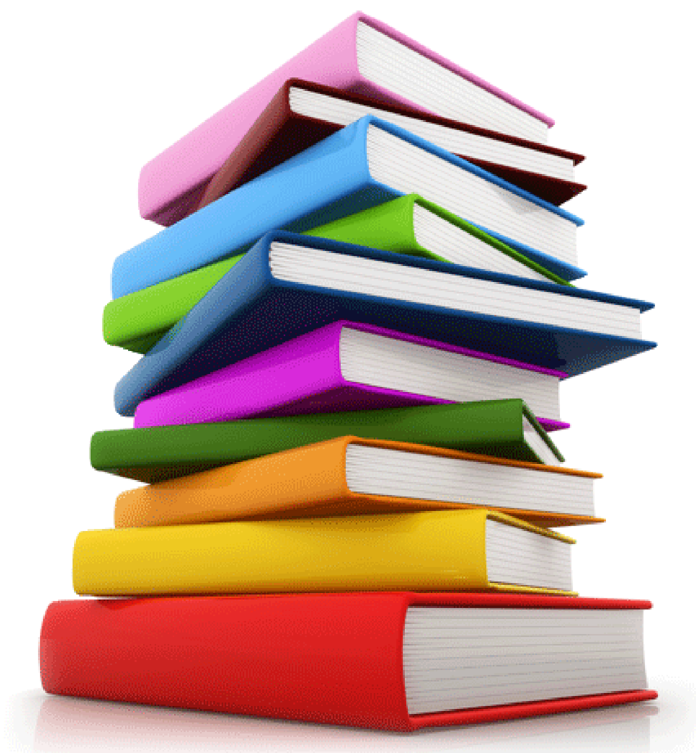
Fall 2015 COMS 4115

Programming Languages & Translators

Final Report

StoryBook



Authors

Anna Lawson (aal2150)

Beth Green (blg2132)

Nina Baculinao (nb2406)

Pratishta Yerakala (py2211)

Table of Contents

A. Full Source Code

A.1 ast.ml

A.2 scanner.mll

A.3 parser.mly

A.4 sast.ml

A.5 semantic_analyzer.ml

A.6 cast.ml

A.7 pretty_print.ml

A.8 codegen.ml

A.9 Makefile

A.10 Test Script

A.11 Tests

A. Project Log

I. Introduction

1. Overview

StoryBook is a pedagogical programming language targeted toward novice programmers who are just beginning to understand the basics of computer science and logical thinking. The language uses intuitive, "story-like" syntax and structure to make object-oriented programming easier for children and adult-beginners to read and implement.

This language analogizes the structure of object-oriented programming to the structure of a story. An object-oriented language is comprised of functions and classes with instance variables and methods that come together to create a program. Likewise, a story consists of chapters and characters with traits and actions that come together to create a story. StoryBook synthesizes these structures to create a platform to introduce object oriented programming and computer science to children as well as adults. In storybook, *characters* are objects. Each character can have its own *traits* (instance variables) and *actions* (methods). These characters can act in *chapters* (functions), and multiple chapters can come together in sequence to form the *plot* (main function) of a story. Just as one character in a story can pass critical information along to another, and what happens in one chapter might influence later events in a story, the subcomponents of a Storybook program can communicate by sharing (returning) information about their "conclusions." This allows users to learn the basics of computer science, and more specifically, object-oriented programming, in the familiar, intuitive context of stories.

Colloquial words are used for reserved keywords instead of symbols that are not intuitive to most. Moreover, common symbols, such as =, are used as they are in basic math and vernacular, rather than adhering to computer science conventions that may be counterintuitive to novices. This minimizes the syntax learning curve for beginners, allowing them to instead focus on mastering basic concepts of computer science and learning to think in a more logical way before moving on to more complex languages.By allowing users to implement basic algorithms using simple syntax and the familiar structure of stories, StoryBook serves as an introductory programming language that can be a gateway to more complex languages and computer science concepts.

2. Motivation

Computer Science, programming especially, is rapidly becoming a larger part of the educational curriculum from grades K-12. However, many students are expected to simply jump into logical thinking via courses like AP Computer Science or haphazardly pick it up by reading endless threads on StackOverflow. We propose StoryBook, a programming language that focuses on readability and intuitiveness to help younger kids learn to think in a logical way and embrace computational thinking. It helps bridge the gap between passively making logical sentences and the deliberate steps it takes to use logic in solving a problem.

II. Language Tutorial

1. Crash Course in StoryBook

Types:

number	(float)
words	(string)
letter	(char)
tof	(boolean)

Operators:

```
+ - * / % is > < >=
<= , 's and or not
```

Comments:

```
~Block~
~~In line~~
```

Basic Program Structures:

Characters	(objects)
Actions	(methods)
Chapters	(function)

Example:

```
Character Monster( words n; number s ) {
  words name is n.
  number size is s.

  Action scare(words scream) returns
nothing {
    say(scream).
  }
}

Chapter plot() returns nothing {
  Character Monster Frank is new
Monster("Frankenstein"; 99).
  say(Frank's name + ":").
  Frank, scare("GLABARGHHHHH!").
}
```

2. Examples

A simple "Hello world"-like program in StoryBook.

```
Chapter plot() returns nothing {
  say("Once upon a time...").
}
```

3. Installing the Compiler

Installation of the StoryBook compiler requires the OCaml and C (gcc) compiler. If you are not in possession of the tar files and have Git version control on your machine, you can download the compiler via:

```
git clone git@github.com:blanksblanks/StoryBook.git
```

Run `make` in the source directory and voila, the compiler should be installed!

4. Running the Compiler

A StoryBook program file has a `.sbk` extension. Running make from the top level directory will generate an executable, `run`, which converts StoryBook code into C code. The `storybook` script uses `run` to pipe the C code into a C file, then compile it with `gcc -std=c99`, and run the output of the C program, therefore producing the output of the StoryBook program.

Example of running the "Hello world" program:

```
./storybook hello_world.sbk
```

Output:

```
Once upon a time...
```

III. Language Reference Manual

1. Introduction

Once upon a time, the creators of Storybook were learning how to code for the first time. At first, they fumbled with the tricky and alien syntax. It took a while for them to discover the joyful creativity of computer programming.

StoryBook is a programming language targeted toward novice programmers who are just starting to understand the basics of computer science and computational thinking. The language uses intuitive, "story-like" syntax and structure to make object-oriented programming easier for children and adult-beginners to read and implement. The backend of StoryBook generates C code which can be compiled and run to produce the desired output from the StoryBook program.

2. Syntax Notation

The syntax notation of this manual is as follows. Any literals or words that belong to the StoryBook language will be written in `monospaced typeface.` Syntactic categories are written in *italic*. Any items with *-list* appended to it refers to 1 or more of those items, while the subscript $_{\text{opt}}$ is for optional terminal or nonterminal symbols. Sometimes these items will appear in shortened form. Therefore, *expr-list*$_{\text{opt}}$ means 0 or more *expressions*.

Grammar patterns are expressed throughout the document using regular expressions. r* means the pattern r may appear zero or more times, r+ means r will appear one or more times, and r? means r will appear one or zero times. r1|r2 means that the pattern has either r1 or r2. r1r2 means that the pattern r1 is concatenated with r2.

3. Lexical Conventions

StoryBook programs are lexically composed of three elements: comments, tokens, and whitespace.

3.1 Comments

Symbol	Description	Example
`~~`	single line comment	`~~Single line comment`
`~ ~`	block comment	`~Multi-line comment~`

Single line comments may be nested in block comments, but block comments may not be nested within other block comments.

3.2 Tokens

A token in StoryBook is a group of characters that hold meaning when considered as a group. These consist of *keywords*, *identifiers*, *operators*, *separators*, and *constants*.

3.2.1 Keywords

These are all the keywords in StoryBook:

`Chapter`	`Character`	`Action`	`new`	`my`	`returns`	`endwith`	`list`
`number`	`words`	`letter`	`tof`	`true`	`false`	`nothing`	`not`
`and`	`or`	`is`	`if`	`else`	`repeatwhile`	`repeatfor`	`say`

3.2.2 Identifiers

identifier → ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*

Identifiers are a collection of characters, numbers, and/or underscores, which must begin with at least one character. The characters are the ASCII characters 'a'-'z' and 'A' -'Z', numbers are digits 0-9, and underscore '_'. StoryBook is case sensitive. Identifiers hold values that are of the type to which they are assigned.

3.2.3 Operators

operator →

```
+
-
*
/
%
<
>
<=
```

```
>=
=
!=
is
,
's
 and
 or
 not
 my
```

In StoryBook there are arithmetic, comparison, logical, access and instantiation operators. The syntax and use of these operators are described in 6.2, 6.3 and 6.4.

3.2.4 Constants

boolean → `[true false]`

letter → `['a'-'z' 'A'-'Z']`

string → `'"'( ('\\'('/'|'\\'| 'b' | 'f' | 'n' | 'r' | 't'))|([^'"']) )*'"'`

digit → `[0-9]`

number → `['-']?(`*digit*`+) | ['-']?(`*digit*`*'.'`*digit*`+) | ['-']?(`*digit*`+'.'`*digit*`*)`

constant → *boolean*
letter
string
number

Constant are values in StoryBook that always have the same value include `true`, `false`, any character, any string, and any negative or non-negative number.

3.2.5 Separators

separator → `;`
`.`

StoryBook uses `;` to separate items in a list of function arguments or in a `list` data structure. A `.` is used to mark the end of an expression.

3.2.6 Newlines

StoryBook uses newlines to identify the end of a single line comment. Otherwise, newlines are ignored by the compiler.

3.3 Whitespace

Tabs and spaces are used by StoryBookers to make their programs more readable. However, whitespace is ignored by the compiler.

4. Data Types

4.1 Primitive Data Types

There are five primitive data types in Storybook: `letter`, `words`, `tof`, and `number`.

Type	Example	Definition
`letter`	`'a'`	- Single character
`words`	`"apple"`	- Grouping of consecutive characters, a string
`tof`	`true`	- Boolean type, holds a value of `true` or `false`
`number`	`1.5`	- A decimal floating-point number with precision of about 24 bits or about seven decimal digits

4.2 Non-Primitive Data Types

Beyond the four basic types, there is a class of derived data types that can be constructed from the basic types into conceivably endless varieties of user configuration.

4.2.1 Chapters

A `Chapter` is a user-defined function that performs operations and may or may not take parameters of a certain type and may or may not return a variable of a given type. Prewritten functions in StoryBook are known as Library `Chapter`s.

4.2.2 Characters

A `Character` is a user-defined data type comprised of traits (instance variables) and `Action`s (methods). Traits can be of a primitive type or of a `Character` type.

4.2.3 Lists

A `list` is a built-in data structure that can hold multiple instances of *letter*s*, tofs, number*s, or user-defined *Characters*; all values in a list must be of the same type.

4.3 Scoping and Lifetime

A variable's scope is the block *after* the variable is declared, with the exception of traits.

In the case of nested blocks, if a variable declared within an inner block and shares the same name as a variable declared in the parent block, then the variable declared in the inner block takes precedence, effectively overriding the one in the outer block. Thus, in this case, the outer block's variable with the shared name is inaccessible from the inner block. If two variables are declared within the same block level, consequently sharing the same scope, with the same name, the one declared later will take precedence and the earlier one will be inaccessible after the point of the later variable's declaration.

Traits have the lifetime of their object. All non-trait variables have a lifetime from their declaration's execution to when the program counter exits the block in which the variable was defined.

There are no global variables in StoryBook. However, Chapters and Characters are global and their identifiers are accessible to all Chapters for the life of the program.

5. Purpose of Identifiers

What's in a name? An identifier is an alphanumeric sequence of characters that amounts to either a *keyword* or the name of a `Chapter`, `Character`, `Action` or a variable. This sections details the purpose and scope of the possible types of non-keyword identifiers.

5.1 Chapters

In Storybook, a `Chapter` is any function that is not inside of a `Character`. `Chapter`s enable users to create reusable and versatile blocks of code that can be called in other `Chapter`s. `Chapter`s can take zero or more arguments. Each `Chapter` can have zero or one return value. All argument and return types must be declared in the `Chapter` header.

5.2.1 Plot

The `plot` is the main function and entry point for the program. The minimum requirement for a valid and executable Storybook program is the declaration of a `Chapter` with the `plot` identifier and nothing to return. Users can construct concise `plot`s that are either comprised of or include complex sequences of `Chapter` calls.

5.2 Characters

In Storybook, classes are called Characters. Characters are user-defined data types that represent a type of object. Users can then instantiate Character objects of a specific Character type. Each Character object has its own copy of instance variables known as traits and can perform `Action`s.

5.2.1 Inheritance

Inheritance can be employed to create subclasses of `Character`s and avoid duplication of code for shared functionality. This structure allows users to define reusable data types and to abstract the implementation details of story characters. `Character`s allow computer science novices to begin to understand the key concepts object-oriented programming and inheritance in the familiar context of story characters.

5.3 Actions

`Action`s are methods that can be invoked on instances of a `Character`. `Action`s are defined inside the `Character` class definition. Like `Chapter`s, they can have zero or more arguments and zero or one return values, and all typed arguments and return types must be listed in the `Action`s header.

5.4 Variables

In StoryBook, variables are statically-typed. A variable is an identifier that is bound to a value of one of the following types: `Character`, `letter`, `words`, `tof`, `list`, or a `number`. Variables of type `number` are dynamically typed in that they can be initialized and re-assigned to any type of number. The variables in StoryBook are mutable.

5.5 Traits

In StoryBook, traits represent the object-oriented concept of instance variables. Traits are variables that are defined at the scope of a `Character` type. Each instantiated object of that `Character` type has its own instance of each trait. When traits are inherited from a parent class to a subclass, the traits from the parent class take precedence in terms of order when passing them in as parameters to to instantiate a new subclass object. After the parent's traits, the subclass' own traits can be initialized in the parameters.

6. Expressions

This section describes the syntax of StoryBook *expressions*. StoryBook uses postfix, prefix, or infix operators. The precedence of expression operators mirrors the order of the major subsections of this section, highest precedence first. Within each subsection, the operators have the same precedence. The grammar of StoryBook incorporates the precedence and associativity of the operators.

6.1 Primary Expressions

primary-expr → *constant*
identifier
`(` *expression* `)`

Primary expressions include *identifiers*, *constants*, or *expressions* that can be evaluated to a single value in parentheses.

6.1.1 Identifiers

An *identifier* for a variable is a primary expression, provided it has been fully declared and holds a value. A variable *a* is a primary expression whose type is the same as the type of *a*. Evaluation of an identifier actually entails evaluation of the expression bound to that variable. Identifiers are described in section 3.2.2.

6.1.2 Constants

A *constant* is a primary expression with the same type as the type of the literal, which can be of type `tof`, `letter`, `string`, or `number`. See 3.2.4 for a discussion of constants.

6.1.3 Parenthesized Expressions

`(` *expression* `)`

A parenthesized expression is a primary expression whose type and value are identical to the final evaluation of an un-parenthesized expression.

6.1.4 Lists

expr-list → `[` $expression_1$ `;` $expression_2$ `;` … `;` $expression_n$ `]`

where $1 \leq i \leq n$ and *n* is the length of the *expr-list*.

A `list` is a primary expression that can contain zero or more expressions. The expressions in a list must all be of the same type. List elements are assigned one by one.

6.2 Postfix Expressions

postfix-expr → *primary-expr* `[` *expression* `]`
primary-expr `'s` *expression*
primary-expr`,` *expression*
primary-expr `(` $expr\text{-}list_{opt}$ `)`

The operators in postfix expressions group from left to right.

6.2.1 List Access and List Instantiation

The expression $expression_1$`[` $expression_2$ `]` denotes the accessing of list elements. First $expression_1$ is evaluated, then $expression_2$, then the `[]` operator. It returns the value at the position denoted by $expression_2$ in the list denoted by $expression_1$. Position numbers in the list begin at 0 and end with the length of the list minus 1. List elements are assigned one by one. This is done to explicitly show students how each element in a list is assigned to a position.

6.2.2 Character Access: Traits

The `'s` operator is used to access a `Character`'s `trait`s in external `Chapter`s.

6.2.3 Character Access: Actions

The `,` operator is used to access a `Character`'s `Action`s. The second expression should be an `Action` call expression.

6.2.4 Chapter Call, Action Call, and Character Instantiation

`Chapter`s and `Character Action`s can be called in the scope in which they were created by the `Chapter` *identifier* and the appropriate arguments wrapped in a pair of parentheses `()`. A function without any arguments is simply called with *primary-expr* `()`.

Unlike Chapters and Actions, a Character instantiation cannot happen without specifying the Character's traits (instance variables) in the parameters. This is done so that students will not be confused about how an object can have a certain trait but not define it from the very beginning.

6.3 Prefix Expressions

prefix-expr → `not` *expression*
`my` *expression*
`new` *expression*

Expressions with unary operators group right-to-left.

6.3.1 Logical Negation

The operand of the `not` operator must have a `tof` type. The result of the prefix expression `not true` is `false` and the value of `not false` is `true`.

6.3.2 Character Instantiation

`new` is used to construct new `Character` instances.

6.3.3 Character Access

`my` is used in a `Character`'s `Action` methods to access its own traits, as opposed to the `'s` operator used in external functions.

6.4 Binary Operator Expressions

binary-expr → *expression*$_1$ *operator* *expression*$_2$

The following categories of binary *operators* exist in StoryBook, and are listed in order of decreasing precedence: arithmetic, concatenation, comparison, logical, assignment and sequence.

6.4.1 Arithmetic Operators

arithmetic-expr → $expression_1$ `*` $expression_2$
$expression_1$ `/` $expression_2$
$expression_1$ `%` $expression_2$
$expression_1$ `+` $expression_2$
$expression_1$ `-` $expression_2$

Operator	Description	Example
`*`	multiply	`5*10    ~~evaluates to 50`
`/`	divide	`55/10   ~~evaluates to 5.5`
`%`	modulo	`5%2     ~~evaluates to 1` `5%2.5   ~~evaluates to 1`
`+`	add	`1+1     ~~evaluates to 2`
`-`	subtract	`10-4    ~~evaluates to 6`

The multiplicative operator `*`, the division operator `/` and the remainder operator `%` are all grouped left-to-right. The operands must have `number` type. The binary operator `*` denotes multiplication of the two operands. The binary `/` operator yields the quotient, which is always the result of floating point division of the first operand by the second. The `%` operator converts the float operands to integers, then takes the remainder of a product of the integer division. The remainder result is a float `number` type. If the second operand is 0 for the `/` or `%` operator, the result is undefined.

Of lower precedence than the multiplicative operators, the additive operator + and subtractive operator `-` also group left-to-right. As long as the operands are of the `number` type, the result of the `+` operator is the sum of the operands. The `+` operator can also have operands of other types, in which case the function of the operator changes to concatenation, which is discussed in the next subsection. The result of the `-` operator is the difference of the operands. The operands for subtraction must be of `number` type.

6.4.2 Concatenation Operator

concatenation-expr → $expression_1$ `+` $expression_2$

Allowed Operand Types	Example
`words` and `words`	`"Story" + "Book". ~~"StoryBook"`
`words` and `letter`	`"Hello" + '!'     ~~"Hello!"`
`words` and `number`	`"Alibaba and the " + 40 + " thieves".` `~~ "Alibaba and the 40 thieves"`
`words` and `tof`	`"Today you are you! That is truer than " + true + "! There is no one alive who is you-er than you!"` `~~ "Today you are you! That is truer than true! There is no one alive who is you-er than you!"`

The + operator is distinguished from the other arithmetic operators because its operands do not have to be of `number` type or even of the same type as each other. So long as one of the right-hand or left-hand operands is of type `words`, then the + operator performs concatenation, such that the result of the + operator is the concatenated result of the two operands and is of type `words`. If one operand is of type `words` and the other operand is a different data type, the non-`words` operand is cast to type `words`; then regular string concatenation takes place, and the final concatenated result is of type `words`.

The concatenation operator has the same level of precedence as addition and subtraction, and also groups left-to-right. By this logic, `1 + 1 + "one"` evaluates to `"2one"` while `"one" + 1 + 1` evaluates to `"one11"`.

6.4.3 Comparison Operators

comparison-expr → $expression_1$ `<` $expression_2$
$expression_1$ `>` $expression_2$
$expression_1$ `<=` $expression_2$
$expression_1$ `>=` $expression_2$
$expression_1$ `=` $expression_2$
$expression_1$ `!=` $expression_2$

Operator	Description
`<`	is less than
`>`	is greater than
`<=`	is less than or equal to
`>=`	is greater than or equal to
`=`	tests equality
`!=`	tests inequality

The operands must have `number` type. The final result of a comparison expression is of type `tof`. The equality operator and the inequality operator have lower precedence than the other comparison operators. Thus, $expression_1$ *op* $expression_2$ and $expression_3$ *op* $expression_4$ evaluate to `true` if both $expression_1$ *op* $expression_2$ and $expression_3$ *op* $expression_4$ share the same `tof` value. In other words, both the expressions `1 < 2 = 3 < 4` and `1 > 2 = 3 > 4` will evaluate to `true` as they are respectively equivalent to `(1 < 2) = (3 < 4)` and `(1 > 2) = (3 > 4).`

6.4.5 Logical Operators

logical-expr → $expression_1$ `and` $expression_2$

expression$_1$ `or` *expression*$_2$

Operator	Description	Example
`and`	logical and	`true and true ~~evaluates to true`
`or`	logical or	`false or true ~~evaluates to true`

The logical operators group left-to-right. The operands have to be of `tof` type and the result of a logical expression is always of type `tof.`

`and` returns `true` if both its operands are unequal to `false`, otherwise it returns `false`. It guarantees left-to-right evaluation and adopts short-circuit evaluation. The first operand is evaluated; if it is equal to `false`, the value of the entire expression is immediately set to `false`. Otherwise, the right operand is evaluated, and if it equal to `false`, the whole expression is `false`, otherwise `true`.

`or` returns `true` if either of its operands are not equal to `false`, otherwise it returns `false`. It also guarantees left-to-right evaluation and adopts short-circuit evaluation. The first operand is evaluated; if it is equal to `true`, the value of the entire expression is immediately set to `true`. Otherwise, the right operand is evaluated, and if it equal to `true`, the whole expression is `true,` otherwise the expression is equal to `false`.

6.4.6 Assignment Operator

assignment-expr → *lhs* `is` *expression*

The `is` assignment operator groups right-to-left. The value of the expression replaces the value stored in the identifier of the *lhs*. It can be of any of the primitives types or a `Character` type, but must not be of type `Chapter`. The left operand must be a declared identifier. The type of an expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place. The operand on the right must have the same type as the left operand.

For instance:

identifier `is 5. ~~the identifier is set to the value of 5`
identifier `is (5+1). ~~the same identifier changes to the value of 6`

6.4.7 Sequence Operator

sequence → [*expression; expression;*]+

Expressions separated by semicolons are evaluated left-to-right. The expression `x is 5;  x + 6` is therefore equivalent to the two expression statements:

```
x is 5.
x is x + 6.
```

See 7.1 immediately below for a discussion of expression statement syntax.

7. Statements

Statements are executed in sequence.

7.1 Expression Statement

expression `.`

Statements are marked with a period `.` to resemble regular English sentences. Most statements are expression statements, and most expression statements are assignments or function calls.

7.2 Block Statement

block-stmt → `{` *statement-list* `}`

statement-list → *statement*

statement statement-list

Block statements can contain one or more statements.

7.2 Conditional Statement

conditional-expr → `if (` *expression* `)` *block-stmt*

`if (` *expression* `)` *block-stmt* `else` *block-stmt*

Each expression after an `if` must be an expression of `tof` type that evaluates to `true` or `false`. Parentheses around the *expression* condition are required. If the expression evaluates to `true` then the expression following the subsequent `then` is executed. Otherwise, the expression following the subsequent `else` is executed. In the case of multiple `if` statements preceding an `else` clause, then the `else` binds to the immediately preceding `if` block with the `{` *statement-list* `}` immediately preceding the `else` keyword. In the example below the last statement would be evaluated as the first two conditions are not `true`.

```
if (1 = 2) {
   statement1
} else if (1 = 2) {
  statement2
} else {
  statement3
}
```

7.3 Loop Statements

loops-stmt → `repeatwhile (` *expression* `)` *block-stmt*

`repeatfor` *expression1*`;` *expression2*`;` *expression3* *block-stmt*

The substatement in the block executes iteratively until the value of the expression is no longer `true` (hence the *expression* must be of type `tof`).

7.3.2 While Statement

`repeatwhile (` *expression* `)` *block-stmt*

The block of code defined in the `repeatwhile` loop will be executed while the *expression*, which must be of type `tof`, evaluates to `true`. If the condition is always `true`, such as `( 7 = 7 )`, the result would be infinite loop. The parentheses around the *expression* is required.

Hypothetically, if we were to have a `Character` instance with the identifier `SleepingBeauty` with the traits `age` and `snore Action`, then a while loop could look like:

```
repeatwhile ( SleepingBeauty's age != 100 ) {
       SleepingBeauty, snore.
       SleepingBeauty's age is SleepingBeauty's age + 1.
}
```

7.3.2 For Statement

`repeatfor` *expression1*`;` *expression2*`;` *expression3 block-stmt*

This is the syntax of a StoryBook `repeatfor` loop is equivalent to:

> *expression1*.
> `repeatwhile` *expression2* `{`
>   *statement*`.`
>   *expression3*`.`
> `}`

The first expression is evaluated only once and initializes the loop. There is no restriction on its type. The second expression must evaluate to type `tof` and is typically a condition for the loop to continue. Once the second expression evaluates to false, the loop ends. The third expression is evaluated after each iteration and specifies a re-initialization for the loop. There is no restriction on its type. The block of code defined after the `repeatfor` line will be executed as long as the *tof-expression* evaluates to `true`.

The suggested pattern used in a *for-loop* is to have the sequence below, an assignment expression, then a comparison expression to check for `tof`-ness, followed at last by a reassignment expression to change the original variable assignment in the sequence. In the example below, an *identifier* is assigned to an initial `number` value, then this value is checked by a comparison operator, and then the value is incremented with each iteration of the loop.

```
repeatfor (number x is 5; x < 10; x is x + 1) {
```

```
    ~~this will print "hello" 5 times
    say("hello").
}
```

7.4 Return Statements

return-stmt → `endwith` *expression*

A `Chapter` returns values to its caller via the `endwith` statement. Not all `Chapter`s have `endwith` statements, depending on their `Chapter` declarations, which will be discussed in the next section. If the expression that is returned is a `number` literal, parentheses are required because a statement like `endwith(0).` could be construed as an unfinished return statement as the period is interpreted by the scanner and parser as a decimal point for the returning `number` value.

8. Declarations and Types

8.1 Type Signatures

type-signature → *type identifier*

type → `number`
`letter`
`words`
`tof`
`charlist`
`toflist`
`numberlist`
`characterlist`
`Character` *identifier*

When declaring a variable prepend each declaration with the data type, for example:

```
number age   letter initial     words dialogue     tof asleepOrNot
```

8.2 Declarations

variable-declaration → *type identifier*

chapter-declaration → `Chapter` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`

character-declaration → `Character` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`
`Character` *identifier* `is` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`

action-declaration → `Action` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`

8.2.1 Variable Declarations

variable-declaration → *type identifier*

Variable declarations serve in many ways. When they immediately follow the *identifier* in a *chapter-declaration, action-declaration* and *character-declaration*, these variables serve as parameters. Inside the a *stmt-list* , *variable declarations* are often assigned values. Within a `Chapter` body, a list of *variable-declaration*s serve as local variables. Inside a `Character` body, a list of *variable-declaration*s serve as traits or instance variables. They are often defined immediately after declaration when they appear in a statement block, as in.

```
tof asleepOrNot is false.
```

8.2.2 Chapter Declarations

chapter-declaration → `Chapter` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`

`Chapters` are declared with zero or more parameters, separated by semicolons, and a return value preceded by the keyword `returns`. For instance:

```
Chapter sum (number x; words y) returns number {
  endwith( x + y ).
}
```

8.2.3 Character Declaration and Instantiation

character-declaration → `Character` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`

`Character` variable names are capitalized by convention. Inside the braces of a `Character` declaration the user can declare zero or more traits and `Action`s. To create an instance of a `Character` the `Character` *identifier* is prepended to the instance *identifier* and assigned to a `new Character` of that type. `Trait`s are defined during instantiation by passing the values in as arguments.

```
Character Monster( words n; number s) {
  words name is n.
  number size is s.

Action scare(words scream) returns nothing {
    ~~ print scream and name separated by a space
    say ( scream + ' ' + (my name) ).
  }
}

Monster Frank is new Monster(name is "Frankenstein"; size is 99).
say(Frank’s name). ~~prints Frankenstein
Frank, scare("AHHHHHH"). ~~prints AHHHHHH
```

8.2.4 Character Subtype Declaration

character-declaration → `Character` *identifier* `is` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`

Subtypes are declared with very similar syntax as a normal `Character`. However note that after its identifier, in the signature, the type signature of the `Character` is assigned to its superclass. The resulting declaration reads very intuitively:

```
~We can also call scare on Giant because of inheritance, and Giant gets the same
traits: name and size~

Character Giant is Monster() {
 Action crush(number personHeight) returns tof {
    tof crushed is false.
    if personHeight < size {
      crushed is true.
    } else {
      crushed is false.
    }
    endswith crushed.
  }

}
```

`Character Giant` has inherited all of `Monster`'s traits. Therefore, in order to to create a `Giant` instance, the appropriate Character `instantiation is:`

```
Character Giant Fum is new Giant("Fum!" , 500). ~~need arguments
Fum, scream("Fee Fi Fo!") ~~print out "Fee Fi Fo! Fum!"
Fum, crush(6). ~~returns true
```

8.2.5 Action Declarations

action-declaration → `Action` *identifier* `(` *var-decl-list*$_{opt}$ `)` `returns` *type* `{` *stmt-list* `}`

`Action` declarations are nearly identical to those of `Chapter`s except the first keyword is `Action` instead of `Chapter`. Otherwise, they are also are declared with zero or more parameters, separated by semicolons, and a return value preceded by the keyword `returns`. These declarations have to be inside of a `Character` body.

```
Action makeMoney(number initialAmnt, number salaryPerMonth, number monthsWorked)
returns number {
  endwith(initialAmnt + salaryPerMonth + monthsWorked).
}
```

8.2.6 Trait Declarations

Trait declarations contain the exact same syntax as variable declarations, but must be declared inside of a `Character` body. However, to access a trait variable outside of the Character requires the `'s` operator to access the variable, while accessing the trait variable inside an Action requires the `my` operator, as discussed in 6.3.3 Trait Access.

8.2.7 List Declarations

list-signature → *type* `list` *identifier*

`List`s are treated as variables in the compiler and can be declared as a regular data type by the user. `List`s can only contain one type of data type and therefore each type of list has its own type to distinguish this fact: `numberlist`, `letterlist`, `toflist`, and `characterlist`.

```
numberlist dwarfAges is new numberlist[5]. ~declares an empty list of numbers of
                                   length 5 called dwarfAges~
```

8. Program Structure

program → *chapter-decl-list* ; *character-decl-list*$_{opt}$

A program in StoryBook comes down to a list of optional `Chapter` declarations and a list of `Character` declarations. Such declarations have program-wide scope.

8.1 Required for Every Good StoryBook: a `plot`

The order of `Chapter` declaration in a StoryBook program is not important as all are visible when the program starts. However one requirement for any executable StoryBook program is the declaration of a `Chapter` by the identifier of `plot` with the signature:

```
Chapter plot() returns nothing
```

When a StoryBooker runs a Storybook program, the first function that is called is the `plot`.
As discussed in 5.2.1, the `plot` is the main function and entry point for the program. The program executes with the first statement in the code block of the `plot` function. The program executes sequentially, statement by statement, until the closing brace of the `plot` function. Only one `plot` is allowed in a StoryBook program.

8.2 Library Chapter: `say`

The `say` function is a built-in `Chapter` that prints its input to standard output. It is a unique `Chapter` in that it accepts <u>any</u> type of expression that can be evaluated to any of the four basic data types: `letter`, `words`, `tof`, or `number`. It prints its input out as a string.

For instance:

```
words pirateName = "Captain Jack Sparrow".
say("Ah" + 0 + "y, " + pirateName + '!'). ~~prints "Ah0y, Captain Jack Sparrow!"
```

IV. Project Plan

The processes for planning started with regularly meeting after our weekly meeting with our TA, Richard Townsend, to discuss and work on the various aspects of the language. Initially, the meetings consisted of how the planning will work, scheduling times when everyone is free to meet and work, discussing strategies on teaming up for certain features of the program, version control, etc. Facebook Messenger was used to schedule meetings and for checking in to see how each member is proceeding and whether to adjust the timeline by either prioritizing and pushing some features back or teaming up to push forward.

1. Team Responsibilities

While we had initial team roles, we discovered later that some members were better and more efficient with certain aspects of the language. These are the final "roles" to which we stuck, but are definitely not reduced to as the members have contributed to many more aspects than the role strictly defines.

Student	**Role**
Anna Lawson	System Architect
Beth Green	Language Guru
Nina Baculinao	Testing
Pratishta Yerakala	Project Manager

Throughout the process, responsibilities were broken down into features. Specifically, Anna was in charge of the main plot and functions; Beth was in charge of returns, assignment and loops; Nina was in charge of types such as numbers and floats; Pratishta was in charge of arithmetic and logical operations. Anna and Beth were the dynamic duo that brought `Characters` with inheritance and `Lists` to life. A more detailed list of how features were implemented here (generally in this order):

Main function and functions: Anna
Say: Anna
Return values: Beth, Anna
Numbers and comments: Nina
Binary and Unary operators: Pratishta
Concatenation: Anna, Beth, Pratishta, Nina
Assignment: Beth, Anna, Nina
Loops: Beth
Characters: Anna, Beth, Nina, Pratishta
Inheritance: Beth, Anna, Nina
Lists: Anna, Beth

2. Style Guide

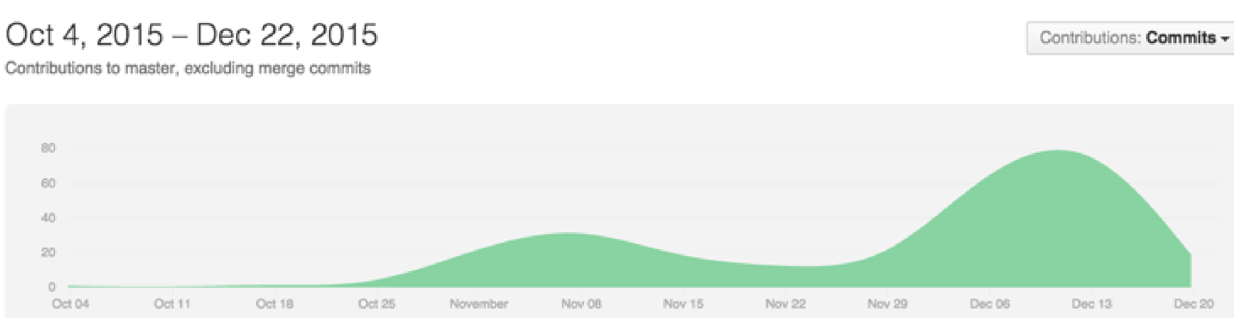
- Git commits and logs have been used extensively to see changes made by members
- Facebook Messenger was used most frequently to communicate whatever has been updated
- Every new feature added called for an accepted and rejected test case with special syntax for being named. Each test case also called for dependent files such as the generated C file, expected output file, the resultant output file in order for the automated test script to work
- The main branch was used to push changes that always compiled and ran as expected. Occasionally, binary files or generated files were committed or pushed, but they were added to the .gitignore and removed from the repository. Any feature that needed extended periods of time to be worked on was done on a branch (e.g. arithmetic, lists, etc.)

3. Project Timeline and Log

Task	Planned	Completed
Meeting in Lerner: Language Ideas, Graphical, Educational...	-	9/18 F
Meeting in Diana: Proposal Rough Draft	-	9/27 U
Proposal Submission	9/30 W	9/30 W
Meeting 1: Proposal Feedback	-	10/8 R
Meeting 2: Target Language?	-	10/15 R
LRM Outline and Division of Responsibilities	10/11 U	10/11 U
LRM Peer Review and Edit	10/13 T	10/13 T
Meeting 3: LRM Draft Feedback	-	10/21 R

LRM Submission	10/26 M	10/26 M
Meeting 4: LRM Feedback	-	10/29 R
Meeting 5: Parser/Scanner	-	11/5 R
Tests for parser / scanner (acceptable input / output)	11/6 F	11/6 F
Parser and Scanner MVP	11/9 M	11/9 M
Meeting 6: Compiling down to C	11/12 R	11/12 R
"Hello World"	11/15 M	11/15 M
Meeting 7: Plan for implementation of Objects	11/19 R	11/19 R
Miscellaneous functions (arithmetic, functions, loops, etc)	12/2 W	12/9 W
Meeting 8: CAST	12/10	12/10
Implementing Basic Objects	12/17 R	12/17 R
Implementing Inheritance	-	12/18 F
Implementing Lists	12/17 R	12/22
Code debugging, writing tests for demo, presentation slides	12/18 F	12/20 S
Final Presentation Demo	-	12/20 S
Final Report	-	12/22 T

4. Git Activity

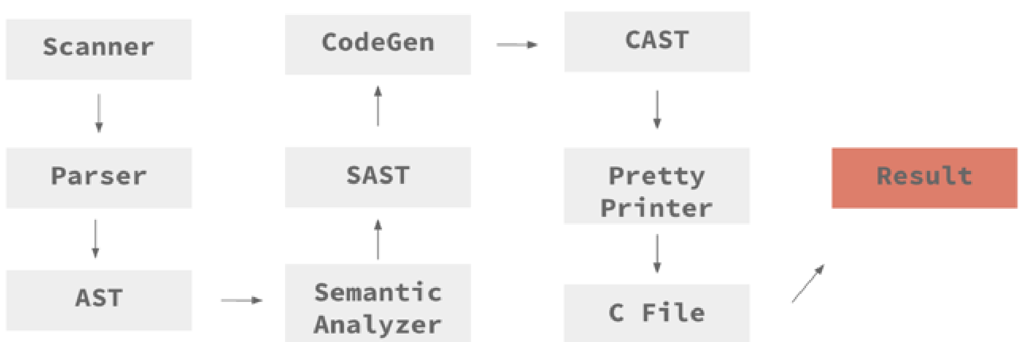


5. Development Environment

The development environment included: OCaml to write the compiler components, Bash to automate testing, and C to compile and test the generated code. For the coding environment, we used the Sublime Text Editor and Vim. Generally, Sublime was used for the compiler files (i.e. AST, SAST, semantic analyzer, pretty printer, etc.) and Vim was used to debug and look at the generated C files. The terminal was used to run the programs. Menhir was used to debug when building the scanner, parser and AST. The command `export OCAMLRUNPARAM=p` was used to debug parser errors. Github was used for version control. Final report and LRM were created using Google Docs, and the presentation slides were created using Google Presentations.

V. Architectural Design

Overview



The above diagram describes the overall architecture of our language. The scanner reads input and to generate tokens. The parser parsers through the tokens and defines the grammar rules of the language. The AST builds a syntactic tree that represents basic types in the language using the items from the parser. The semantic analyzer goes through the AST nodes, checking types and determining whether the semantics of the program input are correct, converting the AST nodes to SAST nodes as it walks through the program structure. The generated SAST nodes are used by the codegen to generate the syntax tree for C (CAST). The key job of the CAST was to decouple the `Actions` and traits in a `Character` object and represent them as virtual tables with function pointers and structs holding different typed fields including a pointer to the structs virtual table. This is used by the pretty printer to finally generate a C file that could be compiled and produce a result.

Anna and Nina started the base code for the framework of how the semantic analyzer would be when working on the "Hello World" program and demo. After that MVP, the project was divided into Language Features amongst group members and were supposed to be done by the set deadline.

After this, members made to the appropriate files so that when their respective features were implemented, the dependent files were updated to accommodate them.

Language Proposal: Anna, Beth, Nina, Pratishta
LRM write-up: Beth, Nina, Anna, Pratishta
Scanner, Parser, AST: Anna, Beth, Nina, Pratishta
Static Semantic Analysis, SAST: Anna, Beth, Nina, Pratishta
CAST, Pretty Printer: Beth, Anna, Nina, Pratishta
Code Generator: Anna, Pratishta, Beth
Test cases: Nina, Pratishta, Anna, Beth
Testing automation: Anna, Nina, Pratishta
Final writeup: Pratishta, Nina, Beth
Powerpoint slides: Pratishta, Anna, Nina, Beth

VI. Testing Plan

1. Testing Structure

A shell script `test.sh` was used to automate the regression testing process. This script runs through all the StoryBook files in the `test` folder. All unit test files ended in either `_Accept.sbk` or `_Reject.sbk` to indicate whether that particular program in StoryBook should throw a compiler error or pass. Additionally, for each acceptable unit test, `_Out.txt` and `_Exp.txt` files were created to be passed into the main testing shell script and `diff` them. In this process a C file would also be generated but because the generated C file had a lot more whitespace, syntax, and other miscellaneous text dictated by the StoryBook compiler, we didn't do a diff between the generated and expected C code.
Each member made test cases for the feature they worked on and generally it was a test-driven development process, where we tried to write tests first, then implement the features such that "accept" cases were successfully passed and "reject" tests fail for the right reasons.
As the project grew, implementing more complex features (e.g. objects) sometimes caused other features to break (e.g. assignment, function parameters, etc). This called for testing with a slightly more complex StoryBook program to ensure that all the parts could come together and not simply just work individually.

2. Automated Test Suite

Calling `./test.sh` in the `test` directory starts the automated test suite.

`test.sh`

```
#!/bin/sh

cd ../
make clean
make

cd test
echo "Accept Tests:" >> test_results.txt
failcount=0
passcount=0
if ls $1*_Accept.sbk 1> /dev/null 2>&1
then
    for acceptname in $1*_Accept.sbk;do
        program=`basename $acceptname _Accept.sbk`
        echo "Test: $program" >> errors.txt
        ../../run < "$acceptname" > "${program}.c" 2>> errors.txt
        if [ -s "$program.c" ]
        then
        gcc -g -std=c99 $program.c -o $program
        if [ -f "$program" ]
        then
            ./$program > "${program}_Out.txt"
            rm $program
            if  diff -q "${program}_Out.txt" "${program}_Exp.txt"
            then
            let "passcount += 1"
            echo ": $program" >> test_results.txt;
            else
            let "failcount += 1"
            echo ": $program -- Compiled and ran, but wrong output." >> test_results.txt
            echo ": $program -- Compiled and ran, but wrong output."
            fi
        else
            let "failcount += 1"
            echo ": $program -- C Code wouldn't compile" >> test_results.txt;
            echo ": $program"
        fi
        else
        let "failcount += 1"
        echo ": $program -- Storybook didn't compile" >> test_results.txt;
        echo ": $program -- Storybook didn't compile"
        fi
    done
fi

if ls $1*_Reject.sbk 1> /dev/null 2>&1
then
    for rejectname in $1*_Reject.sbk;do
        program=`basename $rejectname _Reject.sbk`
        echo "Test: $program" >> errors.txt
        ../../run < "$rejectname" > "${program}.c" 2>> errors.txt
        if [ ! -s "$program.c" ]
        then
        let "passcount += 1"
        echo ": $program" >> test_results.txt
        else
        let "failcount += 1"
        echo ": $program -- Storybook compiled but should not have" >> test_results.txt
        echo ": $program -- Storybook compiled but should not have"
        fi
```

```
    done
fi

echo "$passcount tests passed"
echo "$failcount tests failed"
rm -rf *.dSYM
```

./storybook

```
#!/bin/sh

# Compile c file and run
program=`basename $1 .sbk`
../run < $1 > "${program}.c"

if [ -s "$program.c" ]
then
  gcc -g -std=c99 $program.c -o $program
  if [ -f "$program" ]
        then
    ./$program
    rm -rf *.dSYM
  else
        echo "C code didn't compile"
  fi

else
        echo "Storybook didn't compile"
echo "  /\_/\\ !! _
=( °∩° )= //
  )   (  //
 (__ __)//"
fi
```

3. Sample Tests

3. 1 Return String

FncOneArg_Accept.sbk

```
Chapter whatTimeIsIt(words x) returns words {
  endwith("It's " + x + " o'clock." ).
}

Chapter plot() returns nothing {
  say(whatTimeIsIt("now" + " five")).
}
```

FncOneArg_Accept.c

```
char * whatTimeIsIt(char * x) {
    char buf__1[ strlen("It's ") + strlen(x) + 1];
    sprintf(buf__1, "%s","It's ");
    sprintf(buf__1 + strlen(buf__1), "%s",x);
    char *_1 = buf__1;char buf__2[ strlen(_1) +
    strlen(" o'clock.") + 1];
    sprintf(buf__2, "%s",_1);
    sprintf(buf__2 + strlen(buf__2), "%s"," o'clock.");
    char *_2 = buf__2;
    char *_3 = malloc(strlen(_2));
    strcpy(_3, _2);
    return _3;
}

int main() {
    char buf__4[ strlen("now") + strlen(" five") + 1];
    sprintf(buf__4, "%s","now");
    sprintf(buf__4 + strlen(buf__4), "%s"," five");
    char *_4 = buf__4;char *_5 = whatTimeIsIt (_4 );
    char _6[strlen(_5)];
    strcpy(_6, _5);
    free(_5);
    printf ( "%s\n", _6);
}
```

```
./storybook FncOneArg_Accept.c
```

Output in the console:

```
It's now five o'clock.
```

3.2 Princesses Audition

PrincessesAudition_Accept.sbk

```
Character Princess( words n; number a; tof f) {
  words name is n.
  number age is a.
  tof famous is f.

  Action introduceSelf() returns nothing {
       say( my name + ": Hi, my name is " + my name + "!").
  }

  Action audition(words part; words experience; words movie) returns nothing {
   if(my famous = true) {
       say(my name + ": I am auditioning for the part of " + part + " in " + movie +
".").
       say("In case you didn't recognize me, I was in Disney's " + experience + ".").
       }
       else {
       say(my name + ": I'm auditioning for the part of " + part + " in " + movie +
".").
       say("I don't have any experience, but I think I have great potential! Plus,
all of these old princesses only know how to play roles that depend on men. I can be
a strong, independent, and fearless princess!!").
       }
  }
}

Character DisneyPrincess is Princess( words m ) {
       words movie is m.
  Action salary(number b) returns number {
       number incSal is 2 * b.
       say(my name + ": Just so you know, Walt payed me " + b + " dollars so I expect
at least " + incSal).
       endwith(b).
  }
}

Chapter findActress(tof f; number s) returns nothing {
       if(f = true and s < 10000){
       say("Producers: You're hired!").
       }
       else if(f = false) {
       say("Producers: You're hired! And we'll pay you " + s * 2 + " dollars!").
       }
       else{
       say("Producers: No thanks.").
       }

}
```

```
Chapter plot() returns nothing {

  Character DisneyPrincess Aurora is new DisneyPrincess( "Aurora"; 16; true;
"Sleeping Beauty" ).
  Character Princess Anna is new Princess( "Anna"; 16; false).
  Aurora, introduceSelf().
  Aurora, audition("Elsa"; Aurora's movie; "Frozen").
  number money is Aurora, salary(10000000).
  findActress(Aurora's famous; money).
  Anna, introduceSelf().
  Anna, audition("Anna"; "No exprience"; "Frozen").
  findActress(Anna's famous; 5000).
}
```

PrincessesAudition_Accept.c

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>

void *ptrs[2];

struct Princess;

struct table_Princess {
  void(*audition)(char * part, char * experience, char * movie, struct Princess *_1);
  void(*introduceSelf)(struct Princess *_2);

};

struct Princess{
  const struct table_Princess *vtable;
  bool famous;
  float age;
  char * name;

};

void Princess_audition(char * part, char * experience, char * movie, struct Princess*_3) {

  if(_3 -> famous == 1) {

    char buf__4[ strlen(_3 -> name) + strlen(": I am auditioning for the part of ") + 1];
    sprintf(buf__4, "%s",_3 -> name);
    sprintf(buf__4 + strlen(buf__4), "%s",": I am auditioning for the part of ");
    char *_4 = buf__4;char buf__5[ strlen(_4) + strlen(part) + 1];
    sprintf(buf__5, "%s",_4);
    sprintf(buf__5 + strlen(buf__5), "%s",part);
    char *_5 = buf__5;char buf__6[ strlen(_5) + strlen(" in ") + 1];
    sprintf(buf__6, "%s",_5);
    sprintf(buf__6 + strlen(buf__6), "%s"," in ");
    char *_6 = buf__6;char buf__7[ strlen(_6) + strlen(movie) + 1];
    sprintf(buf__7, "%s",_6);
    sprintf(buf__7 + strlen(buf__7), "%s",movie);
    char *_7 = buf__7;char buf__8[ strlen(_7) + strlen(".") + 1];
    sprintf(buf__8, "%s",_7);
```

```
    sprintf(buf__8 + strlen(buf__8), "%s",".");
    char *_8 = buf__8;
    printf ("%s\n",_8);

    char buf__9[ strlen("In case you didn't recognize me, I was in Disney's ") +
strlen(experience) + 1];
    sprintf(buf__9, "%s","In case you didn't recognize me, I was in Disney's ");
    sprintf(buf__9 + strlen(buf__9), "%s",experience);
    char *_9 = buf__9;char buf__10[ strlen(_9) + strlen(".") + 1];
    sprintf(buf__10, "%s",_9);
    sprintf(buf__10 + strlen(buf__10), "%s",".");
    char *_10 = buf__10;
    printf ("%s\n",_10);

  }
  else {    ;
    char buf__11[ strlen(_3 -> name) + strlen(": I'm auditioning for the part of ") + 1];
    sprintf(buf__11, "%s",_3 -> name);
    sprintf(buf__11 + strlen(buf__11), "%s",": I'm auditioning for the part of ");
    char *_11 = buf__11;char buf__12[ strlen(_11) + strlen(part) + 1];
    sprintf(buf__12, "%s",_11);
    sprintf(buf__12 + strlen(buf__12), "%s",part);
    char *_12 = buf__12;char buf__13[ strlen(_12) + strlen(" in ") + 1];
    sprintf(buf__13, "%s",_12);
    sprintf(buf__13 + strlen(buf__13), "%s"," in ");
    char *_13 = buf__13;char buf__14[ strlen(_13) + strlen(movie) + 1];
    sprintf(buf__14, "%s",_13);
    sprintf(buf__14 + strlen(buf__14), "%s",movie);
    char *_14 = buf__14;char buf__15[ strlen(_14) + strlen(".") + 1];
    sprintf(buf__15, "%s",_14);
    sprintf(buf__15 + strlen(buf__15), "%s",".");
    char *_15 = buf__15;
    printf ("%s\n",_15);

    printf ( "I don't have any experience, but I think I have great potential! Plus, all of
these old princesses only know how to play roles that depend on men. I can be a strong,
independent, and fearless princess!!\n");
  }

}
void Princess_introduceSelf(struct Princess*_16) {
  ;
  char buf__17[ strlen(_16 -> name) + strlen(": Hi, my name is ") + 1];
  sprintf(buf__17, "%s",_16 -> name);
  sprintf(buf__17 + strlen(buf__17), "%s",": Hi, my name is ");
  char *_17 = buf__17;char buf__18[ strlen(_17) + strlen(_16 -> name) + 1];
  sprintf(buf__18, "%s",_17);
  sprintf(buf__18 + strlen(buf__18), "%s",_16 -> name);
  char *_18 = buf__18;char buf__19[ strlen(_18) + strlen("!") + 1];
  sprintf(buf__19, "%s",_18);
  sprintf(buf__19 + strlen(buf__19), "%s","!");
  char *_19 = buf__19;
  printf ("%s\n",_19);

}

static const struct table_Princess vtable_for_Princess = {
  Princess_audition, Princess_introduceSelf};
  struct DisneyPrincess;
  struct table_DisneyPrincess {
    float(*salary)(float b, struct DisneyPrincess *_20);
```

```
    void(*DisneyPrincess_audition)(char * part, char * experience, char * movie, struct
DisneyPrincess *_21);
    void(*DisneyPrincess_introduceSelf)(struct DisneyPrincess *_22);

  };

  struct DisneyPrincess{
    const struct table_DisneyPrincess *vtable;
    char * movie;
    bool famous;
    float age;
    char * name;
  };

  float DisneyPrincess_salary(float b, struct DisneyPrincess*_23) {
    float incSal = 2.* b;

    char buf__24[ strlen(_23 -> name) + strlen(": Just so you know, Walt payed me ") + 1];
    sprintf(buf__24, "%s",_23 -> name);
    sprintf(buf__24 + strlen(buf__24), "%s",": Just so you know, Walt payed me ");
    char *_24 = buf__24;char buf__25[ strlen(_24) + 5000 + 1];
    sprintf(buf__25, "%s",_24);
    sprintf(buf__25 + strlen(buf__25), "%g", b);
    char *_25 = buf__25;char buf__26[ strlen(_25) + strlen(" dollars so I expect at least ") +
1];
    sprintf(buf__26, "%s",_25);
    sprintf(buf__26 + strlen(buf__26), "%s"," dollars so I expect at least ");
    char *_26 = buf__26;char buf__27[ strlen(_26) + 5000 + 1];
    sprintf(buf__27, "%s",_26);
    sprintf(buf__27 + strlen(buf__27), "%g", incSal);
    char *_27 = buf__27;
    printf ("%s\n",_27);

    return b;

  }

  void DisneyPrincess_DisneyPrincess_audition(char * part, char * experience, char * movie,
struct DisneyPrincess*_28) {

    if(_28 -> famous == 1) {
      char buf__29[ strlen(_28 -> name) + strlen(": I am auditioning for the part of ") + 1];
      sprintf(buf__29, "%s",_28 -> name);
      sprintf(buf__29 + strlen(buf__29), "%s",": I am auditioning for the part of ");
      char *_29 = buf__29;char buf__30[ strlen(_29) + strlen(part) + 1];
      sprintf(buf__30, "%s",_29);
      sprintf(buf__30 + strlen(buf__30), "%s",part);
      char *_30 = buf__30;char buf__31[ strlen(_30) + strlen(" in ") + 1];
      sprintf(buf__31, "%s",_30);
      sprintf(buf__31 + strlen(buf__31), "%s"," in ");
      char *_31 = buf__31;char buf__32[ strlen(_31) + strlen(movie) + 1];
      sprintf(buf__32, "%s",_31);
      sprintf(buf__32 + strlen(buf__32), "%s",movie);
      char *_32 = buf__32;char buf__33[ strlen(_32) + strlen(".") + 1];
      sprintf(buf__33, "%s",_32);
      sprintf(buf__33 + strlen(buf__33), "%s",".");
      char *_33 = buf__33;
      printf ("%s\n",_33);
      char buf__34[ strlen("In case you didn't recognize me, I was in Disney's ") +
strlen(experience) + 1];
      sprintf(buf__34, "%s","In case you didn't recognize me, I was in Disney's ");
```

```
      sprintf(buf__34 + strlen(buf__34), "%s",experience);
      char *_34 = buf__34;char buf__35[ strlen(_34) + strlen(".") + 1];
      sprintf(buf__35, "%s",_34);
      sprintf(buf__35 + strlen(buf__35), "%s",".");
      char *_35 = buf__35;
      printf ("%s\n",_35);

    }
    else {
      char buf__36[ strlen(_28 -> name) + strlen(": I'm auditioning for the part of ") + 1];
      sprintf(buf__36, "%s",_28 -> name);
      sprintf(buf__36 + strlen(buf__36), "%s",": I'm auditioning for the part of ");
      char *_36 = buf__36;char buf__37[ strlen(_36) + strlen(part) + 1];
      sprintf(buf__37, "%s",_36);
      sprintf(buf__37 + strlen(buf__37), "%s",part);
      char *_37 = buf__37;char buf__38[ strlen(_37) + strlen(" in ") + 1];
      sprintf(buf__38, "%s",_37);
      sprintf(buf__38 + strlen(buf__38), "%s"," in ");
      char *_38 = buf__38;char buf__39[ strlen(_38) + strlen(movie) + 1];
      sprintf(buf__39, "%s",_38);
      sprintf(buf__39 + strlen(buf__39), "%s",movie);
      char *_39 = buf__39;char buf__40[ strlen(_39) + strlen(".") + 1];
      sprintf(buf__40, "%s",_39);
      sprintf(buf__40 + strlen(buf__40), "%s",".");
      char *_40 = buf__40;
      printf ("%s\n",_40);
      printf ( "I don't have any experience, but I think I have great potential! Plus, all of
these old princesses only know how to play roles that depend on men. I can be a strong,
independent, and fearless princess!!\n");
    }

  }

  void DisneyPrincess_DisneyPrincess_introduceSelf(struct DisneyPrincess*_41) {

    char buf__42[ strlen(_41 -> name) + strlen(": Hi, my name is ") + 1];
    sprintf(buf__42, "%s",_41 -> name);
    sprintf(buf__42 + strlen(buf__42), "%s",": Hi, my name is ");
    char *_42 = buf__42;char buf__43[ strlen(_42) + strlen(_41 -> name) + 1];
    sprintf(buf__43, "%s",_42);
    sprintf(buf__43 + strlen(buf__43), "%s",_41 -> name);
    char *_43 = buf__43;char buf__44[ strlen(_43) + strlen("!") + 1];
    sprintf(buf__44, "%s",_43);
    sprintf(buf__44 + strlen(buf__44), "%s","!");
    char *_44 = buf__44;
    printf ("%s\n",_44);

  }

  static const struct table_DisneyPrincess vtable_for_DisneyPrincess = {
    DisneyPrincess_salary, DisneyPrincess_DisneyPrincess_audition,
DisneyPrincess_DisneyPrincess_introduceSelf};
    void findActress(bool f, float s) {

      if(f == 1 && s < 10000.) {

        printf ( "You're hired!\n");

      }
      else {
        if(f == 0) {
```

```
        char buf__45[ strlen("Producers: You're hired! And we'll pay you ") + 5000 + 1];
        sprintf(buf__45, "%s","Producers: You're hired! And we'll pay you ");
        sprintf(buf__45 + strlen(buf__45), "%g", s* 2.);
        char *_45 = buf__45;char buf__46[ strlen(_45) + strlen(" dollars!") + 1];
        sprintf(buf__46, "%s",_45);
        sprintf(buf__46 + strlen(buf__46), "%s"," dollars!");
        char *_46 = buf__46;
        printf ("%s\n",_46);

      }
      else {}
    }

  }

  int main() {
    ptrs[0] = malloc((int)sizeof(struct DisneyPrincess ));
    ((struct DisneyPrincess *)ptrs[0])  -> name = "Aurora";
    ((struct DisneyPrincess *)ptrs[0])  -> age = 16.;
    ((struct DisneyPrincess *)ptrs[0])  -> famous = 1;
    ((struct DisneyPrincess *)ptrs[0])  -> movie = "Sleeping Beauty";
    ((struct DisneyPrincess *)ptrs[0])  ->vtable = &vtable_for_DisneyPrincess;
    struct DisneyPrincess * Aurora = ptrs[0];
    ptrs[1] = malloc((int)sizeof(struct Princess ));
    ((struct Princess *)ptrs[1])  -> name = "Anna";
    ((struct Princess *)ptrs[1])  -> age = 16.;
    ((struct Princess *)ptrs[1])  -> famous = 0;
    ((struct Princess *)ptrs[1])  ->vtable = &vtable_for_Princess;
    struct Princess * Anna = ptrs[1];
    Aurora->vtable->DisneyPrincess_introduceSelf  (Aurora );
    Aurora->vtable->DisneyPrincess_audition  ("Elsa", Aurora -> movie, "Frozen", Aurora );
    float money =      Aurora->vtable->salary  (10000000., Aurora );
    findActress  (Aurora -> famous, money );
    Anna->vtable->introduceSelf  (Anna );
    Anna->vtable->audition  ("Anna", "No exprience", "Frozen", Anna );
    findActress  (Anna -> famous, 1000. );
  }
```

```
./storybook PrincessesAudition_accept.sbk
```

Output in the console:

```
Aurora: Hi, my name is Aurora!
Aurora: I am auditioning for the part of Elsa in Frozen.
In case you didn't recognize me, I was in Disney's Sleeping Beauty.
Aurora: Just so you know, Walt payed me 1e+07 dollars so I expect at least 2e+07
Producers: No thanks.
Anna: Hi, my name is Anna!
Anna: I'm auditioning for the part of Anna in Frozen.
I don't have any experience, but I think I have great potential! Plus, all of these
old princesses only know how to play roles that depend on men. I can be a strong,
independent, and fearless princess!!
Producers: You're hired! And we'll pay you 10000 dollars!
```

3.3 Character List Loop

CharacterListLoop_Accept.sbk

```
Character Hero(words n; number st; words sp){
   words name is n.
   number strength is st.
   words superpower is sp.

   Action introduceYourself() returns nothing{
       say(my name + ": Hi there! My name is " + my name + " and I have " + my
superpower + "! Nice to meet you guys.").
   }

}

Chapter plot() returns nothing {
       characterlist heroes is new characterlist[5].
       heroes[0] is new Hero("Wonder Woman"; 2000; "the power of flight").
       heroes[1] is new Hero("Spider-Man"; 1500; "Spidey powers").
       heroes[2] is new Hero("Superman"; 100000; "the power of flight and super
strength").
       heroes[3] is new Hero("Invisible Woman"; 200; "the power of invisibility").
       heroes[4] is new Hero("The Flash"; 500; "the power of speed").
       repeatfor(number i is (0).; i < 5; i is i + 1){
              Character Hero h is heroes[i].
              h, introduceYourself().
       }
       say("Narrator: And then all the superheroes joined together to save the
world.").
       say("THE END.").
}
```

CharacterListLoop_Accept.c

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>
void *ptrs[5];
struct Hero;
struct table_Hero {
  void(*introduceYourself)(struct Hero *_1);
};
struct Hero{
  const struct table_Hero *vtable;
  char * superpower;
  float strength;
  char * name;

};
void Hero_introduceYourself(struct Hero*_2) {
  char buf__3[ strlen(_2 -> name) + strlen(": Hi there! My name is ") + 1];
```

```
  sprintf(buf__3, "%s",_2 -> name);
  sprintf(buf__3 + strlen(buf__3), "%s",": Hi there! My name is ");
  char *_3 = buf__3;char buf__4[ strlen(_3) + strlen(_2 -> name) + 1];
  sprintf(buf__4, "%s",_3);
  sprintf(buf__4 + strlen(buf__4), "%s",_2 -> name);
  char *_4 = buf__4;char buf__5[ strlen(_4) + strlen(" and I have ") + 1];
  sprintf(buf__5, "%s",_4);
  sprintf(buf__5 + strlen(buf__5), "%s"," and I have ");
  char *_5 = buf__5;char buf__6[ strlen(_5) + strlen(_2 -> superpower) + 1];
  sprintf(buf__6, "%s",_5);
  sprintf(buf__6 + strlen(buf__6), "%s",_2 -> superpower);
  char *_6 = buf__6;char buf__7[ strlen(_6) + strlen("! Nice to meet you guys.") + 1];
  sprintf(buf__7, "%s",_6);
  sprintf(buf__7 + strlen(buf__7), "%s","! Nice to meet you guys.");
  char *_7 = buf__7;
  printf ("%s\n",_7);

}
static const struct table_Hero vtable_for_Hero = {
  Hero_introduceYourself};
  int main() {
    void ** heroes = malloc(5 * sizeof(void *));
    ptrs[0] = malloc((int)sizeof(struct Hero ));
    ((struct Hero *)ptrs[0])  -> name = "Wonder Woman";
    ((struct Hero *)ptrs[0])  -> strength = 2000.;
    ((struct Hero *)ptrs[0])  -> superpower = "the power of flight";
    ((struct Hero *)ptrs[0])  ->vtable = &vtable_for_Hero;
    heroes[(int)0.] = ptrs[0];
    ptrs[1] = malloc((int)sizeof(struct Hero ));
    ((struct Hero *)ptrs[1])  -> name = "Spider-Man";
    ((struct Hero *)ptrs[1])  -> strength = 1500.;
    ((struct Hero *)ptrs[1])  -> superpower = "Spidey powers";
    ((struct Hero *)ptrs[1])  ->vtable = &vtable_for_Hero;
    heroes[(int)1.] = ptrs[1];
    ptrs[2] = malloc((int)sizeof(struct Hero ));
    ((struct Hero *)ptrs[2])  -> name = "Superman";
    ((struct Hero *)ptrs[2])  -> strength = 100000.;
    ((struct Hero *)ptrs[2])  -> superpower = "the power of flight and super strength";
    ((struct Hero *)ptrs[2])  ->vtable = &vtable_for_Hero;
    heroes[(int)2.] = ptrs[2];
    ptrs[3] = malloc((int)sizeof(struct Hero ));
    ((struct Hero *)ptrs[3])  -> name = "Invisible Woman";
    ((struct Hero *)ptrs[3])  -> strength = 200.;
    ((struct Hero *)ptrs[3])  -> superpower = "the power of invisibility";
    ((struct Hero *)ptrs[3])  ->vtable = &vtable_for_Hero;
    heroes[(int)3.] = ptrs[3];
    ptrs[4] = malloc((int)sizeof(struct Hero ));
    ((struct Hero *)ptrs[4])  -> name = "The Flash";
    ((struct Hero *)ptrs[4])  -> strength = 500.;
    ((struct Hero *)ptrs[4])  -> superpower = "the power of speed";
    ((struct Hero *)ptrs[4])  ->vtable = &vtable_for_Hero;
    heroes[(int)4.] = ptrs[4];
    float i = 0.;
    while(i < 5.){
      struct Hero * h = heroes[(int)i];
      h->vtable->introduceYourself  (h);
      i = i + 1.;
      }
    printf ( "Narrator: And then all the superheroes joined together to save the world.\n");
    printf ( "THE END.\n");
  }
```

```
./storybook CharacterListLoop_Accept.sbk
```

Output in the console:

```
Wonder Woman: Hi there! My name is Wonder Woman and I have the power of flight! Nice
to meet you guys.
Spider-Man: Hi there! My name is Spider-Man and I have Spidey powers! Nice to meet
you guys.
Superman: Hi there! My name is Superman and I have the power of flight and super
strength! Nice to meet you guys.
Invisible Woman: Hi there! My name is Invisible Woman and I have the power of
invisibility! Nice to meet you guys.
The Flash: Hi there! My name is The Flash and I have the power of speed! Nice to meet
you guys.
Narrator: And then all the superheroes joined together to save the world.
THE END.
```

These test cases were chosen because they embody the main aspects of what StoryBook has to offer: objects and inheritance. These two are the main parts of an object oriented programming language. And because StoryBook is targeted toward beginners and one of the first things beginners generally learn is object oriented programming, learning about objects and how they can be useful and flexible to produce robust programs is important. The specific objects and the features they use is also important in that it provides a narrative that the user can follow along and know what to expect from. The built in list object also makes it much easier for beginners to learn the concept of iteration, optimization, and code design. This way StoryBook not only can provide a start base for novices but also a path for them to take on more challenging computer science concepts.

1. Overview

2. Test Suite File Listings

```
_99BottlesOfBeer_Accept.sbk
_99BottlesOfBeer_Exp.txt
AssnBoolF_Accept.sbk
AssnBoolF_Exp.txt
AssnBoolT_Accept.sbk
AssnBoolT_Exp.txt
AssnChar_Accept.sbk
AssnChar_Exp.txt
AssnExpr_Accept.sbk
AssnExpr_Exp.txt
AssnNmbr_Accept.sbk
```

```
AssnNmbr_Exp.txt
AssnNum_Reject.sbk
AssnStr_Accept.sbk
AssnStr_Exp.txt
AssnStr_Reject.sbk
AssnTwice_Reject.sbk
boolListTest_Accept.sbk
boolListTest_Exp.txt
c_exp
CharacterListLoop_Accept
CharacterListLoop_Accept.c
CharacterListLoop_Accept.sbk
CharacterListLoop_Exp.txt
CharacterListTest_Accept.sbk
CharacterListTest_Exp.txt
CharImproperParams_Accept.sbk
CharImproperParams_Exp.txt
charListTest_Accept.sbk
charListTest_Exp.txt
charListTestExp.txt
CommentMultiline_Accept.sbk
CommentMultiline_Exp.txt
CommentNested_Accept.sbk
CommentNested_Exp.txt
CommentNested_Reject.sbk
CommentNoEnd_Reject.sbk
CommentSingle_Accept.sbk
CommentSingle_Exp.txt
CompareBool_Accept.sbk
CompareBool_Exp.txt
CompareBool_Reject.sbk
CompareChar_Reject.sbk
CompareEqChars_Accept.sbk
CompareEqChars_Exp.txt
CompareEqNums2_Accept.sbk
CompareEqNums2_Exp.txt
CompareEqNums_Accept.sbk
CompareEqNums_Exp.txt
CompareEqNumString2_Reject.sbk
CompareEqNumString_Reject.sbk
CompareEqString_Accept.sbk
CompareEqString_Exp.txt
CompareGreatEqual1_Accept.sbk
CompareGreatEqual1_Exp.txt
CompareGreatEqual2_Accept.sbk
CompareGreatEqual2_Exp.txt
CompareGreatEqual3_Accept.sbk
CompareGreatEqual3_Exp.txt
```

```
CompareGreaterFalse_Accept.sbk
CompareGreaterFalse_Exp.txt
CompareGreaterTrue_Accept.sbk
CompareGreaterTrue_Exp.txt
CompareLessEqual1_Accept.sbk
CompareLessEqual1_Exp.txt
CompareLessEqual2_Accept.sbk
CompareLessEqual2_Exp.txt
CompareLessEqual3_Accept.sbk
CompareLessEqual3_Exp.txt
CompareLessFalse_Accept.sbk
CompareLessFalse_Exp.txt
CompareLessTrue_Accept.sbk
CompareLessTrue_Exp.txt
CompareString_Reject.sbk
ConcatBooleanandChar_Reject.sbk
ConcatBooleanAndString_Accept.sbk
ConcatBooleanAndString_Exp.txt
ConcatNumberAndBoolean_Reject.sbk
ConcatNumberandChar_Reject.sbk
ConcatNumberAndString1_Accept.sbk
ConcatNumberAndString1_Exp.txt
ConcatNumberAndString2_Accept.sbk
ConcatNumberAndString2_Exp.txt
ConcatNumberAndString_Accept.sbk
ConcatNumberAndString_Exp.txt
ConcatStringandBooleanExpr_Accept.sbk
ConcatStringandBooleanExpr_Exp.txt
ConcatStringandChar_Accept.sbk
ConcatStringandChar_Exp.txt
ConcatStringandNumberExpr1_Accept.sbk
ConcatStringandNumberExpr1_Exp.txt
ConcatStringandNumberExpr2_Accept.sbk
ConcatStringandNumberExpr2_Exp.txt
ConcatStringandNumberExpr3_Accept.sbk
ConcatStringandNumberExpr3_Exp.txt
ConcatStringandString_Accept.sbk
ConcatStringandString_Exp.txt
ConcatStringNumberExprandBoolean_Accept.sbk
ConcatStringNumberExprandBoolean_Exp.txt
FncArgMissingID_Reject.sbk
FncConcatArg_Accept.sbk
FncConcatArg_Exp.txt
FncDeclSay_Reject.sbk
FncHasArgs_Accept.sbk
FncHasArgs_Exp.txt
FncHasArgs_Reject.sbk
FncInvalidParamTypes_Reject.sbk
```

```
FncNoArgs_Accept.sbk
FncNoArgs_Exp.txt
FncNoArgs_Reject.sbk
FncNoPlot_Reject.sbk
FncNoReturnInDecl_Reject.sbk
FncOneArg_Accept.sbk
FncOneArg_Exp.txt
FncTakingCharacterParam_Accept.sbk
FncTakingCharacterParam_Exp.txt
FncTooFewArgs_Reject.sbk
FncTooManyArgs_Reject.sbk
FncTwoSameName_Reject.sbk
FncUndefined_Reject.sbk
FncWrongTypeArg_Reject.sbk
ForLoop_Accept.sbk
ForLoop_Exp.txt
_GCD_Accept.sbk
_GCD_Exp.txt
_HelloWorld_Accept.sbk
_HelloWorld_Exp.txt
IfElse_Accept.sbk
IfElse_Exp.txt
IfElseIfElse_Accept.sbk
IfElseIfElse_Exp.txt
IfElseSimple_Accept.sbk
IfElseSimple_Exp.txt
IfNestedIfIfElse_Accept.sbk
IfNestedIfIfElse_Exp.txt
IfNoElse_Accept.sbk
IfNoElse_Exp.txt
IfSimple_Accept.sbk
IfSimple_Exp.txt
listAccessChar_Reject.sbk
ListAccess_Reject.sbk
listTestInstant_Accept
listWrongType_Reject.sbk
LogicalAnd2_Accept.sbk
LogicalAnd2_Exp.txt
LogicalAnd3_Accept.sbk
LogicalAnd3_Exp.txt
LogicalAnd4_Accept.sbk
LogicalAnd4_Exp.txt
LogicalAnd_Accept.sbk
LogicalAndBoolExpr_Accept.sbk
LogicalAndBoolExpr_Exp.txt
LogicalAndChain2_Accept.sbk
LogicalAndChain2_Exp.txt
LogicalAndChain_Accept.sbk
```

```
LogicalAndChain_Exp.txt
LogicalAnd_Exp.txt
LogicalAndNum_Reject.sbk
LogicalAndOrChain2_Accept.sbk
LogicalAndOrChain2_Exp.txt
LogicalAndOrChain_Accept.sbk
LogicalAndOrChain_Exp.txt
LogicalOr2_Accept.sbk
LogicalOr2_Exp.txt
LogicalOr3_Accept.sbk
LogicalOr3_Exp.txt
LogicalOr4_Accept.sbk
LogicalOr4_Exp.txt
LogicalOr_Accept.sbk
LogicalOrBoolExpr_Accept.sbk
LogicalOrBoolExpr_Exp.txt
LogicalOrChain_Accept.sbk
LogicalOrChain_Exp.txt
LogicalOrDiffTypes_Reject.sbk
LogicalOr_Exp.txt
LogicalOrStringChar_Reject.sbk
MathAdd_Accept.sbk
MathAdd_Exp.txt
MathDivide_Accept.sbk
MathDivide_Exp.txt
MathMod2_Accept.sbk
MathMod2_Exp.txt
MathMod_Accept.sbk
MathMod_Exp.txt
MathMultiply_Accept.sbk
MathMultiply_Exp.txt
MathSubtract_Accept.sbk
MathSubtract_Exp.txt
memtest.sh
NoReturn_Reject.sbk
NotEq2_Accept.sbk
NotEq2_Exp.txt
NotEq_Accept.sbk
NotEqDifTypes_Reject.sbk
NotEq_Exp.txt
NotGreater_Accept.sbk
NotGreaterEq_Accept.sbk
NotGreaterEq_Exp.txt
NotGreater_Exp.txt
NotLess_Accept.sbk
NotLessEq_Accept.sbk
NotLessEq_Exp.txt
NotLess_Exp.txt
```

```
Not_Reject.sbk
numberListTest_Accept.sbk
numberListTest_exp.txt
ObjectActionConcatParam_Accept.sbk
ObjectActionConcatParam_Exp.txt
ObjectActionWithMyInheritedTrait_Accept.sbk
ObjectActionWithMyInheritedTrait_Exp.txt
ObjectHasActions_Accept.sbk
ObjectHasActions_Exp.txt
ObjectHasTraits_Accept.sbk
ObjectHasTraitsAndActions_Accept.sbk
ObjectHasTraitsAndActions_Exp.txt
ObjectHasTraits_Exp.txt
ObjectInheritance_Accept.sbk
ObjectInheritance_Exp.txt
ObjectInstInLoop_Accept.sbk
ObjectInstInLoop_Exp.txt
ObjectMonster_Accept.sbk
ObjectMonster_Exp.txt
ObjectOverrideFunc_Accept.sbk
ObjectOverrideFunc_Exp.txt
ObjectsMultiple_Accept.sbk
ObjectsMultiple_Exp.txt
ObjectTraitAssignment_Accept
ObjectTraitAssignment_Accept.sbk
ObjectTraitAssignment_Exp.txt
ObjectTraitWrongType_Reject.sbk
PrincessCharacterAsParam_Accept.sbk
Princesses_Accept.sbk
PrincessesAudition_Accept
PrincessesAudition_Accept.c
PrincessesAudition_Accept.sbk
PrincessesAudition_Exp.txt
PrintBool_Accept.sbk
PrintBool_Exp.txt
PrintFncRet_Accept.sbk
PrintFncRet_Exp.txt
PrintNum_Accept.sbk
PrintNum_Exp.txt
PrintVar_Accept.sbk
PrintVar_Exp.txt
ReAssnNum2_Accept.sbk
ReAssnNum2_Exp.txt
ReAssnNum_Accept.sbk
ReAssnNum_Exp.txt
ReAssnStr_Accept.sbk
ReAssnStr_Exp.txt
RecursionSimple_Accept.sbk
```

```
RecursionSimple_Exp.txt
ReturnEndswithWithoutParens_Accept.sbk
ReturnEndswithWithoutParens_Exp.txt
ReturnInvalidType_Reject.sbk
ReturnNum_Accept.sbk
ReturnNum_Exp.txt
ReturnVoid_Accept.sbk
ReturnVoid_Exp.txt
ReturnVoid_Reject.sbk
ReturnWrongStringNotNumber_Reject.sbk
ScopeSimple_Reject.sbk
ScopingObjects_Accept
ScopingObjects_Accept.sbk
ScopingObjects_Exp.txt
ScopingObjectsNoReturn_Reject.sbk
storybook
test
testfileayo
test.sh
testsuitefilelist.txt
testTree.sh
TraitInheritRightHandSide_Accept
TraitInheritRightHandSide_Accept.sbk
TraitInheritRightHandSide_Exp.txt
TraitOverride_Reject.sbk
tree
unitTests.sh
WhileLoop_Accept
WhileLoop_Accept.c
WhileLoop_Accept.sbk
WhileLoop_Exp.txt
```

2.1 Functions

```
FncArgMissingID_Reject.sbk
FncConcatArg_Accept.sbk
FncConcatArg_Exp.txt
FncDeclSay_Reject.sbk
FncHasArgs_Accept.sbk
FncHasArgs_Exp.txt
FncHasArgs_Reject.sbk
FncInvalidParamTypes_Reject.sbk
FncNoArgs_Accept.sbk
FncNoArgs_Exp.txt
FncNoArgs_Reject.sbk
FncNoPlot_Reject.sbk
FncNoReturnInDecl_Reject.sbk
FncOneArg_Accept.sbk
FncOneArg_Exp.txt
```

FncTakingCharacterParam_Accept.sbk
FncTakingCharacterParam_Exp.txt
FncTooFewArgs_Reject.sbk
FncTooManyArgs_Reject.sbk
FncTwoSameName_Reject.sbk
FncUndefined_Reject.sbk
FncWrongTypeArg_Reject.sbk

2.2 Print Statements

PrintBool_Accept.sbk
PrintBool_Exp.txt
PrintFncRet_Accept.sbk
PrintFncRet_Exp.txt
PrintNum_Accept.sbk
PrintNum_Exp.txt
PrintVar_Accept.sbk
PrintVar_Exp.txt

2.3 Comments

CommentMultiline_Accept.sbk
CommentMultiline_Exp.txt
CommentNested_Accept.sbk
CommentNested_Exp.txt
CommentNested_Reject.sbk
CommentNoEnd_Reject.sbk
CommentSingle_Accept.sbk
CommentSingle_Exp.txt

2.4 Arithmetic Operators

MathAdd_Accept.sbk
MathAdd_Exp.txt
MathDivide_Accept.sbk
MathDivide_Exp.txt
MathMod2_Accept.sbk
MathMod2_Exp.txt
MathMod_Accept.sbk
MathMod_Exp.txt
MathMultiply_Accept.sbk
MathMultiply_Exp.txt
MathSubtract_Accept.sbk
MathSubtract_Exp.txt

2.5 Concatenation

ConcatBooleanandChar_Reject.sbk
ConcatBooleanAndString_Accept.sbk
ConcatBooleanAndString_Exp.txt
ConcatNumberAndBoolean_Reject.sbk
ConcatNumberandChar_Reject.sbk
ConcatNumberAndString1_Accept.sbk

ConcatNumberAndString1_Exp.txt
ConcatNumberAndString2_Accept.sbk
ConcatNumberAndString2_Exp.txt
ConcatNumberAndString_Accept.sbk
ConcatNumberAndString_Exp.txt
ConcatStringandBooleanExpr_Accept.sbk
ConcatStringandBooleanExpr_Exp.txt
ConcatStringandChar_Accept.sbk
ConcatStringandChar_Exp.txt
ConcatStringandNumberExpr1_Accept.sbk
ConcatStringandNumberExpr1_Exp.txt
ConcatStringandNumberExpr2_Accept.sbk
ConcatStringandNumberExpr2_Exp.txt
ConcatStringandNumberExpr3_Accept.sbk
ConcatStringandNumberExpr3_Exp.txt
ConcatStringandString_Accept.sbk
ConcatStringandString_Exp.txt
ConcatStringNumberExprandBoolean_Accept.sbk
ConcatStringNumberExprandBoolean_Exp.txt

2.6 Comparison Operators

CompareBool_Accept.sbk
CompareBool_Exp.txt
CompareBool_Reject.sbk
CompareChar_Reject.sbk
CompareEqChars_Accept.sbk
CompareEqChars_Exp.txt
CompareEqNums2_Accept.sbk
CompareEqNums2_Exp.txt
CompareEqNums_Accept.sbk
CompareEqNums_Exp.txt
CompareEqNumString2_Reject.sbk
CompareEqNumString_Reject.sbk
CompareEqString_Accept.sbk
CompareEqString_Exp.txt
CompareGreatEqual1_Accept.sbk
CompareGreatEqual1_Exp.txt
CompareGreatEqual2_Accept.sbk
CompareGreatEqual2_Exp.txt
CompareGreatEqual3_Accept.sbk
CompareGreatEqual3_Exp.txt
CompareGreaterFalse_Accept.sbk
CompareGreaterFalse_Exp.txt
CompareGreaterTrue_Accept.sbk
CompareGreaterTrue_Exp.txt
CompareLessEqual1_Accept.sbk
CompareLessEqual1_Exp.txt
CompareLessEqual2_Accept.sbk

CompareLessEqual2_Exp.txt
CompareLessEqual3_Accept.sbk
CompareLessEqual3_Exp.txt
CompareLessFalse_Accept.sbk
CompareLessFalse_Exp.txt
CompareLessTrue_Accept.sbk
CompareLessTrue_Exp.txt
CompareString_Reject.sbk

2.7 Logical Operators

LogicalAnd2_Accept.sbk
LogicalAnd2_Exp.txt
LogicalAnd3_Accept.sbk
LogicalAnd3_Exp.txt
LogicalAnd4_Accept.sbk
LogicalAnd4_Exp.txt
LogicalAnd_Accept.sbk
LogicalAndBoolExpr_Accept.sbk
LogicalAndBoolExpr_Exp.txt
LogicalAndChain2_Accept.sbk
LogicalAndChain2_Exp.txt
LogicalAndChain_Accept.sbk
LogicalAndChain_Exp.txt
LogicalAnd_Exp.txt
LogicalAndNum_Reject.sbk
LogicalAndOrChain2_Accept.sbk
LogicalAndOrChain2_Exp.txt
LogicalAndOrChain_Accept.sbk
LogicalAndOrChain_Exp.txt
LogicalOr2_Accept.sbk
LogicalOr2_Exp.txt
LogicalOr3_Accept.sbk
LogicalOr3_Exp.txt
LogicalOr4_Accept.sbk
LogicalOr4_Exp.txt
LogicalOr_Accept.sbk
LogicalOrBoolExpr_Accept.sbk
LogicalOrBoolExpr_Exp.txt
LogicalOrChain_Accept.sbk
LogicalOrChain_Exp.txt
LogicalOrDiffTypes_Reject.sbk
LogicalOr_Exp.txt
LogicalOrStringChar_Reject.sbk

2.8 Not Operator

NotEq2_Accept.sbk
NotEq2_Exp.txt
NotEq_Accept.sbk

```
NotEqDifTypes_Reject.sbk
NotEq_Exp.txt
NotGreater_Accept.sbk
NotGreaterEq_Accept.sbk
NotGreaterEq_Exp.txt
NotGreater_Exp.txt
NotLess_Accept.sbk
NotLessEq_Accept.sbk
NotLessEq_Exp.txt
NotLess_Exp.txt
Not_Reject.sbk
```

2.9 Assignment

```
AssnBoolF_Accept.sbk
AssnBoolF_Exp.txt
AssnBoolT_Accept.sbk
AssnBoolT_Exp.txt
AssnChar_Accept.sbk
AssnChar_Exp.txt
AssnExpr_Accept.sbk
AssnExpr_Exp.txt
AssnNmbr_Accept.sbk
AssnNmbr_Exp.txt
AssnNum_Reject.sbk
AssnStr_Accept.sbk
AssnStr_Exp.txt
AssnStr_Reject.sbk
AssnTwice_Reject.sbk
ReAssnNum2_Accept.sbk
ReAssnNum2_Exp.txt
ReAssnNum_Accept.sbk
ReAssnNum_Exp.txt
ReAssnStr_Accept.sbk
ReAssnStr_Exp.txt
```

2.10 For & While Loops

```
CharacterListLoop_Accept
CharacterListLoop_Accept.c
CharacterListLoop_Accept.sbk
CharacterListLoop_Exp.txt
ForLoop_Accept.sbk
ForLoop_Exp.txt
ObjectInstInLoop_Accept.sbk
ObjectInstInLoop_Exp.txt
WhileLoop_Accept
WhileLoop_Accept.c
WhileLoop_Accept.sbk
WhileLoop_Exp.txt
```

2.11 If Else Statements

IfElse_Accept.sbk
IfElse_Exp.txt
IfElseIfElse_Accept.sbk
IfElseIfElse_Exp.txt
IfElseSimple_Accept.sbk
IfElseSimple_Exp.txt
IfNestedIfIfElse_Accept.sbk
IfNestedIfIfElse_Exp.txt
IfNoElse_Accept.sbk
IfNoElse_Exp.txt
IfSimple_Accept.sbk
IfSimple_Exp.txt

2.12 Return Statements

ReturnEndswithWithoutParens_Accept.sbk
ReturnEndswithWithoutParens_Exp.txt
ReturnInvalidType_Reject.sbk
ReturnNum_Accept.sbk
ReturnNum_Exp.txt
ReturnVoid_Accept.sbk
ReturnVoid_Exp.txt
ReturnVoid_Reject.sbk
ReturnWrongStringNotNumber_Reject.sbk

2.13 Scoping

ScopeSimple_Reject.sbk
ScopingObjects_Accept
ScopingObjects_Accept.sbk
ScopingObjects_Exp.txt
ScopingObjectsNoReturn_Reject.sbk

2.14 Recursion

RecursionSimple_Accept.sbk
RecursionSimple_Exp.txt

2.15 Objects and Inheritance

ObjectActionConcatParam_Accept.sbk
ObjectActionConcatParam_Exp.txt
ObjectActionWithMyInheritedTrait_Accept.sbk
ObjectActionWithMyInheritedTrait_Exp.txt
ObjectHasActions_Accept.sbk
ObjectHasActions_Exp.txt
ObjectHasTraits_Accept.sbk
ObjectHasTraitsAndActions_Accept.sbk
ObjectHasTraitsAndActions_Exp.txt
ObjectHasTraits_Exp.txt
ObjectInheritance_Accept.sbk

```
ObjectInheritance_Exp.txt
ObjectInstInLoop_Accept.sbk
ObjectInstInLoop_Exp.txt
ObjectMonster_Accept.sbk
ObjectMonster_Exp.txt
ObjectOverrideFunc_Accept.sbk
ObjectOverrideFunc_Exp.txt
ObjectsMultiple_Accept.sbk
ObjectsMultiple_Exp.txt
ObjectTraitAssignment_Accept
ObjectTraitAssignment_Accept.sbk
ObjectTraitAssignment_Exp.txt
ObjectTraitWrongType_Reject.sbk
```

VII. Lessons Learned

1. Anna Lawson

Ocaml is great. Use pattern matching lots--it's awesome and if/elses are horrendously clunky in ocaml so get a good grasp of pattern matching before you start. Dividing by feature may not be the best because features vary extremely in difficulty. Start by doing a small feature end to end--thing become a lot easier once you've got the "hello world" running. Don't try to do too much for hello world, because a lot of it you'll have to go back and fix because you didn't know what you were doing the first time. Think ahead when you start code--and think ahead to the smallest details--is this structure going to work for all possible inputs? Set standards early for naming, indentation, etc; it makes the code a lot more readable/workable if everyone's code looks the same.

2. Beth Green

At first, Ocaml seems crazy and extremely difficult to use, but once you get the hang of it you soon realize how cool functional programming is! I've come to really appreciate pattern matching. This project has removed another black box from my programming knowledge. I have gained a very very good understanding of what happens when you compile code. Additionally, I've learned a lot about working with a team. You have to be willing to tell your teammates when they need to pick up the slack and you have to be able to recognize when you yourself need to be contributing more. If everyone is honest and hardworking you'll end up with a great team and really getting along. Additionally, it is very helpful to split up the basic features in the beginning. However, once you get into the heart of your language it is helpful to code pair. It is easy to get lost in the code and having an extra set of eyes and another person to bounce ideas off of it extremely productive.

3. Nina Baculinao

Test driven development and version control are awesome. Pair programming, especially early on, both reduce merges conflicts and bridges gaps in knowledge. Don't be afraid to ask questions or take ownership and always strive for open communication. Keep your eye on the goal and try to get the end to end flow as early as possible. Split up your task into small features to feel more manageable, think about what's the bare minimum, and try to keep your focus narrow rather than be overwhelmed by the unfinished ocean before you. I wasted valuable time writing this beautiful and efficient C file to represent virtual tables only to realize it was a rather unfeasible, so spend as much time as you can writing this quirky and fun lingo of OCaml. There will be compromises between how you had hoped to implement a thing and how it turns out, but don't be nitpicky, be hacky, and keep trying!

4. Pratishta Yerakala

One of the most important lessons is to organize so that every member of the team can work on a feasible part of the language. It's easy to fall behind or lose track of where people are, especially when there's a break in consistency (e.g. not communicating as much over Thanksgiving break or spring break, etc). Regardless of how long, there should be frequent check ins even just to make sure everyone's on the same page and can efficiently do work. Also, we should have compiled to C++ instead of C. We would have still been able to use pointers but avoid verbose generated C files that would be difficult to read and understand to debug.

VIII. Appendix

A. Full Source Code

A.1 ast.ml

```
(* Possible data types *)
type data_type =
  | Void
  | Number
  | Boolean
  | String
  | Char
  | Object of string
  | NumberList
  | BooleanList
  | CharList
  | CharacterList (* list of pointers to point to Characters *)

(* Operators *)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq| Mod|
OR | AND | NOT

(* Expressions *)
type expr =
```

```
  LitNum of float
| LitBool of bool
| LitString of string (* quoted string literal *)
| LitChar of char (* 'c' *)
| Noexpr
| Id of string (* foo_unquoted *)
| Assign of string * expr (* x is 5 *)
| TraitAssign of expr * expr (* SleepingBeauty's x is 5 *)
| Instantiate of string * expr list (* object type and constructor parameters *)
| ListInstantiate of data_type * expr (* type, size  -> e.g. int, 5 *)
| ListAccess of string * expr
| ListAssign of expr * expr (* myList[2 + 3] = 5+ 7 *)
| Access of string * string (* Member value access: SleepingBeauty's x or my x within class
itself *)
| Binop of expr * op * expr (* a + b *)
| Unop of op * expr
| FCall of string * expr list (* chapter1() *)
| ACall of string * string * expr list (* SleepingBeauty, setX(5) *)

(* Variable Declarations *)
type var_decl = {
  vtype: data_type;
  vname : string;
  vexpr : expr;
}

(* Statements *)
type stmt =
  Block of stmt list
| Expr of expr
| VarDecl of var_decl
| Return of expr
| If of expr * stmt * stmt
| For of stmt * expr * expr * stmt
| While of expr * stmt

(* Functions *)
type func_decl = {
  fname : string; (* name of the function *)
  fformals : var_decl list; (* formal params *)
  freturn : data_type; (* return type *)
  fbody : stmt list; (* statements, including local variable declarations *)
}

(* Actions *)
type act_decl = {
  mutable aname : string;  (* Name of the action *)
  aformals: var_decl list; (* formal params *)
  areturn : data_type;  (* return type *)
  abody : stmt list;    (* statements, including local variable declarations *)
}

(* Class Declarations *)
type cl_decl = {
  cname : string; (* name of the class *)
  cparent : string;
  cformals: var_decl list;   (* formal params *)
  cinstvars : var_decl list; (*instance variables *)
  cactions: act_decl list;   (*lists of actions (methods) *)
}
```

```
(* Program is class declarations and function declarations *)
(* Method declarations are contained in class declarations *)
type program = cl_decl list * func_decl list (* classes, funcs. no global vars *)
```

A.2 scanner.mll

```
{ open Parser }

let whitespace = [' ' '\t' '\r' '\n']
let comment = "~~" [^ '\n']* "\n"
let digit = ['0'-'9']

rule token = parse
  (* Whitespace and Comments *)
  whitespace { token lexbuf }
  | comment  { token lexbuf }
  | '~'      { comment lexbuf }

  (* Punctuation *)
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE }
  | '}'      { RBRACE }
  | '['      { LBRACK }
  | ']'      { RBRACK }
  | ';'      { SEMI }
  | ','      { COMMA }
  | '.'      { PERIOD }
  | "'s"     { APOST }

  (* Binary Operators *)
  | '+'      { PLUS }
  | '-'      { MINUS }
  | '*'      { TIMES }
  | '/'      { DIVIDE }
  | '%'      { MOD }
  | "<"      { LT }
  | "<="     { LEQ }
  | ">"      { GT }
  | ">="     { GEQ }
  | "="      { EQ }
  | "!="     { NEQ }
  | "is"     { ASSIGN }

  (* Logical Operators *)
  | "not"    { NOT }
  | "and"    { AND }
  | "or"     { OR }
```

```
  (* Control flow *)
  | "if"                  { IF }
  | "else"                { ELSE }
  | "repeatfor"           { FOR }
  | "repeatwhile"         { WHILE }
  | "endwith"             { ENDWITH }
  | "returns"             { RETURNS }

  (* Primitives--booleans, chars, strings, numbers *)
  | "tof"                 { BOOL }
  | "true" as bool_val    { LIT_BOOL(bool_of_string bool_val)}
  | "false" as bool_val   { LIT_BOOL(bool_of_string bool_val)}
  | "number"              { NUMBER }
  | "words"               { STRING }
  | "letter"              { CHAR }
  | "numberlist"          { NUMBERLIST }
  | "toflist"             { BOOLLIST }
  | "letterlist"          { CHARLIST }
  | "characterlist"       { CHARACTERLIST }
  | "nothing"             { VOID }
  | "Chapter"             { FUNCTION }
  | "Character"           { CHARACTER }
  | "Action"              { METHOD }
  | "trait"               { TRAIT }
  | "new"                 { NEW }
  | "my"                  { MY }
  | eof                   { EOF }
  | ['-']?(digit+'.'digit*)|['-']?(digit*'.'digit+)|['-']?(digit+) as lxm {
LIT_NUM(float_of_string lxm) }
  (* String regex modified from:
   realworldocaml.org/v1/en/html/parsing-with-ocamllex-and-menhir.html *)
  | '"'( ('\\'('/'|'\\'| 'b' | 'f' | 'n' | 'r' | 't'))|([^'"']) )*'"' as lxm {
LIT_STRING(lxm) }
  | ['\''] (_ as l) ['\''] {LIT_CHAR(l) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "~"  { token lexbuf }
| _    { comment lexbuf }
```

A.3 parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA PERIOD APOST
%token PLUS MINUS TIMES DIVIDE ASSIGN MOD
```

```
%token EQ NOT AND OR NEQ LT LEQ GT GEQ
%token ENDWITH
%token RETURNS IF ELSE FOR WHILE
%token VOID NUMBER BOOL TRUE FALSE STRING CHAR FUNCTION
%token NUMBERLIST BOOLLIST CHARLIST CHARACTERLIST
%token CHARACTER METHOD TRAIT NEW MY
%token <float> LIT_NUM
%token <bool> LIT_BOOL
%token <string> LIT_STRING
%token <char> LIT_CHAR
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NEW
%right NOT
%left COMMA APOST /* function call and member access */

%start program
%type <Ast.program> program

%%

/* Program is comprised of class declarations and function declarations */
program:
  decls EOF { $1 }

decls:
   /* nothing */ { [], [] }
 | decls cdecl { ($2 :: fst $1), snd $1 } /* class decl */
 | decls fdecl { fst $1, ($2 :: snd $1) } /* func decl */

/* Function declarations */
fdecl:
   FUNCTION ID LPAREN formals_opt RPAREN RETURNS type_label LBRACE stmt_list RBRACE
       { { fname = $2;
             fformals = $4;
       freturn = $7;
             fbody = List.rev $9 } }
```

```
formals_opt:
      /* nothing */ { [] }
  | formal_list   { List.rev $1 }

/* Formal param list. */
/* Params are represented as variable declarations with no expr for assignment */
formal_list:
      type_label ID
      { [ { vtype = $1;
            vname = $2;
            vexpr = Noexpr } ] }
  | formal_list SEMI type_label ID
      { { vtype = $3;
      vname = $4;
      vexpr = Noexpr } :: $1}

/* Data type names */
type_label:
   VOID      { Void }
 | NUMBER  { Number }
 | BOOL      { Boolean }
 | STRING  { String }
 | CHAR      { Char }
 | CHARACTER ID    { Object($2) }
 | NUMBERLIST { NumberList }
 | BOOLLIST { BooleanList }
 /*| STRING LIST { List(String)}*/
 | CHARLIST { CharList }
 | CHARACTERLIST { CharacterList}

/* Variable Declarations */
vdecl_list:
  /* nothing */ { [] }
  | vdecl_list vdecl { $2 :: $1}

vdecl:
      /* Uninitialized regular variable */
      type_label ID PERIOD
      { { vtype=$1;
      vname=$2;
      vexpr = Noexpr } }
  /* Uninitialized instance variable */
  | type_label TRAIT ID PERIOD
      { { vtype = $1;
      vname = $3;
      vexpr = Noexpr } }

  /* Initialized regular variable */
```

```
  | type_label ID ASSIGN expr PERIOD
      { { vtype = $1;
      vname = $2;
      vexpr = $4 } }

  /* Uninitialized instance variable */
  | type_label TRAIT ID ASSIGN expr PERIOD
      { { vtype = $1;
      vname = $3;
      vexpr = $5 } }

/* Character (Class) Declarations */
cdecl:
      CHARACTER ID LPAREN formals_opt RPAREN LBRACE vdecl_list action_list RBRACE
      {{  cname = $2;
      cparent = $2;
      cformals = $4;
      cinstvars = $7;
      cactions = $8;
      }}
  /* inheritance */
  | CHARACTER ID ASSIGN ID LPAREN formals_opt RPAREN LBRACE vdecl_list action_list
RBRACE
      {{
      cname = $2;
      cparent = $4;
      cformals = $6;
      cinstvars = $9;
      cactions = $10;
      }}

/* Action (Method) Declarations */
action_list:
  /* nothing */ {[]}
  | action_list adecl {$2::$1}

adecl:
  METHOD ID LPAREN formals_opt RPAREN RETURNS type_label LBRACE stmt_list RBRACE
  {{
      aname = $2;
      aformals = $4;
      areturn = $7;
      abody = List.rev $9;
  }}

/* Statements */
stmt_list:
      /* nothing */  { [] }
```

```
  | stmt_list stmt { $2 :: $1 }

/* added vdecl to statements so that stmt list could include vdecls */
stmt:
      expr PERIOD { Expr($1) }
  | vdecl {VarDecl($1)}
  | ENDWITH LPAREN expr RPAREN PERIOD { Return($3) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt      { If($3, $5, $7) }
  | FOR LPAREN stmt SEMI expr SEMI expr RPAREN stmt
      { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt{ While($3, $5) }

/* Expressions */
expr:
      LIT_NUM      {LitNum($1)}
  | LIT_BOOL       {LitBool($1)}
  | LIT_STRING     {LitString($1)}
  | LIT_CHAR       {LitChar($1)}
  | ID             {Id($1)}
  | expr PLUS   expr {Binop($1, Add, $3)}
  | expr MINUS  expr {Binop($1, Sub, $3)}
  | expr TIMES  expr {Binop($1, Mult, $3)}
  | expr DIVIDE expr {Binop($1, Div, $3)}
  | expr MOD expr {Binop($1, Mod, $3)}
  | expr EQ  expr {Binop($1, Equal, $3)}
  | expr NEQ expr {Binop($1, Neq, $3)}
  | expr LT  expr {Binop($1, Less, $3)}
  | expr LEQ expr {Binop($1, Leq, $3)}
  | expr GT  expr {Binop($1, Greater, $3)}
  | expr GEQ expr {Binop($1, Geq, $3)}
  | expr OR  expr {Binop($1, OR, $3)}
  | expr AND expr {Binop($1, AND, $3)}
  | NOT expr {Unop(NOT, $2)}
  | LPAREN expr RPAREN {$2}
  | ID ASSIGN expr   {Assign($1, $3)} /* variable assign */
  | ID LPAREN actuals_opt RPAREN {FCall($1, $3)} /* function call */
   /* Object stuff */
  | NEW ID LPAREN actuals_opt RPAREN {Instantiate($2, $4)} /*object declaration  */
  | ID APOST ID     {Access($1, $3)} /* member access */
  | MY ID           {Access("my", $2)} /* self member access */
  | ID APOST ID ASSIGN expr {TraitAssign(Access($1, $3), $5)} /* member assign */
  | MY ID ASSIGN expr {TraitAssign(Access("my", $2), $4)}
  | ID COMMA ID LPAREN actuals_opt RPAREN {ACall($1, $3, $5)} /* action call */
   /* List stuff */
  | ID LBRACK expr RBRACK {ListAccess($1, $3)} /* myList [1 + 1] */
```

```
  | ID LBRACK expr RBRACK ASSIGN expr {ListAssign(ListAccess($1, $3), $6)} /* List
assign a[5] = 3 */
  | NEW NUMBERLIST LBRACK expr RBRACK {ListInstantiate(NumberList, $4)} /* new int
list[5 + 3]  -> ListInstantiate (int, 8) */
  | NEW BOOLLIST LBRACK expr RBRACK { ListInstantiate(BooleanList, $4)}
  | NEW CHARLIST LBRACK expr RBRACK { ListInstantiate(CharList, $4) }
  | NEW CHARACTERLIST LBRACK expr RBRACK { ListInstantiate(CharacterList, $4) }

/* Actual Parameters */
actuals_opt:
      /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
      expr                      { [$1] }
  | actuals_list SEMI expr { $3 :: $1 }
```

A.4 sast.ml

```
open Ast

type data_type =
      Void
  | Number
  | Boolean
  | String
  | Char
  | Object of class_decl
  | NumberList
  | BooleanList
  | CharList
  | CharacterList (* list of pointers to point to Characters *)

and expr_detail =
      LitNum of float
  | LitBool of bool
  | LitString of string (* quoted string literal *)
  | LitChar of char (* 'c' *)
  | Noexpr
  | Id of variable_decl
  | Assign of string * expression (* x is 5 *)
  | TraitAssign of expression * expression (* SleepingBeauty's x is 5 *)
  | Instantiate of class_decl * expression list (* object type and constructor
parameters *)
  | ListInstantiate of data_type * expression   (* list type and size *)
  | ListAccess of variable_decl * expression  (* ages[5] *)
  | ListAssign of expression * expression     (* ages[5] is 10 *)
```

```
  | Access of variable_decl * variable_decl   (* Member value access:
SleepingBeauty's. Or, my to access traits within a character*)
  | FCall of function_decl * expression list
  | ACall of variable_decl * action_decl * expression list
  | StrCat of expression * expression
  | MathBinop of expression * op * expression (* a + b *)
  | Unop of op * expression

(* Tuple of expression and the type it evaluates to *)
and expression = expr_detail * data_type

(* Variable declaration *)
(* All variable declarations have a type and a name
   If variable is initialized upon instantiation, the variable declaration
   also has an expression attached to it *)
and variable_decl =
{
  vtype: data_type;
  mutable vname : string;
  mutable vexpr : expression; (* e.g.: 5+3 in : "number x is (5 + 3)." *)
  istrait: bool;   (* boolean denoting whether variable is character trait *)
  listsize: float; (* to prevent list variables to access beyond the lenght of the
list*)
}


(* Statements *)
and statement =
  Block of statement list
| Expression of expression
| VarDecl of variable_decl
| Return of expression
  (* If statements: boolean expr, if statement, else statement *)
| If of expression * statement * statement
  (* For loops: variable decl, boolean stopping condition, increment expr, loop body
*)
| For of statement * expression * expression * statement
  (* While loops: boolean expr, loop body *)
| While of expression * statement


(* Functions *)
and function_decl = {
  fname : string; (* name of the function *)
  fformals : variable_decl list; (* formal params *)
  freturn : data_type; (* return type *)
  funcbody : statement list; (* statements, including local variable declarations *)
  isLib : bool; (* boolean denoting whether function is library function *)
```

```
}

(* Actions *)
and action_decl = {
  aname : string; (* Name of the action *)
  aclass : string;
  aformals: variable_decl list; (* formal params *)
  areturn : data_type;    (* return type *)
  abody : statement list; (* statements, including local variable declarations *)
}

(* Class Declarations *)
and class_decl = {
  cname : string; (* name of the class *)
  cparent: string;
  cformals: variable_decl list;   (* formal params *)
  cinstvars : variable_decl list; (*instance variables *)
  cactions: action_decl list;    (*lists of actions (methods) *)
}

(* Program --class declarations and function declarations *)
and program = class_decl list * function_decl list
```

A.5 semantic_analyzer.ml

```
open Ast
open Sast

let new_count = ref 0
let increment_new_count() = new_count := !new_count + 1

type symbol_table = {
  name : string;
  parent : symbol_table option;
  mutable functions: Sast.function_decl list;
  mutable variables : Sast.variable_decl list;
  mutable characters: Sast.class_decl list;
  mutable actions: Sast.action_decl list;
}

type translation_environment = {
  scope : symbol_table;
  return_type : Sast.data_type;
}

type func_wrapper =
  Some of Sast.function_decl
  | None
```

```
let rec find_function (scope: symbol_table) name =
      try
            List.find(fun f -> f.fname = name) scope.functions
      with Not_found ->
      match scope.parent with
            Some(parent) -> find_function parent name
      | _ -> raise (Failure("function '" ^ name ^ "' not found"))

let rec is_func_name_already_used (scope: symbol_table) name : func_wrapper =
  try
      Some(List.find(fun f -> f.fname = name) scope.functions)
  with Not_found ->
  match scope.parent with
      Some(parent) -> is_func_name_already_used parent name
  | _ -> None

let find_plot (l : Sast.function_decl list) =
      try
      List.find(fun f -> f.fname = "plot") l
      with Not_found -> raise (Failure("No plot found"))

let has_super (scope : symbol_table) =
  (List.length scope.characters) > 0

(* Check to see that character trait exists *)
let find_trait env name =
  let is_inherit = has_super env.scope in match
  is_inherit with
  true -> List.find(fun x-> x.vname = name) (List.nth env.scope.characters
0).cinstvars
  | false -> raise(Failure("var not found: " ^ name))

(* Check to see if variable exists *)
let rec find_variable (scope : symbol_table) orig_env name =
  try
      List.find(fun v -> v.vname = name) scope.variables
  with Not_found ->
      match scope.parent with
      Some(parent) -> find_variable parent orig_env name
      | _ -> find_trait orig_env name

(* Find all characters in scope *)
let rec find_class_decl (scope: symbol_table) name =
  try
      List.find(fun c -> c.cname = name) scope.characters
  with Not_found ->
  match scope.parent with
```

```
      Some(parent) -> find_class_decl parent name
  | _ -> raise (Failure("Character '" ^ name ^ "' not found"))

(* Check that all variables in a character are in scope *)
let rec find_class_var (scope: symbol_table) c_dec name =
  try List.find(fun v-> v.vname = name) c_dec.cinstvars
  with Not_found ->
      raise(Failure("invalid trait name: " ^ name))

let find_character (scope: symbol_table) =
   let len = List.length scope.characters in
   match len with
      0 ->
      ( match scope.parent with
      Some(parent) -> List.nth parent.characters 0
      | _ -> raise(Failure("No inherited characters"))
      )
   | _ -> List.nth scope.characters 0

(* Ensure only 's and my access on characters *)
let get_class_decl_from_type (scope: symbol_table) ctype =
  match ctype with
  Sast.Object(typDecl) -> find_class_decl scope typDecl.cname
  | _ -> raise(Failure("Not an object. can't access instance vars"))

(* Check for action in scope*)
let find_action_decl (actions : Sast.action_decl list) name className =
  try
      List.find(fun a -> (a.aname = name)) actions
  with Not_found ->
      try
      List.find(fun a -> (a.aname = (className ^ "_" ^ name))) actions
      with Not_found -> raise (Failure("action not found " ^ name))

(* Checking types for binop; takes the op anad the two types to do checking *)
let analyze_binop (scope: symbol_table) op t1 t2 = match op with
  Add ->
      if (t1 == Sast.String || t2 == Sast.String) then Sast.String
      else if (t1 == Sast.Number || t2 == Sast.Number) then
             if (t1 == Sast.Boolean || t2 == Sast.Boolean) then raise
(Failure("Invalid use of + for operands' types"))
             else if (t1 == Sast.Number && t2 == Sast.Number) then Sast.Number
      else if (t1 == Sast.Char || t2 == Sast.Char) then raise (Failure("Invalid use
of + for operands' types"))
             else Sast.String
      else if (t1 == Sast.Char || t2 == Sast.Char) then
             if (t1 == Sast.Boolean || t2 == Sast.Boolean) then raise
(Failure("Invalid use of + for operands' types"))
```

```
      else Sast.String
      else raise (Failure("Invalid use of + for operands' types"))

  | Sub ->   if (t1 <> Sast.Number || t2 <> Sast.Number) then raise (Failure("Invalid
use of - for operands' types")) else Sast.Number
  | Mult ->  if (t1 <> Sast.Number || t2 <> Sast.Number) then raise (Failure("Invalid
use of * for operands' types")) else Sast.Number
  | Div ->   if (t1 <> Sast.Number || t2 <> Sast.Number) then raise (Failure("Invalid
use of / for operands' types")) else Sast.Number
  | Equal -> if (t1 <> t2) then raise (Failure("Invalid use of = for operands'
types")) else Sast.Boolean
  | Neq ->   if (t1 <> t2) then raise (Failure("Invalid use of not= for operands'
types")) else Sast.Boolean
  | Less ->  if (t1 <> Sast.Number || t2 <> Sast.Number) then raise (Failure("Invalid
use of < for operands' types")) else Sast.Boolean
  | Leq ->   if (t1 <> Sast.Number || t2 <> Sast.Number) then raise (Failure("Invalid
use of <= for operands' types")) else Sast.Boolean
  | Greater ->  if (t1 <> Sast.Number || t2 <> Sast.Number) then raise
(Failure("Invalid use of > for operands' types")) else Sast.Boolean
  | Geq ->   if (t1 <> Sast.Number || t2 <> Sast.Number) then raise (Failure("Invalid
use of >= for operands' types")) else Sast.Boolean
  | Mod ->   if (t1 <> Sast.Number || t2 <> Sast.Number) then raise (Failure("Invalid
use of % for operands' types")) else Sast.Number
  | OR ->    if (t1 <> Sast.Boolean || t2 <> Sast.Boolean) then raise
(Failure("Invalid use of or for operands' types")) else Sast.Boolean
  | AND ->   if (t1 <> Sast.Boolean || t2 <> Sast.Boolean) then raise
(Failure("Invalid use of and for operands' types")) else Sast.Boolean
  | NOT ->   raise (Failure("Invalid use of ! for two operands"))

let analyze_unop (scope: symbol_table) op t1 = match op with
NOT ->       if (t1 <> Sast.Boolean) then raise (Failure("Invalid use of ! for
operand type")) else Sast.Boolean
| _ ->       raise (Failure("Invalid unary operator"))


let listClass = {cname = "listAcc"; cparent = "None"; cformals = []; cinstvars = [];
cactions = []}

(* Mainly used for errors to display types as strings *)
let rec type_as_string t = match t
with
     Sast.Number -> "float"
   | Sast.Boolean -> "bool"
   | Sast.String -> "char *"
   | Sast.Char -> "char"
   | Sast.Void -> "void"
   | Sast.Object(n) -> "object" ^ n.cname
   | Sast.NumberList -> "float list"
```

```
    | Sast.BooleanList -> "bool list"
    | Sast.CharList -> "char list"
    | Sast.CharacterList -> "character list"

(* Convert list type to type of element on list access *)
let find_listAcc_type t = match t with
      Sast.NumberList -> Sast.Number
  | Sast.BooleanList -> Sast.Boolean
  | Sast.CharList -> Sast.Char
  | Sast.CharacterList -> Sast.Object(listClass) (* check type in analyze expr *)
  | _ -> raise(Failure("Not list type"))

(* AST data type to SAST data type *)
let rec convert_data_type env old_type = match old_type with
  | Ast.Void -> Sast.Void
  | Ast.Number -> Sast.Number
  | Ast.Boolean -> Sast.Boolean
  | Ast.String -> Sast.String
  | Ast.Char -> Sast.Char
  | Ast.Object(n) ->
      let obj_dec = try find_class_decl env.scope n
      with Not_found -> raise(Failure("classdecl not found")) in
      Sast.Object(obj_dec)
  | Ast.NumberList -> Sast.NumberList
  | Ast.BooleanList -> Sast.BooleanList
  | Ast.CharList -> Sast.CharList
  | Ast.CharacterList -> Sast.CharacterList

(* Compare parameter types *)
let rec compare_p_types formalVars actualExprs = match formalVars, actualExprs with
      [], [] -> true
      |[], y::ytail  ->raise(Failure("wrong number of params"))
      | x::xtail, [] -> raise(Failure("wrong number of params"))
      |x::[], y::y2::ytail -> raise(Failure("wrong number of params"))
      |x::x2::[], y::[] -> raise(Failure("wrong number of params"))
      |x::xtail, y::ytail -> let (_, actual_typ) = y in
      if(actual_typ) == x.vtype then begin compare_p_types xtail ytail end
      else raise(Failure("wrong parameter type"))

(* Expression Environment *)
let rec analyze_expr env = function
      Ast.LitNum(v) -> Sast.LitNum(v), Sast.Number
      | Ast.LitBool(v) -> Sast.LitBool(v), Sast.Boolean
      | Ast.LitString(v) -> Sast.LitString(v), Sast.String
      | Ast.LitChar(v) -> Sast.LitChar(v), Sast.Char
      | Ast.Id(vname) ->
      let vdecl = try
            find_variable env.scope env vname (* locate a variable by name *)
```

```
with Not_found ->
raise (Failure("undeclared identifier " ^ vname))
in Sast.Id(vdecl), vdecl.vtype (* return type *)

| Ast.Assign(vname, expr) ->
let vdecl = try
find_variable env.scope env vname
with Not_found ->
raise (Failure("undeclared identifier " ^ vname))
in let (e, expr_typ) = analyze_expr env expr
in if vdecl.vtype <> expr_typ then raise(Failure("Expression does not match
variable type"))
else
Sast.Assign(vname, (e, expr_typ)), expr_typ

| Ast.TraitAssign(objAccess, ex) ->
let (var, vtype ) = analyze_expr env objAccess in
let (e, exp_type) = analyze_expr env ex in
if vtype <> exp_type then
raise(Failure("Incorrect type assignment to character trait"))
else
Sast.TraitAssign((var, vtype), (e, exp_type)), exp_type

| Ast.Instantiate(objType, exprs) ->
let objDecl = try
find_class_decl env.scope objType
with Not_found ->
      raise (Failure("class not found " ^ objType))
in
let actual_p_typed = List.map (fun e -> analyze_expr env e) exprs in
if (compare_p_types objDecl.cformals actual_p_typed) = true then begin
      increment_new_count(); (Sast.Instantiate(objDecl, actual_p_typed),
Sast.Object(objDecl))
end
else raise (Failure("invalid parameters to function"))

| Ast.ListInstantiate(list_type, s) ->
(
let ltype = convert_data_type env list_type in
let (size, typ) = analyze_expr env s in
if typ = Sast.Number then
(Sast.ListInstantiate(ltype, (size, typ)), ltype)
else raise(Failure("Must specify size of list as number"))
)

| Ast.ListAccess(id, indx) ->
(
let var = try
```

```
        find_variable env.scope env id
        with Not_found ->
        raise (Failure("Undeclared identifier " ^ id)) in
        let (e, etype) = analyze_expr env indx in
        let accessType = find_listAcc_type var.vtype in
        (
        match (e, etype) with
        (Sast.LitNum(n), Sast.Number) ->
                (* print_string (string_of_float n); *)
                if (n > (var.listsize -. 1.0)) then (* prevent access beyond end of list
*)
                raise(Failure("Cannot access beyond the size of the list"))
                else
                (Sast.ListAccess(var, (e, etype)), accessType)
        | _ -> (Sast.ListAccess(var, (e, etype)), accessType)
        )
        )

        | Ast.ListAssign(access, assn) ->
        (
        let (access, etype) = (analyze_expr env access) in
        let (new_val, vtype) = analyze_expr env assn in
        (
        match etype with
        (* If list is character type, use dummy class object to check that the
assignment type is also a Sast.Object(n) *)
        (* Using void * so actual class decl equivalence doesn't matter, only the fast
that it is a Sast.Object *)
        | Sast.Object(n) ->
                (
                match vtype with
                | Sast.Object(x) -> (Sast.ListAssign((access, etype), (new_val, vtype)),
vtype)
                | _ -> raise(Failure("Assignment value type does not match type of list
element"))
                )
        | _ ->
                (
                if etype <> vtype then
                raise(Failure("Assignment value type does not match type of list
element"))
                else
                (Sast.ListAssign((access, etype), (new_val, vtype)), vtype)
                )
        )
        )

        | Ast.Access(objName, varName) -> (* character access *)
```

```
	(* "self" reference *)
	(
	if (objName = "my") then begin
	let classDec = find_character env.scope in (*only class dec in scope is
itself, may be in outer scope because of block *)
	let classVar = try find_class_var env.scope classDec varName
		with Not_found ->
		raise(Failure("instance variable not found" ^ varName))
	in let objVar = {vtype = Object(classDec); vname = ""; vexpr = (Sast.Noexpr,
Sast.Void); istrait = true; listsize = 0.0 } in
	(Sast.Access(objVar, classVar), classVar.vtype)
	end
	(* Regular access *)
	else begin
	let objDec = try find_variable env.scope env objName
		with Not_found ->
		raise(Failure("object variable not found" ^ objName))
	in let classDec = try get_class_decl_from_type env.scope objDec.vtype
		with Not_found -> raise(Failure("class not found"))
	in let class_var =
	try find_class_var env.scope classDec varName
		with Not_found ->
		raise(Failure("instance variable not found" ^ varName))
	in (Sast.Access(objDec, class_var), class_var.vtype)
	end
	)

	| Ast.Binop(e1, op, e2) ->
	let e1 = analyze_expr env e1 (* Check left and right children *)
	and e2 = analyze_expr env e2 in
	let _, t1 = e1 (* Get the type of each child *)
	and _, t2 = e2 in (*let valid = *)
	let validbinop = try analyze_binop env.scope op t1 t2
	with Not_found -> raise (Failure("Invalid binary operator"))
	in if validbinop = Sast.String then Sast.StrCat(e1, e2), validbinop
	else Sast.MathBinop(e1, op, e2), validbinop (* Success: result is int *)

	| Ast.Unop(op, e1) ->
	let e1 = analyze_expr env e1 in
	let _, t1 = e1 in
	let validunop = try
		analyze_unop env.scope op t1
	with Not_found -> raise (Failure("Invalid unary operator"))
	in Sast.Unop(op, e1), validunop

	| Ast.FCall(fname, params) ->
	let actual_p_typed = List.map (fun e -> analyze_expr env e) params in
	let fdecl = try
```

```
	find_function env.scope fname
	with Not_found -> raise (Failure("function '" ^ fname ^ "' not found"))
	in let formal_p_list = fdecl.fformals in
	let ret_type = fdecl.freturn in
	if fname <> "say" then begin
	if (compare_p_types formal_p_list actual_p_typed) = true then
		(Sast.FCall(fdecl, actual_p_typed), ret_type)
	else raise (Failure("invalid parameters to function"))
	end
	else Sast.FCall(fdecl, actual_p_typed), ret_type

	| Ast.ACall(objName, actName, expr_list) ->
	(* Grab object variable *)
	let objDec = try find_variable env.scope env objName
	with Not_found -> raise(Failure("variable not found " ^ objName))
	(* Find corresponding class variable *)
	in let classDec =
	get_class_decl_from_type env.scope objDec.vtype

	(* Check that action is valid *)
	in let actionDec = try find_action_decl classDec.cactions actName
classDec.cname
	with Not_found -> raise (Failure("action not found " ^ actName))
	in
	(*check that params are correct *)
		let formal_p_list = actionDec.aformals in
		let actual_p_typed = List.map( fun a -> analyze_expr env a) expr_list in
		let ret_type = actionDec.areturn in
		if (compare_p_types formal_p_list actual_p_typed) = true then
		(Sast.ACall(objDec, actionDec, actual_p_typed), ret_type)
		else raise (Failure("invalid parameters to action " ^ actName))

	| Ast.Noexpr -> Sast.Noexpr, Sast.Void

(* convert ast.var_decl to sast.variable_decl*)
(* check types and add variable to scope's variable list *)
let check_var_decl (env: translation_environment) (var: Ast.var_decl) =
  let reserve_var_names = Str.regexp "['_']['0'-'9']*" in
  let is_reserved = Str.string_match reserve_var_names var.vname 0 in
  if is_reserved <> true then begin
	let typ = convert_data_type env var.vtype in
	let (e, expr_typ) = analyze_expr env var.vexpr in match e
	(* If Uninitialized and var type is a character, throw error *)
	(* Else, the variable is valid *)
	with Sast.Noexpr ->
		(match typ with
		Sast.Object(o) -> raise(Failure("must assign to character variable on
declaration"))
```

```
            | _ ->
                    let sast_var_decl = { vtype = typ; vname = var.vname; vexpr = (e,
expr_typ); istrait = false; listsize = 0.0 }
                    in env.scope.variables <- List.append env.scope.variables
[sast_var_decl];
                    sast_var_decl)
        (* If variable is initialized, check that the two types match *)
        | _ ->
            if typ <> expr_typ && (type_as_string expr_typ) <> "objectlistAcc" then
begin
            raise(Failure(
            "Variable assignment does not match variable type " ^(type_as_string
typ) ^ " " ^ (type_as_string expr_typ)))
            end
            else begin
            (
            match (e, expr_typ) with
            (Sast.ListInstantiate( _, (size, _ )), _ ) ->
            (
                    match size with
                    | LitNum(s) ->
                    let sast_var_decl = { vtype = typ; vname = var.vname; vexpr = (e,
expr_typ); istrait = false; listsize = s }
                    in env.scope.variables <- List.append env.scope.variables
[sast_var_decl];
                    sast_var_decl
                    | _ -> raise(Failure("List size must be specified as number"))
            )
            | _ ->
            let sast_var_decl = { vtype = typ; vname = var.vname; vexpr = (e,
expr_typ); istrait = false; listsize = 0.0 }
            in env.scope.variables <- List.append env.scope.variables
[sast_var_decl];
            sast_var_decl
            )
            end
  end
  else begin
        raise(Failure(
        "variable name " ^ var.vname ^ "invalid. cannot use \"_\"" ^ "or \"_\"
followed only by numerical digits"
        )) end

let rec analyze_stmt env = function
        Ast.Expr(e) -> Sast.Expression(analyze_expr env e) (* expression *)
  | Ast.VarDecl(var_decl) ->
        if List.exists (fun x -> x.vname = var_decl.vname) env.scope.variables then
            raise(Failure("Variable already declared in this scope"))
```

```
	else
		let sast_var = check_var_decl env var_decl in
		let _ = env.scope.variables <- sast_var :: env.scope.variables in (*
save new var_decl in symbol table *)
		Sast.VarDecl(sast_var);
  | Ast.If(e, s1, s2) ->
	let sastexpr = analyze_expr env e in (* Check the predicate *)
	let (_, typ) = sastexpr in
	if typ = Sast.Boolean then
	Sast.If(sastexpr, analyze_stmt env s1, analyze_stmt env s2)
	else raise(Failure("invalid if condition"))
  | Ast.Return(e) -> let sastexpr = analyze_expr env e in Sast.Return(sastexpr)
  | Ast.For(e1, e2, e3, s) ->
	let sastexpr1 = analyze_stmt env e1 in
	let sastexpr2 = analyze_expr env e2 in
	let (_, typ) =  sastexpr2 in
	if typ <> Sast.Boolean then
	raise(Failure("For loop must have boolean condition"))
	else let sastexpr3 = analyze_expr env e3 in
	let s = analyze_stmt env s in
	Sast.For(sastexpr1, sastexpr2, sastexpr3, s)
  | Ast.While(e, s) ->
	let sastexpr = analyze_expr env e in
	let (_, typ) = sastexpr in
	if typ <> Sast.Boolean then
	raise(Failure("While condition must be a boolean expression"))
	else let s = analyze_stmt env s in
	Sast.While(sastexpr, s)
  | Ast.Block(stmts) ->
	let scope' = {name = "new block"; parent = Some(env.scope); functions = [];
variables = []; characters = []; actions = []}
	in let env' = { env with scope = scope'} in
	let sast_blck =
	List.map( fun s -> analyze_stmt env' s) stmts in
	Sast.Block(sast_blck)
	let library_funcs = [
	{

	fname = "say";
	fformals = [{vtype = (Sast.String);
			vname = "str";
			vexpr = (Sast.Noexpr, Sast.String);
			istrait = false;
			listsize = 0.0
			}];
	freturn = Sast.String;
	funcbody = [Sast.Expression(Sast.LitString(""), Sast.String)];
	isLib = true;
```

```
	}
	]
(* Check return type matches return *)
let check_ret (expTyp: Sast.data_type) (env: translation_environment) (f:
Sast.statement) = match f with
  Sast.Return(e) -> let (_, typ) = e in
	if expTyp = typ  then true
	else if expTyp = Sast.Void then raise (Failure("Void function cannot return a
value"))
	else raise (Failure ("Incorrect return type"))
  | _ -> false

(* If return is not void, ensure value is returned *)
let find_return (body_l : Sast.statement list) (env: translation_environment)
(expTyp: Sast.data_type) =
  try
	List.find(check_ret expTyp env) body_l
  with Not_found -> if expTyp <> Sast.Void then raise (Failure("No return found"))
else Expression(Noexpr, Void)

let analyze_func (fun_dcl : Ast.func_decl) env : Sast.function_decl = (*Why is env of
type Sasy.function_decl??*)
  let name = fun_dcl.fname in
  if name = "say"
	then raise(Failure("Cannot use library function name: " ^ name))
  else begin
	let is_name_taken = is_func_name_already_used env.scope name in
	if is_name_taken != None then raise(Failure("Function name: " ^ name ^ "is
already in use."))
	else begin
	let old_ret_type = fun_dcl.freturn
	and old_body = fun_dcl.fbody in (*?*)
	let ret_type = convert_data_type env old_ret_type in
	let formals = List.map(fun st-> check_var_decl env st) fun_dcl.fformals in
	env.scope.functions <- List.append env.scope.functions [{fname=name; fformals
= formals; freturn = ret_type; funcbody= []; isLib = false}];
	let body = List.map (fun st ->  analyze_stmt env st) old_body in
	let _ = find_return body env ret_type in
	let sast_func_dec = {fname = name; fformals = formals; freturn = ret_type;
funcbody= body; isLib = false} in
	env.scope.functions <- List.filter (fun f -> f.fname <> name)
env.scope.functions; (* remove dummy func for recursion *)
	env.scope.functions <- List.append env.scope.functions [sast_func_dec]; (* add
real func *)
	sast_func_dec
	end
  end
```

```
let has_super (scope : symbol_table) =
  (List.length scope.characters) > 0

let check_parent (var : Ast.var_decl) (class_env : translation_environment) =
      if has_super class_env.scope then
      (* Check direct parent, which will have all inherited traits *)
      if List.exists (fun x -> x.vname = var.vname) (List.nth
class_env.scope.characters 0).cinstvars then
      raise(Failure("Cannot override inherited trait: " ^ var.vname))
      else if List.exists (fun x -> x.vname = var.vname) class_env.scope.variables
then
      raise(Failure("Trait " ^ var.vname ^ " already declared in this Character"))

(* Check trait not declared twice. Don't allow overriding of inherited traits. *)
let analyze_classvars (var : Ast.var_decl) (class_env : translation_environment) =
      let _ = check_parent var class_env in
      let sast_var = check_var_decl class_env var in
      let _ = class_env.scope.variables <- sast_var :: class_env.scope.variables in
(* save new class variable in symbol table *)
      sast_var

let analyze_acts (act : Ast.act_decl) (class_env : translation_environment) =
  if List.exists (fun x -> x.aname = act.aname) class_env.scope.actions then
      raise(Failure("Action " ^ act.aname ^ " already declared for this character"))
  else
  let name = act.aname in
      if name = "say" then raise(Failure("Cannot use library function name: " ^
name))
      else
      let ret_type = convert_data_type class_env act.areturn in
      let formals = List.map (fun param -> check_var_decl class_env param)
act.aformals in
      let body = List.map (fun st -> analyze_stmt class_env st) act.abody in
      let cdecl = find_character class_env.scope in
      let sast_act = {aname = name; aclass = cdecl.cname; aformals = formals;
areturn = ret_type; abody = body} in
      let _ = class_env.scope.actions <- sast_act :: class_env.scope.actions in
      sast_act

let find_parent parent child (env: translation_environment)=
  (* if parent and child name same, then no inheritance, otherwise yes inheritance *)
  if parent <> child then
      (* If inheriting, find parent class *)
      List.find (fun c -> c.cname = parent) env.scope.characters
  else {cname = child; cparent = child; cinstvars = []; cactions = []; cformals = []}

let analyze_class (clss_dcl : Ast.cl_decl) (env: translation_environment) =
  let name = clss_dcl.cname in
```

```
  let parent = clss_dcl.cparent in
  if List.exists (fun x -> x.cname = name) env.scope.characters then
      raise(Failure("Character " ^ name ^ " already exists"))
  else if (parent <> name) && ((List.exists (fun x -> x.cname = clss_dcl.cparent)
env.scope.characters) = false)
      then raise(Failure("Character " ^ clss_dcl.cparent ^ " does not exist"))
  else
      let full_parent =  find_parent parent name env in
      (* First get parent instance variables and actions and store *)
      let parent_acts =
      if full_parent.cname <> name then
      List.map(fun a ->
      {aname = (name ^ "_" ^ a.aname); aclass = name; aformals = a.aformals; areturn
= a.areturn; abody = a.abody}
      ) full_parent.cactions
      else [] in
      let parent_ivars = if full_parent.cname <> name then full_parent.cinstvars
else [] in
   (* create new scope for the class *)
   (* let self = {cname = name; cinstvars = []; cactions = []; cformals = []} in *)
      let class_scope =
      {name = name; parent = None; functions = library_funcs; variables = [];
characters = [full_parent]; actions = []} in
      let class_env = {scope = class_scope; return_type = Sast.Void} in

      (* Now check current inst vars and formals *)
      let newcformals = List.map(fun f-> check_var_decl class_env f)
clss_dcl.cformals in
      let inst_vars = List.map (fun st -> analyze_classvars st class_env)
clss_dcl.cinstvars in

      (* Combine parent and child instance variables and formal parameters *)
      let all_ivars = inst_vars @ parent_ivars in
      let all_formals = full_parent.cformals @ newcformals in

      (* Add class to it's own character scope list so that "self" references work
*)
      class_env.scope.characters <-
      {cname = name; cparent = name; cinstvars = all_ivars; cactions = parent_acts;
cformals = all_formals} :: class_env.scope.characters;
      let all_actions = (List.map (fun a -> analyze_acts a class_env)
clss_dcl.cactions) @ parent_acts in
      let new_class = {cname = name; cparent = full_parent.cname; cinstvars =
all_ivars; cactions = all_actions; cformals = all_formals} in
      (* add the new class to the list of classes in the symbol table *)
      let _ = env.scope.characters <- new_class :: (env.scope.characters) in
      new_class
```

```
let analyze_semantics prgm: Sast.program =
  let prgm_scope = {name= "prgrm"; parent = None; functions = library_funcs;
variables = []; characters = []; actions = []} in
  let env = {scope = prgm_scope; return_type = Sast.Number} in
  let (class_decls, func_decls) = prgm  in
  let new_class_decls = List.map (fun f -> analyze_class f env)
(List.rev(class_decls)) in
  let new_func_decls = List.map (fun f -> analyze_func f env) (List.rev(func_decls))
in
  (* Search for plot *)
  let plot_decl= try
      find_plot new_func_decls
      with Not_found -> raise (Failure("No plot was found.")) in
  match plot_decl.freturn with
  Sast.Void -> (new_class_decls, List.append new_func_decls library_funcs)
  | _ -> raise(Failure("plot cannot return anything"))
```

A.6 cast.ml

```
open Ast
open Sast

(* Objects in storybook are converted to structs *)
type class_struct = {
       sname: string;
       sivars: Sast.variable_decl list;
       svtable: vtable
}

(* Each struct points to a virtual table containing pointers to their functions *)
and vtable = {
       class_name: string; (* will tell us the name of the struct to create a ptr to
*)
       vfuncs: action_decl list;
}

(* C Program consists of structs and function declarations *)
(* Virtual tables are held by class_struct record *)
and prgrm = class_struct list * Sast.function_decl list
```

A.7 pretty_print.ml

```
open Printf
open Ast
open Sast
open Cast
open Semantic_analyzer
open Lexing
```

```
open Codegen

(* current_ptr keeps track of index of each object in the array of c structs *)
let current_ptr = ref (-1)
let increment_cur_ptr() = current_ptr := !current_ptr + 1

(* current_var is an int that keeps track of the current variable name
   used in the code -- we convert this to string name *)
let current_var = ref 0
let increment_current_var() = current_var := !current_var + 1
let get_next_var_name() = increment_current_var(); "_" ^ (string_of_int !current_var)

(* Convert operations to strings *)
let get_op o = match o
with Add -> " + "
   | Sub -> " - "
   | Mult -> "* "
   | Div -> " / "
   | Mod -> " % "
   | Equal -> " == "
   | Neq ->  " != "
   | Less -> " < "
   | Leq -> " <= "
   | Greater -> " > "
   | Geq -> " >= "
   | OR -> " || "
   | AND -> " && "
   | NOT -> " !"

let type_as_string t = match t
with
      Sast.Number -> "float"
   | Sast.Boolean -> "bool"
   | Sast.String -> "char *"
   | Sast.Char -> "char"
   | Sast.Void -> "void"
   | Sast.Object(n) -> "struct " ^ n.cname ^ " *"
   | Sast.NumberList -> "float *"
   | Sast.BooleanList -> "bool *"
   | Sast.CharList -> "char *"
   | Sast.CharacterList -> "void **"

let listClass = {cname = "listAcc"; cparent = "None"; cformals = []; cinstvars = [];
cactions = []}

(* find type of element returned on list access *)
let find_listAcc_type t = match t with
      Sast.NumberList -> Sast.Number
```

```
  | Sast.BooleanList -> Sast.Boolean
  | Sast.CharList -> Sast.Char
  | Sast.CharacterList -> Sast.Object(listClass)
  | _ -> raise(Failure("Not list type"))

let get_bool_str b = match b with
      true -> "1"
   | _ -> "0"

let get_str_len expr_str typ = match typ
with Sast.Number -> "5000"
   | Sast.Boolean -> "5"
   | Sast.String -> "strlen(" ^ expr_str ^ ")"
   | Sast.Char -> "1"
   | _ -> "10000"


let get_str_cat_code expr1_str typ1 expr2_str typ2 v_name=
      let buf_name = "buf_" ^ v_name in
      let convert_expr1 = match typ1
      with Sast.Number -> "sprintf(" ^ buf_name ^ ", \"%g\"," ^ expr1_str ^ ");\n"
      | Sast.Boolean -> "sprintf(" ^ buf_name ^ ", \"%s\", " ^ expr1_str ^ " ?
\"true\" : \"false\");\n"
      | Sast.String -> "sprintf(" ^ buf_name ^ ", \"%s\"," ^ expr1_str ^ ");\n"
      | Sast.Char -> "sprintf(" ^ buf_name ^ ", \"%c\", \'" ^ expr1_str ^ "\');\n"
      | _ -> "" in
      let convert_expr2 = match typ2
      with Sast.Number -> "sprintf(" ^ buf_name ^ " + strlen(" ^ buf_name ^ "),
\"%g\", " ^ expr2_str ^ ");\n"
      | Sast.Boolean -> "sprintf(" ^ buf_name ^ " + strlen(" ^ buf_name ^ "),
\"%s\"," ^ expr2_str ^ " ? \"true\" : \"false\");\n"
      | Sast.String -> "sprintf(" ^ buf_name ^ " + strlen(" ^ buf_name ^ "),
\"%s\"," ^ expr2_str ^ ");\n"
      | Sast.Char -> "sprintf(" ^ buf_name ^ " + strlen(" ^ buf_name ^ "), \"%c\", "
^ expr2_str ^ ");\n"
      | _ -> "" in

      let expr1_len = get_str_len expr1_str typ1 in
      let expr2_len = get_str_len expr2_str typ2 in
      let buf_code = "char " ^ buf_name ^ "[ " ^ expr1_len ^ " + " ^ expr2_len ^ " +
1];\n" in

      buf_code ^ convert_expr1 ^ convert_expr2 ^ "char *" ^ v_name ^ " = buf_" ^
v_name ^ ";"

let idx = ref (0)
let increment_idx() = idx := !idx + 1
```

```
let rec get_init_str frm actl name =
  let (actl_exp_str, prec_code) = get_expr actl in
  let init_str = prec_code ^ "\n" ^
  "((struct " ^ name ^" *)ptrs[" ^ (string_of_int !current_ptr) ^ "])  -> " ^
frm.vname ^ " = "
  ^ actl_exp_str ^ ";\n" in
  init_str

and initalize_inst_vars (forms: Sast.variable_decl list) actuals name =
  let p_list = List.fold_left (
       fun str f ->
       let actl_i = List.nth actuals !idx in
       let new_str = get_init_str f actl_i name
       in
       increment_idx();
       str ^ new_str
       ) "" forms
  in let vtable_str = "((struct " ^ name ^" *)ptrs[" ^ (string_of_int !current_ptr) ^
"])  ->" ^
       "vtable = &vtable_for_" ^ name ^ ";\n\n" in
  (p_list ^ vtable_str)


and get_expr (e, t) = match e
with Sast.LitString(s) ->  (s, "")
   | Sast.LitBool(b) -> let b_str = get_bool_str b in (b_str, "")
   | Sast.LitNum(n) -> (string_of_float n, "")
   | Sast.LitChar(c) -> ("\'" ^ Char.escaped c ^ "\'", "")
   | Sast.Id(var) -> (var.vname, "")

   | Sast.Assign(id, e) ->
       let (exp, prec_assign) = get_expr e in
       (id ^ " = " ^ exp, prec_assign)

   | Sast.Instantiate(c_dec, exprs) ->
       increment_cur_ptr();
       let rev_vars = List.rev c_dec.cinstvars in
       let _ = idx := 0 in
       let init_str = (initalize_inst_vars rev_vars exprs c_dec.cname) in
       let obj_inst_str = "\tptrs[" ^ string_of_int !current_ptr ^ "]" ^
       " = malloc((int)sizeof(struct " ^ c_dec.cname ^ " ));\n" ^ init_str in
       ("ptrs[" ^ string_of_int !current_ptr ^ "];\n", obj_inst_str)

   | Sast.ListInstantiate(typ, s) ->
       let dtyp = type_as_string typ in
       let dataType = String.sub dtyp 0 (String.length dtyp - 1) in (* get rid of ptr
to get size*)
```

```
      let (size, prec_code) = get_expr s in
      let intSize = String.sub size 0 (String.length size - 1) in (* turn float into
int *)
      ("malloc(" ^ intSize ^ " * sizeof(" ^ dataType ^ "))", prec_code)

   | Sast.ListAccess(vdecl, i) ->
      let (indx, prec_access) = get_expr i in
      let listId = vdecl.vname in
      (listId ^ "[(int)" ^ indx ^ "]", prec_access)

      | Sast.ListAssign(access, v) ->
      let (elem, prec_access) = get_expr access in
      let (assn, prec_assign) = get_expr v in
      (elem ^ " = " ^ assn, prec_access ^ prec_assign)

   | Sast.Unop(op, expr) ->
      let op_str = get_op op in let (expr_str, prec_unop) = get_expr expr in
      (op_str ^ "(" ^ expr_str ^ ")", prec_unop)

   | Sast.MathBinop(expr1, op, expr2) ->
      let (expr_str_1, prec_bin1) = get_expr expr1 in
      let (expr_str_2, prec_bin2) = get_expr expr2 in
      if op = Equal then
      let op_str = get_op op in
            let (det1, typ1) = expr1 in
                        match typ1 with
                        Sast.String -> ("strcmp(" ^ expr_str_1 ^ "," ^ expr_str_2
^ ") " ^ op_str ^ " 0", prec_bin1^prec_bin2)
                        | _ -> (expr_str_1^op_str ^ expr_str_2,
prec_bin1^prec_bin2)
      else if op = Mod then
      let op_str = get_op op in
      ("(double)" ^ "((int) (" ^ expr_str_1^ ") " ^ op_str ^ "(int) ( " ^ expr_str_2
^ "))", prec_bin1^prec_bin2)
      else
      let op_str = get_op op in
      (expr_str_1^op_str ^ expr_str_2, prec_bin1^prec_bin2)

   | Sast.StrCat(expr1, expr2) ->
      let (expr1_str, prec_strcat1) = get_expr expr1 in
      let (expr2_str, prec_strcat2) = get_expr expr2 in
      let (_, typ1) = expr1 and (_, typ2) = expr2 in
      let v_name = get_next_var_name() in
      let str_cat_code = get_str_cat_code expr1_str typ1 expr2_str typ2 v_name in
      (v_name, prec_strcat1 ^ prec_strcat2 ^ str_cat_code)

      | Sast.TraitAssign(accessVar, expr) ->
      let (varAccess, prec_var) = get_expr accessVar in
```

```
	let (new_value, prec_new) = get_expr expr in
	(varAccess ^ "=" ^ new_value, prec_var ^ prec_new)

	| Sast.Access(obj_dec, var_dec) ->
	(obj_dec.vname ^ " -> " ^ var_dec.vname ,"")

	| Sast.FCall (f_d, e_l) ->
	if f_d.fname = "say" then begin
	let (strExp, typ) = (List.nth e_l 0) in match strExp
	with Sast.LitString(s) ->
		let lit_str = (String.sub s 0 (String.length s - 1)) ^ ("\\n\"") in
		("printf" ^ " ( " ^ lit_str ^ ")", "")
		| Sast.LitNum(n) -> ("printf" ^ " ( \"%g\", " ^ (string_of_float n) ^
")", "")
		| Sast.LitBool(b) -> ("printf( \"%s\", " ^ (get_bool_str b) ^ " ?
\"true\" : \"false\");\n", "")
		| Sast.LitChar(c) -> ("printf( \"%c\", \'" ^ Char.escaped c ^  "\')",
"")
		| Sast.MathBinop(e1, op, e2) ->
		let (expr_str, prec_expr) = get_expr (strExp, typ) in
		if typ = Sast.Number
		then ("printf ( \"%g\\n\", " ^ expr_str ^ ")" , prec_expr)
		else
		("printf ( \"%s\\n\", " ^ expr_str ^ " ? \"true\" : \"false\")",
prec_expr)
		| Sast.Unop(op, e) ->
		let (expr_str, prec_expr) = get_expr(strExp, typ) in
		if typ = Sast.Number
		then ("printf (\" oops, unops for numbers are not implemented \")",
prec_expr)
		else
		("printf ( \"%s\\n\", " ^ expr_str ^ " ? \"true\" : \"false\")",
prec_expr)
		| Sast.StrCat(e1, e2) -> let (str_expr, prec_code_str) = get_expr
(strExp, typ) in
		let whole_str = prec_code_str ^ "\n\tprintf (\"%s\\n\"," ^str_expr ^ ")"
in
		(whole_str, "")
		| Sast.Id(var) -> let typ = var.vtype in
		( match typ with
			Sast.String -> ("printf ( \"%s\\n\", " ^ var.vname ^ ")", "")
			| Sast.Number -> ("printf (\"%g\"," ^ var.vname ^ ")", "")
			| Sast.Boolean -> ("printf(\"%d\\n\", " ^ var.vname ^ ")", "")
			| Sast.Char -> ("printf( \"%c\", " ^ var.vname ^  ")", "")
			| _ -> ("", "") )
		| Sast.ListAccess(vdecl, i) ->
		let (indx, _) = get_expr i in
		let listId = vdecl.vname in
```

```
            let listAccess = (listId ^ "[(int)" ^ indx ^ "]") in
            let accessType = find_listAcc_type vdecl.vtype in
            (match accessType with
                  Sast.Number -> ("printf(\"%f\", " ^ listAccess ^ ")", "")
                  | Sast.Boolean -> ("printf(\"%d\"," ^ listAccess ^ ")", "")
                  | Sast.Char -> ("printf(\"%c\", " ^ listAccess ^ ")", "")
                  | _ -> ("", "")
            )
            | Sast.Access(objVar, instVar) ->
            let typ = instVar.vtype in
            let (expr_str, prec_code) =  get_expr (strExp, typ) in
            (match typ with
                  Sast.Number -> ("\tprintf ( \"%g\\n\", " ^ expr_str ^ ")",
prec_code)
                  | Sast.Boolean ->("\tprintf (\"%d\"," ^ expr_str ^ ")",
prec_code)
                  | Sast.String -> ("\tprintf(\"%s\\n\", " ^ expr_str ^ ")",
prec_code)
                  | Sast.Char ->  ("\tprintf( \"%c\", " ^ expr_str ^  ")",
prec_code)
                  | _ -> raise(Failure("not a printable type")))

            | Sast.FCall(f_d_inner, e_l_inner) ->
            let (inner_func_str, prec_inner_func) = get_expr (strExp, typ) in
            ( match typ with
                  Sast.String -> ("\tprintf ( \"%s\\n\", " ^ inner_func_str ^ ")",
prec_inner_func)
                  | Sast.Number -> ("\tprintf (\"%g\"," ^ inner_func_str ^ ")",
prec_inner_func)
                  | Sast.Boolean -> ("\tprintf(\"%d\\n\", " ^ inner_func_str ^ ")",
prec_inner_func)
                  | Sast.Char -> ("\tprintf( \"%c\", " ^ inner_func_str ^  ")",
prec_inner_func)
                  | _ -> ("", "") )
            (*  | Sast.ACall(objDec, actDec, exprs) -> *)
            | Sast.Noexpr -> ("", "")
            | _ -> ("", "")
      end
      (* Regular function call --i.e., not "say" *)
      else begin
      let (param_str, prev_code) = List.fold_left(fun str_tup e ->
            let (cur_str, cur_prec_code) = get_expr e in
            let (prev_str, prev_prec_code) = str_tup in
            (prev_str ^ cur_str ^ ", ", prev_prec_code ^ "\n" ^ cur_prec_code)
      ) ("", "") e_l in
      let clean_param_str =
      if (String.length param_str) > 0 then (String.sub param_str 0 ((String.length
param_str) - 2))
```

```
      else param_str in

      let fcall_str = "\t " ^ f_d.fname ^ " " ^ " (" ^  clean_param_str ^ " )" in
      match f_d.freturn with
      | Sast.String ->
            let ret_var = get_next_var_name() in
            let call_and_store = "char *" ^ ret_var ^ " = " ^ fcall_str ^ ";\n" in
            let save_var = get_next_var_name() in
            let save_buf = "char " ^ save_var ^ "[strlen(" ^ ret_var ^ ")];\n" in
            let copy = "strcpy(" ^ save_var ^ ", " ^ ret_var ^ ");\n" in
            let free = "free(" ^ ret_var ^ ");\n" in
            (save_var, (prev_code ^ call_and_store ^ save_buf ^ copy ^ free))

      | _ -> (fcall_str, prev_code)
      end

  (* Action call: takes in object variable declaration, action declaration,
      and actual parameters *)
  | Sast.ACall(objDec, actDec, exprs) ->
      let (param_str, prev_code) = List.fold_left(fun str_tup e ->
      let (cur_str, cur_prec_code) = get_expr e in
      let (prev_str, prev_prec_code) = str_tup in
      (prev_str ^ cur_str ^ ", ", prev_prec_code ^ "\n" ^ cur_prec_code)
      ) ("", "") exprs in
      let full_param_str = param_str ^ objDec.vname in
      let access_vtbl_act = objDec.vname ^ "->vtable->" ^ actDec.aname in
      let acall_str = "\t " ^ access_vtbl_act ^ " " ^ " (" ^  full_param_str ^ " )"
in

      (* Figure out what type the return is *)
      (match actDec.areturn with
      (* If action returns a string, must free the malloc'ed string *)
      | Sast.String ->
            let ret_var = get_next_var_name() in
            let call_and_store = "char *" ^ ret_var ^ " = " ^ acall_str ^ ";\n" in
            let save_var = get_next_var_name() in
            let save_buf = "char " ^ save_var ^ "[strlen(" ^ ret_var ^ ")];\n" in
            let copy = "strcpy(" ^ save_var ^ ", " ^ ret_var ^ ");\n" in
            let free = "free(" ^ ret_var ^ ");\n" in
            (save_var, (prev_code ^ call_and_store ^ save_buf ^ copy ^ free))
      (* If action returns anything else, no need to malloc *)
      |_ -> (acall_str, prev_code) )

      | Sast.Noexpr -> ("", "")

let get_form_param (v: Sast.variable_decl) =
  let typ = type_as_string v.vtype in
  typ ^ " " ^ v.vname
```

```
let get_formals params =
  let p_list = List.fold_left (fun str v -> let v_str = get_form_param v in str ^
v_str ^ ", ") "" params in (* need to remove the last comma if function not action*)
  p_list

let rec write_stmt s = match s with
      Sast.Expression(e) ->
      let (expr_str, prec_expr) = get_expr e in
      print_string ("\t" ^ prec_expr); print_string ";\n\t";
      print_string expr_str; print_string ";\n\t"

    | Sast.Block(stmts) -> List.iter (fun s -> write_stmt s) stmts

    | Sast.VarDecl(vdecl) ->
      let vtyp = type_as_string vdecl.vtype in
      let vname = vdecl.vname in let (vexp, prec_expr) = get_expr vdecl.vexpr in
      print_string ( "\t" ^ prec_expr ^ vtyp ^ " " ^ vname ^ " = " ^ vexp);
print_string ";\n"

    | Sast.While(e, s) ->
      let (boolEx, prec_code) = get_expr e in
      print_string("\t" ^ prec_code ^ "\n\t");
      print_string ("while(" ^ boolEx ^ "){ \n\t");
      write_stmt s;
      print_string "\n\n\t}\n\t";

    | Sast.For( ex1, ex2, ex3, s) ->
      let (boolEx, bool_prec_code) = get_expr ex2 in
      let (incr, incr_prec_code) = get_expr ex3 in
      print_string("\t" ^ bool_prec_code ^ "\n\t");
      print_string("\t" ^ incr_prec_code ^ "\n\t");
      write_stmt ex1;
      print_string ("\twhile(" ^ boolEx ^ "){ \n\t");
      write_stmt s;
      print_string (incr ^ ";\n\t");
      print_string "\n}\n\t";

    | Sast.Return(e) ->  let (expr_str, prec_code) = get_expr e in
      let (det, typ) = e in (match typ with
      | Sast.String ->
      (* If return type is a string, malloc *)
      (* MUST FREE IN FUNCTION CALLER *)
      let next_var = get_next_var_name() in
      let malloc_str = "char *" ^ next_var ^ " = " ^
      "malloc(strlen(" ^ expr_str ^ "));\n" ^
      "strcpy(" ^ next_var ^ ", " ^ expr_str ^ ");\n" in
      print_string(prec_code ^ "\t\n");
```

```
      print_string(malloc_str ^ "return " ^ next_var ^ ";\n")
      | _ -> print_string (prec_code ^ "\t\n");
             print_string "return "; print_string expr_str; print_string ";\n")

    | Sast.If(condExpr, ifstmt, elsestmt) ->
        let (condExprStr, condPrec) = get_expr condExpr in
        print_string (condPrec ^ "\nif(" ^ condExprStr ^ ") {\n");
        write_stmt ifstmt;
        print_string ("\n}\nelse {");
        write_stmt elsestmt;
        print_string("}\n")

let write_func funcdec =
  let ret_and_name_str =
  if funcdec.fname = "plot"
      then "int main"
  else begin
      let typ_str = type_as_string funcdec.freturn in typ_str ^ " " ^ funcdec.fname
  end in
  let forms = get_formals funcdec.fformals in
  let len = String.length forms in
  let clean_forms =
      if len > 0 then (String.sub forms 0 ((String.length forms) - 2))
      else forms in (* remove the extra comma from the formals list *)
  print_string ret_and_name_str;
  print_string ("(" ^ clean_forms ^ ")");
  print_string " { \n\t";
  List.iter (fun s -> write_stmt s) funcdec.funcbody;
  print_string " \n} \n"

(* Convert my expression--ie: my name--to use pointer of struct *)
let rec convert_my_expr (e, t) sptr = match e with
      Sast.Access(v, _) -> if v.istrait then v.vname <- sptr
   | Sast.Assign(_, e) -> convert_my_expr e sptr
   | Sast.Unop(_, exp) -> convert_my_expr exp sptr
   | Sast.MathBinop(ex1, _, ex2) -> convert_my_expr ex1 sptr; convert_my_expr ex2 sptr
   | Sast.StrCat(ex1, ex2) -> convert_my_expr ex1 sptr; convert_my_expr ex2 sptr
   | Sast.FCall(_, el) -> List.iter(fun e -> convert_my_expr e sptr) el
   | Sast.ACall(_, _, exps) -> List.iter(fun e -> convert_my_expr e sptr) exps
   | Sast.TraitAssign(v, e) -> convert_my_expr v sptr; convert_my_expr e sptr;
   | _ -> ()

let rec convert_my_stmt (stmt: Sast.statement) sptr =
  match stmt with
      Sast.Expression(e) ->  convert_my_expr e sptr
    | Sast.Block(stmts)  -> List.iter (fun s -> convert_my_stmt s sptr) stmts
    | Sast.VarDecl(v) -> convert_my_expr v.vexpr sptr
    | Sast.While(e, s) -> convert_my_expr e sptr; convert_my_stmt s sptr
```

```
    | Sast.For(ex1, ex2, ex3, s) -> convert_my_stmt ex1 sptr; convert_my_expr ex2
sptr;
                                 convert_my_expr ex3 sptr; convert_my_stmt s sptr
    | Sast.Return(e) -> convert_my_expr e sptr
    | Sast.If(c, ifst, elst) -> convert_my_expr c sptr; convert_my_stmt ifst sptr;
convert_my_stmt elst sptr

let write_action s_ptr_name action =
  let ret_type = type_as_string action.areturn in
  let ret_and_name = ret_type ^ " " ^ action.aclass ^ "_" ^ action.aname in
  let formals = get_formals action.aformals in
  let ptr_name = get_next_var_name() in
  let ptr = ("struct " ^ s_ptr_name ^ "*" ^ ptr_name) in
  let all_formals = (formals ^ ptr) in
  List.iter (fun s -> convert_my_stmt s ptr_name) action.abody;
  print_string ret_and_name;
  print_string ("(" ^ all_formals ^ ")");
  print_string " { \n\t";
  List.iter (fun s -> write_stmt s) action.abody;
  print_string " \n\t} \n\t"

let create_fptrs cname (cact: Sast.action_decl) =
  let fptr = ("(*" ^ cact.aname ^ ")") in
  let freturn = type_as_string cact.areturn in
  let fforms = get_formals cact.aformals in
  let ptr_name = get_next_var_name() in
  let ptr = ("struct " ^ cname^ " *" ^ ptr_name) in
  let all_formals = ("(" ^ fforms ^ ptr ^ ");\n") in
  (freturn ^ fptr ^ all_formals)

let write_structs (cstruct: Cast.class_struct) =
  let dec_struct = "struct " ^ cstruct.sname ^ ";\n" in
  let vtable_def = "struct table_" ^ cstruct.sname ^ " {\n" in
  let func_ptrs = (List.fold_left(fun str f -> let f_str =  create_fptrs
cstruct.sname f in str ^ f_str) "" cstruct.svtable.vfuncs) in
  let ivars = (List.map (fun v -> get_form_param v) cstruct.sivars) in
  let vtable_dec = ("static const struct table_" ^ cstruct.sname ^ " vtable_for_" ^
cstruct.sname ^ " = { \n") in
  let vtable_fncs = List.fold_left(fun str a -> str ^ cstruct.sname ^ "_" ^ a.aname ^
", ") "\t" cstruct.svtable.vfuncs in
  let clean_vtable_fncs = (let len = String.length vtable_fncs in
       if len > 1 then (String.sub vtable_fncs 0 (len-2))
       else vtable_fncs) in
  print_string (dec_struct ^ vtable_def ^ func_ptrs ^ "\n};\n");
  print_string ("struct " ^ cstruct.sname ^ "{\n");
  print_string ("\tconst struct table_" ^ cstruct.sname ^ " *vtable;\n");
  List.iter (fun v -> print_string ("\t" ^ v ^ "; \n")) ivars;
  print_string "\n}; \n";
```

```
  List.iter (fun a -> write_action cstruct.sname a) cstruct.svtable.vfuncs;
  print_string (vtable_dec ^ clean_vtable_fncs ^ "};\n")

let print_code pgm =
      let (cstructs, funcdecs) = pgm in
      print_string "#include <stdio.h> \n#include <string.h> \n#include
<stdbool.h>\n #include <stdlib.h> \n\t";
      print_string ("void *ptrs[" ^ string_of_int !new_count ^ "];\n");
      List.iter (fun c -> write_structs c) cstructs;
      let userFuncs = List.filter (fun f -> f.isLib = false) funcdecs in
      List.iter (fun f -> write_func f) userFuncs;
  flush

  let lexbuf = Lexing.from_channel stdin
  let ast = Parser.program Scanner.token lexbuf
  let sast = analyze_semantics ast
  let cast = sast_to_cast sast
  let _ = print_code cast
```

A.8 codegen.ml

```
open Sast
open Cast
open Semantic_analyzer

(* To handle inheritance, make virtual tables for each object type *)
let class_to_vtable (cdecl: Sast.class_decl) =
      {class_name = cdecl.cname; vfuncs = cdecl.cactions}

(* Convert Storybook classes to C Struct types *)
let class_to_struct (cdecl: Sast.class_decl) =
      let vtable = class_to_vtable cdecl in
      {sname = cdecl.cname; sivars = cdecl.cinstvars; svtable = vtable }

(* Convert classes to structs *)
let sast_to_cast prgm: Cast.prgrm =
  let (c_dcs, f_dcs) = prgm in
  let cstructs  = List.map (fun c -> class_to_struct c) c_dcs in
  (cstructs, f_dcs)
```

A.9 Makefile

```
OBJS = parser.cmo scanner.cmo semantic_analyzer.cmo ast.cmo sast.cmo cast.cmo
codegen.cmo pretty_print.cmo

# TARFILES = Makefile scanner.mll parser.mly \
#     $(TESTS:%=tests/test-%.mc) \
#     $(TESTS:%=tests/test-%.out)
```

```
run : $(OBJS)
	ocamlc -o run str.cma $(OBJS)

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

.PHONY : clean
clean :
	rm -f test/*.c test/*Out.txt test/test_results.txt test/errors.txt
	rm -f parser.ml parser.mli scanner.ml
	rm -f test/tree/test_results.txt
	rm -f *.cmo *.cmi *.out *.diff run
	rm -rf *.dSYM
# Generated by ocamldep *.ml *.mli
semantic_analyzer.cmo : sast.cmo ast.cmo
semantic_analyzer.cmx : sast.cmx ast.cmx
code_gen.cmo : sast.cmo
code_gen.cmx : sast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
run.cmo : scanner.cmo sast.cmo parser.cmi codegen.cmo ast.cmo semantic_analyzer.cmo
run.cmx : scanner.cmx sast.cmx parser.cmx codegen.cmx ast.cmx semantic_analyzer.cmx
sast.cmo : ast.cmo
sast.cmx : ast.cmx
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
parser.cmi : ast.cmo
```

A.10 Test Script

```
#!/bin/sh

cd ../
make clean
make

cd test
echo "Accept Tests:" >> test_results.txt
```

```
failcount=0
passcount=0
if ls $1*_Accept.sbk 1> /dev/null 2>&1
then
	for acceptname in $1*_Accept.sbk;do
	program=`basename $acceptname _Accept.sbk`
	echo "Test: $program" >> errors.txt
	../.⁄run < "$acceptname" > "${program}.c" 2>> errors.txt
	if [ -s "$program.c" ]
	then
	gcc -g -std=c99 $program.c -o $program
	if [ -f "$program" ]
	then
		./$program > "${program}_Out.txt"
		rm $program
		if  diff -q "${program}_Out.txt" "${program}_Exp.txt"
		then
		let "passcount += 1"
		echo ": $program" >> test_results.txt;
		else
		let "failcount += 1"
		echo ": $program -- Compiled and ran, but wrong output." >>
test_results.txt
		echo ": $program -- Compiled and ran, but wrong output."
		fi
	else
		let "failcount += 1"
		echo ": $program -- C Code wouldn't compile" >> test_results.txt;
		echo ": $program"
	fi
	else
	let "failcount += 1"
	echo ": $program -- Storybook didn't compile" >> test_results.txt;
	echo ": $program -- Storybook didn't compile"
	fi
	done
fi

if ls $1*_Reject.sbk 1> /dev/null 2>&1
then
	for rejectname in $1*_Reject.sbk;do
	program=`basename $rejectname _Reject.sbk`
	echo "Test: $program" >> errors.txt
	../.⁄run < "$rejectname" > "${program}.c" 2>> errors.txt
	if [ ! -s "$program.c" ]
	then
	let "passcount += 1"
	echo ": $program" >> test_results.txt
```

```
	else
	let "failcount += 1"
	echo ": $program -- Storybook compiled but should not have" >>
test_results.txt
	echo ": $program -- Storybook compiled but should not have"
	fi
	done
fi

echo "$passcount tests passed"
echo "$failcount tests failed"
rm -rf *.dSYM
```

A.11 Tests

```
==> _99BottlesOfBeer_Accept.sbk <==
Chapter Sing99BottlesOfBeer() returns nothing {
  number bottles is (99).
  repeatwhile(bottles > 0) {
	say(bottles + " bottles of beer on the wall").
	say(bottles + " bottles of beer").
	say("Take 1 down, pass it around").
	bottles is (bottles - 1).
	say(bottles + " bottles of beer on the wall").
  }
}

Chapter plot() returns nothing {
  Sing99BottlesOfBeer().
}

==> AssnBoolF_Accept.sbk <==
Chapter plot() returns nothing {
	tof x is false.
	say(x).
}

==> AssnBoolT_Accept.sbk <==
Chapter plot() returns nothing {
	tof x is true.
	say(x).
}

==> AssnChar_Accept.sbk <==
Chapter plot() returns nothing {
	letter x is 'h'.
	say(x).
}
```

```
==> AssnExpr_Accept.sbk <==
Chapter plot() returns nothing {
	number x is (0).
	number y is (1).
	x is y + (1).
}

==> AssnNmbr_Accept.sbk <==
Chapter plot() returns nothing {
	number x is (5).
	say(x).
}

==> AssnNum_Reject.sbk <==
Chapter plot() returns nothing {
	number x is "hi".
	say(x).
}

==> AssnStr_Accept.sbk <==
Chapter plot() returns nothing {
	words x is "hi".
	say(x).
}

==> AssnStr_Reject.sbk <==
Chapter plot() returns nothing {
	words x is true.
	say(x).
}

==> AssnTwice_Reject.sbk <==
Chapter plot() returns nothing {
	words x is "hi".
	number x is (7).
	say(x).
}

==> boolListTest_Accept.sbk <==
Chapter plot() returns nothing {
	toflist truth is new toflist[10].
	repeatfor(number i is (0).; i < 10; i is i + 1){
		if(i % 2 = 0){
			truth[i] is true.
		}
		else {
			truth[i] is false.
```

```
            }
        }
}

==> CharacterListLoop_Accept.sbk <==
Character Hero(words n; number st; words sp){
   words name is n.
   number strength is st.
   words superpower is sp.

   Action introduceYourself() returns nothing{
       say(my name + ": Hi there! My name is " + my name + " and I have " + my
superpower + "! Nice to meet you guys.").
   }

}

Chapter plot() returns nothing {
       characterlist heroes is new characterlist[5].
       heroes[0] is new Hero("Wonder Woman"; 2000; "the power of flight").
       heroes[1] is new Hero("Spider-Man"; 1500; "Spidey powers").
       heroes[2] is new Hero("Superman"; 100000; "the power of flight and super
strength").
       heroes[3] is new Hero("Invisible Woman"; 200; "the power of invisibility").
       heroes[4] is new Hero("The Flash"; 500; "the power of speed").
       repeatfor(number i is (0).; i < 5; i is i + 1){
              Character Hero h is heroes[i].
              h, introduceYourself().
       }
       say("Narrator: And then all the superheroes joined together to save the
world.").
       say("THE END.").
}

==> CharacterListTest_Accept.sbk <==
Character Hero(words n; number st; words sp){
       words name is n.
       number strength is st.
       words superpower is sp.
}

Chapter plot() returns nothing {
       characterlist heroes is new characterlist[5].
       heroes[0] is new Hero("Wonder Woman"; 2000; "fly").
       Character Hero a is heroes[0].
       say(a's name).
}
```

```
==> CharImproperParams_Accept.sbk <==
Character Monster() {
}

Chapter plot() returns nothing {
	say("hello world").
}

==> charListTest_Accept.sbk <==
Chapter plot() returns nothing {
	letterlist alphabet is new letterlist[26].
	alphabet[0] is 'a'.
	alphabet[1] is 'b'.
	alphabet[2] is 'c'.
	alphabet[3] is 'd'.
	alphabet[4] is 'e'.
	alphabet[5] is 'f'.
	alphabet[6] is 'g'.
	alphabet[7] is 'h'.
	repeatfor(number i is (0).; i < 8; i is i +1){
		say(alphabet[i]).
	}
}

==> CommentMultiline_Accept.sbk <==
~This is a
 multiline
 comment.~

Chapter plot() returns nothing{
  say("Once upon a time...").
}

==> CommentNested_Accept.sbk <==
~ Hello
  ~~ This is a nested comment.
~

Chapter plot() returns nothing {
  say("Once upon a time...").
}

==> CommentNested_Reject.sbk <==
~ Hello
  ~~ This is a nested comment.
  ~ This is another nested comment that will result in rejection.
	Because you cannot have a block comment inside another block comment.
  ~
```

```
~

Chapter plot() returns nothing {
  say("Hello World").
}

==> CommentNoEnd_Reject.sbk <==
~This is a
 multiline
 comment.

Chapter plot() returns nothing {
  say(Once upon a time...).
}

==> CommentSingle_Accept.sbk <==
~~Hello, this is a single line comment.

Chapter plot() returns nothing {
  say("Once upon a time...").
}

==> CompareBool_Accept.sbk <==
Chapter plot() returns nothing {
	say((true = true)).
}

==> CompareBool_Reject.sbk <==
Chapter plot() returns nothing {
	say((-8 < true)).
}

==> CompareChar_Reject.sbk <==
Chapter plot() returns nothing {
	say((-8 < 'a')).
}

==> CompareEqChars_Accept.sbk <==
Chapter plot() returns nothing {
	say(('a' = 'b')).
}

==> CompareEqNums2_Accept.sbk <==
Chapter plot() returns nothing {
	say((8 = 8)).
}

==> CompareEqNums_Accept.sbk <==
```

```
Chapter plot() returns nothing {
	say((-8 = 8)).
}

==> CompareEqNumString2_Reject.sbk <==
Chapter plot() returns nothing {
	say((-8 = "-8")).
}

==> CompareEqNumString_Reject.sbk <==
Chapter plot() returns nothing {
	say((-8 = hi)).
}

==> CompareEqString_Accept.sbk <==
Chapter plot() returns nothing {
	say(("hi" = "hi")).
}

==> CompareGreatEqual1_Accept.sbk <==
Chapter plot() returns nothing {
	say((5 >= 1)).
}

==> CompareGreatEqual2_Accept.sbk <==
Chapter plot() returns nothing {
	say((-5 >= -5)).
}

==> CompareGreatEqual3_Accept.sbk <==
Chapter plot() returns nothing {
	say((-8 >= -6)).
}

==> CompareGreaterFalse_Accept.sbk <==
Chapter plot() returns nothing {
	say((3 > 3)).
}

==> CompareGreaterTrue_Accept.sbk <==
Chapter plot() returns nothing {
	say((3 > 1)).
	}

==> CompareLessEqual1_Accept.sbk <==
Chapter plot() returns nothing {
	say((-5 <= 1)).
}
```

```
==> CompareLessEqual2_Accept.sbk <==
Chapter plot() returns nothing {
	say((-5 <= -5)).
}

==> CompareLessEqual3_Accept.sbk <==
Chapter plot() returns nothing {
	say((-5 <= -6)).
}

==> CompareLessFalse_Accept.sbk <==
Chapter plot() returns nothing {
	say((-5 < -5)).
}

==> CompareLessTrue_Accept.sbk <==
Chapter plot() returns nothing {
	say((-5 < 1)).
}

==> CompareString_Reject.sbk <==
Chapter plot() returns nothing {
	say(-8 >= hello).
}

==> ConcatBooleanandChar_Reject.sbk <==
Chapter plot() returns nothing {
	say( true + 'c' ).
}

==> ConcatBooleanAndString_Accept.sbk <==
Chapter plot() returns nothing {
	say(true + "string").
}

==> ConcatNumberAndBoolean_Reject.sbk <==
Chapter plot() returns nothing{
	say(1 + true).
}

==> ConcatNumberandChar_Reject.sbk <==
Chapter plot() returns nothing{
	say( 1 + 'c' ).
}

==> ConcatNumberAndString1_Accept.sbk <==
Chapter plot() returns nothing {
```

```
	say("hello" + 1).
}


==> ConcatNumberAndString2_Accept.sbk <==
Chapter plot() returns nothing {
	say(1 + "hello").
}

==> ConcatNumberAndString_Accept.sbk <==
Chapter plot() returns nothing {
	say("hello" + 1).
}


==> ConcatStringandBooleanExpr_Accept.sbk <==
Chapter plot() returns nothing {
	say("This is " + (1 = 1)).
}

==> ConcatStringandChar_Accept.sbk <==
Chapter plot() returns nothing {
	say( "hello" + 'i' ).
	}

==> ConcatStringandNumberExpr1_Accept.sbk <==
Chapter plot() returns nothing {
	say("hello" + (1+1)).
	}

==> ConcatStringandNumberExpr2_Accept.sbk <==
Chapter plot() returns nothing {
	say( 1+1 + "hello" + (1+3) ).
	}


==> ConcatStringandNumberExpr3_Accept.sbk <==
Chapter plot() returns nothing {
	say( "hello" + 1+3 ).
	}

==> ConcatStringandString_Accept.sbk <==
Chapter plot() returns nothing {
	say("This is " + "Sparta!").
   }


==> ConcatStringNumberExprandBoolean_Accept.sbk <==
```

```
Chapter plot() returns nothing {
       say( "hello" + 1 + 1 + true).
}

==> FncArgMissingID_Reject.sbk <==
Chapter whatTimeIsIt(number) returns words {
  endwith("It's crunchy time").
}
Chapter plot() returns nothing {
  whatTimeIsIt(1).
}

==> FncConcatArg_Accept.sbk <==
Chapter whatTimeIsIt(words x) returns words {
  endwith("It's " + x + " o'clock." ).
}
Chapter plot() returns nothing {
  say(whatTimeIsIt("hi" + " friend")).
}
==> FncDeclSay_Reject.sbk <==
Chapter say(words w) returns words {
  endwith(w).
}

Chapter plot() returns nothing {
  say("Hello").
}

==> FncHasArgs_Accept.sbk <==
Chapter whatTimeIsIt(number x; number y) returns words {
  endwith("It's " + x + " o' " + y ).
}
Chapter plot() returns nothing {
  say( whatTimeIsIt(9; 5) ).
}

==> FncHasArgs_Reject.sbk <==
Chapter whatTimeIsIt(number x; number y) returns words {
  endwith("It's crunchy time").
}
Chapter plot() returns nothing {
  whatTimeIsIt().
}

==> FncInvalidParamTypes_Reject.sbk <==
Chapter whatTimeIsIt(blah x) returns words {
  endwith("It's crunchy time").
}
```

```
Chapter plot() returns nothing {
  whatTimeIsIt(1).
}

==> FncNoArgs_Accept.sbk <==
Chapter whatTimeIsIt() returns words {
  endwith("It's crunchy time.").
}

Chapter plot() returns nothing {
  say("What time is it?").
  say(whatTimeIsIt()).
}

==> FncNoArgs_Reject.sbk <==
Chapter whatTimeIsIt() returns words {
  endwith("It's crunchy time").
}
Chapter plot() returns nothing{
  whatTimeIsIt(1; 2).
}

==> FncNoPlot_Reject.sbk <==
Chapter noPlot() returns nothing {
  say("Once upon a time").
}

==> FncNoReturnInDecl_Reject.sbk <==
Chapter plot() {
  say("I won't work. I refuse.").
}

==> FncOneArg_Accept.sbk <==
Chapter whatTimeIsIt(number x) returns words {
  endwith("It's " + x + " o'clock." ).
}
Chapter plot() returns nothing {
  say(whatTimeIsIt(9)).
}

==> FncTakingCharacterParam_Accept.sbk <==
Character Princess( words n) {
  words name is n.

  Action goToDinner() returns nothing {
       say (my name + " is at dinner.").
  }
}
```

```
Chapter createMonster(Character Princess p is new Princess("Mulan")) returns
Character Princess{
  Character Princess p is new Princess("Mulan").
  endwith(p).
}

Chapter plot() returns nothing {
  Character Princess x is new Princess("Dummy").
  x is createMonster(x).

  x, goToDinner().
}
==> FncTooFewArgs_Reject.sbk <==
Chapter whatTimeIsIt(number x; number y) returns words {
  endwith("It's crunchy time").
}
Chapter plot() returns nothing {
  whatTimeIsIt(1).
}

==> FncTooManyArgs_Reject.sbk <==
Chapter whatTimeIsIt(number x) returns words {
  endwith("It's crunchy time").
}
Chapter plot() returns nothing {
  whatTimeIsIt(1; 2).
}

==> FncTwoSameName_Reject.sbk <==
Chapter whatTimeIsIt() returns words {
  endwith("It's crunchy time").
}
Chapter whatTimeIsIt() returns words {
  endwith("It's crunchy time").
}
Chapter plot() returns nothing {
  whatTimeIsIt().
}

==> FncUndefined_Reject.sbk <==
Chapter plot() returns nothing {
  print("Once upon a time").
}

==> FncWrongTypeArg_Reject.sbk <==
Chapter whatTimeIsIt(number x) returns words {
  endwith("It's crunchy time").
```

```
}
Chapter plot() returns nothing {
  whatTimeIsIt("hello").
}

==> ForLoop_Accept.sbk <==
Chapter plot() returns nothing{
 repeatfor(number i is (0).; i < 5; i is 6){
       say("hi").
 }
}

==> _GCD_Accept.sbk <==
Chapter GCD(number a; number b) returns number {
  repeatwhile ( not(a=b) ){
       if (a > b) {
       a is (a - b).
       } else {
       b is (b - a).
       }
  }
  endwith(a).
}

Chapter plot() returns nothing {
  say(GCD(30; 60) + "").
}

==> _HelloWorld_Accept.sbk <==
Chapter plot() returns nothing {
  say("Once upon a time...").
}

==> IfElse_Accept.sbk <==

Chapter plot() returns nothing {
  if ( 1 = 2 ) {
       say("so true").
  } else {
       say("so not true").
  }
}

==> IfElseIfElse_Accept.sbk <==
Chapter plot() returns nothing {
  if (1 = 2) {
       say("nothing").
  } else if (2 = 2) {
```

```
        say("2 true!").
    } else {
        say("nothing").
    }
}

==> IfElseSimple_Accept.sbk <==
Chapter plot() returns nothing {
    if (1 = 0) {
        say ("if was true").
    } else {
     say ("if was false").
    }
}

==> IfNestedIfIfElse_Accept.sbk <==
Chapter plot() returns nothing {
    if ( 1 = 1 ) {
        say("so true").
        if ( 1 = 1 ) {
        say("doubly true").
        }
    } else {
        say("so not true").
    }
}

==> IfNoElse_Accept.sbk <==

Chapter plot() returns nothing {
    if ( 1 = 1 ) {
        say("so true").
    }
}

==> IfSimple_Accept.sbk <==
Chapter plot() returns nothing {
    if (1 = 1) {
        say ("if was true").
    }
}

==> listAccessChar_Reject.sbk <==
Chapter plot() returns nothing {
        letterlist alphabet is new letterlist[3].
        alphabet[1] is 'a'.
        alphabet[4] is 'b'.
}
```

```
==> ListAccess_Reject.sbk <==
Chapter plot() returns nothing {
	numberlist scores is new numberlist[3].
	scores[5] is (92).
}

==> listWrongType_Reject.sbk <==
Chapter plot() returns nothing {
	numberlist scores is new numberlist[5].
	scores[4] = 'a'.
}

==> LogicalAnd2_Accept.sbk <==
Chapter plot() returns nothing {
  say(true and false).
}

==> LogicalAnd3_Accept.sbk <==
Chapter plot() returns nothing {
  say(false and true).
}

==> LogicalAnd4_Accept.sbk <==
Chapter plot() returns nothing {
  say(false and false).
}

==> LogicalAnd_Accept.sbk <==
Chapter plot() returns nothing {
  say(true and true).
}

==> LogicalAndBoolExpr_Accept.sbk <==
Chapter plot() returns nothing {
  say(1>2 and true).
}

==> LogicalAndChain2_Accept.sbk <==
Chapter plot() returns nothing {
  say(true and true and false).
}

==> LogicalAndChain_Accept.sbk <==
Chapter plot() returns nothing {
  say(true and true and true).
}
```

```
==> LogicalAndNum_Reject.sbk <==
Chapter plot() returns nothing {
  say(1 and 2).
}

==> LogicalAndOrChain2_Accept.sbk <==
Chapter plot() returns nothing {
  say(false or true and false or true).
}

==> LogicalAndOrChain_Accept.sbk <==
Chapter plot() returns nothing {
  say((false or true) and (false and true)).
}

==> LogicalOr2_Accept.sbk <==
Chapter plot() returns nothing {
  say(true or false).
}

==> LogicalOr3_Accept.sbk <==
Chapter plot() returns nothing {
  say(false or true).
}

==> LogicalOr4_Accept.sbk <==
Chapter plot() returns nothing {
  say(false or false).
}

==> LogicalOr_Accept.sbk <==
Chapter plot() returns nothing {
  say(true or true).
}

==> LogicalOrBoolExpr_Accept.sbk <==
Chapter plot() returns nothing {
  say(false or (1=1)).
}

==> LogicalOrChain_Accept.sbk <==
Chapter plot() returns nothing {
  say(true or true or false).
}

==> LogicalOrDiffTypes_Reject.sbk <==
Chapter plot() returns nothing {
  say(true or 2).
```

```
}

==> LogicalOrStringChar_Reject.sbk <==
Chapter plot() returns nothing {
  say(me or 'u').
}

==> MathAdd_Accept.sbk <==
Chapter plot() returns nothing {
   say(4+5).
}

==> MathDivide_Accept.sbk <==
Chapter plot() returns nothing {
 say(4/2).
}

==> MathMod2_Accept.sbk <==
Chapter plot() returns nothing {
 say(5.5 % 4).
}

==> MathMod_Accept.sbk <==
Chapter plot() returns nothing {
 say(5%4).
}

==> MathMultiply_Accept.sbk <==
Chapter plot() returns nothing {
   say(4*4).
}

==> MathSubtract_Accept.sbk <==
Chapter plot() returns nothing {
  say(4 - 3).
}

==> NoReturn_Reject.sbk <==
Chapter whatTimeIsIt() returns words {
  say("time to return").
}

Chapter plot() returns nothing {
  say(whatTimeIsIt()).
}

==> NotEq2_Accept.sbk <==
Chapter plot() returns nothing {
```

```
   say( not(5 = 4) ).
}

==> NotEq_Accept.sbk <==
Chapter plot() returns nothing {
   say(5 != 4).
}

==> NotEqDifTypes_Reject.sbk <==
Chapter plot() returns nothing {
       say((-8 not = hi)).
}

==> NotGreater_Accept.sbk <==
Chapter plot() returns nothing {
       say(not(-8 > 8)).
}

==> NotGreaterEq_Accept.sbk <==
Chapter plot() returns nothing {
       say(not(8 >= 8)).
}

==> NotLess_Accept.sbk <==
Chapter plot() returns nothing {
       say(not(-8 < 8)).
}

==> NotLessEq_Accept.sbk <==
Chapter plot() returns nothing {
       say(not (-8 <= -18)).
}

==> Not_Reject.sbk <==
Chapter plot() returns nothing {
       say((8 not 8)).
}

==> numberListTest_Accept.sbk <==
Chapter plot() returns nothing {
       numberlist ages is new numberlist[5].
       ages[4] is (6).
       say(ages[4]).
}

==> ObjectActionConcatParam_Accept.sbk <==
Character Monster() {
  Action scare(words scream) returns nothing {
```

```
       say (scream).
  }
}

Chapter plot() returns nothing {
  Character Monster Frank is new Monster().
  Frank, scare("GLABARGHHHHH!" + "wahhhhhhhh").
}
==> ObjectActionWithMyInheritedTrait_Accept.sbk <==
Character Monster( words n; number s ) {
  words name is n.
  number size is s.

  Action scare(words scream) returns nothing {
       say (my name).
  }
}

Character Zombie is Monster(number a) {
  number age is a.

  Action sayhi() returns nothing {
       say("BOO! I'm a Zombie. My name is " + my name).
  }

}
Chapter plot() returns nothing {
  Character Monster Frank is new Monster("Frankenstein"; 99).
  Frank, scare("Aghhhhhhhh").
  Character Zombie Zoe is new Zombie("Zoe"; 6; 16).
  Zoe, sayhi().
}

==> ObjectHasActions_Accept.sbk <==
Character Monster() {
  Action scare(words scream) returns nothing {
       say (scream).
  }
}

Chapter plot() returns nothing {
  Character Monster Frank is new Monster().
  Frank, scare("GLABARGHHHHH!").
}

==> ObjectHasTraits_Accept.sbk <==
Character Monster( words n; number s ) {
  words name is n.
```

```
  number size is s.
}

Chapter plot() returns nothing {
  Character Monster Frank is new Monster("Frankenstein"; 99).
  say(Frank's name).
  say(Frank's size).
}

==> ObjectHasTraitsAndActions_Accept.sbk <==
Character Monster( words n; number s ) {
  words name is n.
  number size is s.

  Action scare(words scream) returns nothing {
       say (my name).
  }
}

Chapter plot() returns nothing {
  Character Monster Frank is new Monster("Frankenstein"; 99).
  Frank, scare("GLABARGHHHHH!").
}

==> ObjectInheritance_Accept.sbk <==
Character Monster( words n; number s ) {
  words name is n.
  number size is s.

Action scare(words scream) returns nothing {
       say (scream + " I'm a Monster").
       say (" My name is " + my name).
  }
}

Character Zombie is Monster(number a) {
  number age is a.

  Action sayhi() returns nothing {
       say("BOO! I'm a Zombie.").
  }
}

Character Person(words nam; number pos){
  words name is nam.
  number position is pos.

  Action run() returns number {
```

```
       endwith(my position + 100).
  }
}

Chapter plot() returns nothing {
  Character Monster Frank is new Monster("Frankenstein"; 99).
  Frank, scare("Aghhhhhhhh").
  Character Zombie Zoe is new Zombie("Zoe"; 6; 16).
  Zoe, sayhi().
  Zoe, scare("RAAAAAWWWWRRRR").
  Character Person Stephen is new Person("Stephen"; 100).
  number mpos is (100).
  number ppos is Stephen, run().
  say(Stephen's name + "'s position is " + ppos + ".").
  say(Frank's name + "'s position is " + mpos + ".").
  if(ppos < mpos){
       say(Stephen's name + " will be eaten by " + Frank's name + ".").
  }
  else {
       say(Stephen's name + " will outrun " + Frank's name + "!").
  }
}

==> ObjectInstInLoop_Accept.sbk <==
Character Animal(words n; words s)
{
       words name is n.
       words species is s.
}

Chapter plot() returns nothing {
       repeatfor(number i is (0).; i < 3; i is i + 1){
              Character Animal dog is new Animal("skip"; "canine").
       }
}

==> ObjectMonster_Accept.sbk <==
Character Monster (words n) {

}

Chapter plot() returns nothing {
  say("hello world").
}

==> ObjectOverrideFunc_Accept.sbk <==
Character Monster( words n; number s ) {
  words name is n.
```

```
  number size is s.

  Action scare(words scream) returns nothing {
       say (scream + "I'm a Monster").
       say (my name).
  }
}

Character Zombie is Monster(number a) {
  number age is a.

  Action sayhi() returns nothing {
       say("BOO! I'm a Zombie.").
       say (my name).
  }
  Action scare(words scream) returns nothing {
       say ("I'm overriding!!!!").
  }
}
Chapter plot() returns nothing {
  Character Monster Frank is new Monster("Frankenstein"; 99).
  Frank, scare("Aghhhhhhhh").
  Character Zombie Zoe is new Zombie("Zoe"; 6; 16).
  Zoe, sayhi().
  Zoe, scare("ZOE SCREAMING").
}

==> ObjectsMultiple_Accept.sbk <==
Character Monster( words n; number s ) {
  words name is n.
  number size is s.

  Action scare(words scream) returns nothing {
       say (my name).
  }
}

Character Zombie (number a; words n) {
  number age is a.
  words name is n.

  Action sayhi() returns nothing {
       say("Hi! I'm a Zombie. My name is " + my name).
  }
}
Chapter plot() returns nothing {
  Character Monster Frank is new Monster("Frankenstein"; 99).
  Frank, scare("GLABARGHHHHH!").
```

```
  Character Zombie Zoe is new Zombie(5; "Zoe").
  Zoe, sayhi().
}

==> ObjectTraitAssignment_Accept.sbk <==
Character Princess( words n; words s ) {
  words name is n.
  words sister is s.
}

Chapter plot() returns nothing {
  Character Princess Elsa is new Princess("Elsa"; "Anna").
  say(Elsa's name).
  say(Elsa's sister).
  Elsa's name is "Anna".
  say(Elsa's name).
}

==> ObjectTraitWrongType_Reject.sbk <==
Character Princess( words n; words s ) {
  words name is n.
  words sister is s.
}

Chapter plot() returns nothing {
  Character Princess Elsa is new Princess("Elsa"; "Anna").
  say(Elsa's name).
  say(Elsa's sister).
  Elsa's name is (9).
  say(Elsa's name).
}

==> PrincessCharacterAsParam_Accept.sbk <==
Character Princess( words n) {
  words name is n.

  Action introduceSelf() returns nothing {
       say("Hi, I'm " + my name + "!").
  }
}


Character LittleMermaid is Princess( Character Princess p ) {
       words ability is a.

  Action talkToPrincess( Character Princess b) returns nothing {
       say("Hi " + b's name).
  }
```

```
}

Chapter plot() returns nothing {
  Character Princess Cinderella is new Princess("Cinderella").
  Character LittleMermaid Ariel is new LittleMermaid( Cinderella ).
  Ariel, talkToPrincess( Cinderella ).
}
==> Princesses_Accept.sbk <==
Character Princess( words n) {
  words name is n.

  Action goToDinner() returns nothing {
       say (my name + "is at dinner.").
  }
}

Character LittleMermaid is Princess( Character Princess p ) {
  Character Princess prin is p.

  Action talkToPrincess( Character Princess b) returns nothing {
       say("Hi " + b's name + ",  I'm " + my name + "!").
  }

}

Chapter plot() returns nothing {
  Character Princess Cinderella is new Princess("Cinderella").
  Character LittleMermaid Ariel is new LittleMermaid( "Ariel"; Cinderella ).
  Ariel, talkToPrincess(Cinderella).
}
==> PrincessesAudition_Accept.sbk <==
Character Princess( words n; number a; tof f) {
  words name is n.
  number age is a.
  tof famous is f.

  Action introduceSelf() returns nothing {
       say( my name + ": Hi, my name is " + my name + "!").
  }

  Action audition(words part; words experience; words movie) returns nothing {
   if(my famous = true) {
       say(my name + ": I am auditioning for the part of " + part + " in " + movie +
".").
       say("In case you didn't recognize me, I was in Disney's " + experience + ".").
       }
       else {
```

```
	say(my name + ": I'm auditioning for the part of " + part + " in " + movie +
".").
	say("I don't have any experience, but I think I have great potential! Plus,
all of these old princesses only know how to play roles that depend on men. I can be
a strong, independent, and fearless princess!!").
	}
  }
}

Character DisneyPrincess is Princess( words m ) {
	words movie is m.
  Action salary(number b) returns number {
	number incSal is 2 * b.
	say(my name + ": Just so you know, Walt payed me " + b + " dollars so I expect
at least " + incSal).
	endwith(b).
  }
}

Chapter findActress(tof f; number s) returns nothing {
	if(f = true and s < 10000){
	say("Producers: You're hired!").
	}
	else if(f = false) {
	say("Producers: You're hired! And we'll pay you " + s * 2 + " dollars!").
	}
	else{
	say("Producers: No thanks.").
	}

}

Chapter plot() returns nothing {

  Character DisneyPrincess Aurora is new DisneyPrincess( "Aurora"; 16; true;
"Sleeping Beauty" ).
  Character Princess Anna is new Princess( "Anna"; 16; false).
  Aurora, introduceSelf().
  Aurora, audition("Elsa"; Aurora's movie; "Frozen").
  number money is Aurora, salary(10000000).
  findActress(Aurora's famous; money).
  Anna, introduceSelf().
  Anna, audition("Anna"; "No exprience"; "Frozen").
  findActress(Anna's famous; 5000).
}

==> PrintBool_Accept.sbk <==
Chapter plot() returns nothing {
```

```
   say(true).
}

==> PrintFncRet_Accept.sbk <==
Chapter getSum() returns number {
   endwith(5 + 3).
}
Chapter plot() returns nothing {
   say(getSum()).
}

==> PrintNum_Accept.sbk <==
Chapter plot() returns nothing {
   say(5).
}

==> PrintVar_Accept.sbk <==
Chapter plot() returns nothing {
   words x is "hi".
   say(x).
}

==> ReAssnNum2_Accept.sbk <==
Chapter plot() returns nothing {
       number x is (5).
       x is (6).
       x is (10).
}

==> ReAssnNum_Accept.sbk <==
Chapter plot() returns nothing {
       number x is (5).
       number y is (1).
       number z is (x + y).
       say(z).
}

==> ReAssnStr_Accept.sbk <==
Chapter plot() returns nothing {
       words x is "hi".
       x is "bye".
       x is "cow".
       say(x).
}

==> RecursionSimple_Accept.sbk <==
Chapter gcd(number a; number b) returns number {
  if (b = 0) {
```

```
      endwith (a).
  }
  endwith(gcd(b; a %b)).
}

Chapter plot() returns nothing {
  say(gcd(54; 24)).
}

==> ReturnEndswithWithoutParens_Accept.sbk <==
Chapter plot() returns nothing {
  say("Once upon a time...").
      }

==> ReturnInvalidType_Reject.sbk <==
Chapter plot() returns blah {
   endwith(0).
}

==> ReturnNum_Accept.sbk <==
Chapter fncReturnsNumber() returns number {
      endwith(1).
}

Chapter plot() returns nothing {
  say(fncReturnsNumber()).
}

==> ReturnVoid_Accept.sbk <==
Chapter plot() returns nothing {
  say("nothing returned").
}

==> ReturnVoid_Reject.sbk <==
Chapter plot() returns nothing {
  endwith (0).
}

==> ReturnWrongStringNotNumber_Reject.sbk <==
Chapter moo() returns number {
  say ("Once Upon a time").
  endwith ("cow").
}

Chapter plot() returns nothing {
      moo().
}
```

```
==> ScopeSimple_Reject.sbk <==
Chapter plot() returns nothing {
  if (1=1) {
       number five is 5.
  }
  say (five).
}

==> ScopingObjects_Accept.sbk <==
Character Monster( words n; number s ) {
  words name is n.
  number size is s.

  Action scare() returns nothing {
       say (my name).
  }
}

Chapter createMonster() returns Character Monster{
  Character Monster Frank is new Monster("Frankenstein"; 99).
  endwith(Frank).
}

Chapter plot() returns nothing {
  Character Monster f is new Monster("Dummy"; 69).
  f is createMonster().
  f, scare().
}

==> ScopingObjectsNoReturn_Reject.sbk <==
Character Monster( words n; number s ) {
  words name is n.
  number size is s.

  Action scare() returns nothing {
       say (my name).
  }
}

Chapter createMonster() returns nothing{
  Character Monster Frank is new Monster("Frankenstein"; 99).
}

Chapter plot() returns nothing {
  Character Monster f is new Monster("Dummy"; 69).
  f is createMonster().
  f, scare().
}
```

```
==> TraitInheritRightHandSide_Accept.sbk <==
Character Princess( words n; number a; tof f) {
  words name is n.
  number age is a.
  tof famous is f.

  Action introduceSelf() returns nothing {
       say( my name + ": Hi, my name is " + my name + "!").
  }
}

Character DisneyPrincess is Princess( words m ) {
       words movie is m.

  Action growup() returns nothing {
       my age is (my age + 1).
  }
}


Chapter plot() returns nothing {

  Character DisneyPrincess Aurora is new DisneyPrincess( "Aurora"; 16; true;
"Sleeping Beauty" ).
  Aurora, growup().
  say(Aurora's age).
}

==> TraitOverride_Reject.sbk <==
Character Princess( words n; number a; tof f) {
  words name is n.
  number age is a.
  tof famous is f.

  Action introduceSelf() returns nothing {
       say( my name + ": Hi, my name is " + my name + "!").
  }

  Action audition(words part; words experience; words movie) returns nothing {
   if(my famous = true) {
       say(my name + ": I am auditioning for the part of " + part + " in " + movie +
".").
       say("In case you didn't recognize me, I was in Disney's " + experience + ".").
       }
       else {
       say(my name + ": I'm auditioning for the part of " + part + " in " + movie +
".").
```

```
       say("I don't have any experience, but I think I have great potential! Plus,
all of these old princesses only know how to play roles that depend on men. I can be
a strong, independent, and fearless princess!!").
       }
  }
}

Character DisneyPrincess is Princess( words m ) {
  words movie is m.
  tof famous is true.

  Action salary(number b) returns number {
       number incSal is 2 * b.
       say(my name + ": Just so you know, Walt payed me " + b + " dollars so I expect
at least " + incSal).
       endwith(b).
  }
}

Chapter findActress(tof f; number s) returns nothing {
       if(f = true and s < 10000){
       say("You're hired!").
       }
       else if(f = false) {
       say("Producers: You're hired! And we'll pay you " + s * 2 + " dollars!").
       }

}

Chapter plot() returns nothing {

  Character DisneyPrincess Aurora is new DisneyPrincess( "Aurora"; 16; true;
"Sleeping Beauty" ).
  Character Princess Anna is new Princess( "Anna"; 16; false).
  Aurora, introduceSelf().
  Aurora, audition("Elsa"; Aurora's movie; "Frozen").
  number money is Aurora, salary(10000000).
  findActress(Aurora's famous; money).
  Anna, introduceSelf().
  Anna, audition("Anna"; "No exprience"; "Frozen").
  findActress(Anna's famous; 1000).
}

==> WhileLoop_Accept.sbk <==
Chapter plot() returns nothing{
 number x is (10).
 repeatwhile( x > 5){
       say("hi").
```

```
      x is (x - 1).
 }
}
```

A. Project Log

Committers:

ast.ml: Anna, Beth, Nina

scanner.mll: Anna, Beth, Nina

parser.mly: Anna, Beth, Nina

sast.ml: Anna, Beth, Nina, Pratishta

semantic_analyzer.ml: Anna, Beth, Nina, Pratishta

cast.ml: Anna, Beth

pretty_print.ml: Anna, Beth, Pratishta, Nina

codegen.ml: Anna, Beth, Pratishta

Makefile: Anna, Beth, Nina, Pratishta

test.sh: Anna, Beth, Nina, Pratishta

tests/: Anna, Beth, Nina, Pratishta