Stint

String and Integer Language

Final Report

Jiang Wu, jw3026

Ningning Xia, nx2120

Sichang Li, sl3484

Tingting Ai, ta2355

Yiming Xu, yx2213

Dec. 19th, 2012

Table of Contents

1. Introduction

1.1 Background and Motivation

String is one of the most commonly used types in programming languages. While in traditional languages we usually access part of a string by index of character or pattern matching with REs, we believe we should be able to do that more intelligently. Strings are usually mixtures of blocks of characters and integers in most real-life applications, so we should be able to access parts of a string by what they are instead of index range. Code becomes complicated in order to do this in traditional languages. As for scripting languages, they are usually a little bit too generalized and heavy-weighted in order to solve such problems.

Motivated by these factors, in this project we try to design a language that provides easy ways to process strings directly and conveniently by specifically designed operators and built-in functions.

1.2 Objectives of Stint

The programming language Stint is a text-processing language, which contains a useful collection of built-in string manipulation functions. Stint provides solutions for manipulations and conversions between the most common-used data types: string and int. While processing textual data and test file, Stint shows its advantages in several aspects:

- Intelligently distinguish between string and int.
- Provide a simple and direct way to do mathematical operations for numbers in string without extracting them and transferring data type.
- Define more effective text-manipulation via operators. Meanwhile, maintain traditional string features and functions.
- Make input and output functions more convenient.

2. Language Tutorial

2.1 Basics

Stint has three types of variables: string, int, and boolean. The return type of a function can be void, int, string or boolean.

2.1.1 Comments

Comments are notes by the programmer in the code and will be ignored by the compiler. Comments begin with /* and end with */.

```
… /*This is a comment*/
```

2.1.2 Declarations

Both global and local variables must be declared before being used. Declarations define the type of a variable.

```
string str;           /*Declare string variable */
int num;              /*Declare int variable */
boolean value;        /*Declare boolean variable */
```

2.1.3 Assignments

We use "=" to assign the values to a corresponding variables or expressions. The values can be string constants, integer constants, boolean constants or expressions. The values of both sides should be in the same type. The most common assignments are as following.

```
/*Assign "abc" to a string variable 'str'*/
string str = "abc";
/*Assign ("abc" + "def") to addstr*/
string addstr = "abc" + "def";
/*Assign 123 to an int variable 'num'*/
int num = 123;
/*Assign (1+2) to an int variable 'addnum'*/
int addnum = 1+2;
/*Assign (1+2) to an int variable 'addnum'*/
boolean value = true;
```

Stint provides solutions for assigning values to expression.

```
string s = "abc123def456";  /*Declare a string 's'*/
s<|0|> = "cba"; /*Assign "cba" to the first string of s*/
s.<|0|> = 0;    /*Assign 0 to the first int of s*/
s[4] = "g";     /*Assign "g" to the 4th index of s*/
s[5,2]= "hi";   /*Assign "hi" from 5th index*/
```

2.1.4 Operators

Stint provides multiple operators on the number, string, boolean and expression.

The operators on numbers contain arithmetic operators (+, -, * /) and relational operators (>, <, >=, <=, !=, ==).

The operators on strings contain append(+), delete(-), sub-string([index], [index1,index2]), integer set extractor(.<|index|>), string set extractor(<|index|>), string splitter (string|sep), string finder (string #), string remover (~ expression). (See more details in Section 3.3.3)

```
str = “ab” + “cd”;      /*Append “cd” to the end of “ab*/
str = “ab” + “cd” @ 0;  /*Append “cd” to the  str*/

str = “cabc” - “c”;     /*Delete first “c” in str*/
str = “cabc” - “c” @ 1;  /*Delete first “c“ after 1st index*/

/*Get substring from the 1st index */
str = “abcd”; str = str[1];
/*Get length 2 substring from 1st index*/
str = “abcd”; str = str[1,2];
str = “ab12cdef”;
int num = str.<|0|>;    /*Get the set of integer with index 0*/
string subs = str<|0|>; /*Get the set of string with index 0*/

string str = “this is a sentence in string”;
/*Split str by “ ”, return amount of separated set num = 6*/
int num = str | “ ”;
string str = “this is his thesis”;
/*Return the number of substr in string, num = 4*/
int num = str # “is”;
/*Find and change the substr in the 2nd index*/
str<2> = “er”;

string str = “a12b56c”;
~str.<0>;   /*Remove the integer with the index 0 in the str*/
~str[0,3];  /* Remove a length 3 substr from index 0*/
```

The operators on boolean contain And, Or, and Not.

2.1.5 Expressions

Expressions are some operations that can be executed and evaluated. Literals, variable names and binary and unary operations are expressions, among others.

2.1.6 IOStream

Standard input/ouput

```
/* Print a sentence to screen */
std << “This is a sentence”;
/* Read and store input as variable str */
std >> str;
```

File input/output

```
string in_buffer = "";
/*Read a line from the text file "input_file" and store it into
in_buffer*/
input_file >> in_buffer;
string out_buffer = "A sentence for output";
/*Output the content of out_buffer
 *to the text file "output_file" */
output_file << out_buffer;
```

2.1.7 Functions

A function in *Stint* is typically the same as a function in C or a method in Java. It takes a series of parameters and returns a value after execution of the code within the function.

```
/* Function "add" takes two arguments of int type, and returns
the calculated result of int type */
int add (int a, int b)
{
/* The type of result (a+b) must be int, compatible with the
return type of the "add function" */
  return a + b;
}
```

All functions have global scope within the program it was create. And the names of functions are unique among all the functions and variables.

The return type must be one of **void**, **int**, **string**, and **boolean**.

2.1.8 Invoking Functions

Defined functions can be used by invoking them. Invoking a function requires specifying the values of variables as input. These values must be in the same type with arguments when defining the function. The output of the invoked function can be assigned to a variable which is in the same type with it, using the standard assignment operator in Stint.

```
void main()
{
   int num;        /* Declare a variable 'num' of int type */
   num = add (39, 3);
   std << num;     /* Output the value of num in the std way */
   return;         /* The end of the main function */
}
```

2.2 Compiling and Running Stint

In order to run Stint, the user should go into the "compiler" directory, and type "make" in the terminal to get the executable of Stint. Then type the following command to compile a **.sti** source file:

./stint <option> <source file>

Any invalid command will lead to the following help message:

```
Usage: ./stint <option> <source file>

where possible options include:

        -a file.sti          (Output the AST of source file)
        -s file.sti          (Output the SAST of source file)
        -j file.sti          (Compile source code to .java file)
        -c file.sti          (Compile source code to .java and .class files)
        -help                (Print a synopsis of standard options )
        -v                   (Display version information)
```

3. Language Reference Manual

3.1 Lexical Conventions

3.1.1 Identifiers

An identifier can only include 26 alphabets in lower or upper case (a-z, A-Z), digits (1-9), and underline (‘ _ ’); the first character must be alphabets in lower case.

3.1.2 Comments

Double-slash (“ `/* */` ”) is the only comment sign.

```
/* This is a comment
 * and for multiple lines */
open “data.txt”;  /* can append to a statement */
```

3.1.3 End-of-Statement

Semicolon (“ `;` ”) is used to indicate the end of one statement.

3.1.4 Keywords

The following identifiers are reserved as keywords/special function and may not be used otherwise:

```
int     string    boolean
if      else      return
while   open      close
false   true      std
break   void
```

3.1.5 Constants

In *Stint*, there are 3 types of constants: integer constant, string constant and boolean constant.

- Integer Constants
  An integer constant consists of a sequence of numbers without a decimal point.
  ```
  int n = 100; /* 100 is an integer constant */
  ```
- String Constants
  A string constant is enclosed in double quote marks (“ `" "` ”).
  ```
  /* below are examples of string constants */
  string str = “This is a string”;
  string title = “Product Name\tPrince\t\n”;
  ```
  Stint also contains the following escape sequence as constants in order to represent some special characters such as new line, tab and backslash, as shown in Table 1:

Table 1. Escape Sequence in String Constants

Character Name	Escape Sequence
Newline	\n
Horizontal tab	\t
Double quotation marks	\”
Backslash	\\

- Boolean Constants
  The reserved boolean constants are **true** and **false**.
  ```
  boolean a = true;
  boolean b = false;
  ```

3.2 Data Type & Conversion

3.2.1 Data Types

There are three types of data in *Stint* – integer, string and boolean.

- Integer
  In *Stint*, the only supported integer type is **int**, which represents a sequence of digits. Integers are signed and with fixed size of 32 bits.
- String
  The string in *Stint* is defined as a sequence of ASCII characters enclosed in double quotes, e.g. “abc”. A string can have potentially unlimited length as long as the computing resource, e.g. memory, allows.

  A string in *Stint* is a dynamic structure, which keeps a list of sub-strings or integers. *Stint* predefined some special operators in order to modify or search the sub-strings dynamically, e.g. “ .<| |> ” (extracting), “ | ” (splitting), “ # ” (highlighting), etc. (See details in Section 4.3)
- Boolean
  A boolean type can take either **true** or **false** as its value, which is used for condition determination.

  Some main features of the three data types are illustrated in the Table 2 below.

Table 2. Summary of Data Type in Stint

Primitive Types	Length	Range	Default Value
int	32 bits	-2147483648 to 2147483647	0
string	>= 8 bits	Any permutation of ASCII characters	“” (empty)
boolean	8 bits	**true** or **false**	**false**

3.3.2 Type Conversion

In *Stint*, there are two types of conversions, i.e. from integer to string and from boolean to string, both of which are done implicitly. No explicit conversion is allowed. All other invalid type conversions, e.g. from string to integer or from integer to boolean, will lead to a compilation error.

- Integer-to-String
  An integer can be converted to a string implicitly, but not vice versa. After casting, the new variable name cannot be the same as the original one; otherwise there will be a compilation error.
  ```
  int num = 12;
  string snum = num; /*snum="12" after conversion*/
  string snum = "12";
  int num = snum;    /*this assignment is invalid*/
  ```
  Besides assigning, the implicit conversion can be done automatically as well when an integer is used in a string operation.
  ```
  int num = 12;
  string snum = "abc" + num; /*snum = "abc12"*/
  ```
  Even though a string cannot be converted into an integer directly, this conversion can be done by using a special operator defined in *Stint*, i.e. " `.< >` ". (See details in Section 3.3.3)
- Boolean-to-String
  A boolean value can be converted to a string, but not the other way around.
  ```
  boolean b = true;
  string s= b;      /*s = "true" when b is true*/
  ```
  Trying to convert an integer or a string to a boolean will lead to a compilation error though it seems make sense.
  ```
  int n = 1;
  boolean b = n;    /*Compilation error*/
  ```

3.3 Expressions & Operators

3.3.1 General Expression

- Assignment: *destin* = *source*
  Copy and assign the value of *source* to *destin*.
- Parenthesis: (*expression*)
  Expression in parenthesis will be assigned a higher precedence than those isn't in.
- Equality: *expr1* == *expr2* *expr1* != *expr2*
  Check whether *expr1* and *expr2* have the same value.
- Input Stream: *source* >> *expr*
- Output Stream: *destin* << *expr*
  The subject of stream is indicated by *source*/*destin* that can be file name or **std** for standard I/O. Read a whole line from *source* to *expr* or write the content of *expr* to *destin*.
  The input stream operator will return Boolean value to indicate whether the reading is successful.
- Function Call: *func-name* (*arg1, arg2, …*)

3.3.2 Numeric Operator

- Arithmetic Operator: *expr1* + *expr2* *expr1* – *expr2* *expr1 * expr2*   *expr1 / expr2*

Basic arithmetic operation for integer: add, subtraction, multiplication, and division.

- Relational Operator: *expr1 > expr2*   *expr1 >= expr2*

  expr1 < expr2   *expr1 <= expr2*

  Compare expr1 and expr2 with above relations. Return a Boolean value.

3.3.3 String Operator

- Position Indicator: @ *position*

  Indicate specific position where the operation will be done. Use with string operator "+" and "-" (See examples in 4.3.2/4.3.3).

- Append (Insert): *expr1 + expr2*

  expr1 + expr2 @ pos

  Append *expr2* to the end of *expr1*. When using "@", add *expr2* before the character indicated by *pos* of *expr1* (use as insert).

  Use a number smaller than the lowest index (negative number) or larger than the highest index will put *expr2* at the head/tail of *expr1*.

```
str = "ab" + "c";        /* str = "abc" */
str = "ab" + "c" @ 0;    /* str = "cab" */
```

- Delete: *expr1 – expr2*

  expr1 – expr2 @ pos

  Delete the first *expr2* in *expr1*. When using "@", delete the first *expr2* occurring after the character indicated by *pos* in *expr1* (include this character).

  The operator will give *expr1* back if no matching of *expr2* found in *expr1*.

```
str = "cabc" - "c";          /* str = "abc" */
str = "cabc" - "c" @ 1;      /* str = "cab" */
```

- Sub-string Extractor: [*index*]

  [*index* , *length*]

  Get the sub-string of a single character at *index*, or starting from *index* of *length*.

```
str = "abcd"; str = str[0];       /* str = "a" */
str = "abcd"; str = str[1, 2];    /* str = "bc" */
```

- Integer Set Extractor: .<| *index* |>

  Get the set of integer with *index*.

```
string str = "a12b56c";
int integer = str.<|0|>;     /* integer = 12 */
```

- String Set Extractor: <| *index* |>

  Get the set of string with *index*. Set of string may vary with user's setting (see detail in 4.37/4.38).

- String Splitter: *string* | *sep*

  Change the set of *string* as sub-strings separated by *sep*. Return the amount of sets having been split.

  In default, the string was separated by integer sets. Use `str | ""` can cancel previous splitter can go back to default splitting.

```
string str = "this is a sentence in string";
int num = str | " ";    /* num = 6 */
string temp = str<|0|>;/* temp = "this" */
temp = str<|4|>;        /* temp = "sentence" */
```

- String Finder: *string # substr*

  Change the set of *string* as all sub-strings equaling *substr*. Return the number of *substr* in *string*.

  Every time the string is modified, the indicated key will be eliminated. User has to reset key to find if need.

```
string str  = "this is his thesis";
int num = str # "is";   /* num = 4 */
str<|2|> = "er";   /* str = "this is her thesis" */
```

- String Remover: *~ expression*

  Remove *expression* from the string it belongs to.

```
string str = "a12b56c";
~str.<|0|>;        /* str = "ab56c" */
~str[0, 3];        /* str = "6c" */
```

3.3.4 Boolean Operator

- And: *expr1 && expr2*
- Or: *expr1 || expr2*
- Not: ! *expression*

3.3.5 Precedence

The precedence of above operators is typically in the following order from high to low. All three boolean operators are in the same order, so they are not included in the Table 3 below.

Table 3. Precedence of Stint Operators

Precedence	Operator
1 (highest)	( )
2	|, #
3	[ ], <| |>, .<| |>
4	*, /
5	+, -
6	>, >=, <, <=, ==, !=
7 (lowest)	=, <<, >>

3.4 Declarations

3.4.1 Variable Declaration

Variables must be declared before they are used in the program. A variable declaration has the following form:

var_type var_name;

The *var_type* can be **int**, **boolean** or **string**. The *var_name* can be any valid identifier. If a variable is declared, in the following assignment, value assigned to the variable must have exactly the same type as declared. Otherwise, it's a syntax error. A single semicolon must be followed by the declaration.

Variables can also be initialized during the declaration. A declaration with initialization has the following form:

var_type var_name = expression;

The expression must have exactly the same type as *var_type*. Otherwise, it's a syntax error.

3.4.2 Function Declaration

A function declaration has the following form:

return-type function-name (type parameter1, type_parameter2….)

The detail of this part is talk in Section 3.6.1.

3.5 Statements

3.5.1 Expression Statement

An expression statement is composed of primary expressions separated by a single semicolon at the end of each expression.

3.5.2 If Statement

If statement consists of keywords **if** and **else**. It has the following two varieties:

```
if ( expression ) {
    statement
}
if ( expression ) {
    statement1
} else {
    statement2
}
```

The expression must be of **boolean**. Statements must be surrounded by open and closed curly bracket. In the first case, if the *expression* is evaluated to **true**, then *statement* is executed. Otherwise statements after the *if* statement is executed. In the second case, if the *expression* is evaluated to **true**, then *statement1* is executed, otherwise *statement2* is executed.

3.5.3 While Statement

While statements consists of keyword **while** and it allows a statement to be executed for any number of times. It has the following format:

```
while ( expression ) {
    statement
}
```

The expression must be **`boolean`**. Statements must be surrounded by open and closed curly bracket. The *expression* is evaluated before the execution of the statement and *statement* will be executed until the expression is evaluated to **`false`**.

3.5.4 Break Statement

Break statement consists of keyword **`break`**. It's used to jump out of the while loop. It's followed by a single semicolon. The following is an example of using break statement:

```
while ( true ) {
    break;
}
```

3.5.5 Open Statement

Open statement consists of keyword **`open`**. It's used to open a file. It takes a string as the file name. It has the following format:

open filename

3.5.6 Close Statement

Close statement consists of keyword **`close`**. It's used to close a file. It takes a string as the file name. It has the following format:

close filename

3.5.7 Return Statement

Return statement consists of keyword **`return`**. A function must have a return statement to return its value to its caller. It can return an expression that is evaluated to type **`int`**, **`bool`** or **`string`**, or it can return nothing when the function uses **`void`** as its return type.

```
return expression;
return;
```

The return statement must be followed by a single semicolon.

3.6 Functions

3.6.1 Function Definition

A function in *Stint* is typically the same as a function in C or a method in Java. It takes a series of parameters and returns a value after execution of the code within the function.

The signature of a function is as follows:

```
return-type function-name ( type parameter1, type_parameter2.... )
{
    statements
    ……
    return value;
}
```

All functions have global scope within the program it was create, and in other words, a function cannot be declared inside another function. And the names of functions are unique among all the functions.

The return type must be one among **`void`**, **`int`**, **`string`**, and **`boolean`**. We suggest to use **`boolean`** as the return type to replace **`void`**, **`true`** indicating execution success, otherwise **`false`**.

The body of a function must be included in a " `{ }` " just like C and Java do. The variables defined within the body of a function have local scope. That's to say, all variables have the same local scope. We don't recommend creating variables in the body of a function except for temporary variables. If a function with variable definitions within its body gets called multiple times, these variables will be reinitiated.

3.6.2 Main Function

An executable program should include an entry function **`void`** `main()`, where the entire program will be started.

However, `main()` is not imperative by compiler. A file without main function can be accepted but will not be able to run.

3.6.3 Function Return

To simply make sure the function can return values with defined return type, *Stint* force the last statement of the function to be **`return`**. Otherwise, the compiler will complain about this.

3.6.4 Build-in Functions

- **`boolean`** `PrintFile(`**`string`** `s,` **`string`** `filename)`

  This function is used to print String s to the end of a text file. It can be a file that hasn't been opened yet, the function will automatically do the rest of work.
- **`string`** `toUpperCase(`**`string`** `s)`

  The return string will be the same as input screen but in all upper case.
- **`string`** `toLowerCase(`**`string`** `s)`

  Similar as toUpperCase(), this one return string s with all characters in lower case.
- **`string`** `replceAll(`**`string`** `dest,` **`string`** `s)`

  This function is used to replace all the matched sections in dest to s.

3.7 Scoping and Naming

3.7.1 Scoping

In *Stint*, the scope is defined as the region within a program in which a certain identifier/function can be accessed.

Identifiers within a function are defined as local scope, i.e. they can only be accessed within the function body in which it is declared. Identifiers outside any functions (at same level with functions) will be treated as global variables that can be accessed anywhere after declared. Identifiers/variables cannot be accessed until declared.

Functions are of global scope from the position they are defined till the end of code. Function calls are possible as long as the target function has been defined before the current position.

3.7.2 Naming

Naming of variables and functions are effective and unique within its scope. The local variable has higher priority to be accessed than the global one when they have same name.

4. Project Plan

4.1 Project Processes

4.1.1 Planning

In the first meeting, the team decided the form, place and frequency of meetings, and a generalized time plan for the project. Members would communicate through emails to decide specific time for each meeting. During each meeting, members should report their progress and discuss the direction and distribution of remaining work.

4.1.2 Specification

Considering the drawbacks of some commonly used languages, we decided the core functionality and major features of our language in the second meeting as presented in our proposal. Later on, we thoroughly discussed the details of our language, e.g. format, names, operators, functions, scope, etc., and carefully considered the feasibility of proposed functionalities, and the way to implement them if feasible. Then, we came up with our Language Reference Manual (LRM).

4.1.3 Development

Based on the structure of the Micro C compiler, we built up the skeleton of our compiler. Team members built the Scanner and Parser together. However, when constructing the rules, we found that some special operator defined in our original LRM will lead to shift/reduce conflicts. So we refined the signs for those operators.

The remaining modules of the compiler, as well as test suites are developed in a parallel mode. We divided into three small groups and each group focus on one or two of the remaining modules. In this phase, we met more frequently because communications between each group are important to achieve the consistency and integrity of the components. The programming style guide is used in this phase to ensure the consistency and make the code easier to read for other members. Each component was tested as much as possible before integration.

After all components were done, the compiler was built up by Makefile, and we began testing of the whole compiler.

4.1.4 Testing

Testing is the longest phase in the project, which began alongside with the development phase. Each member tested their own code before pushing onto the server.

Testing of the whole compiler is tough, but vital. Yiming developed a C program for regression tests. Whenever a bug was fixed, the C program will test all the test cases by the Stint compiler and compare the outputs with expected results. More details of testing are discussed in Section 6 of this document.

4.2 Programming Style Guide

All source code files use standard indented format.

Function and variable names: all lower case letters and underscores.

Module and type names: capitalized initial letter.

Tokens: all upper case letters.

Pattern matching: keep aligned.

Avoid making lines longer than 120 characters.

Parentheses should be used in expressions to specify order of precedence, and to help others to read the code easily.

Commas and semicolons are always followed by whitespace.

Declare variables as close as possible to where they are used.

4.3 Project Timeline

The Gantt chart of this project is shown in Figure 1 below.

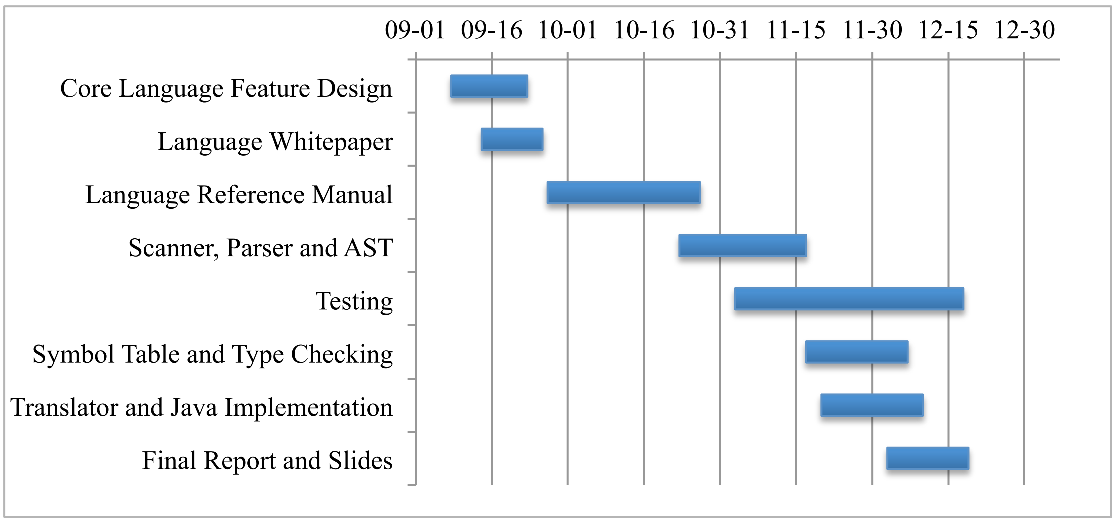


Figure 1. Gantt Chart of the Stint Project

4.4 Team Responsibilities

All team members participated actively throughout this project. All members collaborated in the Language Whitepaper, Language Reference Manual, the implementation of Scanner and Parser, and the Final Report. In the latter half of compiler implementation, the team is divided into sub-groups of one or two members, the compiler modules are distributed to each sub-group. The detailed responsibilities of each member are listed in.

Table 4. Individual Responsibilities

Team member	Compiler Implementation	Final Report
Jiang Wu	Scanner, Parser, Java Library, Build-In Functions, Test Cases	Section 3, Section 5, Section 7
Ningning Xia	Scanner, Parser, Translator, Test Cases	Section 2, Section 3, Section 7
Sichang Li	Scanner, Parser, Type Checking, AST, Test Cases	Section 3, Section 5, Section 7
Tingting Ai	Scanner, Parser, Translator, Command Line Interface, Test Cases	Section 1, Section 4, Section 7
Yiming Xu	Scanner, Parser, Type Checking, SAST, Test Scripts, Test Cases	Section 3, Section 6, Section 7

4.5 Software Development Environment

The compiler for Stint is written in Objective Caml (OCaml) and Java, and is developed over Cygwin on Windows and Mac OS. Table 3 details the tools and languages used in this project.

Table 5. Tools and Languages

Tool / Language	Description
Ocaml 4.00.0	Primary coding language for the Stint compiler
Ocamllex	Ocaml utility for lexical analysis
Ocamlyacc	Ocaml utility for parsing
Java (JDK 1.6 or above)	Java library, helper class, and output code the Stint compiler
C	The stintc.c file (in the 'bin' directory) provides the ability to run and test a bunch of Stint source files simultaneously. This is used for regression test in our project.
Shell Scripts	Shell scripts aiding some operations in testing
Makefile	Make the Stint compiler from modules
git	Version control of source code
Eclipse and Sublime Text 2	Coding Environment
Google Doc and Dropbox	Documentation and file sharing

4.6 Project Log

The project log, as shown in Table 6 below, presents major milestones in the project, which are extracted from the **github** commit history. Minor changes and bug fixing are omitted.

Table 6. Detailed Project Log

Date	Milestone
09/08/2012	Project begun, first meeting
09/22/2012	Core language feature decided
09/26/2012	Language Whitepaper completed
10/20/2012	Grammar decided
10/27/2012	Language Reference Manual completed
10/29/2012	Skeleton of compiler decided based on Microc
11/12/2012	Scanner and Parser completed
11/07/2012	AST and Symbol Table completed
11/07/2012	Typechecking and SAST completed

11/27/2012	Command Line Interface completed (with -a, -s, -j, -v, -help options)
11/28/2012	Scanner and Parser tested (using -s option to print out the SAST)
12/03/2012	C program for regression tests, stint.c, completed
12/04/2012	Translator completed
12/05/2012	Java Implementation completed
12/08/2012	Compiler working, passed basic test cases
12/10/2012	Delete EOF in AST and add return value of << and >> operators
12/10/2012	Add -c option for generation of .class files
12/13/2012	Translator modified for better formatting in output files
12/15/2012	Bugs fixed, compiler passed robust test cases
12/18/2012	Presentation
12/19/2012	Final Report completed

5. Architectural Design

5.1 Architecture Overview

The stint compiler takes a stint program in source file named *.sti and a source file named *BuildInFunctions.sti* containing stint build-in functions as input. Then it checks whether the source code has lexical errors, syntax errors and semantic errors. After everything is checked, it will output the Java source code having the same functionality as the stint source program. The stint compiler totally consists of 6 parts: scanner, parser, symbol table, type checker, and translator and Java helper functions library. The output of each part is the input of its next part. Figure 2 is the structure diagram of our stint compiler:

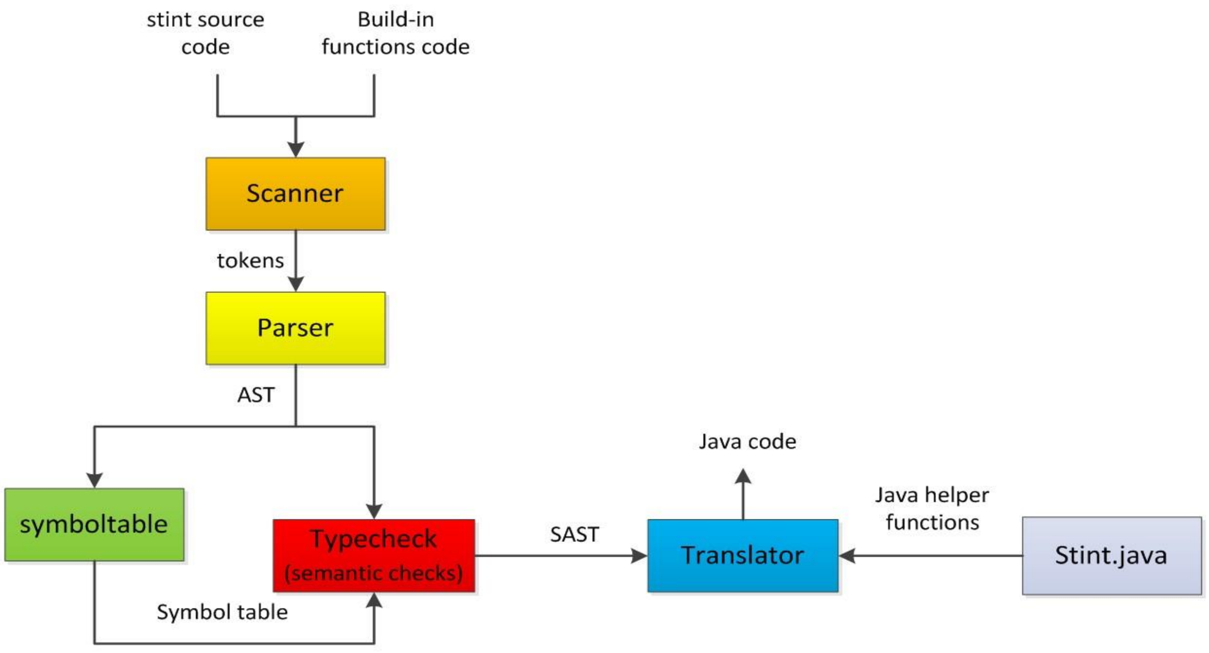


Figure 2. Major Components of the Stint Compiler

5.2 Scanner

The scanner (*scanner.mll*) was implemented with Ocamllex. It takes the source code in the source file and the code of build-in functions as input and generates a list of tokens for the parser to process.

5.3 Parser

The parser (*parser.mly*) is implemented with Ocamlyacc. It takes in a sequence of tokens generated by the scanner through the program declaration and produces an abstract syntax tree (*ast*). The parser can detect any syntax error and it was implemented with Ocamlyacc. If the parser detects a syntax error, it will raise a failure and the entire process will stop immediately.

5.4 Symbol Table

To avoid using duplicate names or unnamed variables and functions, we define a new type called env in the symbol table (*symboltable.ml*). The env contains 3 members: locals, globals and functions. They are implemented by StringMap to store the names of local variables, global variables and functions used in the source, along with their types. There are also some helper functions such as functions to find whether a name of a variable exists in the env or functions to add a name of a variable to the env.

5.5 Type Checking

The type checker checks the code in top-down order. Generally a program consists of two parts: a list of global variables and a list of functions and these are what the type checker needs to check. Firstly, the type checker will check whether the declarations of global variables are compatible, i.e., it will check whether the type of a global variable matches the type of the expression that is assigned to it. It will also check whether a variable is defined repeatedly. If not, it will add the name and type of the new variable to globals in env. Secondly, the type checker will check each function in the function list. It will first check whether it is main function. If it is main function, it cannot have formal list and can only have type void. Otherwise, it will check whether the body of the function is empty. If it's empty, it will raise a failure because it must have one return statement even it has void type. It will also check whether the last statement of a function is a return statement. The last statement has to be return statement.

Then it will check the formal list of the function. Before it checks, it will empty the list of locals in the env and check whether the function name is already in the list of functions in the env. If it is, it will raise a failure. If not, it will add the function name, return type and formal list to the list of functions in the env. For each formal variable, it will first check whether the variable is already defined, if yes, it will raise a failure. If no, it will add its name and type to the list of formals in the env.

After the formal list is checked, the body of the function will be checked. The body contains a list of statements.

5.6 Translator

The translator (*translator.ml)* serves as code generator in our program, responsible for taking the type checked OCaml code and turning it into Java Source code. The translator should also add proper import statements and create a class at the beginning of Java code in order to match the Java rule.

5.7 Java Utilities

In order to make the code generation process easier and make code reusable, we developed some utilities as libraries to help us.

We have developed a utility Java class: Stint class. In it we implemented all the functionalities we have provided in our own language. This Stint class is used as an alternative of the original String class Java provided. Every string in our own language will become a Stint object in the generated Java code, and all the operation we provided on string is mapped to one or two method call.

We also implemented a Utility.java class, which is used to maintain and retrieve all the IO interfaces, including File Objects, Readers and Writers for a program. During translation, whenever we need to create, check, or use the IO interfaces, we simply call the static method provided by this class.

5.7.1 Stint.java

Basically, this is a re-designed String class in Java according to our demands. As our language provide more complicated functions than Java provides in its String class, we need to maintain a more complicated data structure and implement more function than String class does. In the Java code we generated, Stint is used whenever a String is needed. For example, *string a="abc"* in our language will be translated to *Stint a=new Stint("abc).* As all the operations we provide a implemented as methods, this makes is convenient to translate our code to Java code and make the generated Java code readable. The data structure of Stint class is shown in Figure 3 below.

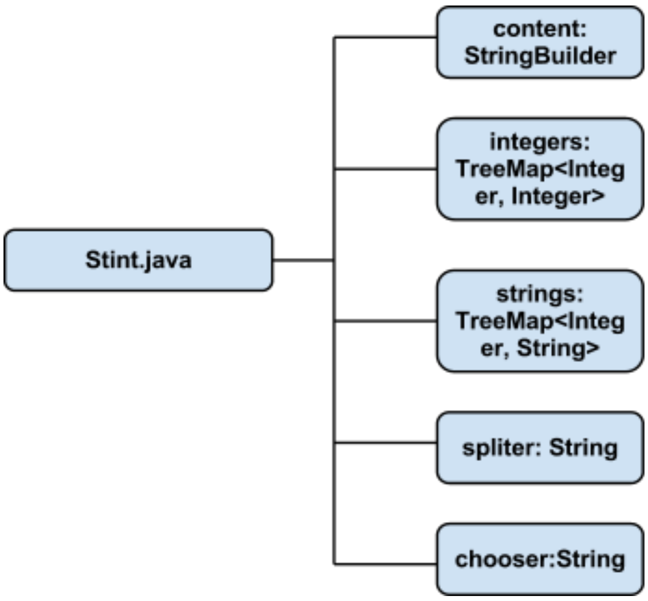

Stint.java
content: StringBuilder
integers: TreeMap<Integer, Integer>
strings: TreeMap<Integer, String>
spliter: String
chooser:String

Figure 3. Data Structure of Stint Class

The content variable stores the whole content of the string, as our "string" is modifiable; we use a StringBuilder to store all the text content in order to have better performance. We also maintain two tree maps to store all the integers and text fragments. The key is the index of the fragment and the value is the integer or text. The splitter and chooser is used to record the regular expression to split or change the content.

Every string operation we provided in our language is mapped to one or two method call to the methods implemented in Stint.java. So the number of lines of code is greatly reduced. This also brings us a lot of convenience during the translation process.

The list of method in Stint.java can be found in Appendix I.

5.7.2 Utility.java

In our own language we have simplified the file input and output to only one line of code, but in java, we have to create a File object and a Scanner or FileWriter to to this. We find it tricky to deal with the extra variables because we need to keep track of with scanner or writer certain stream will use. In order to make this simpler, we designed a utility class to store and retrieve all the File objects, Scanners and Writers.

The getFile method, getScanner method and getWriter method can be called with filename as parameter no matter a IO interface or File object has been created or not. It will first search in the hashmap for the requested object, if it already there, simply return the object, if not, it will create one with the parameter and then add it to the map. This way, our generated Java program do not need to deal with Java I/O directly, the static methods in Utility class is used instead. We never need to worry about the naming and existence of any I/O variable.

The outlines of Stint class and Utility class can be found in Appendix I.

6. Test Plan

6.1 Test Suites

To reduce confusion and enhance efficiency while testing, each part of the translator will be tested individually before combined to the entire structure.

6.1.1 Scanner and Parser

First, scanner and parser should be well compiled indicating there are no shift/reduce conflicts. Then, they will be evaluated by passing test cases to make sure the syntax defined in *Stint* can be parsed without error.

6.1.2 AST and SAST

To exam parser and type-checker can produce correct syntax tree, we provide two functions to print the trees out, which re-translate AST and SAST types into tokens. As long as the output shows a piece of code similar to the inputted one, it can be concluded that the translator has created correct trees for translation.

6.1.3 Translator

Translator was tested after we made sure previous parts till SAST worked fine. Then, combining translator to the whole structure will generate Java source code based on *Stint* code passed in. We evaluated its correctness by compiling the Java source code to eliminate compilation errors during translation. Meanwhile, we revise its format for the convenience while debugging.

6.1.4 Java Helper Class

We also have separated test methods to debug Java Helper Class before combing with the generated code. All test cases for it is included in file StintTester.java (*stint/StintTester.java*).

6.2 Test Case Examples

6.2.1 Scenario: “Modify Sale Record Table”

To test and demo Stint’s language features, we defined an applied scenario shown as below:

> Here is a list for last month's computer sales' information (*sn1_table.txt*).
>
> The columns separated by white spaces are represented as:
>
> *<name> <price> <sales> <inventory>*
>
> And there's another file recorded a single number per line for the sale amount in this month (*sn1_input.txt*).
>
> Now, the program is required to update the information list and output the new list to another file called "*sn1_output.txt*".
>
> The necessary output includes modify the number of sales and inventory up to this month based on the number in "sn_input.txt". Calculate a new column for the total amount of income according to sales and price for each kind of computer, and append it to the end of line.
>
> Meanwhile, for those having total income larger than $300,000, mark "(BEST)" at the end of computer name; for those lower than $60,000, mark "(OFFER)" at the end of computer name, and reduce its price to half as special offer.

This test case is used to demonstrate how Stint can deal with the integers in a string, like getting its value and doing arithmetic calculations.
To finish the tasks described above, here is the source code written in *Stint*:

```
void main ()
{
        string input_file = "sn1_table.txt";
        string data_file = "sn1_input.txt";
        string output_file = "sn1_output.txt";

        string in_buff = "";
        string data_buff = "";
        int sales;

        /* open input and output files */
        open input_file;
        open data_file;
        open output_file;

        /* read line by line */
        while (input_file >> in_buff) {

                /* read sales of this month */
                data_file >> data_buff;
                sales = data_buff.<|0|>;        /* string -> int */

                /* modify sales and inventory */
                in_buff.<|1|> = in_buff.<|1|> + sales;
                in_buff.<|2|> = in_buff.<|2|> - sales;

                /* calculate total sale income */
                in_buff = in_buff + " "
                              + (in_buff.<|0|> * in_buff.<|1|>);

                /* mark tag if necessary */
                in_buff | " ";
                if (in_buff.<|3|> > 300000) {
                        in_buff<|0|> = in_buff<|0|> + "(BEST)"; }

                if (in_buff.<|3|> < 100000) {
                        in_buff<|0|> = in_buff<|0|> + "(OFFER)";
                        in_buff.<|0|> = in_buff.<|0|> / 2;
                }

                /* output */
                output_file << in_buff + "\n";
        }

        close input_file;
        close data_file;
        close output_file;

        return;
}
```

The above code can be generated to Java source code:

6.2.2 String Operation Test

We worked out another test case to test full functionality for the string operations we defined. It does not have a particular scenario, but only try to include the operations as much as possible, making sure all operations work well and the dynamic data structure Stint kept does change based on those operations.

```
void main ()
{
        string str = "abc123def456gh";

        /* expect “abc” and “123” */
        std << str<|0|> + ", " + str.<|0|> + "\n";
        ~str<|1|>;      /* remove “def” from str,
                         * expect “123” and “456” merge */
        /* expect “abc” and “123456” */
        std << str<|0|> + ", " + str.<|0|> + "\n";
        ~str.<|0|>;     /* remove “123456” */
        std << str<|0|> + "\n";
        std << str[0] + ", " + str[2, 2] + "\n";

        /* add something into it, and test indexing again */
        str = str + 9 @ 0;          /* add 9 at beginning */
        str = str + 78 @ 4;         /* insert 78 before 4 */
        std << str + "\n";
        std << str<|1|> + ", " + str.<|1|> + "\n";
        str<|0|> = 2;
        std << str + ", " + str.<|0|> + "\n";

        return;
}
```

The target java code of theses two test cases can be found in Appendix II.

6.2.3 Test Cases

There is a directory containing all the test cases and its expected output in *stint/testcase*.

6.3 Test Automation

We wrote a test binary in C to automatically running test cases called *stintc* under *stint/bin*. The binary should be run under directory *stint/compiler* by either `stintc -d` to execute all test cases listed under *stint/testcase*, or `stintc -d key` to only run those test cases containing the keyword of “key” you indicated in their names. The binary will automatically makefile for you and print testing report.

7. Lessons Learned and Advice for Future Teams

7.1 Lessons Learned

Sichang Li:

Participating in this project gives me a deeper understanding of the principles and techniques of compiler construction. It also gives me the experience of programming in Ocaml, which is very different from other popular programming languages. To be honest, at the beginning, I really hated this language. But when it comes to the compiler construction, Ocaml becomes very powerful. It is so clear and succinct that our compiler has less than 1000 lines of Ocaml code. I have never seen any other languages can do such magical things.

Jiang Wu:

Designing a programming language from scratch enhanced my understanding of how a compiler works. During the whole project we work on several different languages, this help me realized the strengths and shortcomings of different languages (OO languages like Java, functional language, and scripting language).

Tingting Ai:

Besides the main idea of building a compiler, one of the important things I learned from this project is the OCaml language and how powerful it can be. At the beginning the strong error checking ability of the OCaml compiler seems annoying, but later, I was amazed by the fact that “when it compiles, it always works”. Especially when I learned how to build a compiler myself, the powerfulness of OCaml becomes even more astonishing.In addition, working and collaborating with my teammates gave me such a good experience. Each member in the group is capable of accomplishing the assigned task in time, so that our project can progress gradually. The version control system git provided so much convenience for us to work in distributed mode. Good communication is also important in teamwork.

Ningning Xia:

Designing and developing a compiler of our own is interesting, especially when the program turns to be what we expected and imaged before. During this process, I basically found a concept of functional language and learned how to apply recursion thoughts. Finally, effective communication and schedule tasks among team members are necessary for a project.

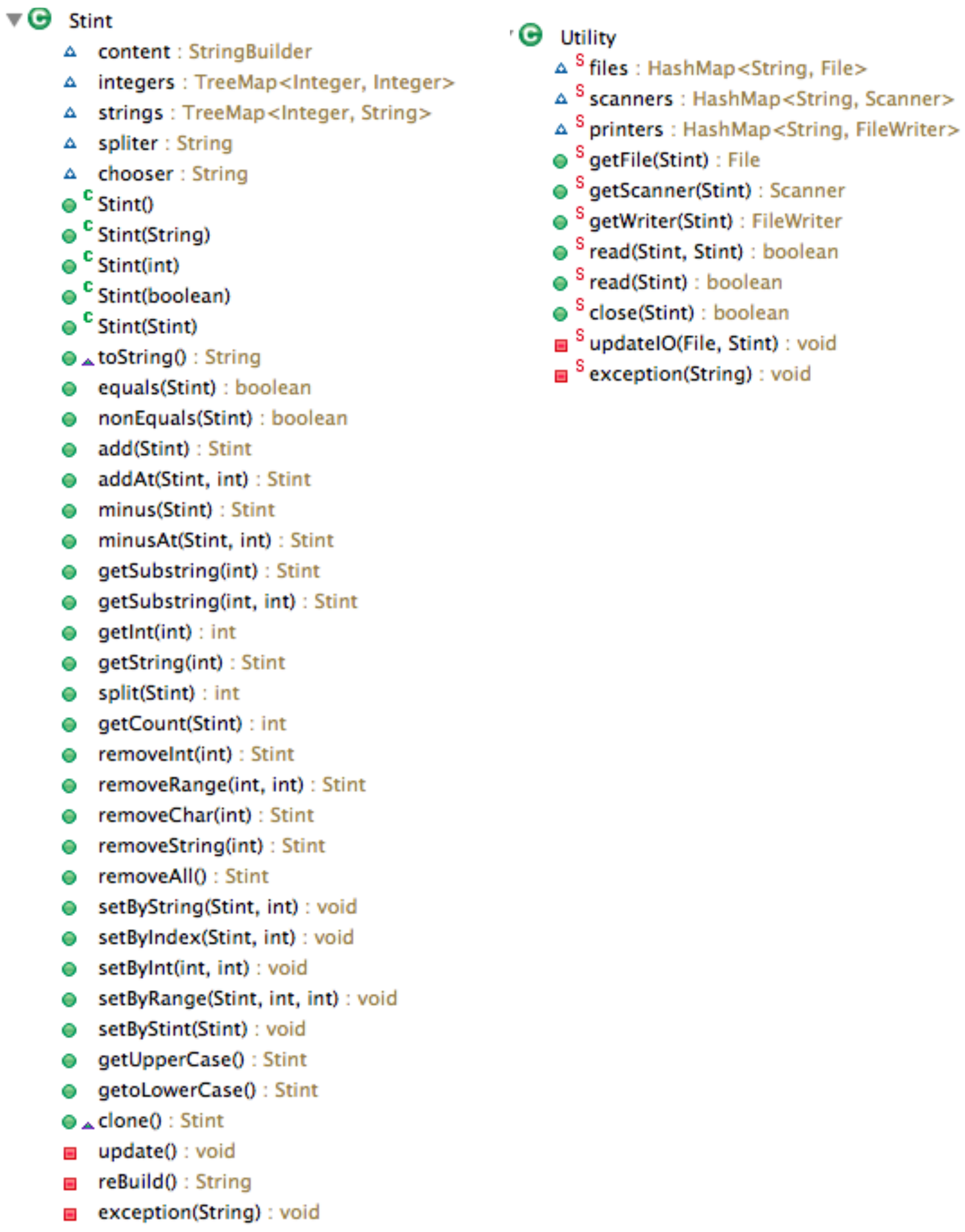
Yiming Xu:

Actually, what I learnt most in this project is using different language for different aims. Our project includes several languages when we were trying to do different things. Like, according to requirement, we had to use Ocaml to write compiler, and I found it did save us many lines of code though we spent more time to learn its syntax. So, choosing correct language would save time in doing many things that computer actually can do for you. When testing our compiler, I was charged in writing automation program to run the test cases. In order to save time, I decided to use C because I was familiar with this. However it turned out I wrote quite a long piece of code. So, choosing language before coding sometimes is quite necessary. By the way, using script language should be a better choice.

7.2 Advice for Future Teams

- This is a complicated project, schedule everything ahead of time and leave enough time for testing.
- If you don’t have any experience with functional language, don’t be afraid, it will save you time as your project goes on.
- Teamwork is usually more efficient that work individually. Group members should meeting at least once a week even when it seems there's not much thing to do. Good ideas might come anytime.
- Ask questions. TAs are experienced and helpful, communicate more with you TA, they will give you advice on the whole process.

Appendix I



```
Stint
    content : StringBuilder
    integers : TreeMap<Integer, Integer>
    strings : TreeMap<Integer, String>
    spliter : String
    chooser : String
    Stint()
    Stint(String)
    Stint(int)
    Stint(boolean)
    Stint(Stint)
    toString() : String
    equals(Stint) : boolean
    nonEquals(Stint) : boolean
    add(Stint) : Stint
    addAt(Stint, int) : Stint
    minus(Stint) : Stint
    minusAt(Stint, int) : Stint
    getSubstring(int) : Stint
    getSubstring(int, int) : Stint
    getInt(int) : int
    getString(int) : Stint
    split(Stint) : int
    getCount(Stint) : int
    removeInt(int) : Stint
    removeRange(int, int) : Stint
    removeChar(int) : Stint
    removeString(int) : Stint
    removeAll() : Stint
    setByString(Stint, int) : void
    setByIndex(Stint, int) : void
    setByInt(int, int) : void
    setByRange(Stint, int, int) : void
    setByStint(Stint) : void
    getUpperCase() : Stint
    getoLowerCase() : Stint
    clone() : Stint
    update() : void
    reBuild() : String
    exception(String) : void
```

```
Utility
    files : HashMap<String, File>
    scanners : HashMap<String, Scanner>
    printers : HashMap<String, FileWriter>
    getFile(Stint) : File
    getScanner(Stint) : Scanner
    getWriter(Stint) : FileWriter
    read(Stint, Stint) : boolean
    read(Stint) : boolean
    close(Stint) : boolean
    updateIO(File, Stint) : void
    exception(String) : void
```

Figure 4. Outlines of Stint Class and Utility Class

Appendix II

Sn1_test.java

```
import java.util.Scanner;
import java.io.FileWriter;
import java.io.File;
import java.io.IOException;

class Sn1_test{

static boolean printFile(Stint s, Stint filename)

{
    try {
        Utility.getFile(filename);
        Utility.getWriter(filename).write((s).toString());
        Utility.getWriter(filename).flush();

        Utility.close(filename);
    }
    catch (Exception e) {
        System.err.println (e);
    };
  return true;
}


static Stint replaceAll(Stint dest, Stint d, Stint s)

{
  int n = dest.getCount(d);
  while (n > 0)
  {
    dest.setByString(s, 0);
    n = dest.getCount(d);
  }

  return dest;
}


static Stint toUpperCase(Stint s)

{
  replaceAll(s, new Stint("a"), new Stint("A"));
  replaceAll(s, new Stint("b"), new Stint("B"));
  replaceAll(s, new Stint("c"), new Stint("C"));
  replaceAll(s, new Stint("d"), new Stint("D"));
  replaceAll(s, new Stint("e"), new Stint("E"));
  replaceAll(s, new Stint("f"), new Stint("F"));
  replaceAll(s, new Stint("g"), new Stint("G"));
  replaceAll(s, new Stint("h"), new Stint("H"));
  replaceAll(s, new Stint("i"), new Stint("I"));
  replaceAll(s, new Stint("j"), new Stint("J"));
  replaceAll(s, new Stint("k"), new Stint("K"));
  replaceAll(s, new Stint("l"), new Stint("L"));
  replaceAll(s, new Stint("m"), new Stint("M"));
  replaceAll(s, new Stint("n"), new Stint("N"));
  replaceAll(s, new Stint("o"), new Stint("O"));
  replaceAll(s, new Stint("p"), new Stint("P"));
  replaceAll(s, new Stint("q"), new Stint("Q"));
  replaceAll(s, new Stint("r"), new Stint("R"));
  replaceAll(s, new Stint("s"), new Stint("S"));
  replaceAll(s, new Stint("t"), new Stint("T"));
  replaceAll(s, new Stint("u"), new Stint("U"));
  replaceAll(s, new Stint("v"), new Stint("V"));
  replaceAll(s, new Stint("w"), new Stint("W"));
  replaceAll(s, new Stint("x"), new Stint("X"));
  replaceAll(s, new Stint("y"), new Stint("Y"));
  replaceAll(s, new Stint("z"), new Stint("Z"));
  return s;
}
```

```
static Stint toLowerCase(Stint s)

{
  replaceAll(s, new Stint("A"), new Stint("a"));
  replaceAll(s, new Stint("B"), new Stint("b"));
  replaceAll(s, new Stint("C"), new Stint("c"));
  replaceAll(s, new Stint("D"), new Stint("d"));
  replaceAll(s, new Stint("E"), new Stint("e"));
  replaceAll(s, new Stint("F"), new Stint("f"));
  replaceAll(s, new Stint("G"), new Stint("g"));
  replaceAll(s, new Stint("H"), new Stint("h"));
  replaceAll(s, new Stint("I"), new Stint("i"));
  replaceAll(s, new Stint("J"), new Stint("j"));
  replaceAll(s, new Stint("K"), new Stint("k"));
  replaceAll(s, new Stint("L"), new Stint("l"));
  replaceAll(s, new Stint("M"), new Stint("m"));
  replaceAll(s, new Stint("N"), new Stint("n"));
  replaceAll(s, new Stint("O"), new Stint("o"));
  replaceAll(s, new Stint("P"), new Stint("p"));
  replaceAll(s, new Stint("Q"), new Stint("q"));
  replaceAll(s, new Stint("R"), new Stint("r"));
  replaceAll(s, new Stint("S"), new Stint("s"));
  replaceAll(s, new Stint("T"), new Stint("t"));
  replaceAll(s, new Stint("U"), new Stint("u"));
  replaceAll(s, new Stint("V"), new Stint("v"));
  replaceAll(s, new Stint("W"), new Stint("q"));
  replaceAll(s, new Stint("X"), new Stint("x"));
  replaceAll(s, new Stint("Y"), new Stint("y"));
  replaceAll(s, new Stint("Z"), new Stint("z"));
  return s;
}


public static void main (String args[])

{
  Stint input_file = new Stint("sn1_table.txt");
  Stint data_file = new Stint("sn1_input.txt");
  Stint output_file = new Stint("sn1_output.txt");
  Stint in_buff = new Stint("");
  Stint data_buff = new Stint("");
  int sales= 0;
    try {
        Utility.getFile(input_file);
      try {
          Utility.getFile(data_file);
        try {
            Utility.getFile(output_file);
        while ( Utility.read(input_file, in_buff))
        {
            Utility.read(data_file, data_buff);
          sales = data_buff.getInt(0);
          in_buff.setByInt(in_buff.getInt(1) + sales, 1);
          in_buff.setByInt(in_buff.getInt(2) - sales, 2);
          in_buff = new Stint (in_buff.add(new Stint(" ").add(new
Stint(in_buff.getInt(0) * in_buff.getInt(1)))));
          in_buff.split(new Stint(" "));
          if (in_buff.getInt(3) > 300000)

          {
            in_buff.setByString(in_buff.getString(0).add(new Stint("(BEST)")), 0);
          }

          if (in_buff.getInt(3) < 100000)

          {
            in_buff.setByString(in_buff.getString(0).add(new Stint("(OFFER)")), 0);
            in_buff.setByInt(in_buff.getInt(0) / 2, 0);
          }

            Utility.getWriter(output_file).write((in_buff.add(new
Stint("\n"))).toString());
            Utility.getWriter(output_file).flush();
        }


            Utility.close(input_file);
        }
        catch (Exception e) {
            System.err.println (e);
        };

          Utility.close(data_file);
      }
      catch (Exception e) {
          System.err.println (e);
        };

        Utility.close(output_file);
    }
    catch (Exception e) {
        System.err.println (e);
    };
  return ;
}



}
```

Test_sp2.java

```
import java.util.Scanner;
import java.io.FileWriter;
import java.io.File;
import java.io.IOException;

class Test_sp2{

static boolean printFile(Stint s, Stint filename)

{
    try {
        Utility.getFile(filename);
        Utility.getWriter(filename).write((s).toString());
        Utility.getWriter(filename).flush();

        Utility.close(filename);
    }
    catch (Exception e) {
        System.err.println (e);
    };
  return true;
}


static Stint replaceAll(Stint dest, Stint d, Stint s)

{
  int n = dest.getCount(d);
  while (n > 0)
  {
    dest.setByString(s, 0);
    n = dest.getCount(d);
  }

  return dest;
}


static Stint toUpperCase(Stint s)

{
  replaceAll(s, new Stint("a"), new Stint("A"));
  replaceAll(s, new Stint("b"), new Stint("B"));
  replaceAll(s, new Stint("c"), new Stint("C"));
  replaceAll(s, new Stint("d"), new Stint("D"));
  replaceAll(s, new Stint("e"), new Stint("E"));
  replaceAll(s, new Stint("f"), new Stint("F"));
  replaceAll(s, new Stint("g"), new Stint("G"));
  replaceAll(s, new Stint("h"), new Stint("H"));
  replaceAll(s, new Stint("i"), new Stint("I"));
  replaceAll(s, new Stint("j"), new Stint("J"));
  replaceAll(s, new Stint("k"), new Stint("K"));
  replaceAll(s, new Stint("l"), new Stint("L"));
  replaceAll(s, new Stint("m"), new Stint("M"));
  replaceAll(s, new Stint("n"), new Stint("N"));
  replaceAll(s, new Stint("o"), new Stint("O"));
  replaceAll(s, new Stint("p"), new Stint("P"));
  replaceAll(s, new Stint("q"), new Stint("Q"));
  replaceAll(s, new Stint("r"), new Stint("R"));
  replaceAll(s, new Stint("s"), new Stint("S"));
  replaceAll(s, new Stint("t"), new Stint("T"));
  replaceAll(s, new Stint("u"), new Stint("U"));
  replaceAll(s, new Stint("v"), new Stint("V"));
  replaceAll(s, new Stint("w"), new Stint("W"));
  replaceAll(s, new Stint("x"), new Stint("X"));
  replaceAll(s, new Stint("y"), new Stint("Y"));
  replaceAll(s, new Stint("z"), new Stint("Z"));
  return s;
}


static Stint toLowerCase(Stint s)

{
  replaceAll(s, new Stint("A"), new Stint("a"));
  replaceAll(s, new Stint("B"), new Stint("b"));
  replaceAll(s, new Stint("C"), new Stint("c"));
  replaceAll(s, new Stint("D"), new Stint("d"));
  replaceAll(s, new Stint("E"), new Stint("e"));
  replaceAll(s, new Stint("F"), new Stint("f"));
  replaceAll(s, new Stint("G"), new Stint("g"));
  replaceAll(s, new Stint("H"), new Stint("h"));
  replaceAll(s, new Stint("I"), new Stint("i"));
  replaceAll(s, new Stint("J"), new Stint("j"));
  replaceAll(s, new Stint("K"), new Stint("k"));
  replaceAll(s, new Stint("L"), new Stint("l"));
  replaceAll(s, new Stint("M"), new Stint("m"));
  replaceAll(s, new Stint("N"), new Stint("n"));
  replaceAll(s, new Stint("O"), new Stint("o"));
  replaceAll(s, new Stint("P"), new Stint("p"));
  replaceAll(s, new Stint("Q"), new Stint("q"));
  replaceAll(s, new Stint("R"), new Stint("r"));
  replaceAll(s, new Stint("S"), new Stint("s"));
  replaceAll(s, new Stint("T"), new Stint("t"));
  replaceAll(s, new Stint("U"), new Stint("u"));
  replaceAll(s, new Stint("V"), new Stint("v"));
  replaceAll(s, new Stint("W"), new Stint("q"));
  replaceAll(s, new Stint("X"), new Stint("x"));
  replaceAll(s, new Stint("Y"), new Stint("y"));
  replaceAll(s, new Stint("Z"), new Stint("z"));
  return s;
}


public static void main (String args[])

{
  Stint str = new Stint("abc123def456gh");
  System.out.print((str.getString(0).add(new Stint(", ").add(new
Stint(str.getInt(0)).add(new Stint("\n"))))).toString());
  str.removeString(1);
  System.out.print((str.getString(0).add(new Stint(", ").add(new
Stint(str.getInt(0)).add(new Stint("\n"))))).toString());
  str.removeInt(0);
  System.out.print((str.getString(0).add(new Stint("\n"))).toString());
  System.out.print((str.getSubstring(0).add(new Stint(", ").add(str.getSubstring(2,
2).add(new Stint("\n"))))).toString());
```

```
  str = new Stint (str.addAt(new Stint(9), 0));
  str = new Stint (str.addAt(new Stint(78), 4));
  System.out.print((str.add(new Stint("\n"))).toString());
  System.out.print((str.getString(1).add(new Stint(", ").add(new
Stint(str.getInt(1)).add(new Stint("\n"))))).toString());
  str.setByString(new Stint(2), 0);
  System.out.print((str.add(new Stint(", ").add(new Stint(str.getInt(0)).add(new
Stint("\n"))))).toString());
  return ;
}


}
```

Appendix III

Source code of the Stint compiler is listed in the following order:

- scanner.mll
- parser.mly
- ast.ml
- symboltable.ml
- typechecking.ml
- sast.ml
- translator.ml
- stint.ml
- Stint.java
- Utility.java
- BuildInFunctions.sti

scanner.mll

```
{ open Parser }

let lowLetter = ['a' - 'z']
let upLetter = ['A' - 'Z']
let digit = ['0' - '9']
let quote = '"'

rule token = parse
    [' ' '\t' '\r' '\n']    { token lexbuf }     | "/*"       { comment lexbuf }
    | '('           { LPAREN }       | ')'       { RPAREN }
    | '{'           { LBRACE }       | '}'       { RBRACE }
    | ';'           { SEMI }         | ','       { COMMA }
    | '+'           { PLUS }         | '-'       { MINUS }
    | '*'           { TIMES }        | '/'       { DIVIDE }
    | '='           { ASSIGN }       | '@'       { AT }
    | "=="          { EQ }           | "!="      { NEQ }
    | '<'           { LESS }         | "<="      { LEQ }
    | '>'           { GRT }          | ">="      { GEQ }
    | '['           { LBRACK }       | ']'       { RBRACK }
    | ".<|"         { LPANGLE }      | "<|"      { LANGLE }
    |  "|>"         { RANGLE }       | '|'       { SPLIT }
    | '#'           { SEARCH }       | '~'       { RM }
    | "&&"          { AND }          | "||"      { OR }
    | '!'           { NOT }          | "<<"      { COUT }
    | ">>"          { CIN }          | "int"     { INT }
    | "string"      { STR }          | "boolean" { BOOL }
    | "if"          { IF }           | "else"    { ELSE }
    | "while"       { WHILE }        | "return"  { RETURN }
    | "open"        { OPEN }         | "close"   { CLOSE }
    | "break"       { BREAK }
    | "void"        { VOID }         | "std"     { STD }
    | "true"        { TRUE }         | "false"   { FALSE }
    | eof           { EOF }          (* do as microC *)
    | digit+ as lit                  { LIT_INT(int_of_string lit) }
    | quote [^'"']* quote as lit     { LIT_STR(lit) }
    | lowLetter (lowLetter | upLetter | digit | '_')* as id     { ID(id) }
    | _ as char          { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
    "*/"            { token lexbuf }

    | _             { comment lexbuf}
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LBRACK RBRACK LPANGLE LANGLE RANGLE   /*
marks */
%token ASSIGN PLUS MINUS TIMES DIVIDE EQ NEQ LESS LEQ GRT GEQ  /* general operators */
%token AT SPLIT SEARCH RM NOT AND OR COUT CIN          /* type-specified operators */
%token INT STR BOOL IF ELSE WHILE RETURN OPEN CLOSE BREAK EOF VOID TRUE FALSE STD   /*
key word */
%token <int> LIT_INT
%token <string> LIT_STR
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc NOAT
%nonassoc AT

%right ASSIGN           /* =, <<, >> */
%nonassoc COUT CIN
%left OPEN CLOSE
%left EQ NEQ            /* ==, != */
%left LESS LEQ GRT GEQ      /* <, <=, >, >= */
%left AND OR            /* &&, || */
%right RM           /* ~ */
%right NOT          /* ! */
%left PLUS MINUS        /* +, - */
%left TIMES DIVIDE      /* *, / */
%left SPLIT SEARCH      /* |, # */
%nonassoc LPANGLE LANGLE RANGLE /* <| |> .<| */

%start program
%type <Ast.program> program

%%

program:
    /* nothing */               { [], [] }
    | program vdecl SEMI        { ($2 :: fst $1), snd $1 }
    | program fdecl             { fst $1, ($2 :: snd $1) }

fdecl:
        var_type ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
        {
            {
                returnType = $1;
                fname = $2;
                formals = $4;
                body = List.rev $7
            }
        }

var_type:
        INT { "int" }
        | STR { "string" }
        | BOOL { "boolean"}
        | VOID { "void" }

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

vdecl:
    var_type ID { ($1, $2, Noexpr) }
    | var_type ID ASSIGN expr { ($1, $2, $4) }

formal_list:
    | vdecl { [$1] }
    | formal_list COMMA vdecl { $3 :: $1}

/*
cont_list:    { [] }
        | cont_list stmt { $2 :: $1 }
        | cont_list vdecl { $2 :: $1}
*/
stmt:
        expr SEMI { Expr($1) }
        | RETURN expr SEMI { Return($2) }
        | RETURN SEMI { Return(Noexpr) }
        | 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) }
        | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
        | BREAK SEMI { Break }
        | vdecl SEMI { Decl($1) }

stmt_list:
        /*Nothing*/ { [] }
        |stmt_list stmt {$2 :: $1}

expr:    LIT_INT { Integer($1) }
        |LIT_STR { String($1) }
        | ID { Id($1) }
        | TRUE { Boolean(True) }
        | FALSE { Boolean(False) }
        /* ___Operator___ */
        | expr PLUS expr %prec NOAT { Oper($1, Add, $3) }
        | expr MINUS expr %prec NOAT { Oper($1, Sub, $3) }
        | expr PLUS expr AT expr{ OperAt($1, Add, $3, $5) }
        | expr MINUS expr AT expr{ OperAt($1, Sub, $3, $5) }
        | expr TIMES expr { Oper($1, Mult, $3) }
        | expr DIVIDE expr { Oper($1, Div, $3) }
        | expr EQ expr { Oper($1, Equal, $3) }
        | expr NEQ expr { Oper($1, Neq, $3) }
        | expr LESS expr { Oper($1, Less, $3) }
        | expr LEQ expr { Oper($1, LessEq, $3) }
        | expr GRT expr { Oper($1, Grt, $3) }
        | expr GEQ expr { Oper($1, GrtEq, $3) }
        | expr AND expr { Oper($1, And, $3) }
        | expr OR expr { Oper($1, Or, $3) }
        | NOT expr { Not($2) }
        /* ___Extract___ */
        | ID LBRACK expr RBRACK { Extract($1, SubChar, $3) }
        | ID LPANGLE expr RANGLE { Extract($1, SubInt, $3) }
        | ID LANGLE expr RANGLE { Extract($1, SubStr, $3) }
        | ID LBRACK expr COMMA expr RBRACK { Sublen($1, $3, $5) }
        /* ___Assign___ */
        | ID ASSIGN expr { Assign($1, $3) }
        | ID LANGLE expr RANGLE ASSIGN expr { AssignSet($1, SubStr, $3, $6) }
        | ID LPANGLE expr RANGLE ASSIGN expr { AssignSet($1, SubInt, $3, $6) }
        | ID LBRACK expr RBRACK ASSIGN expr {AssignSet($1, SubStr, $3, $6)}
        | ID LBRACK expr COMMA expr RBRACK ASSIGN expr { AssignRange($1, $3, $5, $8) }
        | ID SEARCH expr { Chset($1, Fnd, $3) }
        | ID SPLIT expr { Chset($1, Spl, $3) }
        /* ___Remove___ */
        | RM ID LPANGLE expr RANGLE { RemoveSet($2, SubInt, $4) }
        | RM ID LANGLE expr RANGLE { RemoveSet($2, SubStr, $4) }
        | RM ID LBRACK expr RBRACK { RemoveSet($2, SubChar, $4) }
        | RM ID LBRACK expr COMMA expr RBRACK { RemoveRange($2, $4, $6) }
        /* ____Stream___ */
        | expr CIN expr { Stream(In, $1, $3) }
        | STD CIN expr { StreamStd(In, $3) }
        | expr COUT expr { Stream(Out, $1, $3) }
        | STD COUT expr { StreamStd(Out, $3) }
        | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
        | LPAREN expr RPAREN { $2 }
        | OPEN expr { Fop(Open, $2) }
        | CLOSE expr { Fop(Close, $2) }

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

actuals_list:
    expr                    { [$1] }

  | actuals_list COMMA expr { $3 :: $1 }
```

ast.ml

```
type bop =  Add | Sub | Mult | Div      (* +, -, *, / *)
        | Equal | Neq           (* ==, != *)
        | Less | LessEq | Grt | GrtEq   (* <, <=, >, >= *)
        | And | Or          (* &&, ||, ! *)
type subs = SubChar | SubInt | SubStr       (* [] .<||>, <||> *)
type sets = Spl | Fnd               (* |, # *)
type strm = In | Out                (* <<, >>*)
type boolean = True | False
type fop = Open | Close                         (* file operator *)

type expr =
      Integer of int            (* data type: int *)
    | String of string          (* data type: string *)
    | Boolean of boolean
    | Id of string              (* indentifier *)
    | Oper of expr * bop * expr
    | Not of expr
    | OperAt of expr * bop * expr * expr    (* expr1 + expr2 @ pos *)
    | Assign of string * expr       (* iden = *)
    | AssignSet of string * subs * expr * expr
    | AssignRange of string * expr * expr * expr    (*str[,]=*)
    | Extract of string * subs * expr
    | Sublen of string * expr * expr    (* str[index, length] *)
    | Chset of string * sets * expr     (* change set *)
    | RemoveSet of string * subs * expr (* ~str<||>, ~str.<||> *)
    | RemoveRange of string * expr * expr   (* ~str[,] *)
    | Stream of strm * expr * expr      (* normal io stream *)
    | StreamStd of strm * expr      (* std io stream *)
    | Call of string * expr list
    | Fop of fop * expr
    | Noexpr                (* for void arg list *)

type stmt =
    | Block of stmt list
    | Decl of (string * string * expr)
    | Expr of expr
    | Return of expr
    | If of expr * stmt * stmt      (* if() {} else{} *)
    | While of expr * stmt          (* while() {} *)
    | Break

type func_decl = {
        returnType :string;
        fname : string;         (* function name *)
        formals : (string*string*expr) list;    (* Formal argument names *)
        (* local : string list      variables defined in function *)
        body : stmt list;       (* function statements *)
    }

type program = (string * string * expr) list * func_decl list (* global vars, funcs *)

let rec string_of_expr = function
    Integer(i) -> string_of_int i
  | String(s) -> s
  | Id(s) -> s
  | Boolean(b) -> (match b with True -> "true" | False -> "false")
  | Oper(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
    Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | LessEq -> "<=" | Grt -> ">" | GrtEq -> ">=" | And -> "&&" | Or ->
"||") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | AssignSet(v, s, e1, e2) -> v ^
    (match s with
        SubChar -> "[" ^ string_of_expr e1 ^ "]"
        | SubInt -> ".<|" ^ string_of_expr e1 ^ "|>"
        | SubStr -> "<|" ^ string_of_expr e1 ^ "|>")
        ^ " = " ^ string_of_expr e2
  | AssignRange(v, i, l, e) -> v ^"["^ string_of_expr i ^","^ string_of_expr l^"]=" ^
string_of_expr e
  | Extract(v, s, e) -> v ^ (match s with
                SubChar -> "[" ^ string_of_expr e ^ "]"
                | SubInt -> ".<|" ^ string_of_expr e ^ "|>"
                | SubStr -> "<|" ^ string_of_expr e ^ "|>")
  | Sublen(v, e1, e2) -> v ^ "[" ^ string_of_expr e1  ^ ", " ^ string_of_expr e2 ^ "]"
  | Chset(v, s, e) -> v ^ (match s with
                Spl -> " | "
                | Fnd -> " # ") ^ string_of_expr e
  | RemoveSet(v, s, e) -> "~" ^ v ^ (match s with
                SubChar -> "[" ^ string_of_expr e ^ "]"
                | SubInt -> ".<|" ^ string_of_expr e ^ "|>"
                | SubStr -> "<|" ^ string_of_expr e ^ "|>")
  |  RemoveRange(v,  e1,  e2)  ->  "~"  ^  v  ^  "["  ^  string_of_expr  e1  ^  ",  "  ^
string_of_expr e2 ^ "]"
  | Stream(s, v, e) ->  string_of_expr v ^ (match s with
                In -> " >> "
                | Out -> " << ") ^ string_of_expr e
  | StreamStd(s, e) -> "std" ^ (match s with
                In -> " >> " | Out -> " << ") ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  |  Fop(fop,  e)  ->  (match  fop  with  Open  ->  "Open  "  |  Close  ->  "Close  ")  ^
string_of_expr e
  | Not(e) -> "!" ^ string_of_expr e
  | OperAt(e1, bop, e2, e3) -> string_of_expr e1 ^ " " ^ (match bop with
  Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | LessEq -> "<=" | Grt -> ">" | GrtEq -> ">=" | And -> "&&" | Or ->
"||") ^ " " ^ string_of_expr e2 ^ " @ " ^ string_of_expr e3

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Decl(str1, str2, expr) -> str1 ^ " " ^ str2 ^ " = " ^ string_of_expr expr ^ ";\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Break -> "break;\n"

let printFormals = function
  (f, s, l) -> f ^ s ^ string_of_expr l


let string_of_fdecl fdecl =
  fdecl.returnType  ^  "  "  ^  fdecl.fname  ^  "("  ^  String.concat  ",  "  (List.map
printFormals fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map printFormals vars) ^ "\n" ^

  String.concat "\n" (List.map string_of_fdecl funcs)
```

symboltable.ml

```
module StringMap = Map.Make(String)

type env = {
    locals:         string StringMap.t;
    globals:        string StringMap.t;
    functions:      string list StringMap.t;
}

let find_variable name env =
    try StringMap.find name env.locals
    with Not_found -> try StringMap.find name env.globals
    with Not_found -> ""
    (*raise (Failure ("undefined variable " ^ name)) *)

let find_function name env =
    try StringMap.find name env.functions
    with Not_found -> []
    (*raise (Failure ("undefined function " ^ name)) *)

let add_local name v_type env =
    if StringMap.mem name env.locals then StringMap.empty
    else StringMap.add name v_type env.locals

let add_global name v_type env =
    if StringMap.mem name env.globals then StringMap.empty
    else StringMap.add name v_type env.globals

let get_arg_type = function
    (t, _, _) -> t

let add_function name return_type formals env =
    if StringMap.mem name env.functions then StringMap.empty
    else let f = List.map get_arg_type formals in

    StringMap.add name (return_type::f) env.functions
```

typechecking.ml

```
open Ast
open Symboltable

module StringMap = Map.Make(String)

(* get the type of expression:
 *  -> string if one of the two operands having string type
 *  -> int/boolean if both of the operands having the same type *)
let get_expr_type t1 t2 =
    if t1 = "void" || t2 = "void" then raise (Failure ("cannot use void type inside
expression")) else
    if t1 = "string" || t2 = "string" then "string" else
    if t1 = "int" && t2 = "int" then "int" else
    if t1 = "boolean" && t2 = "boolean" then "boolean" else
    raise (Failure ("type error"))

(* mark int & boolean expression to string type *)
let conv_type = function
    (expr, t) -> if t = "void" then raise (Failure ("cannot use void type inside
expression")) else
             if not(t = "string") then Sast.ToStr(expr) else expr

(* get variable type according to the name
 * raise error if no name matching in variable list *)
let get_vartype env id =
    let t = find_variable id env in
    if t = "" then raise (Failure ("undefined variable " ^ id)) else t

let check_string_id expr =
    let (e, t) = expr in
    if not(t = "string") then raise (Failure ("type error")) else
    ( match e with Sast.Id(i) -> e
            | _ -> raise (Failure ("should use identifier")) )

(* check the expression type can be used for
 * the corresponding argument according to definition
 * return the new expression list in expr_t for sast *)
let check_func_arg lst expr arg_t =
    if arg_t = "string" then (conv_type expr)::lst else
    if (snd expr) = arg_t then (fst expr)::lst else
    raise (Failure("unmatched argument type"))

let match_oper e1 op e2 =
    let expr_t = get_expr_type (snd e1) (snd e2) in
    (match op with
       Add -> if expr_t = "int" then (Sast.BinOp(fst e1, Sast.Add, fst e2), "int")
else
          if expr_t = "string" then (Sast.StrOp(conv_type e1, Sast.Adds, conv_type e2),
"string") else
          raise (Failure ("type error"))
     | Sub -> if expr_t = "int" then (Sast.BinOp(fst e1, Sast.Sub, fst e2), "int")
else
          if expr_t = "string" then (Sast.StrOp(conv_type e1, Sast.Subs, conv_type e2),
"string") else
          raise (Failure ("type error"))
     | Mult -> if expr_t = "int" then (Sast.BinOp(fst e1, Sast.Mult, fst e2), "int")
else
           raise (Failure ("type error"))
     | Div -> if expr_t = "int" then (Sast.BinOp(fst e1, Sast.Div, fst e2), "int")
else
          raise (Failure ("type error"))
     | Equal -> if expr_t = "string" then (Sast.StrOp(conv_type e1, Sast.Eqs,
conv_type e2), "boolean") else
            (Sast.BinOp(fst e1, Sast.Equal, fst e2), "boolean")
     | Neq -> if expr_t = "string" then (Sast.StrOp(conv_type e1, Sast.Neqs, conv_type
e2), "boolean") else
          (Sast.BinOp(fst e1, Sast.Neq, fst e2), "boolean")
     | Less -> if expr_t = "string" then raise (Failure ("type error")) else
           (Sast.BinOp(fst e1, Sast.Less, fst e2), "boolean")
     | LessEq -> if expr_t = "string" then raise (Failure ("type error")) else
             (Sast.BinOp(fst e1, Sast.LessEq, fst e2), "boolean")
     | Grt -> if expr_t = "string" then raise (Failure ("type error")) else
```

```
          (Sast.BinOp(fst e1, Sast.Grt, fst e2), "boolean")
     | GrtEq -> if expr_t = "string" then raise (Failure ("type error")) else
            (Sast.BinOp(fst e1, Sast.GrtEq, fst e2), "boolean")
     | And -> if expr_t = "string" then raise (Failure ("type error")) else
          (Sast.BinOp(fst e1, Sast.And, fst e2), "boolean")
     | Or -> if expr_t = "string" then raise (Failure ("type error")) else
          (Sast.BinOp(fst e1, Sast.Or, fst e2), "boolean")
    )

let match_str_oper e1 op e2 pos =
    match op with
      Add -> (Sast.StrOpAt((conv_type e1), Sast.Adds, (conv_type e2), pos), "string")
      | Sub -> (Sast.StrOpAt((conv_type e1), Sast.Subs, (conv_type e2), pos), "string")
      | _ -> raise (Failure ("type error"))

let rec check_expr env = function
    Integer(i) -> Sast.Integer(i), "int"
    | String(s) -> Sast.String(s), "string"
    | Boolean(b) -> (match b with True -> (Sast.Boolean(Sast.True), "boolean")
            | False -> (Sast.Boolean(Sast.False), "boolean"))

    | Id(id) ->
        Sast.Id(id), (get_vartype env id)

    | Oper(e1, op, e2) ->
        match_oper (check_expr env e1) op (check_expr env e2)

    | Not(e) ->
        Sast.UniOp(Sast.Not, (get_expr_with_type env e "boolean")), "boolean"

    | OperAt(e1, op, e2, pos) ->
        let fst_expr = check_expr env e1
        and snd_expr = check_expr env e2
        and position = get_expr_with_type env pos "int" in
        if not((get_expr_type (snd fst_expr) (snd snd_expr)) = "string") then raise
(Failure ("type error"))

        else match_str_oper fst_expr op snd_expr position

    | Assign(id, e) ->
        let t = get_vartype env id in
        if t = "string" then
             Sast.AssignStr(id, (conv_type (check_expr env e))), "void"
        else Sast.Assign(id, (get_expr_with_type env e t)), "void"

    | AssignSet(id, subs, i, e) ->
        if not((get_vartype env id) = "string") then raise (Failure ("type error"))
        else let index = get_expr_with_type env i "int"
             and expr = check_expr env e in
             ( match subs with
              SubChar -> Sast.AssignSet(id, Sast.SubChar, index, (conv_type expr)),
"void"
            | SubInt  -> if not((snd expr) = "int") then raise (Failure ("type error"))
                      else Sast.AssignSet(id, Sast.SubInt, index, (fst expr)), "void"
            | SubStr  -> Sast.AssignSet(id, Sast.SubStr, index, (conv_type expr)),
"void" )

    | AssignRange(id, i, l, e) ->
        if not((get_vartype env id) = "string") then raise (Failure ("type error"))
        else Sast.AssignRange( id,
                    get_expr_with_type env i "int",
                    get_expr_with_type env l "int",
                    conv_type (check_expr env e)), "void"

    | Extract(id, subs, i) ->
        if not((get_vartype env id) = "string") then raise (Failure ("type error"))
        else let index = get_expr_with_type env i "int" in
             ( match subs with
              SubChar -> Sast.Extract(id, Sast.SubChar, index), "string"
            | SubInt  -> Sast.Extract(id, Sast.SubInt, index), "int"
            | SubStr  -> Sast.Extract(id, Sast.SubStr, index), "string" )

    | Sublen(id, i, l) ->
        if not((get_vartype env id) = "string") then raise (Failure ("type error"))
        else Sast.Sublen( id,
                  get_expr_with_type env i "int",
                  get_expr_with_type env l "int"), "string"

    | Chset(id, sets, s) ->
        if not((get_vartype env id) = "string") then raise (Failure ("type error"))
        else let str = get_expr_with_type env s "string" in
             ( match sets with
               Spl -> Sast.Chset(id, Sast.Spl, str), "int"
             | Fnd -> Sast.Chset(id, Sast.Fnd, str), "int" )

    | RemoveSet(id, subs, i) ->
        if not((get_vartype env id) = "string") then raise (Failure ("type error"))
        else let index = get_expr_with_type env i "int" in
             ( match subs with
               SubChar -> Sast.RemoveSet(id, Sast.SubChar, index), "void"
             | SubInt -> Sast.RemoveSet(id, Sast.SubInt, index), "void"
             | SubStr -> Sast.RemoveSet(id, Sast.SubStr, index), "void" )

    | RemoveRange(id, i, l) ->
        if not((get_vartype env id) = "string") then raise (Failure ("type error"))
        else Sast.RemoveRange(  id,
                    get_expr_with_type env i "int",
                    get_expr_with_type env l "int"), "void"
    | Stream(strm, dest, e) ->
        let target = get_expr_with_type env dest "string"
        and expr = check_expr env e in
        ( match strm with
            In -> Sast.Stream(Sast.In, target, check_string_id expr), "boolean"
            | Out -> Sast.Stream(Sast.Out, target, conv_type expr), "void" )

    | StreamStd(strm, e) ->
        let expr = check_expr env e in
        ( match strm with
            In -> Sast.StreamStd(Sast.In, check_string_id expr), "boolean"
            | Out -> Sast.StreamStd(Sast.Out, conv_type expr), "void" )

    | Call(func, el) ->
        let args = find_function func env in    (* return & arguments type list from
definition *)
        ( match args with
            [] -> raise (Failure ("undefined function " ^ func))
            | hd::tl -> let new_list = try List.fold_left2 check_func_arg [] (List.map
(check_expr env) el) tl
                          with Invalid_argument "arg" -> raise(Failure("unmatched
argument list"))
                    in Sast.Call(func, List.rev new_list ), hd )

    | Fop(fop, e) ->
        let target = get_expr_with_type env e "string" in
        ( match fop with
            Open -> Sast.Fop(Sast.Open, target), "void"
            | Close -> Sast.Fop(Sast.Close, target), "void" )

    | Noexpr -> Sast.Noexpr, "void"

(* get expr_t(sast type) by expr(ast type) with given type
 * raise error if the expression type does match requirement *)
and get_expr_with_type env expr t =
    let e = check_expr env expr in
    if not((snd e) = t) then raise (Failure ("type error")) else (fst e)


(* modified: just simply do not allow to assign value in function definition*)
let check_formal env formal =
    let (t, name, expr) = formal in
    let ret = add_local name t env in
    if t = "void" then raise (Failure("cannot use void as variable type")) else
    if StringMap.is_empty ret then raise (Failure ("local variable " ^ name ^ " is
already defined"))
    else let env = {locals = ret; globals = env.globals; functions = env.functions }
in
    ( match expr with
        Noexpr -> (t, name, Sast.Noexpr), env
        | _ -> raise(Failure("cannot assign value inside function definition")) )
    (*let e = check_expr env expr i
        if not(snd e = s1) && not(snd e = "void") && not(s1 = "string") then raise
(Failure ("type error"))
        else let ret = add_local s2 s1 env in if StringMap.is_empty ret then raise
(Failure ("local variable " ^ s2 ^ " is already defined"))
        else if s1 = "string" && (snd e = "int" || snd e = "boolean") then (s1, s2,
Sast.ToStr(fst e))
        else (s1, s2, fst e)*)


let rec check_formals env formals =
    match formals with
      [] -> []
    | hd::tl -> let f, e = (check_formal env hd) in (f, e)::(check_formals e tl)


let rec check_stmt env func = function
      Block(stmt_list) -> (Sast.Block(check_stmt_list env func stmt_list)), env
    | Decl(s1, s2, expr) -> let e = check_expr env expr in
                (*modified: 1. check s1 cannot be void; 2. expr can be void*)
                            if s1 = "void" then raise (Failure("cannot use void as
variable type")) else
                            if not(snd e = s1) && not(snd e = "void") && not(s1 =
"string") then raise (Failure ("type error"))
                            else let ret = add_local s2 s1 env in
                            if StringMap.is_empty ret then raise (Failure ("local
variable " ^ s2 ^ " is already defined"))
                            else let env = {locals = ret; globals = env.globals;
functions = env.functions } in
                            if s1 = "string" && (snd e = "int" || snd e = "boolean")
then (Sast.Decl(s1, s2, Sast.ToStr(fst e))), env
                            else (Sast.Decl(s1, s2, fst e)), env
    | Expr(expr) -> (Sast.Expr(fst (check_expr env expr))), env
    | Return(expr) -> let e = check_expr env expr in
                      if not(snd e = func.returnType) then raise (Failure ("The return
type doesn't match!"))
                      else (Sast.Return(fst e)), env
    | If(expr, stmt1, stmt2) -> let e = check_expr env expr in
                                if not(snd e = "boolean") then raise (Failure ("The
type of the condition in If statement must be boolean!"))
                                else (Sast.If(fst e, fst (check_stmt env func stmt1),
fst (check_stmt env func stmt2))), env     (* if() {} else{} *)
    | While(expr, stmt) -> let e = check_expr env expr in
                           if not (snd e = "boolean") then raise (Failure ("The type
of the condition in While statement must be boolean!"))
                           else (Sast.While(fst e, fst (check_stmt env func stmt))),
env                 (* while() {} *)
    | Break -> (Sast.Break), env

and check_stmt_list env func = function
      [] -> []
    (* | [s] -> (match s with Return(expr) -> [fst (check_stmt env func s)]
                        | _ -> raise (Failure ("The last statement must be return
statement"))) *)
    | hd::tl -> let s,e = (check_stmt env func hd) in s::(check_stmt_list e func tl)

let check_global env global =
    let (v_type, name, expr) = global in
    let e = check_expr env expr in
    if v_type = "void" then raise (Failure("cannot use void as variable type")) else
    if not(snd e = v_type) && not(snd e = "void") && not(v_type = "string") then raise
(Failure ("type error"))
    else let ret = add_global name v_type env in
    if StringMap.is_empty ret then raise (Failure ("global variable " ^ name ^ " is
already defined"))
    else let env = {locals = env.locals; globals = ret; functions = env.functions } in
    (v_type, name, fst e), env

let rec check_globals env globals =
    match globals with
      [] -> []
    | hd::tl -> let g, e = (check_global env hd) in (g, e)::(check_globals e tl)
```

```
let check_function env func =
    if List.length func.body = 0 then raise (Failure ("The last statement must be
return statement"))
    else if func.fname = "main" && (List.length func.formals) > 0
    then raise (Failure ("The main function cannot take any argument"))
    else if  func.fname = "main" && ((func.returnType = "int") || (func.returnType =
"boolean"))
    then raise (Failure ("The main function cannot can only has type void"))
    else
    match List.hd (List.rev func.body) with
      Return(_) ->
        let env = {locals = StringMap.empty; globals = env.globals; functions =
env.functions } in
        let ret = add_function func.fname func.returnType func.formals env in
        if StringMap.is_empty ret then raise (Failure ("function " ^ func.fname ^ " is
already defined"))
        else let env = {locals = env.locals; globals = env.globals; functions = ret }
in
        let f = check_formals env func.formals in
        let formals = List.map (fun formal -> fst formal) f in
        (match f with
        [] -> let body = check_stmt_list env func func.body in
            {Sast.returnType = func.returnType; Sast.fname = func.fname; Sast.formals
= formals; Sast.body = body}, env
        | _ ->  let e = snd (List.hd (List.rev f)) in
            let body = check_stmt_list e func func.body in
            {Sast.returnType = func.returnType; Sast.fname = func.fname; Sast.formals
= formals; Sast.body = body}, e )
      | _ -> raise (Failure ("The last statement must be return statement"))



let rec check_functions env funcs =
    match funcs with
      [] -> []
    | hd::tl -> let f, e = (check_function env hd) in f::(check_functions e tl)


let check_program (globals, funcs) =
(*  let ret = add_function "readFile" "string" [("string", _, _)] StringMap.empty in
    let ret = add_function "printFile" "boolean" [("string", _, _); ("string", _, _)]
ret in
    let ret = add_function "replaceAll" "string"  *)
    let env = { locals = StringMap.empty;
            globals = StringMap.empty;
            functions = StringMap.empty }
    in
    let g = check_globals env globals in
    let globals = List.map (fun global -> fst global) g in
    match g with
     [] -> (globals, (check_functions env (List.rev funcs)))
    | _ -> let e = snd (List.hd (List.rev g)) in (globals, (check_functions e
(List.rev funcs)))
```

sast.ml

```
type bop_t = Add | Sub | Mult | Div | Equal | Neq | Less | LessEq | Grt | GrtEq | And
| Or
type uop_t = Not
type sop_t = Adds | Subs | Eqs | Neqs
type subs_t = SubChar | SubInt | SubStr
type sets_t = Spl | Fnd
type strm_t = In | Out
type fop_t = Open | Close
type boolean_t = True | False

type expr_t =
      Integer of int
    | String of string
    | Boolean of boolean_t
    | Id of string
    | BinOp of expr_t * bop_t * expr_t  (* general operator for int & bollean *)
    | UniOp of uop_t * expr_t        (* special for ! *)
    | StrOp of expr_t * sop_t * expr_t  (* operator for string *)
    | StrOpAt of expr_t * sop_t * expr_t * expr_t   (* with @ *)
    | Assign of string * expr_t
    | AssignStr of string * expr_t  (* assign value to string type *)
    | AssignSet of string * subs_t * expr_t * expr_t
    | AssignRange of string * expr_t * expr_t* expr_t
    | Extract of string * subs_t * expr_t
    | Sublen of string * expr_t * expr_t
    | Chset of string * sets_t * expr_t
    | RemoveSet of string * subs_t * expr_t
    | RemoveRange of string * expr_t * expr_t
    | Stream of strm_t * expr_t * expr_t
    | StreamStd of strm_t * expr_t
    | Call of string * expr_t list
    | ToStr of expr_t        (* type convert *)
    | Fop of fop_t * expr_t
    | Noexpr

type stmt_t =
      Block of stmt_t list
    | Decl of (string * string * expr_t)
    | Expr of expr_t
    | Return of expr_t
    | If of expr_t * stmt_t * stmt_t
    | While of expr_t * stmt_t
    | Break

type func_t = {
    returnType :string;
    fname : string;
    formals : (string * string * expr_t) list;
    body : stmt_t list; }

type prog_t = (string * string * expr_t) list * func_t list


let rec string_of_expr_t = function
    Integer(i) -> string_of_int i
  | String(s) -> s
  | Id(s) -> s
  | Boolean(b) -> (match b with True -> "true" | False -> "false")
  | BinOp(e1, o, e2) ->
      string_of_expr_t e1 ^ " " ^
      (match o with
    Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | LessEq -> "<=" | Grt -> ">" | GrtEq -> ">=" | And -> "&&" | Or ->
"||") ^ " " ^
      string_of_expr_t e2
  | UniOp(o, e) -> "!" ^ string_of_expr_t e
  | StrOp(e1, o, e2) ->
        string_of_expr_t e1 ^ " " ^
        (match o with
        Adds -> "+" | Subs -> "-" | Eqs -> "==" | Neqs -> "!=" ) ^ " " ^
        string_of_expr_t e2
  | StrOpAt(e1, bop, e2, e3) ->
        string_of_expr_t e1 ^ " " ^
        (match bop with
        Adds -> "+" | Subs -> "-" | Eqs -> "==" | Neqs -> "!=" ) ^ " " ^
        string_of_expr_t e2 ^ " @ " ^ string_of_expr_t e3
  | Assign(v, e) -> v ^ " = " ^ string_of_expr_t e
  | AssignStr(v, e) -> v ^ " = " ^ string_of_expr_t e
  | AssignSet(v, s, e1, e2) -> v ^
    (match s with
        SubChar -> "[" ^ string_of_expr_t e1 ^ "]"
        | SubInt -> ".<|" ^ string_of_expr_t e1 ^ "|>"
        | SubStr -> "<|" ^ string_of_expr_t e1 ^ "|>")
        ^ " = " ^ string_of_expr_t e2
  | AssignRange(v, i, l, e) -> v ^"["^ string_of_expr_t i ^","^ string_of_expr_t
l^"]=" ^ string_of_expr_t e
  | Extract(v, s, e) -> v ^ (match s with
                SubChar -> "[" ^ string_of_expr_t e ^ "]"
                | SubInt -> ".<|" ^ string_of_expr_t e ^ "|>"
                | SubStr -> "<|" ^ string_of_expr_t e ^ "|>")
  | Sublen(v, e1, e2) -> v ^ "[" ^ string_of_expr_t e1  ^ ", " ^ string_of_expr_t e2 ^
"]"
  | Chset(v, s, e) -> v ^ (match s with
                Spl -> " | "
                | Fnd -> " # ") ^ string_of_expr_t e
  | RemoveSet(v, s, e) -> "~" ^ v ^ (match s with
                SubChar -> "[" ^ string_of_expr_t e ^ "]"
                | SubInt -> ".<|" ^ string_of_expr_t e ^ "|>"
                | SubStr -> "<|" ^ string_of_expr_t e ^ "|>")
  | RemoveRange(v, e1, e2) -> "~" ^ v ^ "[" ^ string_of_expr_t e1 ^ ", " ^
string_of_expr_t e2 ^ "]"
  | Stream(s, v, e) ->  string_of_expr_t v ^ (match s with
                In -> " >> "
                | Out -> " << ") ^ string_of_expr_t e
  | StreamStd(s, e) -> "std" ^ (match s with
                In -> " >> " | Out -> " << ") ^ string_of_expr_t e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr_t el) ^ ")"
  | ToStr(e) -> string_of_expr_t e
  | Noexpr -> ""
  | Fop(fop, e) -> (match fop with Open -> "Open " | Close -> "Close ") ^
string_of_expr_t e

let rec string_of_stmt_t = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt_t stmts) ^ "}\n"
  | Decl(str1, str2, expr) -> str1 ^ " " ^ str2 ^ " = " ^ string_of_expr_t expr ^
";\n"
  | Expr(expr) -> string_of_expr_t expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr_t expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr_t e ^ ")\n" ^ string_of_stmt_t s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr_t e ^ ")\n" ^
      string_of_stmt_t s1 ^ "else\n" ^ string_of_stmt_t s2
  | While(e, s) -> "while (" ^ string_of_expr_t e ^ ") " ^ string_of_stmt_t s
  | Break -> "break;\n"

let string_of_fdecl_t fdecl =
  fdecl.returnType ^ " " ^ fdecl.fname ^ "(" ^ String.concat ", " (List.map (fun (t, n,
_) -> t ^ n) fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt_t fdecl.body) ^
  "}\n"

let string_of_program_t (vars, funcs) =
  String.concat ";\n" (List.map (fun (t, n, e) -> t ^ " " ^ n ^ " " ^ string_of_expr_t
e) vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl_t funcs)
```

translator.ml

```
type bop =  Add | Sub | Mult | Div       (* +, -, *, / *)
        | Equal | Neq                  (* ==, != *)
        | Less | LessEq | Grt | GrtEq    (* <, <=, >, >= *)
        | And | Or             (* &&, ||, ! *)
type subs = SubChar | SubInt | SubStr        (* [] .<||>, <||> *)
type sets = Spl | Fnd                (* |, # *)
type strm = In | Out                 (* <<, >>*)
type boolean = True | False
type fop = Open | Close                          (* file operator *)

type expr =
      Integer of int                (* data type: int *)
    | String of string              (* data type: string *)
    | Boolean of boolean
    | Id of string                  (* indentifier *)
    | Oper of expr * bop * expr
    | Not of expr
    | OperAt of expr * bop * expr * expr     (* expr1 + expr2 @ pos *)
    | Assign of string * expr         (* iden = *)
    | AssignSet of string * subs * expr * expr
    | AssignRange of string * expr * expr * expr     (*str[,]=*)
    | Extract of string * subs * expr
    | Sublen of string * expr * expr     (* str[index, length] *)
    | Chset of string * sets * expr      (* change set *)
    | RemoveSet of string * subs * expr (* ~str<||>, ~str.<||> *)
    | RemoveRange of string * expr * expr    (* ~str[,] *)
    | Stream of strm * expr * expr       (* normal io stream *)
    | StreamStd of strm * expr       (* std io stream *)
    | Call of string * expr list
    | Fop of fop * expr
    | Noexpr                    (* for void arg list *)

type stmt =
    | Block of stmt list
    | Decl of (string * string * expr)
    | Expr of expr
    | Return of expr
    | If of expr * stmt * stmt       (* if() {} else{} *)
    | While of expr * stmt           (* while() {} *)
    | Break

type func_decl = {
        returnType :string;
        fname : string;           (* function name *)
        formals : (string*string*expr) list;     (* Formal argument names *)
        (* local : string list      variables defined in function *)
        body : stmt list;         (* function statements *)
    }

type program = (string * string * expr) list * func_decl list (* global vars, funcs *)

let rec string_of_expr = function
    Integer(i) -> string_of_int i
  | String(s) -> s
  | Id(s) -> s
  | Boolean(b) -> (match b with True -> "true" | False -> "false")
  | Oper(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
    Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | LessEq -> "<=" | Grt -> ">" | GrtEq -> ">=" | And -> "&&" | Or ->
"||") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | AssignSet(v, s, e1, e2) -> v ^
    (match s with
        SubChar -> "[" ^ string_of_expr e1 ^ "]"
        | SubInt -> ".<|" ^ string_of_expr e1 ^ "|>"
        | SubStr -> "<|" ^ string_of_expr e1 ^ "|>")
        ^ " = " ^ string_of_expr e2
  | AssignRange(v, i, l, e) -> v ^"["^ string_of_expr i ^","^ string_of_expr l^"]=" ^
string_of_expr e
  | Extract(v, s, e) -> v ^ (match s with
                SubChar -> "[" ^ string_of_expr e ^ "]"
                | SubInt -> ".<|" ^ string_of_expr e ^ "|>"
                | SubStr -> "<|" ^ string_of_expr e ^ "|>")
  | Sublen(v, e1, e2) -> v ^ "[" ^ string_of_expr e1  ^ ", " ^ string_of_expr e2 ^ "]"
  | Chset(v, s, e) -> v ^ (match s with
                Spl -> " | "
                | Fnd -> " # ") ^ string_of_expr e
  | RemoveSet(v, s, e) -> "~" ^ v ^ (match s with
                SubChar -> "[" ^ string_of_expr e ^ "]"
                | SubInt -> ".<|" ^ string_of_expr e ^ "|>"
                | SubStr -> "<|" ^ string_of_expr e ^ "|>")
  |  RemoveRange(v,  e1,  e2)  ->  "~"  ^  v  ^  "["  ^  string_of_expr  e1  ^  ",  "  ^
string_of_expr e2 ^ "]"
  | Stream(s, v, e) ->  string_of_expr v ^ (match s with
                In -> " >> "
                | Out -> " << ") ^ string_of_expr e
  | StreamStd(s, e) -> "std" ^ (match s with
                In -> " >> " | Out -> " << ") ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  |  Fop(fop,  e)  ->  (match  fop  with  Open  ->  "Open  "  |  Close  ->  "Close  ")  ^
string_of_expr e
  | Not(e) -> "!" ^ string_of_expr e
  | OperAt(e1, bop, e2, e3) -> string_of_expr e1 ^ " " ^ (match bop with
 Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
     | Equal -> "==" | Neq -> "!="
     | Less -> "<" | LessEq -> "<=" | Grt -> ">" | GrtEq -> ">=" | And -> "&&" | Or ->
"||") ^ " " ^ string_of_expr e2 ^ " @ " ^ string_of_expr e3

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Decl(str1, str2, expr) -> str1 ^ " " ^ str2 ^ " = " ^ string_of_expr expr ^ ";\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Break -> "break;\n"

let printFormals = function
  (f, s, l) -> f ^ s ^ string_of_expr l


let string_of_fdecl fdecl =
  fdecl.returnType  ^  "  "  ^  fdecl.fname  ^  "("  ^  String.concat  ",  "  (List.map
printFormals fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map printFormals vars) ^ "\n" ^

  String.concat "\n" (List.map string_of_fdecl funcs)
```

stint.ml

```
open Unix

type action = Ast | Sast | Java | Class | Help | Version

let version =
  "Stint version 0.1 \"12/18/12\" \n "

let usage =
  "Usage: ./stint <option> <source file>\n where possible options include: \n" ^
    "         -a file.sti        (Output the AST of source file)\n" ^
    "         -s file.sti        (Output the SAST of source file)\n" ^
    "         -j file.sti        (Compile source code to .java file)\n" ^
    "         -c file.sti        (Compile source code to .java and .class files)\n" ^
    "         -help              (Print a synopsis of standard options )\n" ^
    "         -v                 (Display version information)\n"
let _ =
  let action =
    if Array.length Sys.argv > 1 then
      (match Sys.argv.(1) with
          "-a" -> if Array.length Sys.argv == 3 then Ast else Help
        | "-s" -> if (Array.length Sys.argv == 3) then Sast else Help
        | "-j" -> if Array.length Sys.argv == 3 then Java else Help
        | "-c" -> if Array.length Sys.argv == 3 then Class else Help
        | "-v" -> Version
        |   _   -> Help )
    else Help in

    match action with
      Help -> print_endline (usage)
    | Version -> print_endline (version)
    | (Ast | Sast | Java | Class) ->
      let fname = Sys.argv.(2) in
      let index = (if String.contains fname '.' then String.rindex fname '.' else 0 )
in
      let suffix = String.sub fname index 4 in
      if not (suffix = ".sti") then raise (Failure ("Invalid type of source file."))
      else
        let input = open_in fname in
        let input2 = open_in "BuildinFunctions.sti" in
        let lexbuf = Lexing.from_channel input in
        let lexbuf2= Lexing.from_channel input2 in
        let program = Parser.program Scanner.token lexbuf in
        let program2 = Parser.program Scanner.token lexbuf2 in
        let  program_t  =  Typecheck.check_program  (fst  program,  (snd  program  @  snd
program2)) in
        let lindex = ( if ( String.contains fname '/') then ((String.rindex fname '/')
+ 1) else 0 ) in
        let temp = String.capitalize (String.sub fname lindex ( index - lindex) ) in
        let outfilename = temp ^".java" in
        if  action  =  Ast  then  let  listing  =  Ast.string_of_program  program  in
print_string listing
        else
          if  action  =  Sast  then  let  listing  =  Sast.string_of_program_t  program_t  in
print_string listing
          else
            let listing = Translator.to_java program_t temp in
            let out = open_out outfilename in
                output_string out listing; close_out out;
            if action = Class then execvp "javac" [|"javac"; outfilename|]
```

Stint.java

```java
import java.util.ArrayList;
import java.util.Map.Entry;
import java.util.TreeMap;

/***
 * This class is used as a utility class for Stint language to replace the
 * original String class Java offers
 * @author JohnWoo
 *
 */

public class Stint implements Cloneable{

    StringBuilder content;
    TreeMap<Integer,Integer> integers;
    TreeMap<Integer, String> strings;
    String spliter;
    String chooser;

    public Stint(){
        this("");
    }

    public Stint(String arg){
        content=new StringBuilder();
        content.append(arg);
        integers=new TreeMap<Integer, Integer>();
        strings=new TreeMap<Integer, String>();
        update();
    }

    public Stint(int arg){
        this(arg+"");
    }

    public Stint(boolean arg){
        this(arg?"true":"false");
    }

    public Stint(Stint s){
        this(s.toString());
    }

    /* NEVER call this directly */
    public String toString(){
        return content.toString();
    }

    public boolean equals(Stint s){
        return content.toString().equals(s.toString());
    }

    public boolean nonEquals(Stint s){
        return !equals(s);
    }

    public Stint add(Stint s){
        String s1=this.toString();
        String s2=s.toString();
        return new Stint(s1+s2);
    }

    public Stint addAt(Stint s, int index){
        if(index<0)
            index=0;
        String s1=this.toString();
        String s2=s.toString();
        if(s1.length()<index || index<0){
            exception("Stint: Invalid Index: "+index);
            return this;
        }
        if(index==0)
```

```
            return new Stint(s2+s1);
        if(index==s1.length())
            return new Stint(s1+s2);
        return new Stint(s1.substring(0,index)+s2+s1.substring(index));
    }

    //Not Tested
    public Stint minus(Stint s){
        String s1=this.toString();
        String s2=s.toString();
        if(s1.indexOf(s2)==-1)
            return this;
        else{
            return new Stint(s1.replaceFirst(s2,""));
        }
    }

    //Not Tested
    public Stint minusAt(Stint s, int index){
        String s1=this.toString().substring(index);
        String s2=s.toString();
        String s3=(index==0?null:this.toString().substring(0,index));
        if(s1.indexOf(s2)==-1)
            return this;
        else{
            return new Stint(s3+s1.replaceFirst(s2, ""));
        }
    }

    public Stint getSubstring(int index){
        if(index>=this.toString().length()){
            exception("Stint: Invalid Index");
            return this;
        }
        return new Stint(this.toString().substring(index, index+1));
    }

    public Stint getSubstring(int start, int length){
        if(start+length>this.toString().length()){
            exception("Stint: Invalid Length");
            return this;
        }
        return new Stint(this.toString().substring(start, start+length));
    }

    public int getInt(int index){
        if(integers.size()==0){
            exception("Stint: Invalid Index");
        }
        int t=0;
        int key=0;
        for(Integer i:integers.keySet()){
            if(t==index){
                key=i;
                break;
            }
            else t++;
        }
        if(t!=index)
            exception("Stint: Invalid Index");
        return integers.get(key);
    }

    public Stint getString(int index){
        if(spliter==null && chooser==null){
            if(strings.size()==0){
                exception("Stint: Invalid Index");
            }
            int t=0;
            int key=0;
            for(Integer i:strings.keySet()){
                if(t==index){
                    key=i;
                    break;
                }
                else t++;
            }
            if(t!=index)
                exception("Stint: Invalid Index");
            return new Stint(strings.get(key));
        }else if(spliter!=null){
            String[] temp=this.toString().split(spliter);
            if(temp.length-1<index)
                exception("Stint: Invalid Index");
            return new Stint(temp[index]);
        }else{
            return new Stint(chooser);
        }
    }

    //Function not Tested
    public int split(Stint s){
        if(s.equals("")){
            spliter=null;
            return -1;
        }
        else {
            spliter=s.toString();
            chooser=null;
            return this.toString().split(spliter).length;
        }
    }

    public int getCount(Stint s){
        chooser=s.toString();
        spliter=null;
        return
this.toString().endsWith(s.toString())?this.toString().split(chooser).length:this.toSt
ring().split(chooser).length-1;
    }

    public Stint removeInt(int index){
        if(index>integers.size()-1)
            exception("Stint: Invalid Index: "+index);
        else{
            int t=0;
            int key=0;
            for(Integer i:integers.keySet()){
                if(t==index){
                    key=i;
                    break;
                }
                else t++;
            }
            if(t!=index)
                return this;
            integers.remove(key);
            reBuild();
            update();
            //          content=new StringBuilder();
            //          content.append(key);
            //          update();
        }
        return this;
    }

    public Stint removeRange(int start, int length){
        String temp=content.toString();
        if(start==0)
            temp=temp.substring(length);
        else temp=temp.substring(0,start)+temp.substring(start+length);
        content=new StringBuilder();
        content.append(temp);
        update();
        return this;
    }

    public Stint removeChar(int index){
        return removeRange(index,1);
    }

    public Stint removeString(int index){
        this.setByString(new Stint(),index);
        update();
        return this;
    }

    public Stint removeAll(){
        this.content=new StringBuilder("");
        chooser=null;
        spliter=null;
        update();
        return this;
    }

    //a<index>=s
    public void setByString(Stint s,int index){
        if(chooser!=null){
            if(this.getCount(new Stint(chooser))<=index)
                exception("Stint: Invalid Index");
            String s1=this.toString();
            int j=-1;
            for(int i=0;i<=index;i++){
                j=s1.indexOf(chooser,j+1);
            }
            String s2=this.toString().substring(j);
            s2=s2.replaceFirst(chooser,s.toString());
            String f=this.toString().substring(0,j)+s2;
            content=new StringBuilder();
            content.append(f);
            update();
            return;
        }
        if(spliter!=null){
            if(this.split(new Stint(spliter))<=index)
                exception("Stint: Invalid Index");
            String[] temp=this.toString().split(spliter);
            temp[index]=s.toString();
            content=new StringBuilder();
            for(int k=0;k<temp.length;k++){
                content.append(temp[k]);
                if(k!=temp.length-1)
                    content.append(spliter);
            }

            update();
            return;
        }
        if(strings.size()==0){
            exception("Stint: Invalid Index");
        }
        int t=0;
        int key=0;
        for(Integer i:strings.keySet()){
            if(t==index){
                key=i;
                break;
            }
            else t++;
        }
        if(t!=index)
            return;
        strings.remove(key);
        strings.put(key,s.toString());
        reBuild();
        update();
    }

    //a[index]=s
    public void setByIndex(Stint s, int index){
        String temp=content.toString();
        if(index==0){
            temp=s.toString()+temp.substring(1);
            content=new StringBuilder();
            content.append(temp);
        }
        else if(index==content.length()-1){
            temp=temp.substring(0,content.length()-1);
            temp=temp+s.toString();
            content=new StringBuilder();
            content.append(temp);
        }
        else{
            String left=content.substring(0,index);
            String right=content.substring(index+1);
            content=new StringBuilder();
            content.append(left+s.toString()+right);
        }
        update();
    }

    //a.<index>=value
    public void setByInt(int value, int index){
        if(integers.size()==0){
            exception("Stint: Invalid Index");
        }
        int t=0;
        int key=-1;
        for(Integer i:integers.keySet()){
            if(t==index){
                key=i;
                break;
            }
            else t++;
        }
        if(t!=index || key==-1)
            return;
        integers.remove(key);
        integers.put(key,value);
        reBuild();
        update();
    }

    //a[start, length]=s
    public void setByRange(Stint s, int start, int length){
        String temp=content.toString();
        if(start==0)
            temp=s.toString()+temp.substring(length);
        else temp=temp.substring(0,start)+s.toString()+temp.substring(start+length);
        content=new StringBuilder();
        content.append(temp);
        update();
    }

    public void setByStint(Stint s){
        this.content=new StringBuilder(s.toString());
        update();
    }

    /* Below are methods that are used to support build-in functions */

    public Stint getUpperCase(){
        return new Stint(content.toString().toUpperCase());
    }

    public Stint getoLowerCase(){
        return new Stint(content.toString().toLowerCase());
    }

    /* Below are private methods for the maintaince of internal structure */

    public Stint clone(){
        return new Stint(content.toString());
    }

    private void update(){
        chooser=null;
        integers.clear();
        strings.clear();
        String temp=content.toString();
        if(temp.length()==0)
            return;
        String[] ints=temp.split("[^0-9]+");
        ArrayList<String> t=new ArrayList<String>();
        for(String s:ints){
            if(!s.equals(""))
                t.add(s);
        }
        ints=new String[t.size()];
        t.toArray(ints);
        int index=0;
        if(ints.length>0 && temp.startsWith(ints[0])){
            integers.put(0, Integer.parseInt(ints[0]));
            index=2;
        }else{
            index=1;
        }
        for(int i=(index==2?1:0);i<ints.length;i++){
            integers.put(index, Integer.parseInt(ints[i]));
            index=index+2;
        }

        String[] strs=temp.split("[0-9]+");
        t=new ArrayList<String>();
        for(String s:strs){
            if(!s.equals(""))
                t.add(s);
        }
        strs=new String[t.size()];
        t.toArray(strs);
        index=0;
        if(strs.length>0 && temp.startsWith(strs[0])){
            strings.put(0, strs[0]);
            index=2;
        }else{
            index=1;
        }
        for(int i=(index==2?1:0);i<strs.length;i++){
            strings.put(index, strs[i]);
            index=index+2;
        }
    }

    private String reBuild(){
        chooser=null;
        TreeMap<Integer, String> temp=new TreeMap<Integer, String>();
        for(Entry<Integer, String> e:strings.entrySet()){
            temp.put(e.getKey(), e.getValue());
        }
        for(Entry<Integer, Integer> e:integers.entrySet()){
            temp.put(e.getKey(), e.getValue()+"");
        }
        content=new StringBuilder();
        for(Entry<Integer, String> e:temp.entrySet()){
            content.append(e.getValue());
        }
        return content.toString();
    }

    private void exception(String message){
        throw new RuntimeException(message);
    }

}
```

Utility.java

```java
import java.io.File;
import java.io.IOException;
import java.io.FileWriter;
import java.util.HashMap;
import java.util.Scanner;


public class Utility {

    static HashMap<String, File> files= new HashMap<String, File>();
    static HashMap<String, Scanner> scanners=new HashMap<String, Scanner>();
    static HashMap<String, FileWriter> printers=new HashMap<String, FileWriter>();

    public static File getFile(Stint s){
        if(files.containsKey(s.toString())){
            return files.get(s.toString());
        }else{
            File file=new File(s.toString());
            files.put(s.toString(),file);
            updateIO(file,s);
            return file;
        }
    }

    public static Scanner getScanner(Stint s){
        if(scanners.containsKey(s.toString())){
            return scanners.get(s.toString());
        }else{
            exception("Stint: File Not Opened");
            return null;
        }
    }

    public static FileWriter getWriter(Stint s){
        if(printers.containsKey(s.toString())){
            return printers.get(s.toString());
        }else{
            exception("Stint: File Not Opened");
            return null;
        }
    }

    public static boolean read(Stint filename, Stint dest){
        try{
            Scanner sc=getScanner(filename);
            if(sc.hasNextLine()){
                dest.removeAll();
                dest.setByStint(new Stint(sc.nextLine()));
                return true;
            }else
                return false;
        }catch(Exception e){
            e.printStackTrace();
        }
        return false;
    }

    public static boolean read(Stint dest){
        try{
            Scanner sc=new Scanner(System.in);
            if(sc.hasNextLine()){
                dest.removeAll();
                dest.setByStint(new Stint(sc.nextLine()));
                return true;
            }else
                return false;
        }catch(Exception e){
            e.printStackTrace();
        }
        return false;
    }
    public static boolean close(Stint s){
        try{
            if(files.containsKey(s.toString())){
                files.remove(s.toString());
                if(scanners.containsKey(s.toString())){
                    scanners.remove(s.toString());
                    if(printers.containsKey(s.toString())){
                        printers.remove(s.toString());
                        return true;
                    }
                }
            }
            return false;
        }catch(Exception e){
            e.printStackTrace();
            exception("Stint: IO Exception");
        }
        return false;
    }

    private static void updateIO(File f, Stint s){
        try {
            FileWriter fw=new FileWriter(f,true);
            Scanner sc=new Scanner(f);
            scanners.put(s.toString(),sc);
            printers.put(s.toString(),fw);
        } catch (IOException e) {
            e.printStackTrace();
            exception("Stint: IO Exception");
        }

    }

    private static void exception(String message){
        throw new RuntimeException(message);
    }

}
```

BuildInFunctions.sti

```
boolean printFile(string s, string filename){
    open filename;
    filename << s;
    close filename;
    return true;
}
string replaceAll(string dest, string d, string s){
    int n=dest # d;
    while(n>0){
        dest<|0|>=s;
        n=dest # d;
    }
    return dest;
}
string toUpperCase(string s){
    replaceAll(s,"a","A");
    replaceAll(s,"b","B");
    replaceAll(s,"c","C");
    replaceAll(s,"d","D");
    replaceAll(s,"e","E");
    replaceAll(s,"f","F");
    replaceAll(s,"g","G");
    replaceAll(s,"h","H");
    replaceAll(s,"i","I");
    replaceAll(s,"j","J");
    replaceAll(s,"k","K");
    replaceAll(s,"l","L");
    replaceAll(s,"m","M");
    replaceAll(s,"n","N");
    replaceAll(s,"o","O");
    replaceAll(s,"p","P");
    replaceAll(s,"q","Q");
    replaceAll(s,"r","R");
    replaceAll(s,"s","S");
    replaceAll(s,"t","T");
    replaceAll(s,"u","U");
    replaceAll(s,"v","V");
    replaceAll(s,"w","W");
    replaceAll(s,"x","X");
    replaceAll(s,"y","Y");
    replaceAll(s,"z","Z");
    return s;
}
string toLowerCase(string s){
    replaceAll(s,"A","a");
    replaceAll(s,"B","b");
    replaceAll(s,"C","c");
    replaceAll(s,"D","d");
    replaceAll(s,"E","e");
    replaceAll(s,"F","f");
    replaceAll(s,"G","g");
    replaceAll(s,"H","h");
    replaceAll(s,"I","i");
    replaceAll(s,"J","j");
    replaceAll(s,"K","k");
    replaceAll(s,"L","l");
    replaceAll(s,"M","m");
    replaceAll(s,"N","n");
    replaceAll(s,"O","o");
    replaceAll(s,"P","p");
    replaceAll(s,"Q","q");
    replaceAll(s,"R","r");
    replaceAll(s,"S","s");
    replaceAll(s,"T","t");
    replaceAll(s,"U","u");
    replaceAll(s,"V","v");
    replaceAll(s,"W","q");
    replaceAll(s,"X","x");
    replaceAll(s,"Y","y");
    replaceAll(s,"Z","z");
    return s;
}
```