMap`.

3.7.2 Algorithm

All resources are generated in one pass (`ResourceGenerator`):

- **offsets**: each local and formal variable is assigned an offset within the scope of the containing method or constructor. This offset is used in the bytecode for (local and formal) variable access.
- **locals limits**: each method and constructor is required by the *Java Virtual Machine* (JVM) to specify a locals limit, i.e. the projected size of its stack frame during execution. This value is just the maximum offset reached during offset computation for the corresponding method or constructor (+1 for the receiver object reference, also stored in the stack frame).
- **labels for control structures**: control structures, i.e. structures involving branches, translate into bytecode involving *branch instructions*. Labels are needed to map these instructions to their corresponding target in the bytecode stream. Thus each control structure in the AST is assigned the set of (integer) labels needed for its code generation. These control structures consist in any statement or expression that involves testing a condition.
- **labels for expressions to be string-coerced**: coercion of a given expression to a String object also requires testing for a condition, which implies a need for associated labels.

3.7.3 Resource AST Annotations

3.7.3.1 Resource Bundle Representation

We represent resource data associated with a particular program entity (which translates into an AST node) through the following class hierarchy:

- **`Resources`**: an abstract base class representing a *resource bundle* for an arbitrary AST node; this class has 3 subclasses: `LocalResources`, `ExecUnitResources`, and `Labels`;
- **`LocalResources`**: a concrete class representing a resource bundle associated to a local or formal;
- **`ExecUnitResources`**: a concrete class representing a resource bundle associated to a method or constructor;
- **`Labels`**: an abstract class representing a set of named labels associated to an arbitrary AST node; this class has a concrete subclass for each control structure type, plus one for string-coercion (`ToStringLabels`).

3.7.3.2 Summary of AST Annotations

We associate resource bundles to AST nodes through 2 maps: `resourcesMap` and `toStringResourcesMap`.

Here is a summary of the annotations made by `ResourceGenerator`:

AST Node	Map	Annotation	Resources
Method and Constructor	resourcesMap	ExecUnitResources	localslimit
Local and Formal	resourcesMap	LocalResources	offset
If statement	resourcesMap	IfStmLabels	stoplabel
Ifelse statement	resourcesMap	IfelseStmLabels	elselabel, stoplabel
While statement	resourcesMap	WhileStmLabels	startlabel, stoplabel
|| expression	resourcesMap	OrExpLabels	truelabel
&& expression	resourcesMap	AndExpLabels	falselabel
Relational expressions	resourcesMap	RelExpLabels	truelabel, stoplabel
! expression	resourcesMap	NotExpLabels	truelabel, stoplabel
toString-coerced expressions	toStringResourcesMap	ToStringLabels	nulllabel, stoplabel

3.8 Code Generation

3.8.1 Overview

This phase generates an *intermediate representation* (IR) of the bytecode to be generated for the input program. The purpose of this *low-level* IR is to provide an opportunity for *peephole pattern* code optimization, before the final emission stage. Even though there was no optimization phase in our compiler as of writing, it should be easy to add one since an important part of the effort involved in the design of the IR went into making it optimization-friendly.

The code generation algorithm, which consists of one pass over the AST, uses *code templates* to map AST structures to corresponding bytecode instruction sequences. This pass uses all the analysis information accumulated over the preceding phases to execute its task.

The legacy of this phase consists of a Hashtable mapping class, method and constructor AST nodes to their signatures (`signaturesMap`) and another one mapping method and constructor nodes to the IR code stream generated for their body (`astToCodeMap`).

3.8.2 Intermediate Representation

The intermediate representation consists of a representation for code streams, a representation for individual bytecode instructions, and a representation for the control flow graph.

3.8.2.1 Design Goals

The design goals for our intermediate representation can be summarized as follows:

1. allow easy replacement of parts of a code stream, required for by a peephole optimizer;
2. allow sequential and random access (relative to a certain instruction) within a code stream;
3. encapsulate instruction-specific information within instructions themselves (e.g. effect on the stack, textual representation, etc.) in order to minimize the work required by code chain visitors (e.g. bytecode optimizers and code emitter).
4. capture the commonality between instruction kinds

3.8.2.2 Code Chain Representation

Code generation consists of building a sequence of abstract bytecode instructions for each *execution unit* in the input program. The structure for representing such a sequence must be highly versatile, to realize all the goals outlined above. The structure chosen is implemented by the `CodeChain` class, which encapsulates a `HashChain` object, borrowed from the *open source* **Soot** framework. A c*hain* data structure merges the sequential behavior of a linked list with the constant-time random access of a hashtable. The only requirement for using a chain is that each of its elements be uniquely identifiable, since random access is always relative to an existing instruction's position.

Unique characteristics of CodeChain are that, unlike HashChain, it only accepts forward operations (e.g. no *predecessor-of* method), to enforce peephole semantics. It is also alterable through subsequence replacement methods that are ideally suite for peephole pattern optimizations. These methods take an `InstrBox` object that indicates the start of the sequence to be replaced. `InstrBox` represents double indirection in Java: it is equivalent to `Instr**` in C. These methods are then responsible for updating the content of the box if the instruction pointed to is replaced or eliminated. An important restriction on the usage of these replacement methods is that a *label instruction* should never be replaced unless it is *dead* (*indegree* 0).

3.8.2.3 Bytecode Instruction Representation

A large number of different bytecode instructions is used to generate code for JOOS programs. Yet, many attributes are shared between instruction kinds: since they are all Java bytecode instructions, their effect on the execution stack can be evaluated in the same manner; likewise all branching instructions have a target. These are only a few of the commonality that can be taken advantage of in the design of an instruction representation. As usual, a class hierarchy is the best candidate for representing this commonality.

The hierarchy designed thus has the interface `Instr` as root. Every instruction is represented by a class that implements Instr by providing implementations for methods that indicate the effect of the instruction on the stack, and for the `toString` method invoked during *code emission*. Instructions are then clustered according to semantic proximity into sub-hierarchies rooted at intermediate abstract classes (e.g. `BranchInstr`, `ArithmInstr`, `BinaryArithmInstr`, etc.). These intermediate classes are also useful for capturing common stack behavior which usually goes hand in hand with semantic proximity, saving us from having to implement the stack effect methods for each instruction kind separately.

3.8.2.4 Control Flow Graph Representation

The *control flow graph* indicates the flow of execution in the code stream, which usually follows the sequence of instructions in the code stream, except in the presence of *branching instructions*. The target of a branching instruction is always a *label* (assimilated to a non-executable instruction in our IR).

In the reference compiler, the flow graph for the code sequence corresponding to a given execution unit is captured in an array of `LABEL` structures, where the array index is the label's number (generated as a resource in the preceding phase). Each `LABEL` structure contains the textual name of the label, its indegree, and its position in the code sequence (pointer to the corresponding label instruction). This label table is thus an indirection from branch instructions to label instructions.

In our view, this added level of indirection is unnecessary and cumbersome because it decreases the cohesion of the intermediate representation. The control flow graph is as important to opimizability as the rest of the IR and thus should be tightly coupled to the latter. Our design simply removes the indirection, by having branching instructions point directly to their targets, and encapsulating information about labels (index, name, indegree) in the corresponding label instructions themselves. By the same token, the label instruction class can now take responsibility for controlling its indegree attribute, by providing well-defined accessors and modifiers to the outside world. Another advantage of this approach is that the IR for a given execution unit is now entirely contained in one entity: the corresponding `CodeChain`.

3.8.3 Algorithm

This phase consists of one pass over the AST: `CodeGenerator`. This pass uses code templates that translate specific subtrees of the AST into bytecode instruction sequences. Each sequence is appended to the `CodeChain` of the current execution unit as it is built.

The traversal is implemented by overriding the `case<node>` visitors in order to control the order of the traversal, dictated by the code templates.

Noteworthy is the scheme used to associate a `BranchInstr` to its target `LabelInstr`. Label instructions are most often inserted in the code stream *after* the branch instructions that target them. Thus initialization of a `BranchInstr` is not completed until after its instantiation, when the label instruction has been inserted and the target of the branch is finally set. Thanks to the principle of *locality* however, this period of invalidity never lasts long: branches are always established within the boundaries of a certain code template.

3.8.4 Summary of AST Annotations

Here is a summary of all AST annotations (made by the `CodeGenerator` pass):

AST Node	Map	Annotation
Class	signaturesMap	String
Method and Constructor	signaturesMap	String
Method and Constructor	astToCodeMap	CodeChain

3.9 Code Emission

In this phase, we recover code chains and signatures from the `signaturesMap` and `astToCodeMap` `Hashtables` respectively, and use them to emit a Jasmin bytecode description file for each non external class in the input program.

Code emission is made extremely simple by our IR design. It is enough to add a `printTo` method to the `CodeChain` class, that queries each of its embedded instructions in sequence for its `String` representation (by invoking their `toString` method), and prints the entire sequence to the output stream provided by the call to `printTo`.

Of course, Jasmin files are not just code stream printouts. Headers and footers need to be printed for each class, constructor and method. In particular locals limits are recovered from the `resourcesMap` Hashtable.

CONCLUSION

In this document we have covered the design and implementation details of a JOOS compiler written in Java on top of a SableCC framework. We have learnt how to take advantage of the new paradigms introduced by SableCC to develop a real-world compiler in an object-oriented fashion. We have also encountered some of the difficulties introduced by the use of SableCC and discussed our implementation of some possible workarounds.

For the writer of this document too, the journey has certainly been an instructive one, as further understanding of one's own design decisions comes from having to explain them to others. While inevitably some decisions are put into question, the process also brings appreciation for the validity of many others.

The hope is that, by reading this document and peeking as necessary at the accompanying source code, future and prospective users of SableCC will be more aware of the power - and constraints - that come with SableCC and harness this awareness during their own SableCC-based compiler writing endeavors.