SAMPL Project Report

Mike Glass Mike Haskel Navarun Jagatpal
Morgan Rhodes

May 7, 2007

Contents

Chapter 1

Introduction and Motivation

SAMPL is a simple, functional, strictly typed, potentially platform-independent, translated, high-performance signal processing language.

1.1 Introduction

SAMPL is a language used for signal and music processing. It produces programs that take in audio streams as input and outputs the modified audio stream along with text output. SAMPL also allows for the sampling of the audio file to be transparent to the user, with the user defining desired behavior for the program and the actual loop for the sampling happening in the background. As a program designed for the processing of music, it has domain-specific types. SAMPL has primitive types including time, frequency, and intensity, along with string and integer types. SAMPL has the potential for platform-independence as it will be translated to C. Possible applications of SAMPL are as simple as frequency filters, effects(such as reverb), and as complex as identifying and filtering a particular instrument.

1.2 Functional

SAMPL is a functional programming language in which programs define the output stream in terms of various operations on the input stream. Programs consist of a series of recursive definitions. The language provides functions for sampling the input stream, such as

`frequency 200 hz`
`rmsvolume .5 sec`
i.e. samples of the input which vary over time
and
`highpass 400hz stream`
i.e. functions for mutating streams to produce other streams

1.3 Strictly Typed

SAMPL is strictly typed, providing types directly related to signal processing such as frequency, time, and intensity. It also provides string and integer types. Programs specify values of these types by specifying units, as in 300 hz, 20 rad, or 50.3 sec. The language supports basic operations such as comparing, adding, or scaling values of a given type, and provides functions for converting between types in meaningful ways (such as frequency–time).

1.4 Control Flow

One of the key features of SAMPL is that the process of scanning and sampling input is transparent to the user. Ordinarily, a program for signal processing would need to implement a tight loop which encapsulates the process of reading the input and updating state as necessary. This process is conceptually distinct from the specification of the audio transformations and users shouldn't need to implement it. To accommodate this, SAMPL implements this loop in the background, and a program's control flow is based upon constructs for applying filters when certain conditions hold.

Such constructs include "if-then-else", which takes on a different value depending on some condition, as in

```
if rmsvolume .5 sec > .75 lfs
   lowpass 5000hz input
else
   input
end
```

which applies a low-pass filter to the input when and only when the input is sufficiently loud.

These control structures limit the layers of indirection between the programs and their intended effects.

1.5 Possible Applications

SAMPL has a number of possible applications, ranging from simple to complex processing tasks. Simple applications in SAMPL will allow low or high pass filters to be applied to the stream, outputting the filtered stream. Another application will be applying reverb to the input stream. These applications can be made more complex in applying a high pass filter whenever a there is a frequency lower than a specified frequency and apply reverb only when the intensity exceeds a certain value. SAMPL will also make it possible to write applications to recognize particular instruments which will make it possible to filter out the flute, or only filter out the flute while there is a trumpet playing. Programs generated by the SAMPL compiler are themselves inherently modular, as users can easily pipe output from one program to another.

Chapter 2

Tutorial

2.1 Hello, world

SAMPL programs, like programs in other functional languages, consist of a sequence of definitions. Our equivalent of a "hello, world" program is one which defines the output equal to the input, without modification.

```
let intensity output = input
```

The `let` keyword signifies a new definition. `intensity output` is what we are defining, that is, a term `output` whose value is an `intensity` which varies over time. `output` is a special term in that the program's output stream takes on its value. The `output` term must always have type `intensity`. Other possible types in general are `scalar`, `frequency`, `angle`, `time`, and `boolean`. On the right hand side of the `=`, we define that the value of `output` should be equal to the value of the term `input`. `input` is a built-in term representing the input stream.

2.2 Functions

In SAMPL one can define terms which take arguments (functions). For example, consider this program:

```
let intensity halve intensity x = x / 2
let intensity output = halve input
/* comments are C-style
   multi-line comments */
```

`intensity halve` again denotes a term `halve` with values of type `intensity`. `intensity x` denotes a *formal parameter* `x`, also of type intensity. `halve` is a function from intensities to intensities. SAMPL is a *pure* language in that functions may not have side-effects beyond specifying an output value.

2.2.1 Higher-order functions, Currying, and partial application

Functions in SAMPL may be treated as values in that they can be taken as parameters or returned as results. Consider the following program:

```
let intensity output = filter halve input
let intensity filter intensity->intensity f intensity x = f x
let intensity halve intensity x = f x
```

The most important thing to note in this program is how `intensity->intensity` uses the `->` operator to denote a type of functions from intensity to intensity.

Multi-argument functions are implemented in SAMPL using the technique of Currying. A function taking two arguments is a function from its first argument to a function from its second argument to its result. The function can thus be evaluated by passing its arguments successively, that is, by passing the second argument to the result of the first application. The "." operator for building functional types is appropriately *right*-associative—the type signature for a function from `a` and `b` to `c` is `a->b->c`, or, more explicitly, `a->(b->c)`.

An interesting implication of Currying is the ability to partially apply arguments to functions. Given the definitions above, `filter halve` is interchangeable with just `halve`, i.e a function which halves its argument.

2.3 Operators and types other than intensity

In general, types other than `boolean` may be multiplied or divided by scalar to yield the same type. The same (non-boolean) types may be divided by each other to obtain scalars. In addition, multiplying frequency by time yields an angle (one Hz times one second yields one radian); the corresponding rules for division also hold. Booleans may be combined with `&` and `|`. `not` is a built-in function for negation. Any two values of the same (non-boolean)

type may be compared with `<` and `>`. Equality tests are not provided, as variables represent imprecise continuous numbers.

2.3.1 Constants

Constants in SAMPL are by default scalars, as used in the above programs. To denote constants of other types, one must use a unit specifier. For example, a frequency constant may be specified for example with `2.5 hz`. The unit specifiers are

`hz` for frequency
`sec` for time
`rad` for angle (in radians)
`lfs` for intensity

Note that `lfs` stands for "linear full scale", an invented term indicating that a value of `1 lfs` denotes the maximum intensity that fits with in the sampling parameters (full scale), and that `.5 lfs` is half that intensity, et cetera (linear).

2.4 Useful functions and an example program

There are several useful functions built in to SAMPL, including:

`sin x`
`sin` takes an angle —x— and returns a scalar
`cos x` works similarly
`highpass freq stream` takes frequency and intensity arguments
It returns an intensity value corresponding to filtering out all frequencies below the one specified from the provided stream.
`lowpass freq stream` works similarly
`freqStrength freq stream` takes frequency and intensity arguments
It returns an intensity value representing the strength of that frequency in the provided stream.
`volume t stream` takes time and intensity arguments and returns an intensity representing the average volume using the specified window width.

`rand` is a scalar value which represents randomly fluctuating noise between 0 and 1.

The following is a complete program to add static hiss at one tenth the original volume. It demonstrates the conciseness of most SAMPL programs.

```
let intensity output = hiss input
let intensity hiss intensity stream =
  stream + rand * volume .1 sec stream / 10
```

Chapter 3

Manual

3.1 Lexical Conventions

Tokens consist of identifiers, keywords, numbers, the '=' sign, the '`->`' type builder, parentheses, and operators. Whitespace may include spaces, tabs, newlines, and carriage returns, and is ignored except in that it separates tokens.

3.1.1 Identifiers

Identifiers are strings of alphanumeric characters starting with an alphabetic character. They are case sensitive and may be of unlimited length.

3.1.2 Keywords

The keywords have special syntactic meaning and may not be used as identifiers:

```
angle boolean else end frequency hz if intensity let lfs
rad scalar sec then time
```

`if`, `then`, `else`, and `end` are used in conditional expressions. `angle`, `boolean`, `frequency`, `intensity`, and `scalar` denote types and are used for type signatures in definitions. `hz`, `lfs`, textttrad, and `sec` are unit specifiers and are used to indicate the type of numerical constants. `let` is a keyword representing the start of a definition.

3.1.3 Numbers

The convention for numbers is based upon the convention for floating constants in C. They consist of an integer part, a decimal point, a fraction part, an `e` or `E`, an exponent sign, and an integer exponent. The integer part, fraction part, and integer exponent each consist of a sequence of digits. The exponent sign consists of either a `+` or a `-`. Any part may be missing, so long as the integer exponent is present exactly when the `e` or `E` is, the exponent sign is present only when the integer exponent is, the decimal point is present whenever both the integer and fraction parts are present, and at least one of the integer and fraction parts is present. Numbers denote `double` floating numbers in C via the usual translation.

3.2 Behavior of Generated Programs

The programs generated by the SAMPL compiler read and write audio data streams from standard input and output, respectively. The read and write operations occur synchronously in that there is a one-to-one correspondence between input and output samples points, which are strictly interleaved starting with input.

The audio itself is encoded as 16-bit two's-complement little-endian single-channel raw linear PCM data, sampled at 44.1 kHz. It may be possible to configure many of these parameters via command-line options to the generated programs—run the generated programs with the `--help` flag for details.

3.3 Expressions

Expressions in SAMPL have a type and a value. The type of an expression is implicit and does not change throughout the program's execution. The value of an expression may change as the program scans new input, but is otherwise static. Due to the possibility of infinite recursion, an expression's value may at any point be undefined, and the program may loop indefinitely when trying to evaluate it.

3.3.1 Types

An expression's type determines how it may be used, and informally specifies the interpretation of the expression's value. The various mechanisms for forming expressions determine the type of the resulting expression and impose restrictions on the types of their components. Type restrictions are checked at compile-time.

Base Types

The following are base types:

```
time intensity frequency angle scalar boolean
```

Functional Types

Given types a and b, `(`a `->` b`)` is a type. These correspond to functions from type a to type b. Functions in multiple arguments are implemented as curried functions, that is, a function from a and b to c has type `(`a `->` `(`b `->` c`))`.

3.3.2 Values

Values of expressions of type scalar are a `double` in C. These values cannot be infinite or `nan`. Values of expressions of type boolean can either be true or false. The implementation of values in other base types is not defined. We assume the existence of a single canonical value for each; further behavior is specified with rules for how they interact with other values in following sections.

3.3.3 Unit Values

As discussed in the previous section, the values of expressions of base types (excepting scalar and boolean) are based upon some canonical unit value. The unit values are described here. One linear full-scale (`lfs`) is the maximum intensity of the input and output streams. Behavior is undefined upon a value greater than `1 lfs`. One second (`sec`) is a conceptual second as determined by the sample frequency. One radian (`rad`) is a conceptual radian—2π radians form a complete wave period. One Hertz (`hz`) represents the *angular*

frequency of a 1 Hz wave. That is, a component with an angular frequency of one Hz has a period of 2π seconds.

Functional Values

Values of functional types are abstract. The value of an expression of type `(`*a* `->` *b*`)` consists of a mechanism which, when provided a value of type *a*, produces a value of type *b*.

3.4 Definitions

*program : definition**
definition : let *name parameters = expression*
name : type ID
*parameters : (type ID)**

A program in SAMPL consists of a series of definitions. Each of these definitions binds a new name whose scope consists of the entire program, and optionally a collection of formal parameters whose scope consists of the body of the definition. Behavior is undefined when a definition attempts to bind an identifier twice or attempts to bind an identifier bound as a name elsewhere in the program.

There are two special names in SAMPL, `input` and `output`. Input is an expression of type intensity whose value is the intensity of the input signal at any particular point in the scanning proccess. The name `input` must not be bound elsewhere.

One of the program's definitions must bind the name `output` with no arguments as type intensity. After scanning a sample of input and setting the value of the `input` name accordingly, a program outputs the value of the `output` name. If the value of the `output` name is not well defined (i.e. infinite recursion), the program may hang indefinitely.

3.4.1 Interpretation of Definitions

A name bound by a definition may be used in an expression anywhere in the program.

Supposing identifiers in the parameter list to be expressions of their specified type, the expression on the right size of '=' (which may contain the parameter identifiers) must have the same type as the type specified in *name*.

Given that the type specified in *name* is b and the types specified in the parameters are, from left to right, $t_1 \ldots t_n$, the identifier in *name* is an expression which may be used anywhere in the program, and has type `(`t_1 `->` `(` t_2 `->` `(` ... `(` t_n `->` b `)` ... `)` `)` `)`.

The identifier mentioned in *name* is an expression which may be used anywhere is the program. If it has no formal parameters, its value is the value of the expression on the right side of its definition. If it has a single formal parameter, then the value is a mechanism which, given a value of appropriate type, yields the value of the expression on the right side of the definition when taking the value of the parameter identifier to be the provided value.

If the definition has multiple formal parameters, we use the method of curried functions. That is, the value of the defined identifier is a mechanism which is provided the value of the first parameter and returns a mechanism which is provided the value of the second parameter and returns another mechanism, et cetera. The method which is provided the value of the final parameter returns the value of the expression given that the parameter identifiers assume the appropriate provided values.

3.4.2 Type specifiers

type : tatom (-> type)?
tatom : base-type | *LPAREN type RPAREN*
base-type : angle | boolean | frequency | intensity | scalar | time

Type specifiers are used in the left side of definitions to indicate the type signatures of arguments and results. The '`->`' operator is *right* associative, in that *a*`->`*b*`->`*c* is equivalent to `(`*a*`->(`*b*`->`*c*`))`.

3.5 Notation of Expressions

This section describes the various means of constructing expressions, their types, and their values.

3.5.1 Conditional Expressions

expression : logical | if *expression* then *expression* else *expression*

A conditional expression consists of a condition (following `if`), a positive part (following *then*), and a negative part (following *else*).

The condition must have boolean type. The positive and negative parts must have the same type—this is the type of the expression.

Whenever the value of the condition is true the value of the expression is the value of the positive part. Whenever the value of the condition is false the value of the expression is the value of the negative part.

If statements may be nested arbitrarily, including inside the conditional part (though it's not clear how that would be useful).

3.5.2 Operators

SAMPL provides four levels of operator precedence in addition to function application, which can be interpreted as the use of an invisible "apply" operator. They are, from loosest to tightest: logical, comparison, additive, multiplicative, and function application. All operators (including function application) are left associative.

The type and value of all operator expressions depends only on the types and values, respectively, of their arguments.

Logical Operators

logical : comparison | logical LOP comparison

Logical operators consist of '|' (logical or) and '&' (logical and). Note that these have the same precedence, unlike C and most sane languages.

The type of each argument to the operator must be boolean. The type of the entire expression is boolean.

The value a logical "and" expression is true if and only if the value of each of the arguments is true. The value of a logical "or" expression is true if and only if the value of at least one of the arguments is true.

Comparison Operators

comparison : additive | comparison COP additive

Comparison operators consist of '<' and '>'. Tests for equality are not provided as values should be treated as more or less continuous. Notions of equality raise issues of discretization, and SAMPL guides the programmer intentionally away from these issues.

Both arguments to a comparison operator must share the same type. This type may not be boolean. The type of the resulting expression is boolean.

This reference manual does not specify the numerical details of expression values. The behavior of comparison and many other operators is defined in terms of properties they must satisfy.

The operators are opposites in that '>' expressions have the same value as '<' expressions with swapped argument values. '>' is transitive in that, given $a > b$ and $b > c$, $a > c$. Never do both $a > b$ and $a < b$ have true value. Both $a > b$ and $a < b$ have false value if and only if a has the same value as b.

Scalar values compare as `double` values in C.

Additive Operators

additive : multiplicative | additive AOP multiplicative

Additive operators consist of '+' and '−'.

Both arguments to an additive operator must share the same type. This type may not be boolean or frequency. The resulting expression has the same type as the arguments.

Excepting rounding errors, addition and subtraction form an abelian group. This is equivalent to the following conditions: addition is associative and commutative. The value written `0` *UNIT*, written here 0, is such that $a + 0$ has the same value as a for all expressions a. For all expressions a, $a - a$ has the same value as the additive identity. $a - b$ is equivalent to $a + (0 - b)$.

Multiplicative Operators

multiplicative : functional | multiplicative MOP functional

Multiplicative operators consist of '$*$' and '/'.

If one argument to '$*$' has type scalar, the other argument may be of any type other than boolean. The resulting expression has this as its type. Otherwise, one argument must have type frequency and the other time. The resulting expression has type angle.

Scalars multiply as `double` do in C. Excepting rounding errors, scalars multiply with other types as a vector spaces, equivalent to the following conditions. If a and b have scalar type and x has any type other than boolean and scalar, $a * (b * x)$ has the same value as $(a * b) * x$. If c has scalar type

and value zero, $c * x$ has the value of the additive identity. Similarly, if c has value one, $c * x$ has the same value as x. If c has value negative one, $c * x$ has the same value as $d - x$, where d is the additive identity of the same type as x. If a is of scalar type and x and y are of the same type, neither boolean nor scalar, $a * (x + y)$ has the same value as $a * x + a * y$. Finally, if $a * x$ has the same value as $x * a$.

If x has frequency type and y has time type, $x * y$ has the same value as $y * x$. Also, if a has scalar type, $a * (x * y)$ has the same value as $(a * x) * y$ and $(a * y) * x$. Furthermore, `1 hz * 1 s` has the same value as `1 rad`.

Expressions which occur as the denominator of a '/' operator may only have types scalar, time, and frequency. If the denominator is a scalar, the numerator may have any type other than boolean, and the resulting expression has the same type as the numerator. If the denominator is either time or frequency, the numerator must have type angle, and the resulting expression has type either frequency or time, whichever is not the type of the denominator.

If the denominator is a scalar, the resulting expression has the same value as though the numerator were multiplied by the reciprocal (as a `double` in C) of the denominator. Otherwise, if the denominator is either of type frequency or time, the value of the resulting expression is such that, when multiplied by the denominator, the result would have the same value of the numerator.

3.5.3 Function Application

functional : atom | functional atom

Functional expressions consist of a function followed by an argument.

The type of the function must be `a->b` for some types a and b. The argument must have type a. The resulting expression has type b.

As mentioned in 3.3.3, the value of the function consists of the name of a free variable and an result expression. The value of the function application is the value of the result expression when all occurrences of the identifier named by the free variable have been substituted with the argument.

3.5.4 Atoms

atom : constant | LPAREN expression RPAREN

LPAREN consists of a single '(' character, while *RPAREN* consists of a single ')' character. The value and type of a parenthesized atom are the same as of the enclosed *expression.*

Constants

constant : true | false | *numerical*
numerical : NUMBER unit?
unit : lfs | sec | rad | hz

Constants may be `true` or `false`, which have boolean type and correspond to the appropriate values, or they may be numerical constants.

Numerical constants are expressions consisting of a number followed by an optional unit specifier. The type of a constant is determined by its unit specifier—`lfs` (*linear full-scale*) corresponds with intensity, `sec` with time, `rad` with angle, and *hz* with frequency. Numbers with no type specifier are scalars.

The value of a scalar constant is simply the value of the `double` value indicated by *NUMBER*. The value of other numerical constants is the same as the value of the scalar indicated by *NUMBER* multiplied by the unit value of appropriate type.

Chapter 4

Project Plan

4.1 Development overview

The group planned the development of SAMPL largely during weekly meetings, roughly an hour in length. We spent much of this time clarifying various issues with the language and implementation. Also, Mike Haskel (the project leader) would present documents detailing various aspects of the language and the compiler's programming interfaces.

After it was written, the current draft of the language reference manual included in Chapter 3 was the canonical specification for development. Beyond this, there is a file describing the format for the abstract syntax trees. The various helper classes are small and largely self-documenting.

During the development process, individual group members worked individually on large sections of code. We spent much of the coding time together in a computer lab where we could ask each other for assistance or advice when it arose.

Testing consists of a traditional test suite of small, demonstrative programs designed to test a range of functionality. A single command runs all tests and reports errors. New test cases are easily added by adding a test file in the proper file system location.

4.2 Programming style

The team did not formalize any specific programming styles. Individuals were working on largely disjoint sections of code, and legibility was not an

issue when they were not.

4.3 Roles and responsibilities

Mike Haskel was the project leader. He was in charge of language specification and arbitration of language issues, organizing development, overseeing progress and schedules, coordinating meetings, writing all documents, and providing as-needed help and advice. Originally, he was also to be in charge of testing.

Morgan Rhodes was originally responsible for lexing, parsing, and syntax tree generation. This, however, was largely finished before large development efforts were underway. Later she was responsible for the test suite and writing libraries for use with the compiled programs. The library effort was abandoned late in the project as it became clear we would not finish code generation. She provided assistance writing documents.

Nav Jagatpal was originally responsible for static semantic analysis and intermediate code generation. He played a crucial role regarding the direction and motivation of the language. During the development process we decided to implement code generation in antlr working directly from the syntax tree, and Nav was to work on semantic analysis until needed elsewhere. Nav laid the groundwork for semantic analysis but did not end up finishing it.

Mike Glass was originally responsible for code generation and programming for the C environment. Mike designed, wrote, and tested this environment, though it was not used. Mike redirected his attention to semantic analysis near the completion of the project, and saw this through to completion.

4.4 Software environment

The entire project exists in a subversion repository on Mike Haskel's personal computer.

We use antlr's Java target for scanning, parsing, and semantic analysis. We wrote in Java a package of helper classes for use in the antlr source.

The test suite consists of a Java package to scan a directory for test cases.

We implemented a build system in ant with individual targets for running ant, compiling the Java sources, running the test suite, and cleaning

automatically generated files.

Some of the team members (Nav and Mike Glass) used Eclipse for development, and others (Morgan and Mike Haskel) used Emacs.

4.5 Timeline and log

As all group members were new to compiler development, we did not stick to a formal timeline, and our intended schedule changed throughout the semester as we became familiar with the process and requirements. Table 4.5 details an ad-hoc approximation at our desired timeline.

Additionally, the group did not maintain a formal progress log. Table 4.5 represents a summary of both the writer's memory and the log generated by subversion commit messages.

Table 4.1: Project log

Date	**Task**
2007-02-07	Finished proposal
2007-02-25	Lexer completed
2007-02-25	Parser first version completed
2007-03-05	Parser stabilized
2007-03-07	LRM completed, language design solidified
2007-03-21	AST construction finished
2007-04-15	Helper classes written
2007-04-22	"be blocks" (a major feature) removed from language
2007-05-06	Semantic analysis completed
2007-05-06	Test suite completed
2007-05-07	Final report completed

Table 4.2: Project timeline

Week	**Goal**
Week 4	Solid language concept, project proposal
Week 6	Language details congealing, begin to develop lexer/parser
Week 8	Language details solidified, lexer/parser working to spec., reference manual finished
Week 10	Understand implementation strategy, programming interfaces specified
Week 11	Semantic analysis and code generation drafts written
Week 12	More or less working compiler assembled, test suite finished
Week 13	Compiler solid and tested, begin work on support libraries and demo programs and report
Week 14	Everything finished
Week 15	Buffer week (to absorb delays)

Chapter 5

Architectural Design

Figure 5 outlines the major components of the SAMPL system. Components depicted with dotted lines are intended but not implemented.

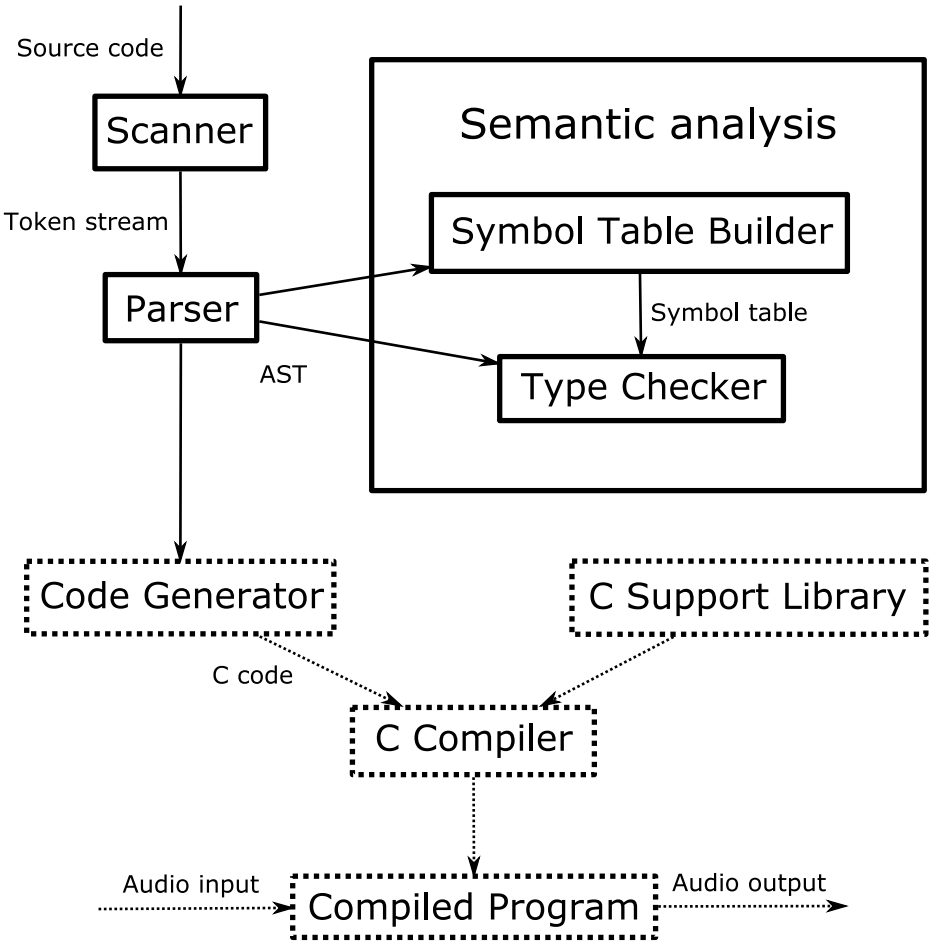
The scanner, parser, semantic analysis, and code generator are defined using the antlr tool, making extensive use of SAMPL-specific helper classes written in Java. The token stream consists of an antlr token stream, and the AST is an antlr tree. The symbol table is a helper class.

All other software interfaces consist of C code or platform-dependent dynamically linked libraries.

As specified in Chapter 3, audio streams are single-channel, 16-bit, two's complement linear pcm data sampled at 44.1 kHz.

Morgan wrote the scanner and parser. Mike Haskel started the symbol table builder, finished and revised by Mike Glass. Nav initially wrote the type checker, which Mike Glass recrafted and finished. Mike Haskel started the helper classes, and Mike Glass refactored them extensively to better suit the needs of semantic analysis.

Figure 5.1: Architecture diagram



Chapter 6

Test Plan

6.1 Testing overview

Our test mechanism consists of a hand-written Java "testing" package, including a routine to run all included tests. We also provide a "test" target for ant such that

```
$ ant test
```

is sufficient to run these tests.

There is a testing directory separate from other source code which contains a hierarchy of test cases written in SAMPL. There is a subdirectory for each type of test (`syntax` and `semantic`), each containing `pass` and `fail` directories. These in turn contain several SAMPL programs which should pass or fail the appropriate compilation phase, respectively.

Morgan wrote the testing functionality, while Mike Haskel wrote in test cases.

6.2 Test details

6.2.1 Syntax tests

The tests for the `syntax` directory are provided by the `testing.SyntacticTest` class. This test creates a scanner from the file stream, a parser from this scanner, and calls the parser's top-level program recognition routine. This test is considered passed exactly when this process does not throw an `Exception`.

6.2.2 Semantic tests

The tests for the `semantic` directory are provided by the `testing.SemanticTest` class. This test creates a scanner from the file stream, a parser from this scanner, and calls the parser's top-level program recognition routine. It then fetches the AST from the parser. With this AST it creates a tree walker. It then calls this walker's symbol table builder and passes the result to the walker's type checker.

This test is considered passed exactly when this process does not throw an `Exception`. If this process throws a `SemanticException`, it is considered to fail appropriately (i.e. a success for tests in the `fail` directory). If it throws some other `Exception`, it fails in a manner inappropriate for either case.

6.3 Role in development

While the test suite was useful in finding the existence and general location of bugs, actual debugging typically required more case-specific tests which the suite could not provide. We wrote these on an informal, as-needed basis.

6.4 Demonstrative examples

There was no fixed system for writing test cases. As the test suite simply looks for files in a directory, Mike Haskel spent some time concocting minimal cross-cutting examples of as many language issues as came to mind.

As code generation does not work, we only provide source code. Here is a listing of one pass and one fail case for each test type.

Figure 6.1: functypes.sampl—syntax pass case

```
let intensity->intensity a = a
let (intensity->intensity) a = a
let intensity->intensity -> intensity a = a
let intensity->(intensity -> intensity) a = a
let (intensity->intensity) ->intensity a = a
let (intensity -> intensity-> intensity) a = a
```

Figure 6.2: numproblem.sampl—syntax fail case

```
let intensity a = 1.1e-
```

Figure 6.3: implicitfunc.sampl—semantic pass case

```
let intensity f scalar x = f x
let scalar a = a
let intensity b = f a
```

Figure 6.4: wrongassign.sampl—semantic fail case

```
let intensity a = b
let frequency b = b
```

Chapter 7

Lessons Learned

7.1 Mike Haskel's lessons

This is my first major group programming assignment. I feel that I have two major regrets regarding our work together. First, our language design was too ambitious. Especially before we removed be blocks from the language, the programs had a dual sense of time which made implementation complex in a way that broke significantly from established designs. Our knowledge from the course would have been more applicable to code generation had we chosen a more traditional approach.

My second regret is regarding the scheduling and organization of programming efforts. I sought to completely specify the software interfaces before people began programming. Fully functional interfaces took a long time to produce, and this is related to the earlier point regarding the implementation being non-obvious. I should have set people on coding as soon as we knew enough to start in order to avoid the fairly significant delays this caused.

7.2 Morgan's lessons

This project taught me the importance of consistent work over the course of the semester. Falling behind a little at a time will eventually aggregate into a large problem. Also, as far as team relations go, this project has taught me the importance of flexibility in regards to what would be worked on. The willingness to work in unfamiliar and previously unassigned portions of the

project in order to help with persistent bugs and other problems is necessary for success.

7.3 Mike Glass' lessons

Planning early is great—prototyping early is necessary. The scope of PLT should not be "original research." as such, being able to produce prototypes of each stage of your compiler reveals the true complexity of a problem (as opposed to the perceived complexity) and allow for more equitable distribution of responsibility early on.

Teamwork is great—but map out tasks and join together only when intersections are present. Initially when coding in teams, I felt the so-called collaboration was a waste of time and only getting in the way of me being in my standard working environment. Later, when we were completing tasks that meshed well together (specifically, writing the final report, writing our test suite, and tweaking the grammar file) collaboration yielded significantly increased productivity. Picking good intersections can speed development by condensing the turn around time on "fix tag" to a number of minutes instead of a number of hours or even days.

Appendix A

Code listing

A.1 Build system

build.xml—Mike Haskel

```
<?xml version="1.1" encoding="UTF-8"?>
<project name="SAMPL" default="test">
  <property environment="env"/>
  <property name="antlrsrc" location="antlr/sampl.g"/>
  <property name="javasrc" location="src"/>
  <property name="antlrdest" location="${javasrc}/parser"/>
  <property name="bin" location="bin"/>
  <property name="jarname" location="sampl.jar"/>
  <property name="testdir" location="testing"/>


  <target name="init">
    <mkdir dir="${antlrdest}"/>
    <mkdir dir="${bin}"/>
  </target>

  <target name="clean">
    <delete dir="${antlrdest}"/>
    <delete dir="${bin}"/>
  </target>
```

```
  <target name="antlr" depends="init"
      description="Compiles antlr files into Java source">
    <java fork="true" classpath="${env.CLASSPATH}"
        classname="antlr.Tool">
      <arg value="-o"/>
      <arg value="${antlrdest}"/>
      <arg value="${antlrsrc}"/>
    </java>
  </target>

  <target name="compile" depends="antlr"
      description="Compiles .java files to /class files">
    <javac srcdir="${javasrc}" destdir="${bin}"/>
  </target>

  <target name="build" depends="compile"
      description="Builds program into .jar file">
    <jar basedir="${bin}" destfile="${jarname}"/>
  </target>

  <target name="test" depends="compile"
      description="Runs the builtin test suite">
    <java fork="true" classpath="${env.CLASSPATH}:${bin}"
        classname="testing.Main">
      <arg value="${testdir}"/>
    </java>
  </target>

</project>
```

A.2 Antlr grammar

antlr/sampl.g—multiple authors; see comments

```
header {package parser;}

class SamplParser extends Parser;
```

```
options {
    buildAST=true;
    exportVocab=SamplParser;
    defaultErrorHandler=false;
}

tokens  {
        PROGRAM;
        ARGS;
        APPLY;
        CONST;
        NAME;
        BASETYPE;
        EXPR;
}

program
    :   definition (definition)* EOF!
        { #program = #([PROGRAM,"PROGRAM"], program); }
    ;

definition
    :   "let"^ name args EQ! expression
    ;

name
    :   ( type ID )
        { #name = #([NAME, "NAME"], name); }
    ;

args
   :   (name)*
        { #args = #([ARGS,"ARGS"], args); }

    ;

type
    :   tatom ( DOT^ type)? //can recurse
```

```
    ;

tatom
    :   "time"! {#tatom = #([BASETYPE, "time"]);}
    |   "intensity"! {#tatom = #([BASETYPE,"intensity"]);}
    |   "frequency"! {#tatom = #([BASETYPE,"frequency"]);}
    |   "angle"! {#tatom = #([BASETYPE,"angle"]);}
    |   "scalar"! {#tatom = #([BASETYPE,"scalar"]);}
    |   "boolean"! {#tatom = #([BASETYPE,"boolean"]);}
    |   LPAREN! type RPAREN!
    ;

expression
:
ifexpression
{#expression = #([EXPR, "EXPR"], expression);}
;

ifexpression
    :
     lexpression

    |   "if"^ ifexpression
           "then"! ifexpression
       "else"! ifexpression
    ;

lexpression
    :   cexpression
        (
            ( AND^ | OR^ )
            cexpression
        )*
    ;

cexpression
    :   aexpression
        (
```

```
            ( LT^ | GT^ )
           aexpression
        )*
    ;

aexpression
    :   mexpression
        (
            ( PLUS^ | MINUS^ )
            mexpression
        )*
    ;

mexpression
    :   fexpression
        (
            ( MULT^ | DIV^ )
            fexpression
        )*
    ;

fexpression!
    :   a:atom
        {#fexpression = #a;}
        (b:atom {#fexpression =
      #([APPLY,"APPLY"], fexpression, b);})*
    ;

atom
    :   ("true" | "false")
    |   ID
    |   FPNUM ( "sec" | "lfs" | "rad" | "hz" )?
        { #atom = #([CONST,"CONST"], atom); }
    |   LPAREN! ifexpression RPAREN!
    ;

class SamplLexer extends Lexer;
```

```
options {
    k=2;
    importVocab=SamplParser;
    defaultErrorHandler=false;
}

ID
    :   ALPHA
        ( ALPHA | NUM )*
    ;

protected
ALPHA
    :   'a'..'z'
| 'A'..'Z'
    ;

protected
NUM
    :   '0'..'9'
    ;

FPNUM
    :   ( NUM )+
        (
            ( '.'
                ( NUM )+
            )?
            (
                ( 'e' | 'E' )
                ( PLUS | MINUS )?
                ( NUM )+
            )?
        )
    |   ( '.' ( NUM )+
            (
                ( 'e' | 'E' )
                ( PLUS | MINUS )?
```

```
                ( NUM )+
            )?
        )
    ;


LPAREN
    :   '('
    ;

RPAREN
    :   ')'
    ;

DOT
    :   "->"
    ;

EQ
    :   '='
    ;

MULT
    :   '*'
    ;

DIV
    :   '/'
    ;

PLUS
    :   '+'
    ;

MINUS
    :   '-'
    ;
```

```
OR
    :   '|'
    ;

AND
    :   '&'
    ;

LT
    :   '<'
    ;

GT
    :   '>'
    ;

/* Lexer rule for comments taken from class slides. */
COMMENT
    :   "/*"
        (   options {greedy=false;}:
            (
                (   ( '\r' '\n' ) => '\r' '\n'
                |   '\r'
                |   '\n'
                )
                { newline(); }
            |   ~('\n' | '\r' )
            )
        )*
        "*/"

        {$setType(Token.SKIP);}
    ;


WS
: (   ' '
|   '\t'
```

```
        |   (   '\r' '\n'
            |   '\n'
            |   '\r'
            )
            {newline();}
        )
        {$setType(Token.SKIP);}
    ;
```

```
{import helper.*;}
class SamplTreeWalker extends TreeParser;

options {
    importVocab=SamplParser;
    defaultErrorHandler=false;
}

/*
common rules (require nothing)
*/

type returns [Type t]
{Type a = null; Type b = null; t = null;}
    :   #(DOT a=type b=type) {t = new FuncType(a,b);}
    |   x:BASETYPE {t = BaseType.getType(x.getText());}
    ;

ignoreExpr
:  #(EXPR trash)
;

trash
: #("if" . . .)
| #(APPLY . .)|#(MINUS . .)|#(PLUS . .)|#(MULT . .)|#(DIV . .)
| #(AND . .)|#(OR . .)|#(LT . .)|#(GT . .)|#(CONST .)
```

```
    |"true"|"false"|ID
;

getName returns [String name]
{name = null;}
: #(NAME type v:ID ) {name = v.getText();}
;

getNameType returns [nameType nt]
{Type t; nt = new nameType();}
: #(NAME t=type v:ID) { nt.name = v.getText(); nt.type = t; }
;
/*
builds SymTable (requires empty / building SymTable)
*/
buildSymTable returns [SymTable s]
{ s = new SymTable();  }
    :   #(PROGRAM (addDefinition[s])*)
    ;

addDefinition[SymTable s]
{ String name = null;
  TypeBuilder b = new TypeBuilder();
  SymTable scopeType = new SymTable();}
: #("let" name=addName[b,null] addArgs[b, scopeType] ignoreExpr)
{ s.put(name, b.build(), scopeType);}
    ;

addArgs[TypeBuilder b, SymTable s]
: #(ARGS (addName[b, s])*)
;


addName[TypeBuilder b, SymTable scope] returns [String name]
{Type t = null; name = null;}
: #(NAME t=type v:ID )
{
b.newArg(t);
```

```
name = v.getText();
if(scope != null)
scope.put(name,t);
}
;

/*
does semantic analysis (requires complete SymTable)
*/


checkProgram[SymTable s]
    :   #(PROGRAM (checkDef[s])*)
    ;

checkDef[SymTable s]
{nameType nt = null; String name; Type r = null, t = null;}
    :   #("let" nt=getNameType {name = nt.name;}
            checkArgs[s] (r=checkExprRoot[s,name]))
        {  if(!r.match(nt.type))
         throw new SemanticException
 ("Inferred type "+r+" does not match specified type "+
      t+" in definition of " + name); }
    ;

checkArgs[SymTable global]
{String name = null;}
    :   #(ARGS (name = getName
        { if(global.getType(name) != null)
            throw new SemanticException
  ("Argument and definition cannot share name " + name);})*)
    ;

checkExprRoot[SymTable s, String parent] returns [Type t]
{t = null;}
: #(EXPR t=checkExpr[s,parent])
;
```

```
checkExpr[SymTable s, String parent] returns [Type t]
{t=null;}
    :   t=checkIf[s, parent]
    |   t=constant
    |   name:ID {t=s.getScopedType(parent, name.getText());}
    |   #(PLUS t=checkAddExpr[s,parent, true])
    |   #(MINUS t=checkAddExpr[s,parent, false])
    |   t=checkMult[s, parent]
    | t=checkDiv[s, parent]
| #(LT t=checkCompExpr[s,parent])
| #(GT t=checkCompExpr[s,parent])
| #(AND t=checkLogExpr[s,parent])
| #(OR t=checkLogExpr[s,parent])
| t=checkApply[s, parent]
    ;

checkIf[SymTable s, String parent] returns [Type t]
{Type condition = null,
      firstblock=null,
      secondblock=null;
      t = null;
 String endEx = " in definition of " + parent;}
:#("if" condition=checkExpr[s,parent]
            firstblock=checkExpr[s,parent]
            secondblock=checkExpr[s,parent])
        {
            if(!condition.match(BaseType.BOOLEAN))
             throw new SemanticException
                  ("Condition is not a boolean"+endEx);
            if(!firstblock.match(secondblock))
             throw new SemanticException
                  ("Block types do not match"+endEx);
            t = firstblock;
        }
     ;

checkMult[SymTable s, String parent] returns [Type t]
{Type a = null;
```

```
 Type b = null;
 t = null;
 String endEx = " in definition of " + parent;}
: #(MULT a=checkExpr[s,parent] b=checkExpr[s,parent])
        {
         if(a.match(BaseType.BOOLEAN) ||
               b.match(BaseType.BOOLEAN))
             throw new SemanticException
                  ("Error: multiplying by a boolean"+endEx);

            // (time)*(freq) = (angle)
            else if ((a.match(BaseType.TIME) &&
                      b.match(BaseType.FREQUENCY) ) ||
             ( b.match(BaseType.TIME) &&
                      a.match(BaseType.FREQUENCY) ) )
             t = BaseType.ANGLE;

            // (scalar)*(anything) = (anything)
            else if(a.match(BaseType.SCALAR))
             t = b;
            else if(b.match(BaseType.SCALAR))
             t = a;
            else throw new SemanticException
                ("Cannot multiply types "+a+" and "+b+endEx);
        }
;

checkDiv[SymTable s, String parent] returns [Type t]
{Type a = null;
 Type b = null;
 t = null;
 String endEx = " in definition of " + parent;}
   : #(DIV a=checkExpr[s,parent] b=checkExpr[s,parent])
        {
            if(a.match(BaseType.BOOLEAN) ||
               b.match(BaseType.BOOLEAN))
             throw new SemanticException
                  ("Error: dividing by a boolean"+endEx);
```

```
//if denom is scalar, num can be anything
else if(b.match(BaseType.SCALAR))
t = a;


        // Something divided by something
            // of the same type is a scalar
else if(a.match(b))
             t = BaseType.SCALAR;

            // Angle divided by ..
            else if(a.match(BaseType.ANGLE)) {
             //..time is frequency
             if(b.match(BaseType.TIME))
             t = BaseType.FREQUENCY;
//..frequency is time
else if (b.match(BaseType.FREQUENCY))
t = BaseType.TIME;
//.. and angle is scalar

else
throw new SemanticException
         ("Can only divide angle by frequency, time, or angle");
            }

            else throw new SemanticException
         ("Cannot divide types "+a+" by "+b+endEx);
            }
;
checkApply[SymTable s, String parent] returns [Type t]
{Type a = null;
 Type b = null;
 t = null;
 String endEx = " in definition of " + parent;}
: #(APPLY a=checkExpr[s,parent] b=checkExpr[s,parent])
{
if(a instanceof FuncType) {
```

```
FuncType f = (FuncType)a;
if(f.arg.match(b))
t = f.result;
else
throw new SemanticException
      ("Type "+b+" wrong argument for functional type "+a+endEx);
}
else
throw new SemanticException
      ("Cannot apply non-functional type "+a+" to type "+b+endEx);
}
;

checkLogExpr[SymTable s, String parent] returns [Type t]
{Type a = null;
 Type b = null;
 t = null;
 String endEx = " in definition of " + parent;}
: a=checkExpr[s,parent] b=checkExpr[s,parent]
{
if(a.match(BaseType.BOOLEAN) && b.match(BaseType.BOOLEAN))
t = BaseType.BOOLEAN;
else throw new SemanticException
 ("Cannot perform logical operation on non-boolean argument"+endEx);

}
;

checkCompExpr[SymTable s, String parent] returns [Type t]
{Type a = null;
 Type b = null;
 t = null; String endEx = " in definition of " + parent;}
: a=checkExpr[s,parent] b=checkExpr[s,parent]
{
if(a.match(BaseType.BOOLEAN) || b.match(BaseType.BOOLEAN))
throw new SemanticException
 ("Cannot perform comparison operation on boolean argument"+endEx);
else if(a.match(b))
```

```
t = BaseType.BOOLEAN;
else
throw new SemanticException
 ("Cannot perform comparison operation between types "+a+
  " and "+b+endEx);
}
;

checkAddExpr[SymTable s,
             String parent,
             boolean isAddition]
            returns [Type t]
{Type a = null;
 Type b = null;
 t = null;
 String endEx = " in definition of " + parent;}
: a=checkExpr[s,parent] b=checkExpr[s,parent]
{
String o;
if(isAddition)
o = "adding";
else
o = "subtracting";
if(a.match(BaseType.BOOLEAN) ||
               a.match(BaseType.FREQUENCY))
             throw new SemanticException
                  ("Error: "+o+" type "+a+endEx);
            else if(b.match(BaseType.BOOLEAN) ||
                    b.match(BaseType.FREQUENCY))
             throw new SemanticException
                  ("Error: "+o+" type "+b+endEx);
            else if(!a.match(b)) {
             if(isAddition)
o="plus";
             else
             o="minus";
             throw new SemanticException
       ("Types "+a+" and "+b+" do not match at "+o+" sign"+endEx);
```

```
            }
            t = a;
        }
;

constant returns [Type t]
{t=null;}
    :
     ("true" | "false")
     {   t = BaseType.BOOLEAN; }
    |   #(CONST FPNUM t=unitOfConstant)
    ;

unitOfConstant returns [Type t]
{t=null;}
    :   "sec"
        {   t = BaseType.TIME;    }
    | "lfs"
        {   t = BaseType.INTENSITY;    }
    |   "rad"
        {   t = BaseType.ANGLE;        }
    |   "hz"
        {   t = BaseType.FREQUENCY;    }
    |   //nothing
        {   t = BaseType.SCALAR;    }
;
```

A.3 Java sources

A.3.1 Package 'helper'

src/helper/BaseType.java—originally Mike Haskel, revised Mike Glass

```
package helper;

public class BaseType implements Type
{
    private BaseType(String s)
```

```
    { type = s; }

    public static BaseType getType(String s) {

     if(s.equalsIgnoreCase("time"))
     return TIME;
     if(s.equalsIgnoreCase("intensity"))
     return INTENSITY;
     if(s.equalsIgnoreCase("angle"))
     return ANGLE;
     if(s.equalsIgnoreCase("scalar"))
     return SCALAR;
     if(s.equalsIgnoreCase("frequency"))
     return FREQUENCY;
     if(s.equalsIgnoreCase("boolean"))
     return BOOLEAN;
     return null;
    }

    public boolean match(Type t)
    {
if(t instanceof BaseType)
    return type.equalsIgnoreCase( ((BaseType) t).type );
else
    return false;
    }

    public String toString() {
     return type;
    }

    public final String type;
    public static final BaseType TIME =
new BaseType("time");
    public static final BaseType INTENSITY =
new BaseType("intensity");
    public static final BaseType FREQUENCY =
new BaseType("frequency");
```

```
    public static final BaseType ANGLE =
new BaseType("angle");
    public static final BaseType SCALAR =
new BaseType("scalar");
    public static final BaseType BOOLEAN =
new BaseType("boolean");
}
```

src/helper/FuncType.java—Mike Haskel

```
package helper;

public class FuncType implements Type
{
    public FuncType(Type arg, Type result)
    {
this.arg = arg;
this.result = result;
    }

    public boolean match(Type t)
    {
    if (t instanceof FuncType){
FuncType f = (FuncType)t;
return (arg.match(f.arg) && result.match(f.result));
}
return false;
}

    public String toString() {
     return "("+arg + "->" + result+")";
    }

    public final Type arg;
    public final Type result;
}
```

src/helper/SymTable.java—originally Mike Haskel, mostly rewritten Mike Glass

```
package helper;

import java.util.HashMap;
import java.util.Set;
import antlr.SemanticException;

public class SymTable {
public HashMap<String, Type> type;
public HashMap<String, SymTable> scope;

public SymTable() {
type = new HashMap<String, Type>();
scope = new HashMap<String, SymTable>();
}

public void put(String name, Type t, SymTable s)
    throws SemanticException {
    if(type.put(name, t) != null) {
throw new SemanticException
    ("Duplicate definitions for name " + name);
    }
    scope.put(name, s);
}

public void put(String name, Type t) throws SemanticException {
if(type.put(name, t) != null) {
throw new SemanticException
    ("Duplicate arguments for name " + name);
}
}

public Set<String> nameSet() {
return type.keySet();
}
public Type getType(String name) {
return type.get(name);
}
```

```
public SymTable getScope(String name) {
return scope.get(name);
}

public Type getScopedType(String parent, String name)
    throws SemanticException {
Type t = type.get(name);
if (t == null) {
t = scope.get(parent).getType(name);
if(t == null)
throw new SemanticException("No name exists: "  + name);
}

return t;

}

public int size() {
return type.size();
}
}
```

src/helper/Type.java—Mike Haskel (does it matter?)

```
package helper;

public interface Type
{
    public boolean match(Type t);
}
```

src/helper/TypeBuilder.java—Mike Haskel, minor refactoring Mike Glass

```
package helper;

public class TypeBuilder {
public TypeBuilder() {
l = null;
}
```

```
public void newArg(Type arg) {
if(l == null)
l = new TypeList(arg, null);
else
l = l.newArg(arg);
}

public Type build() {
return l.build();
}

private TypeList l;

}


class TypeList
{
    TypeList() {
     type = null;
     last = null;
    }
    TypeList(Type result, TypeList prev)
    {
type = result;
last = prev;
    }

    TypeList newArg(Type arg)
    {
return (new TypeList(arg,this));
    }

    TypeList reverse(TypeList tail)
    {
if (last == null)
    return new TypeList(type, tail);
```

```
else
    return last.reverse(new TypeList(type, tail));
    }
    TypeList append(Type x)
    {
if (last == null)
    return new TypeList(type, new TypeList(x,null));
else
    return new TypeList(type, last.append(x));
    }

    TypeList reverseExceptLast()
    {
TypeList x = reverse(null);
if (x.last == null)
    return x;
else
    return x.last.append(x.type);
    }

    Type build ()
    {
return reverseExceptLast().buildInternal();
    }

    Type buildInternal()
    {
if (last == null)
    return type;
else
    return (new FuncType(type, last.build()));
    }

    private Type type;
    private TypeList last;
}
```

src/helper/nameType.java—Mike Glass (because Java doesn't have pairs)

```
package helper;

public class nameType {
public String name;
public Type type;
}
```

A.3.2 Package 'testing'

src/testing/Main.java—Morgan

```
/* Runs Syntactic and Semantic tests on files specified
 * in syntax/pass, syntax/fail, semantic/pass
 * and semantic/fail in the directory passed in
 * as a command-line argument
 */

package testing;

import helper.*;
import parser.*;
import java.io.*;

public class Main
{
    public static void main(String[] args) throws Exception
    {
SyntacticTest st;
SemanticTest sem;
String path;
String pass_path;
String fail_path;
String[] pass_files;
String[] fail_files;
String[] pass_files_sem;
String[] fail_files_sem;
File pass;
File fail;
```

```
int errCount = 0;
int total_err = 0;

path = args[0];
path += "/syntax";
System.out.println("PATH: " + path + "\n");
pass_path = path + "/pass";
fail_path = path + "/fail";

pass = new File(pass_path);
fail = new File(fail_path);

/* fills an array with files in the directory */
pass_files = pass.list();
fail_files = fail.list();

if(path.indexOf("syntax") != -1)
{
    System.out.println("SYNTAX TESTING\nPASS tests:\n");
    for(int i = 0; i < pass_files.length; i++)
    {
String tmp = "";
tmp = pass_path + "/" + pass_files[i];

/* only tests files ending in ".sampl" */
if(tmp.endsWith(".sampl"))
 {
     st = new SyntacticTest(tmp, 0);
     System.out.println(st.test());
     errCount += st.getErr();
 }
    }
    System.out.println("\nFAIL tests:\n");
    for(int j = 0; j < fail_files.length; j++)
    {
String tmp2 = "";
tmp2 = fail_path + "/" + fail_files[j];
```

```
if(tmp2.endsWith(".sampl"))
{
    st = new SyntacticTest(tmp2, 1);
    System.out.println(st.test());
    errCount += st.getErr();
}
    }

    total_err += errCount;
    if(errCount == 0)
System.out.println("\nALL SYNTAX TESTS COMPLETED CORRECTLY\n");
}

errCount = 0;
path = args[0];
path += "/semantic";
System.out.println("PATH: " + path + "\n");

        pass_path = path + "/pass";
        fail_path = path + "/fail";

        pass = new File(pass_path);
        fail = new File(fail_path);

        pass_files_sem = pass.list();
        fail_files_sem = fail.list();

        if(path.indexOf("semantic") != -1)
{
    System.out.println("SEMANTIC TESTING\nPASS tests:\n");
    for(int i = 0; i < pass_files_sem.length; i++)
    {
String tmp3 = "";
                tmp3 = pass_path + "/" + pass_files_sem[i];

                if(tmp3.endsWith(".sampl"))
{
    sem = new SemanticTest(tmp3, 0);
```

```
    System.out.println(sem.test());
    errCount += sem.getErr();
}

    }

    System.out.println("\nFAIL tests:\n");
    for(int j = 0; j < fail_files_sem.length; j++)
    {
String tmp4 = "";
tmp4 = fail_path + "/" + fail_files_sem[j];

if(tmp4.endsWith(".sampl"))
{
    sem = new SemanticTest(tmp4,1);
    System.out.println(sem.test());
    errCount += sem.getErr();
}
    }
}

total_err += errCount;
if(errCount == 0)
{
    System.out.println("\nALL SEMANTIC TESTS COMPLETED CORRECTLY");
}

if(total_err == 0)
{
    System.out.println("\nALL TESTS COMPLETED CORRECTLY");
}
else
        {
    System.out.println("\nTotal Errors: "+
       (total_err - errCount) +
       " syntactic errors\n\t" +
       errCount +
       "semantic errors");
```

```
}

    }
}
```

src/testing/SemanticTest.java—Morgan

```
package testing;

import parser.*;
import helper.*;
import antlr.*;
import antlr.collections.*;
import java.io.*;
import java.util.*;

public class SemanticTest
{
    /* test is file name to be tested for semantic correctness
     * passOrFail = 0 to test for pass, 1 to test for failure
     * prepares lexer and parser
     */
    public SemanticTest(String test, int passOrFail)
throws Exception
    {
ret = "";
errCount = 0;
this.passOrFail = passOrFail;
this.test = test;
t = new FileInputStream(test);
lexer = new SamplLexer(t);
parser = new SamplParser(lexer);
    }

    /* Returns String with results of run,
     * file that was tested, Exceptions for any errors
     * (Semantic errors and other errors),
     * along with stack trace for other errors (debugging help).
     */
```

```
    public String test() throws Exception
    {
if(passOrFail == 0)
{
    /* try to parse, build AST, build SymTable,
     * and then do static semantic analysis.
     * If no exceptions are thrown,
             * assumed success of test
     */
    try {
parser.program();
AST t = parser.getAST();
SamplTreeWalker walker = new SamplTreeWalker();
SymTable s = walker.buildSymTable(t);
walker.checkProgram(t,s);
ret = "SUCCESS " + test;
    } catch (Exception e){
if(e instanceof SemanticException)
{
    ret = "SEMANTIC FAILURE: " +
test +
"\nSemanticException: " +
e.getMessage();
    errCount++;
}
else
{
    ret = "OTHER FAILURE: " +
test +
"\nException: " +
e.getMessage();
    e.printStackTrace();
    errCount++;
}
    }
}
else
{
```

```
    /* fail tests.
     * Should fail due to semantic exceptions,
     * no other exceptions.
     */
    try {
                parser.program();
AST t = parser.getAST();
                SamplTreeWalker walker = new SamplTreeWalker();
SymTable s = walker.buildSymTable(t);
                walker.checkProgram(t,s);
ret = "INCORRECT SUCCESS: " + test;
errCount++;
    } catch (Exception e){
if(e instanceof SemanticException)
{
    ret = "CORRECT FAILURE - SEMANTICS: " +
test +"\nSemanticException: " +
e.getMessage();
}
else
{
    ret = "FAILED WITHOUT SEMANTIC EXCEPTION! " +
test +
"\nException: " +
e.getMessage();
    e.printStackTrace();
    errCount++;
}
    }
}

return ret;
    }

    /* returns errCount (# of files with errors)
     * in semantic analysis
     */
    public int getErr()
```

```
    {
return errCount;
    }

    private FileInputStream t;
    private SamplLexer lexer;
    private SamplParser parser;
    private String test;
    /* passOrFail = 0 => pass testing; = 1 => fail testing */
    private int passOrFail;
    private int errCount;
    private String ret;

}
```

src/testing/SyntacticTest.java—Morgan

```
package testing;

import parser.*;
import antlr.*;
import antlr.collections.*;
import java.io.*;
import java.util.*;

public class SyntacticTest
{
    /* passOrFail: 0 if you are testing for 'pass',
     * 1 if you are testing for 'fail'
     * test is filename
     * Prepares Lexer and Parser to run.
     */
    public SyntacticTest(String test, int passOrFail)
throws Exception
    {
ret = "";
errCount = 0;
this.passOrFail = passOrFail;
this.test = test;
```

```
t = new FileInputStream(test);
lexer = new SamplLexer(t);
parser = new SamplParser(lexer);
    }

    /* returns a string containing the result of the test,
     * the file that was tested
     * and any exceptions that would have been thrown
     */
    public String test() throws Exception
    {
if(passOrFail == 0)
{
    try {
parser.program();
ret = "SUCCESS " + test;
    } catch (Exception e){
ret = "SYNTACTIC FAILURE: " +
    test +
    "\nException: " +
    e.getMessage();
errCount++;
    }
}
else
{
    try {
                parser.program();
ret = "INCORRECT SUCCESS: " + test;
errCount++;
    } catch (Exception e){
                ret = "CORRECT SYNTACTIC FAILURE: " +
    test +
    "\nException: " +
    e.getMessage();
    }
}
```

```
return ret;
    }

    /* returns the error count (number of files with errors)
     * for the syntactic tests
     */
    public int getErr()
    {
return errCount;
    }

    private FileInputStream t;
    private SamplLexer lexer;
    private SamplParser parser;
    private String test;
    /* passOrFail = 0 => pass testing; = 1 => fail testing */
    private int passOrFail;
    private int errCount;
    private String ret;

}
```

A.4 Test cases

testing/semantic/fail/buildtype.sampl—Mike Haskel

```
let intensity f scalar x boolean y = f x y
let scalar a = a
let boolean b = b
let intensity c = f b a
```

testing/semantic/fail/wrongassign.sampl—Mike Haskel

```
let intensity a = b
let frequency b = b
```

testing/semantic/fail/wrongfuncapply.sampl—Mike Haskel

```
let (intensity->scalar) f = f
let intensity x = x
let intensity y = f x
```

testing/semantic/fail/wrongfuncassign.sampl—Mike Haskel

```
let (intensity->intensity) a = b
let intensity b = b
```

testing/semantic/fail/wrongfuncassign2.sampl—Mike Haskel

```
let (intensity->intensity) a = b
let (scalar->intensity) b = b
```

testing/semantic/fail/wrongfuncassign3.sampl—Mike Haskel

```
let (intensity->intensity) a = b
let (intensity->scalar) b = b
```

testing/semantic/fail/wrongfuncresult.sampl—Mike Haskel

```
let (intensity->scalar) f = f
let intensity x = x
let intensity y = f x
```

testing/semantic/pass/additive.sampl—Mike Haskel

```
let intensity a = a + a - a
let scalar b = b + b - b
let time d = d + d - d
let angle e = e + e - e
```

testing/semantic/pass/comparison.sampl—Mike Haskel

```
let intensity a = a
let scalar b = b
let frequency c = c
let time d = d
let angle e = e

let boolean x = a > a | b > b | c > c | d > d | e > e
let boolean y = a < a | b < b | c < c | d < d | e < e
```

testing/semantic/pass/funcargs.sampl—Mike Haskel

```
let intensity->intensity f = f
let intensity a = f a
```

testing/semantic/pass/funcargs2.sampl—Mike Haskel

```
let intensity->scalar->boolean f = f
let intensity a = a
let scalar b = b
let boolean c = f a b
```

testing/semantic/pass/ifthenelse.sampl—Mike Haskel

```
let boolean t = t
let intensity a = a

let intensity b = if t
  then a
  else a
```

testing/semantic/pass/implicitfunc.sampl—Mike Haskel

```
let intensity f scalar x = f x
let scalar a = a
let intensity b = f a
```

testing/semantic/pass/implicitfunc2.sampl—Mike Haskel

```
let boolean f scalar x intensity y = f x y
let scalar a = a
let intensity b = b
let boolean c = f a b
```

testing/semantic/pass/logical.sampl—Mike Haskel

```
let boolean a = a | a
let boolean b = b & b
let boolean c = c | c & c | c
```

testing/semantic/pass/multdiv.sampl—Mike Haskel

```
let scalar a = a

let intensity x = a * x / a

let time t = a * (o / f) / a
let frequency f = a * (o / t) / a
let angle o = a * (t * f) / a
```

testing/semantic/pass/otherassign.sampl—Mike Haskel

```
let intensity a = b
let intensity b = a
```

testing/semantic/pass/otherassignfunc.sampl—Mike Haskel

```
let intensity->intensity a = b
let intensity->intensity b = a
```

testing/semantic/pass/selfassign.sampl—Mike Haskel

```
let intensity a = a
let frequency b = b
let time c = c
let angle d = d
let scalar e = e
let boolean f = f
```

testing/semantic/pass/selfassignfunc.sampl—Mike Haskel

```
let intensity->intensity a = a
let intensity->intensity->intensity b = b
let (intensity->intensity)->intensity c = c
```

testing/semantic/pass/ski.sampl—Mike Haskel

```
let intensity I intensity x = x
let intensity K intensity x intensity y = x
let intensity S
    intensity->intensity->intensity x
    intensity->intensity y
    intensity z =
      x z (y z)
```

testing/syntax/fail/idstartnum.sampl—Mike Haskel

```
let intensity 1a = a
```

testing/syntax/fail/nolet.sampl—Mike Haskel

```
intensity a = a
```

testing/syntax/fail/nolet2.sampl—Mike Haskel

```
let intensity a = a
intensity b = b
```

testing/syntax/fail/numproblem.sampl—Mike Haskel

```
let intensity a = 1.1e-
```

testing/syntax/pass/application.sampl—Mike Haskel

```
let intensity a = a b
let intensity a = a b c
let intensity a = (a b) c
let intensity a = a (b c)
let intensity a = (a b)
```

testing/syntax/pass/arg1.sampl—Mike Haskel

```
let intensity a intensity b = a
```

testing/syntax/pass/arg2.sampl—Mike Haskel

```
let intensity a intensity b intensity c = a
```

testing/syntax/pass/basetypes.sampl—Mike Haskel

```
let scalar a = a
let boolean a = a
let intensity a = a
let time a = a
let frequency a = a
let angle a = a
```

testing/syntax/pass/basic.sampl—Mike Haskel

```
let intensity a = a
```

testing/syntax/pass/binop.sampl—Mike Haskel

```
let intensity a = a + a
let intensity a = a - a
let intensity a = a * a
let intensity a = a / a
let intensity a = a < a
let intensity a = a > a
let boolean a = a | a
let boolean a = a & a

let intensity a = (a < a)
let intensity a = (a * a)
let intensity a = (a + a)
let intensity a = (a < a)
let boolean a = (a | a)
```

testing/syntax/pass/binopmulti.sampl—Mike Haskel

```
let intensity a = a + a + a + a
let intensity a = a + a * a + a
let intensity a = a / a / a + a * a | a
```

testing/syntax/pass/binopnested.sampl—Mike Haskel

```
let intensity a = (a + a) / a + (a - a)
let intensity a = (a + (a + (a - (a + (d * c) / (a | a)))))
```

testing/syntax/pass/functypes.sampl—Mike Haskel

```
let intensity->intensity a = a
let (intensity->intensity) a = a
let intensity->intensity -> intensity a = a
let intensity->(intensity -> intensity) a = a
let (intensity->intensity) ->intensity a = a
let (intensity -> intensity-> intensity) a = a
```

testing/syntax/pass/ifthenelse.sampl—Mike Haskel

```
let boolean a = if a then a else a
let boolean a = (if a then a else a)
```

testing/syntax/pass/ifthenelsenested.sampl—Mike Haskel

```
let boolean a = if a
then a
else if a
     then if a
  then a
                           else a
     else a
let boolean a = (if a
 then a
 else (if (if a then a else a)
       then (if (a)
    then a
    else a)
       else a))
```

testing/syntax/pass/mixexpr.sampl—Mike Haskel

```
let intensity a = if a
              then a + b
      else a < b
let intensity a = if a | b
  then a < foo b
  else (a * b)
```

testing/syntax/pass/multidef.sampl—Mike Haskel

```
let intensity a = a
let intensity b = b
let intensity c = b
```