"Hey guys, we're making a new language!"
"k"

k-AWK: A testing language

Albert Cui, Karen Nan, Michael Raimi, Mei-Vern Then

Introduction

Motivation and Goals

The k-AWK language will facilitate developers in creating automated tests for quality assurance. Testing code is often a time-consuming process, but at the same time a critical phase to ensure the proper functionality of an application. The k-AWK language centers around the best practices of test-driven-development, which encourages developers to design software in a robust manner and can serve as a teaching tool for programmers new to software development.

Overview

k-AWK will check for predefined statements within each struct that when called or initialized all assertions will have to evaluate to true for the program to continue. The AWK-like assertion pattern enforces incremental testing and code quality in object-like structs.

The power of this language comes in the idea of assertions, which simplifies the process of validating changes to the state of the program each time a variable or data structure is initialized or modified. As a *struct* is updated, the language evaluates predicates attached to the *struct* to update the state of the program. If the assertion is false, the attached block executes, essentially acting as an exception handler. However, if all assertions are true in a block, the program proceeds normally to the next statement.

Additionally, *unit* features attached to functions allow the checking of output.

Language Tutorial

Program Execution

A k-AWK program has the extension `.k`. To compile (but not run) a `.k` program, no setup is necessary. Simple use `make` in the top level source directory to create the `code_gen` file, then run the code generator file with your `.k` file as the only argument:

```
$ ./code_gen foobar.k
```

The above command can be piped to a `.java` file, which can then be compiled using `javac`. For convenience's sake, a run script has been provided, which takes one `.k` file as an argument. This script calls the compilation script first, so no compilation is required before it. This runs the program, and outputs any print statements to output_java_<test.k>.txt:

```
$ ./run.sh foobar.k
```

Asserts

k-AWK comes with special assert statements, which can only be used in `structs`, denoted by the `@` symbol with an expression and a block of statements. Asserts work very similar to an `if` statement. The expression must evaluate to a boolean; if the `@` expression evaluates to true, the program continues. If it evaluates to false, the program executes the statements within the attached block (which can be empty, if the programmer wishes). Consider the following example:

```
@(k < 100) { print("k is >= 100!"); }
```

Asserts are evaluated whenever a variable used in the evaluative expression is changed. In the above example, if `k` is less than 100, the program continues. If not, the print statement within the attached block is executed.

Units

k-AWK comes with `unit` declarations, which can be called on any function except main, from any function but itself. Consider the following example:

```
unit:foo(8):equals(1):accept;
```

The unit function calls the function `foo` and passes in 8 as a parameter. `foo` will evaluate the function and return a value, and if that value equals the value in the `equals()` section of the declaration, the `accept` value indicates that the test should pass if a `true` value is returned.

Built-In Functions

k-AWK comes with two built-in functions. Programmers can use the print function to be used as follows:

```
print(10);
```

where the function can take a number or string argument and prints to stdout.
Additionally, an exit function is available and can be used as follows:

```
exit("foobar");
```

which takes only a string as a single argument, and prints it before exiting the program.

k-AWK Program Examples

hello_world.k

A basic k-AWK program is `hello_world.k`, as in the following:

```
void main(){
        print("Hello, world! k-awk says hi!");
}
```

A main function of type `void` (which takes no arguments) must be in every k-AWK program. In `hello_world.k`, the main function simply uses k-AWK's built-in print function to print a string to stdout.

gcd.k

As a second example, consider the following, which uses asserts and and units to find the greatest common denominator (gcd) of any two numbers:

```
int run_gcd (int a, int b) {
        while (a != b) {
                if (a > b) {
                        a = a-b;
                }
                else {
                        b = b-a;
                }
        }
        return a;
}

  void main() {
        unit:run_gcd(24, 54):equals(6):accept;
 }
```

For main to use a function or a struct, it must be defined before main is. In the above `gcd.k` example, `gcd` is a struct with several asserts (characterized by the `@` symbol), which work as `if` statements. If the `@` expression evaluates to true, the program continues. If it evaluates to false, the program executes the statements within the attached block.

Language Reference Manual

Introduction

This manual describes the k-AWK language and is meant to be used as a reliable guide to the language.

For the most part, this document follows the outline of the C Language Reference Manual, as described in Appendix A of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie.

Lexical Conventions

A program consists of one or more translation units stored in files. It is translated in several phases. The first phases do low-level lexical transformations. When the preprocessing is complete, the program has been reduced to a sequence of tokens.

Tokens

There are five classes of tokens: identifiers, keywords, string literals, operators and separators. Blanks, tabs, and newlines will be ignored, except for white space that is required to separate two consecutive tokens.

Comments

The characters `/*` and `*/` introduce a comment and terminates them. Comments do not nest.

Identifiers

An identifier is a sequence of letters and digits of any length. The sequence must start with a letter. all following characters can be any combination of letters, numbers, or the underscore _ and hyphen - (which counts as a letter). Upper and lower case letters are different.

Keywords

The following are reserved as keywords and cannot be used otherwise:

```
if        return    true      equals
else      int       false     accept
while     void      str       reject
for       bool      struct    unit
exit      print
```

Keyword definitions

`accept`	Used in unit, a value that denotes that a unit test has passed.
`reject`	Used in unit, a value that denotes that a unit test has failed.
`unit`	Identifier signifying a unit test statement
`exit`	exit the program at the point of execution
`print`	print the output specified
`true`	A boolean value signifying true
`false`	A boolean value signifying false
`equals`	Used in unit, a keyword that evaluates whether a specified value equals the return value of a function

Constants

Constants are not supported.

String Literals

A string literal is a sequence of one or more escape characters or a non-double quote character (as indicated below), surrounded by double quotes, as in “`String Literal`”. A string has type `str` and is initialized with the given characters.

Escape characters include:

`' '`	space
`\t`	tab
`\r`	carriage return
`\n`	newline

Meaning of Identifiers

Identities or names refer to many things: functions, tags of structures, members within the structures, and variables. Interpretations of variables depend on two main attributes: *scope* and *type*. The scope is the region of the program where it is known and type determines the meaning of the values in the variable.

Basic Types

There are four fundamental types: strings; integers; booleans; and void.

Types	k-AWK Declaration	Use
Strings	`str`	Large enough to store any sequence of combinations from the character set.
Integers	`int`	Have the natural size suggested by the host machine architecture. In this case, the `int` data type has a minimum of -2^{31} and a maximum value of 2^{31}-1, following the 32-bit

		signed two's complement integer.
Booleans	`bool`	Only hold either `true` or `false` values.
Void	`void`	An empty set of values and is the return type of functions that generate no value. This type can only be used where the value is not required

Derived Types
Beside the basic types, there are derived types constructed from the fundamental types in the following ways:

arrays of objects of a given type
functions returning objects of a given type
structures containing a sequence of objects of various types, with optional *asserts* of conditional checks on objects of various types.

Expressions

In k-AWK, expressions include primary expressions, array references, function calls, unit calls, structure references, and operators.

Comma Operator
The comma's function is to separate elements of a formal list of arguments (in a function declaration or call) and in an list of actual arguments:

formal args:
```
type id(type formal_arg1, type formal_arg2){return NULL;};
```

actual args:
```
id(actual_arg1, actual_arg2);
```

Commas can also separate elements in an array:
```
type[size] id = [element1, element2, element3];
```

Colon Operator
The colon's function is to separate elements of a `unit` expression into its subparts. In a unit expression declared within the function, the colon separates the unit expression into three parts:
```
unit(args):logical_operator(args):result_type
```

In unit outer declarations, the colon also serves as a separator:

```
unit:function_name(args):logical_operator(args):result_type
```

Mathematical Operators

These include the arithmetic operator, difference operator and multiplicative operators:

Symbol	Operator Name	Use
+	Arithmetic operator	Calculates the sum of operands
-	Difference operator	Calculates difference of operands
`*`	Multiplicative operator	Multiplication, grouped from left to right
`/`	Divisive operator	Division, grouped from left to right
`%`	Modulo operator	Finds remainder, grouped from left to right

Relational Operators

Relational operators include the following:

Symbol	Use
>	greater than
`<`	less than
`>=`	greater than or equals to
`<=`	less than or equals to

The general form of relational operators take the following form:

relational-expression [relational operator] shift-expression

Such statements evaluate to either `true` or `false` and are grouped from left to right. The type of the result is `bool`.

Equality Operators

Equality operators include the following:

Symbol	Use
==	equal to
`!=`	not equal to

The general form of relational operators take the following form:

equality-expression [equality operator] relational-expression

Equality operators have lower precedence than relational operators. For example:

`a < b == c < d` is returns `true`
if `a < b` and `c < d` return the same truth-value (both `true` or both `false`)

The type of the result is `bool`.

Logical Negation Operator

The logical negation operator includes the following:

Symbol	Use
!	not

The `!` operator is a unary operator and must be applied to a boolean expression:

[logical negation]operand

The result is `true` if the value of its operand is `false`, and `false` if the value of its operand is `true`. The type of the result is `bool`.

Logical AND Operator

There is only one form of the logical AND operator:

Symbol	Use
`&`	and

The AND operator is applied in the following form:

expression `&` *expression*

Logical AND groups left-to-right. It returns `true` if the left and right boolean expressions both evaluate to true. Otherwise it returns `false`. Both left and right expressions are required to return the `boolean` type. The type of the result is `bool`.

Logical OR Operator

There is only one form of the logical OR operator:

Symbol	Use
`|`	or

The AND operator is applied in the following form:

expression | *expression*

Logical AND groups left-to-right. It returns `true` if either the left or the right boolean expressions evaluate to `true`. If both the left or the right expressions return false, the statement returns `false`. Both left and right expressions are required to return the `boolean` type. The type of the result is `bool`.

Assignment Operators

There is only one assignment operator:

Symbol	Use
=	assign

This operator is applied in the following form:

left-value = *right-value*;

In the assignment operator, the value of the left operand is replaced by the expression to the right of the assignment operator. Both operands must have the same arithmetic type.

All require a left-value as a left operand, where the left value must be modifiable. The type of an assignment expression is equal to the type of its left operand, and the value is the value stored in the left operand after the assignment is executed.

Array References

Indexes are indicated between brackets, with its name before it processed in postfix manner. Elements of an index can be accessed in the form:

```
foo[x];
```

where `foo` is an array identifier, and `x` is the index of the element to be accessed. The type returned is the type of the array.

Structure Member References

Structure references are accessed using dot, in the form

```
foo.bar;
```

where `foo` is an identifier of a struct and `bar` is a member of `foo`. The type returned is the type of the member. For example:

```
struct test {
        int mem;
};
struct test s;
s.mem = 10;    /* mem is now 10 */
```

Function Calls

Function calls are postfix expressions constructed with a designator (the name of function followed by a pair of parentheses() .Expressions within the parentheses serve as placeholders for arguments of each function, separated by commas. The following examples are calls to functions:

```
function_name();
        function_name(arg1, arg2);
```

Unit Calls

Unit calls have four major components, and returns either an `accept` or `reject` value.

unit keyword

The `unit` keyword denotes the start of the unit call.

function call

The function call specifies the function to test for this unit. This is specified after the `unit` keyword, separated by a colon:

```
unit:function_name(arg1)
```

The function name and its corresponding arguments must match in number and type. For example, given the following function:

```
int function_name(int arg1, int arg2){return 2;};
```

The *unit keyword* and the *function call* of the statement should be as follows:

```
unit:function_name(actual_1, actual_2):(... rest of unit call ...)
```

where `actual_1` and `actual_2` are of type `int`. Only one function call can be attached per unit call.

logical expression validation
Logical expressions check whether or not the value returned by the function specified in the *explicit function call* or the *implicit function call* evaluates to a certain value.

The logical expression allowed in k-AWK is `equals(value)`, which returns `true` if the return value of the function call is equal to the `value` specified and returns `false` if it does not equal the `value` specified.

Building on top of the previous example, we now have:

```
int function_name(int arg1, int arg2){
        return arg1+arg2;
};
unit(5,4):equals(9)(...rest of unit call ...)
```

In this example, `equals(9)` would check against the call `function_name(5,4)` for equality.

test result type
This component of the unit call tells whether or not a test is accepted or rejected. This is directly related to the validation of the logical expression.
The following chart explains most clearly how the return value behaves:

	result type `accept`	*result type* `reject`
`expression(value) = true`	`accept`	`reject`
`expression(value) = false`	`reject`	`accept`

This implies that if the developer expects a `true` result and the expression yields a `true` result, the developer would want to return an `accept` value, whereas if an expression is expected to return `false` and does return `false`, then the developer should return an `accept` value on the test as well.

Building on top of the previous example, we now have:

```
int function_name(int arg1, int arg2){
        return arg1+arg2;
};

unit:function_name(5,4):equals(9):accept
```

In this example, `equals(9)` would check against the call `function_name(5,4)` for equality, returning `true`. Since the expression returns true and the *test result type* is accept, the unit test returns an `accept`.

Constructing these parts together yields a unit call, as show below:

```
type function1(argtype arg2){
        return arg2;
}
type function2(argtype arg1){
        return arg1;
        unit:function1(actual_arg2):logical_operator(args):result_type;
}
```

exit statements

Exit statements are written with the keyword `exit`, and a string literal or variable of type `str` as an argument. For example:

```
exit("Exiting");
exit(value);
```

Where `value` is of type `str`.

When this statement is reached, `exit` prints the strings inside its argument to console and exits the program at that point in time.

Declarations and Declarators

Declarations specify the interpretation give to each identifier.

Meaning of Declarators

A list of declarators appears after a sequence of type and storage class specifiers. Each declarator declares a unique main identifier. The storage class applies directly to this identifier, but its type depends on the form of its declarator. When an identifier appears in an expression of the same for as the declarator, is will give an object of the specified type.

Type Specifiers

The following are type specifiers:

`void`	Void
`string`	String
`int`	Integer
`bool`	Boolean
`struct`	Structure

Each declaration must have one type-specifier.

Declarators

Declarators have the generic form:

```
type identifier;
```

The identifier can be seen as the name of the variable.

Array Declarators

Arrays can be declared in two ways:

```
type[constant/expression] identifier;
```

or

```
type[] identifier = {element_1, element_2, element_3};
```

In the first example, the *constant/expression* specifies the size of the array. A constant should be of type `int` and an expression should return a value with type `int`.

In the second example, the elements denoted by `element_1, element_2, element_3` represent the items inside the array in order of increasing index. The size of the array does not need to be explicitly specified. Elements declared inside arrays should be the same type as the type of the array as specified with `type`.

Function Declarators

Function declarations take the form:

```
type identifier(argtype1 arg1, argtype2 arg2){
            ...
      return_stmt;
};
```

where `argtype1` and `argtype2` specify the types of `arg1` and `arg2`, respectively. When the function is called, the actual parameters of `arg1` and `arg2` have to have the corresponding types as specified in the declaration.

However, arguments are optional in function declarations, hence:

```
type identifier(){
      return_stmt;
}
```

is valid.

The value of `return_stmt` much match the type declared by `type`. Return statements are not allowed in the event that the function is declared with return type `void.` In such cases, no value will be returned from the function.

Function overriding and overloading

Functions cannot be overloaded or overridden once they are declared. Functions of the same name cannot be declared.

Structure Declarators

A program maintains a list of declared structures. A `struct` is an object optionally consisting of a sequence of named members and assertions on the aforementioned members. The following shows an

Structs can be empty:

```
struct identifier{}
```

Or they can include named members:

```
struct identifier{
        type identifier;      /*member*/
}
```

As well as assert statements:

```
struct identifier{
        type identifier;      /*member*/
        @(boolean_expression);
}
```

Assert Declarations

Asserts are denoted by an @ character in the beginning of the line followed by a logical expression within the parentheses () and a series of statements wrapped in a block {}:

```
@(expression){statements;};
```

Only one expression is allowed per assert. Variables referenced in the assert expression should be within the scope of the struct in which the assert is declared. The assert expression should evaluate to a boolean value, `true` or `false`. Statements within assert blocks cannot return a value.

Every time there is a change in value of a member in an struct, the struct's properties will be checked against a series of assertions optionally declared in the struct. If an assertion statement fails, the attached block of code will be executed.

```
struct Player {
        int hp = 100;
        int size = 100;
        int weight
        /* this will print "Not enough HP" at runtime */
        @(hp > 1000) { print("Not enough HP"); }
        /* this should pass since size 100 > 10 */
        @(size > 10) { print("Not big enough"); }
}
```

Initialization

Initialization of Variables

When an variable is declared, one may specify an initial value (in the form of an expression) for the identifier being declared:

```
type ID = value;
```

After an variable is declared, one may initialize to a value of the same type as the declared variable:

```
the_type ID;
ID = value;
```

Where `value` has type `the_type`.

Initialization of Arrays

Arrays can be declared and then initialized in the same line with the following:

```
type[] identifier = {element_1, element_2, element_3};
```

The elements denoted by `element_1, element_2, element_3` represent the items inside the array in order of increasing index. The size of the array does not need to be explicitly specified. Elements declared inside arrays should be the same type as the type of the array as specified with `type`.

After declaration, indices of arrays can be initialized to a value, provided that the value matches the declared type of the array and the index is smaller than the size of the array:

```
the_type[size] identifier;
identifier[index] = value;
```

The `index` must be of type `int` and less than `size`. The `value` must be of the same type as `the_type`.

Initialization of Structures

`structs` can only be initialized outside functions before function declarations. During `struct` initialization, all member functions inside the block must be initialized as well, in the same order as the declaration. Member functions will be initialized inside the curly brackets `{}` For example:

```
struct test {
	int mem;
	int mem2;
}
struct test s = {10, 11}; /* mem is now 10, mem2 is now 11 */
```

`test` corresponds to the identifier of the declared `struct`, while `s` corresponds to the new instance of the struct of type `test`.

Statements

Unless otherwise specified, statement execution is sequential. Statements are executed for their effect and do not contain values. They fall into several groups: Expression-statements, compound-statements, selection-statements, and iteration-statements.

Expression Statements

Most expression statements are assignments or function calls. All side effects from the expression are completed and evaluated before the next statement is executed.

Expression statements appear in the following form:

```
expression;
```

Compound Statements

To allow for use of several statements where only one is expected, the compound statement, or “block,” is provided. The body of a function definition, as is the body of a structure definition, is a compound statement.

```
struct i {
        type id1;
        type id2; /*variable declarations*/
        type id3;
        statement1; /*statements */
        statement2;
}
        type id4;
        type id5; /*variable declarations outside block*/
        type id6;
        statement3;
        statement4; /*statements outside block*/
        statement5;
```

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended inside the block. An identifier must be unique and declared once inside the block.

Initialization of objects is performed each time the block is entered at the top, and proceeds in the order of declarators. If a jump into the block is executed, the initializations are not performed.

Selection Statements

Selection statements have several flows of control:

`if` (*expression*) *statement* `else` *statement*

In the `if` statements, the expression must be of a boolean type. It is evaluated and includes all side effects, and if the expression evaluates to true, the first sub-statement is executed. Since the `if` statement is followed by an `else`, the second sub-statement is executed if the expression evaluates to false. To get the similar effect of having an `else`-less `if` statement, the attached `else` sub-statement may be left empty.

Iteration Statements

Iteration statements specify looping.

iteration-statement:

`while` (*expression*) *statement;*

`for` (*expression*; *expression*; *expression*) *statement;*

In the `while` statement, the substatement is executed repeatedly provided the value of the expression evaluates to true. The boolean test, including all side effects from the expression, occurs before each execution statement body. Blocks of code can wrap statements using `{}`.

In the `for` statement, the first expression is evaluated once, which specifies initialization for the loop. The second expression must be either of boolean type or omitted. It is evaluated before each iteration, and if it evaluates to false, the `for` is terminated. The third expression is evaluated after each iteration, and thus specifies a reinitialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. A `for` statement must include all three expressions. Blocks of code can wrap statements using `{}`.

Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another based on where they're declared.

Members of structures are unique given that their structures are named uniquely.

Variables declared at the top level (within no block) are able to be access from anywhere in the program. If a variable is declared globally (outside of a blocked section) it is not referred to from within a block of code when there's a local variable with the same name.

Behavior of global variables is undefined when declared after a function declaration.

```
int a;
int func(){
        int a;
        a = 10; /* does not refer to global var */
}
```

Project Plan

Process

All four team members tried to meet together at least once a week. We tried to do all the work when all team members are present to make sure that language design decisions were consistent. We would try to find a room with a TV or monitor, and display the main parts of the code that we are working on at that time (whether it was the parser, semantic checker, code generator). Although one person would be in charge of the file and the typing, all team members would be able to see the code and point out mistakes immediately in this manner. As a result, all four team members were able to contribute to a majority of the project and at the very least know what was going on.

The decision to change the language from a game-based RPG language to a language focused more on testing came after Weiyuan's review of the proposal and the LRM. We were interested in exploring the idea of asserts within structs and the various applications of such a structure in a language. While the RPG aspects of the language were really exciting, we (along with Weiyuan) found a bunch of inconsistencies, and as a result, we had to modify our objective to coincide with the changes in our language structure.

Programming style guide and conventions followed

- Space out different functions by adding extra newlines between each feature
- Comment the parser, scanner, and ast (the front end) so that the LRM will be easy to write
- Write test cases for each stage of development and run previous tests as well
- The main branch is used for architectural development (mostly backend and testing), while the features branch is used for feature additions (front-end)

Timeline

Date	Log
September 22	Meet to discuss proposal and language ideas
September 24	**Proposal Due**
October 8	Met with Weiyuan to discuss proposal feedback
October 24-25	LRM written, Scanner, Parser, and AST started
October 27	**LRM Due**
October 29	LRM reviewed and discussed with TA (Weiyuan)
November 5	Short Meeting with Weiyuan to update on progress
November 8	Final bugs and design issues on Scanner, Parser, and AST resolved, "Pretty Printer" started, LRM revisions
November 12	Short meeting with Weiyuan to discuss testing approaches, check-in
November 14	Pretty printer written and test cases/programs started, LRM revisions
November 22	Automated test script completed, semantic checker started
December 9	Discuss frontend examples, semantic checker
December 10	Semantic checker and testing feature enhancements, create new test suite
December 11	Semantic checker debugging
December 12	Semantic checker debugging, start translation from Sast to Jast (java syntax tree)

December 14	Sast to Jast debugging, start code generation, LRM and final write up
December 15	Code generation, further testing of parser and semantic checker, LRM and final write up, end to end test suite started
December 16	Code generation debugging, integration tests, final writeup, presentation prep
December 17	Code generation debugging, integration tests, final writeup, presentation prep
December 17	**Project Due**

Roles

Our roles shifted based on our areas of expertise and our schedules at certain times in the semester. In general, we stuck to these roles in the end:

Student Name	Roles
Albert Cui	System Architect
Karen Nan	Project Manager / Testing and Validation
Mei-Vern Then	Language Guru / Testing and Validation
Michael Raimi	Language Guru / Testing and Validation

Task Breakdown

Although every team member contributed on every aspect of the project, responsibilities were assigned in the following manner as listed below. Work done by each team member is listed in descending order from most to least contributed.

Language proposal: Karen Nan, Michael Raimi, Albert Cui, Mei-Vern Then
Scanner, Parser, and AST: Albert Cui and Mei-Vern Then, Michael Raimi, Karen Nan
LRM write-up: Karen Nan, Michael Raimi, Mei-Vern Then, Albert Cui
Pretty printer: Mei-Vern Then, Albert Cui, Karen Nan, Michael Raimi
Semantic Checker/SAST: Albert Cui, Mei-Vern Then, Michael Raimi, Karen Nan
Java Intermediate Representation: Albert Cui and Mei-Vern Then
Code Generation: Albert Cui, Mei-Vern Then, Karen Nan, Michael Raimi
Test cases: Michael Raimi, Mei-Vern Then, Karen Nan, Albert Cui
Example program: Mei-Vern Then, Michael Raimi, Karen Nan, Albert Cui
Testing automation*:* Karen Nan, Michael Raimi, Mei-Vern Then, Albert Cui

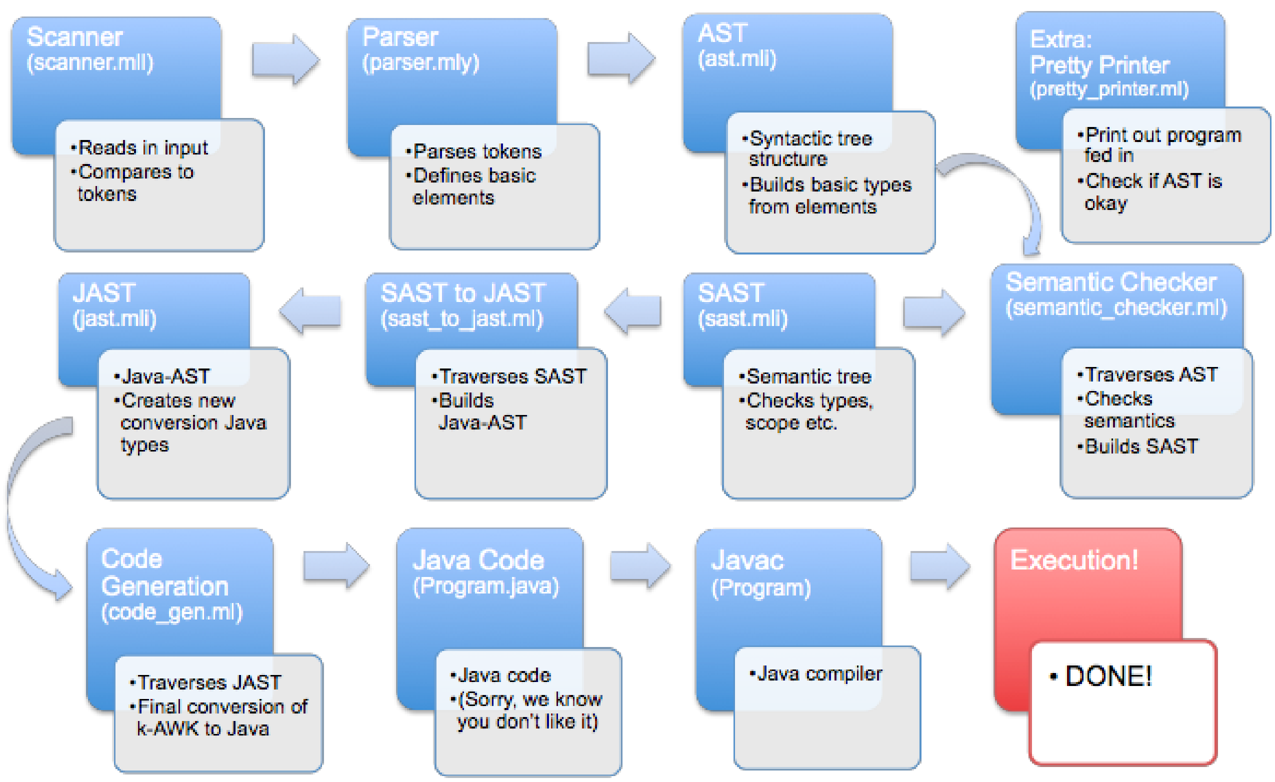
Final writeup: Karen Nan, Michael Raimi, Mei-Vern Then
Powerpoint slides: Mei-Vern Then, Michael Raimi, Karen Nan
Project Management/Scheduling: Karen Nan

Development tools/environments

The languages used to create and test our language were the following: Ocaml for language creation/translation, Bash Shell and Python for testing, and Java for validation of our generated code.

SublimeText was used as a text editor to develop our program. The Mac OSX Terminal was used to run our programs, including the bash scripts for the automated test scripts. For Ocaml debugging, Menhir was used. For version control, github was used for our program. Final writeup and the LRM were generated using Google Docs, and the presentation was generated over Google Presentations.

Architectural Design



The figure above describes the system architecture of our language. After Albert Cui started the basic framework of the scanner, parser, and AST, Michael Raimi and Karen Nan focused mostly on adding features to the front-end (scanner, parser, AST, and pretty printer), while Mei-Vern Then and Albert Cui worked extensively on the backend (Semantic checker, SAST,

SAST to JAST, Code Generation). Testing of the semantic checker and SAST was done by Mei-Vern Then and Michael Raimi, while Karen Nan did most of the syntax checking and debugging of the scanner, parser, and AST.

We made the decision to compile down to Java, since Java yielded an easy way for us to test the output since all of the team members were familiar with the language. The design pattern of getters and setters in Java also yielded an interesting translation for the asserts, since Java would be able to detect changes in the member variables of an object through the setter function. Since you would need to go through the setter, we could write a series of statements to evaluate every time the value of an member function changes and the setter function is called.

To compile down to Java, we needed a series of syntax trees. After semantic checking, we created an intermediate representation in the Java Syntax Tree (JAST) from the Semantic Abstract Syntax Tree (SAST).

Testing Plan

Testing structure

We conducted a variety of unit and integration tests throughout the development cycle of this language. After completing the pretty printer, Michael and Karen wrote some sample code, passed it to the pretty printer, and compared the input program to that of the output string to ensure the proper parsing of each feature and data structure in our language.

Unit testing (having a separate test script loop for the pretty printing generated after the AST, a test loop for the output of the semantic checker, and a test loop for the generated Java Code and then Java compilation) was essential in our complete script to determine when things broke, and if adding new features would break earlier parts of the architecture. By doing this, we were able to expose errors that stemmed from all layers of the program, as well as finding previously undiscovered errors that were more noticeable in the later stages after semantic analysis.

As the program and language started coming together, we found that integration testing was more efficient and helpful to see if the language works and compiles as intended in the LRM. We tried to cover all cases, using test cases to verify known issues to randomly testing features as written in the LRM. Mei-Vern mostly tested known issues in the semantic checker, Michael tested the semantic checker based on rules in the LRM, and Karen tested syntax errors based on rules in the LRM.

Automation was done by writing a shell script and python script *(`test_logic.py`* and *`test_suite.sh`)* that accepted and compared each test program and test case written in k-AWK. Karen Nan wrote these two scripts.

All test cases were placed in the `test` folder:

Test files without a “`semantic_`” prefix went through only the pretty printer (AST) compilation (syntax) tests

Test files with a “`semantic_`” prefix went through testing through the semantic checker stage (SAST)

Test files with a “`_reject`” appended to it were tests that were supposed to fail with a syntax error (thus, if the AST returns an error, the test is Accepted/Passed)

Test files with a “`_rejectsem`” appended to it were tests that were supposed to fail with a semantic error (thus, if the SAST/semantic checker catches a semantic error, the test is Accepted/Passed)

Messages were output into the console in the following format:

`output*.k or output*.txt (java)`

This file either returned a working (parsed and printed) program in the .k format, or recorded errors and exceptions.

The python script does some more complicated logic with whether a test is accepted/rejected, takes out all whitespace in generated inputs and outputs for string comparison.

For each file, the name of the file and its output file name is printed and whether the test passes (ACCEPT) or fails (REJECT).

Calling `./test_suite.sh` starts the automated scripts

Automated test scripts

Calling `./test_suite.sh` starts the automated scripts

`test_suite.sh`

```
#/bin/bash
echo "-----------------------------------------"
echo "-----------------------------------------"
echo "----------BEGIN PRETTY PRINT-------------"
echo "-----------------------------------------"
echo "-----------------------------------------"

cd ../
```

```
make clean
if make pretty
	then
		cd test/
		for filename in *.k; do
			../../pretty < "$filename" > "output_$filename" 2>&1
			python test_logic.py pretty "$filename" "output_$filename"
		done
else
	exit
fi
cd ../
if make
	then
		cd test/
		echo "-----------------------------------------"
		echo "-----------------------------------------"
		echo "-----------BEGIN SEMANTIC CHK------------"
		echo "-----------------------------------------"
		echo "-----------------------------------------"
		rm -f test/output_semantic*.k
		for filename in semantic*.k; do
			../../code_gen < "$filename" > "output_$filename" 2>&1
			python test_logic.py semantic "$filename" "output_$filename"

		done
		echo "-----------------------------------------"
		echo "-----------------------------------------"
		echo "-----------------END---------------------"
		echo "-----------------------------------------"
		echo "-----------------------------------------"

		echo "-----------------------------------------"
		echo "-----------------------------------------"
		echo "---------BEGIN Java Compilation----------"
		echo "-----------------------------------------"
		echo "-----------------------------------------"
		for filename in *.k; do
			../../code_gen < "$filename" > "Program.java" 2>&1
			javac "Program.java"
			java "Program" > "output_java_$filename.txt" 2>&1
		done
		echo "-----------------------------------------"
		echo "-----------------------------------------"
		echo "-----------------END---------------------"
		echo "-----------------------------------------"
		echo "-----------------------------------------"

else
```

```
        exit
fi
```

test_logic.py

```
#A simple test script to test for equality of
#the output after parsed by the trees and printed by the pretty printer
#This will be expanded to accommodate more tests in the future

import sys
mode = sys.argv[1]
inputFileName = sys.argv[2]
outputFileName = sys.argv[3]
semanticReject = False
semantic = False
syntax = True
syntaxReject = False
if (inputFileName[:8] == "semantic"):
        semantic = True
        syntax = False
        if (inputFileName[-11:] == "rejectsem.k"):
                semanticReject = True
else:
        print inputFileName[-8:] == "reject.k"
        if (inputFileName[-8:] == "reject.k"):
                syntaxReject = True

shouldReject = False

#convert the string by taking out spaces, newlines, and tabs
def convert_str(thestring):
        testStr = ''.join(thestring.split())
        return testStr

#read the files
file1 = open(inputFileName, 'r')
inputFileStr = file1.read()
file2 = open(outputFileName, 'r')
outputFileStr = file2.read()

#convert to string
inString = str(inputFileStr)
outString = str(outputFileStr)
inputTestStr = convert_str(inString)
outputTestStr = convert_str(outString)
```

```
if (syntax == True and mode == "pretty"):
	print inputFileName
	print outputFileName
	if (outputTestStr == inputTestStr and syntaxReject == False):
		print "Syntax test ACCEPTED\n"
	elif (outputTestStr != inputTestStr and syntaxReject == True):
		print "Syntax test ACCEPTED\n"
	else:
		print "Syntax test REJECTED"
		print "The input file:"
		print inputFileStr
		print "The output file:"
		print outputFileStr
		print "\n"

elif (semantic == True and mode == "semantic" ):
	print inputFileName
	print outputFileName

	#print outputTestStr
	try:
		outputTestStr.index("Fatalerror:")
		if (semanticReject == True):
			print "Semantic test ACCEPTED\n"
		elif (semanticReject == False):
			print "Semantic Test REJECTED"
			print "The input file:"
			print inputFileStr
			print "The error:"
			print outputFileStr
			print "\n"
	except ValueError:
		if (semanticReject == False):
			print "Semantic test ACCEPTED\n"
		elif (semanticReject == True):
			print "Semantic Test REJECTED"
			print "The input file:"
			print inputFileStr
			print "The error:"
			print outputFileStr
			print "\n"
```

Sample input/outputs

Pretty Printer example:

call_function.k

```
int main(){
        int a;
        a = 3;
        return a;
}
int main2(){
        main();
}
```

```
./pretty < "call_function.k" > "output_call_funtion.k"
```

output_call_function.k

```
int main() {
        int a;
        a = 3 ;
        return a ;
}
int main2() {
        main() ;
}
```

Console output (after going through test suite)

```
call_function.k
output_call_function.k
ACCEPT
```

Input for code generation

./code_gen < test/semantic_int_dollarSign_str_rejectsem.k

```
int main(){
        int a = 1;
        str b = "1";
        if (a $ b)
                return 1;
        return 0;
}
```

Output:

```
Fatal error: exception Failure("illegal character $")
```

Lessons Learned

Albert Cui

A lot of code that we wrote earlier had to be rewritten because we didn't truly understand the specs of the assignment, or how the different layers interacted with each other. For the semantic checker between the AST and the SAST for example, we cut a lot of corners in terms of redefining types, which came back later to haunt us. I think we ended up rewriting the semantic checker about 4 times, which could have been avoided had we had a better understanding of the assignment. We also spend too much time on the pretty printer, which (though it was helpful for writing code generation) was not actually part of the compiler. We probably also should've focused on getting something simple like "Hello World" to print and slowly adding more and more features, but instead we focused on trying to get each layer working by itself, which (again) was difficult because we didn't know how they worked with each other, resulting in a lot of redundant code that had to be rewritten or deleted.

Karen Nan

It goes without saying that it never hurts to start early, and even if a team starts early, to not lose sight of the goal as the semester ends. We met regularly but took a break during Thanksgiving week and the last week of class, which resulted in us falling behind. A big thing that I wish we've done differently was to get the simplest end-to-end compiler working before perfecting the more complicated features in each layer. I feel that if we'd done that it would have been easier to split the work into manageable chunks for members to complete on their own, as opposed to having to work together to try to iron out all the bugs in a complicated, not-compiling semantic checker. Another lesson learned is to write good test cases early, so that major problems in the code will be caught earlier. These test cases should have been better defined based on our decisions for our language, for example about how structs and arrays are accessed and modified. If we had been more decisive and definite about our language from the earlier stages instead of making decisions as we were developing our program, the testing process would have gone smoother.

Mei-Vern Then

Pair programming is definitely an effective way to learn and it was helpful to have at least two people writing code at any given moment. We would plug in our computers to a larger monitor so we could look at each other's code, and help brainstorm on whiteboards whenever we ran into any design issues. This made working rather slow, but at least it improved morale (especially when we worked late into the night). It was also helpful because we were constantly pushing new code when we fixed bugs (which we found a lot of, many by accident), so one of us could constantly test the new corrected code and make the necessary changes. We also probably should've had a better testing system earlier on and clearly defined rules for initializing and accessing data types in our language, because we ended up making a lot of design decisions on the fly, which caused confusion when we coded various stages of the compiler. We also didn't find a lot of bugs until later on (while writing code generation), so we had to go all the way back to the parser and consequently fix every stage after that, thus reducing the time we had to work on actual code generation.

Michael Raimi

We should have managed our time better, and had clearly defined goals and tasks we wanted to accomplish each time we met. We were actually on a pretty good timeline in the earlier stages, but not meeting one week set us back quite a bit. Not having actual goals (e.g. having the semantic checker done by the end of one marathon session) caused us to get pretty sidetracked sometimes. We should also have more clearly defined the rules of our language and properly designed it out at the beginning, as there was a lot of confusion in terms of syntax and semantics on how we wanted to define things. This led to a lot of debugging and forced us to spend more time fixing old things we thought we were done with, rather than working on the next stages of the compiler. A lot of design decisions could have been made earlier on with some forethought, but what ended up happening was we would make decisions while coding, which later made things harder for us in terms of implementation (because we didn't think far enough ahead).

Appendix

```
README.md
-----------------

# kAWK "kay-awk" (formerly: GAWK)
## The Testing Language


## Team members:
* Albert Cui (System Architect)
* Karen Nan (Project Manager/Testing and Validation)
* Michael Raimi (Testing and Validation/Language Guru)
* Mei-Vern Then (System Architect/Language Guru)

### To build and clean up:

make

make clean

### Test suite (in test program):

./test_suite.sh

### Pretty printer:

./pretty testprogram.k

### Code generator:

./code_gen testprogram.k

### To see output from any program, include a input redirect:

./code_gen < testprogram.k

Makefile
-----------------

default: code_gener pretty semantic sast_jast

code_gener: scanner parser semantic sast_to_jast code_gen
	ocamlc -o code_gen scanner.cmo parser.cmo semantic_checker.cmo
sast_to_jast.cmo code_gen.cmo
```

```
sast_jast: scanner parser semantic sast_to_jast
	ocamlc -o sast_to_jast scanner.cmo parser.cmo semantic_checker.cmo
sast_to_jast.cmo

pretty: scanner parser pretty_printer
	ocamlc -o pretty parser.cmo scanner.cmo pretty_printer.cmo

code_gen: jast
	ocamlc -c code_gen.ml

semantic: scanner parser semantic_checker
	ocamlc -o semantic parser.cmo scanner.cmo semantic_checker.cmo

sast_to_jast: jast
	ocamlc -c sast_to_jast.ml

semantic_checker: sast scanner
	ocamlc -c semantic_checker.ml

scanner: parser
	ocamllex scanner.mll; ocamlc -c scanner.ml

parser: ast
	ocamlyacc parser.mly; ocamlc -c parser.mli; ocamlc -c parser.ml

jast: sast ast
	ocamlc -c jast.mli

sast: ast
	ocamlc -c sast.mli

ast:
	ocamlc -c ast.mli

pretty_printer:
	ocamlc -c pretty_printer.ml

.PHONEY: clean

clean:
	rm -f test/output*.k
	rm -f code_gen semantic pretty sast_to_jast *.cmo *.cmi *~ parser.mli
parser.ml scanner.ml
```

ast.mli

```
type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater |
Geq | Or | And | Not
```

```
type expr = (* Expressions *)
      Noexpr (* for (;;) *)
      | Id of string (* foo *)
      | Integer_literal of int (* 42 *)
      | String_literal of string (* "foo" *)
      | Boolean_literal of bool
      | Array_access of string * expr (* foo[10] *)
      | Assign of string * expr (* foo = 42 *)
      | Uniop of op * expr
      | Binop of expr * op * expr (* a + b *)
      | Call of string * expr list (* foo(1, 25) *)
      | Access of string * string (* foo.bar *)
      | Struct_Member_Assign of string * string * expr
      | Array_Member_Assign of string * expr * expr

type stmt = (* Statements *)
      Block of stmt list (* { ... } *)
      | Expr of expr (* foo = bar + 3; *)
      | Return of expr (* return 42; *)
      | If of expr * stmt * stmt (* if (foo == 42) {} else {} *)
      | For of expr * expr * expr * stmt (* for (i=0;i<10;i=i+1) { ... } *)
      | While of expr * stmt (* while (i<10) { i = i + 1 } *)

type var_types =
      Void
      | Int
      | String
      | Boolean
      | Struct of string
      | Array of var_types * expr

type fn_param_decl =
      Param of var_types * string

type var_decl =
      Variable of var_types * string
      | Variable_Initialization of var_types * string * expr
      | Array_Initialization of var_types * string * expr list
      | Struct_Initialization of var_types * string * expr list

type struct_decl = {
      sname: string; (* Name of the struct *)
      variable_decls: var_decl list; (* int foo *)
      asserts: (expr * stmt list) list; (* @ (bar > 1) { ... } *)
}

type unit_decl =
```

```
      Local_udecl of expr list * expr * bool
      | Outer_udecl of string * expr list * expr * bool

type func_decl = {
      ftype: var_types;
      fname : string; (* Name of the function *)
      formals : fn_param_decl list; (* Formal argument names *)
      locals : var_decl list; (* Locally defined variables *)
      body : stmt list;
      units : unit_decl list; (* Series of unit tests *)
}

type program = struct_decl list * var_decl list * func_decl list * unit_decl
list (* global vars, funcs *)
```

99_bottles.k

```
 int run_99_bott (int a) {
      struct bott_beer bb;
      int i;
      bb.bottles_of_beer = a;
      for(i = a; i > 0; (i = (i - 1))) {
            bb.bottles_of_beer = i;
      }
      return 0;
 }

 struct bott_beer {
      int bottles_of_beer;
      @(!(bottles_of_beer > 0)) {
                  print(bottles_of_beer);
                  print("bottles of beer on the wall.")
                  print(bottles_of_beer);
                  print("bottles of beer. take one down , pass it around");
                  print(bottles_of_beer-1);
                  print("bottles of beer on the wall.");
      }
 }

 void main() {
      unit:run_99_bott(99):equals(0):accept;
 }
```

demo_gcd.k

```
int run_gcd (int a, int b) {
      while (a != b) {
```

```
            if (a > b) {
                  a = a-b;
            }
            else {
                  b = b-a;
            }
      }
      return a;
}

  void main() {
      unit:run_gcd(24, 54):equals(6):accept;
 }
```

code_gen.ml

```
(* Code gen*)
open Ast
open Sast
open Jast
open Sast_to_jast
open Semantic_checker
open Lexing

let jast =
      let lexbuf = Lexing.from_channel stdin in
      let ast = Parser.program Scanner.token lexbuf in
      let sast = check_program ast in
      sast_to_jast sast

let (j_struct_decl_list, _, _, _) = jast

let print_op = function
      Add -> print_string "+ "
      | Sub -> print_string "- "
      | Mult -> print_string "* "
      | Div -> print_string "/ "
      | Mod -> print_string "% "
      | Equal -> print_string "== "
      | Neq -> print_string "!= "
      | Less -> print_string "< "
      | Leq -> print_string "<= "
      | Greater -> print_string "> "
      | Geq -> print_string ">= "
      | Or -> print_string "|| "
      | And -> print_string "&& "
      | Not -> print_string "!"
```

```
let get_instance_name = function
      Variable(_, str) -> str
      (* if not a Variable we drop the unnecessary stuff *)
      | Variable_Initialization(_, str, _) -> str
      | Array_Initialization(_, str, _) -> str
      | Struct_Initialization(_, str, _) -> str

let print_checked_var_decl = function
      Variable(checked_var_decl, str) -> Printf.printf "%s[" str
      | _ -> Printf.printf "adsf"


let rec print_expr (e : Sast.expression) =
      let (e, _) = e in match e with
      Noexpr -> print_string ""
      | Id(decl) -> let str = match decl with
            Variable(_, str) -> str
            (* if not a Variable we drop the unnecessary stuff *)
            | Variable_Initialization(_, str, _) -> str
            | Array_Initialization(_, str, _) -> str
            | Struct_Initialization(_, str, _) -> str in
            print_string str
      | IntConst(i) -> Printf.printf "%d " i
      | StrConst(str) -> Printf.printf "%s " str
      | BoolConst(b) -> Printf.printf "%B " b
      | ArrayAccess(checked_var_decl, expr) -> print_string(get_instance_name
checked_var_decl); print_string "["; print_expr expr; print_string "]"
      | Assign(decl, expr) -> let str = match decl with
            Variable(_, str) -> str
            (* if not a Variable we drop the unnecessary stuff *)
            | Variable_Initialization(_, str, _) -> str
            | Array_Initialization(_, str, _) -> str
            | Struct_Initialization(_, str, _) -> str in
            print_string (str^" = "); print_expr expr
      | Uniop(op, expr) -> print_op op; print_string "("; print_expr expr;
print_string ")"
      | Binop(expr1, op, expr2) -> print_expr expr1; print_op op; print_expr
expr2
      | Call(f, expr_list) ->
            if f.fname = "exit" then (print_string "\n\tSystem.out.println(";
List.iter print_expr expr_list; print_string ");\n\tSystem.exit(0)")
            else
                  ((if f.fname = "print" then print_string
"\n\tSystem.out.println("
                  else Printf.printf "%s(" f.fname);
                  let rec print_expr_list_comma = function
                        [] -> print_string ""
```

```
                        | e::[] -> print_expr e
                        | e::tl -> print_expr e; print_string ", ";
print_expr_list_comma tl
                        in print_expr_list_comma (List.rev expr_list);
print_string ")")
      | Access(struc, instance, decl) ->
            let j_s_decl = List.find ( fun j -> j.original_struct = struc)
j_struct_decl_list in
            let var = List.find ( fun j_v -> let (v, _) = j_v.the_variable in v
= decl) j_s_decl.variable_decls in
            print_string (get_instance_name instance);
print_string("."^var.name)
      | Struct_Member_Assign(struc, instance, decl, expr) ->
            let j_s_decl = List.find ( fun j -> j.original_struct = struc)
j_struct_decl_list in
            let var = List.find ( fun j_v -> j_v.the_variable = decl)
j_s_decl.variable_decls in
            if (List.length var.asserts) <> 0 then (print_string
(get_instance_name instance); print_string (".set_" ^ var.name ^ "(");
print_expr expr; print_string ")")
            else
                  (print_string (get_instance_name instance); print_string ".";
print_string (var.name ^ "="); print_expr_semi expr)
      | Array_Member_Assign (decl, idx, expr) ->
            print_string (get_instance_name decl); print_string ("[");
print_expr expr; print_string("] = "); print_expr expr
      (* | _ -> print_string "" *)
and print_expr_semi (e : Sast.expression) =
      print_expr e; print_string ";\n"

let rec print_expr_list_comma (el : Sast.expression list) = match el with
      [] -> print_string ""
      | hd::[] -> print_expr hd
      | hd::tl -> print_expr hd; print_string ", "; print_expr_list_comma tl

let rec print_stmt = function
      Block(stmt_list) -> print_string "{";  List.iter print_stmt (List.rev
stmt_list); print_string "}\n"
      | Expr(expr) -> print_expr_semi expr
      | Return(expr) -> print_string "return "; print_expr_semi expr
      | If(expr, stmt1, stmt2) -> print_string "if ("; print_expr expr;
print_string ") "; print_stmt stmt1; print_string "else "; print_stmt stmt2
      | For(expr1, expr2, expr3, stmt) -> print_string "for ("; print_expr_semi
expr1; print_expr_semi expr2; print_expr expr3; print_string ")"; print_stmt
stmt;
      | While(expr, stmt) -> print_string "while ("; print_expr expr;
print_string ")"; print_stmt stmt
```

```
let rec print_var_types = function
      Void -> print_string "void "
      | Int -> print_string "int "
      | String -> print_string "String "
      | Boolean -> print_string "boolean "
      | Struct(s) -> Printf.printf "%s " (String.capitalize s.sname)
      | Array(var_types, expr) ->
            print_var_types var_types;
            print_string "[";
            print_expr expr;
            print_string "] "

let print_param v =
      let (var_types, _) = v in match var_types with
      Variable(var_types, str) -> print_var_types var_types; print_string str
      (* if not a Variable we drop the unnecessary stuff *)
      | Variable_Initialization(var_types, str, _) -> print_var_types
var_types; print_string str
      | Array_Initialization(var_types, str, _) -> print_var_types var_types;
print_string str
      | Struct_Initialization(var_types, str, _) -> print_var_types var_types;
print_string str

let rec print_var_decl  (v : Sast.variable_decl) =
      let (var_types, _) = v in match var_types with
            Variable(var_types, str) -> (match var_types with
                  Struct(decl) ->
                        let s = List.find (fun j -> j.original_struct = decl)
j_struct_decl_list in
                        print_string (String.capitalize s.sname); Printf.printf
" %s = new %s();\n" str (String.capitalize s.sname)
                  | _ -> print_var_types var_types; print_string (str ^ ";\n"))
            | Variable_Initialization(var_types, str, expr) -> print_var_types
var_types; Printf.printf "%s = " str; print_expr_semi expr
            | Array_Initialization(var_types, str, expr_list) -> (match
var_types with
                  Array(var_types, _) -> print_var_types var_types;
Printf.printf "[] %s = { " str; print_expr_list_comma expr_list; print_string
"};\n"
                  | _ -> raise (Failure "Not an array"))
            | Struct_Initialization(var_types, str, expr_list) -> match
var_types with
                  Struct(decl) ->
                        let s = List.find (fun j -> j.original_struct = decl)
j_struct_decl_list in
                        print_string (String.capitalize s.sname); Printf.printf
" %s = new %s(" str (String.capitalize s.sname); print_expr_list_comma
(List.rev expr_list); print_string ");\n"
```

```
                | _ -> raise (Failure "shouldn't happen")

let rec print_function_params (v : Jast.j_var_struct_decl list) = match v with
      [] -> print_string "";
      | hd::[] -> print_param hd.the_variable;
      | hd::tl -> print_param hd.the_variable; print_string ", ";
print_function_params tl

let print_asserts a_list =
      List.iter (
            fun (expr, stmt_list) ->
                  print_string "if(!(";
                  print_expr expr;
                  print_string ")){\n";
                  List.iter ( fun s ->
                        print_stmt s; print_string "\n"
                  ) stmt_list;
                  print_string "}\n"
      ) a_list

let print_j_var_decl (dec : j_var_struct_decl) =
      print_var_decl dec.the_variable;
      if (List.length dec.asserts) <> 0 then
            (
                  print_string("\npublic void set_" ^ dec.name ^ "(");
                  print_param dec.the_variable;
                  print_string "){\n";
                  print_string ("this." ^ dec.name ^ "=" ^ dec.name ^ ";\n");
                  print_asserts dec.asserts;
                  print_string "}\n"
            )
      else ()

let print_constructors (name : string) (s : Jast.j_var_struct_decl list) =
      print_string ("public " ^ (String.capitalize name) ^ "(");
      print_function_params s;
      print_string "){\n";
      List.iter (
            fun dec -> print_string ("this." ^ dec.name ^ "=" ^ dec.name ^
";\n")
      ) s;
      print_string "\n}\n";
      (* Empty constructor*)
      print_string ("public " ^ (String.capitalize name) ^ "(){}\n")

let print_struct_decl (s : Jast.j_struct_decl) =
      print_string "static class ";
      print_string (String.capitalize s.sname);
```

```
        print_string " {\n\t\t";
        List.iter print_j_var_decl s.variable_decls;
        (* Make the constructors *)
        print_constructors s.sname s.variable_decls;
        print_string "\n}\n"

let print_unit_decl (u : Sast.unit_decl) = match u with
        Outer_udecl(str, udecl_params, udecl_check_val, true) -> print_string
"if("; print_string (str.fname ^ "(");  print_expr_list_comma udecl_params;
print_string ")==("; print_expr udecl_check_val; print_string "))
{System.out.println(\"The test passes\");} else {System.out.println(\"The test
fails\");} \n"
        | Outer_udecl(str, udecl_params, udecl_check_val, false) -> print_string
"if("; print_string (str.fname ^ "(");  print_expr_list_comma udecl_params;
print_string ")==("; print_expr udecl_check_val; print_string "))
{System.out.println(\"The test fails\");} else {System.out.println(\"The test
passes\");}\n"
        | Local_udecl(udecl_params, udecl_check_val, false) -> print_string
"local_inner_false:"
        | Local_udecl(udecl_params, udecl_check_val, true) -> print_string
"local_inner_true:"

let rec print_param_list (p : Sast.variable_decl list) = match p with
        [] -> print_string "";
        | hd::[] -> print_param hd;
        | hd::tl -> print_param hd; print_string ", "; print_param_list tl

let print_func_decl (f : Sast.function_decl) =
        if f.fname = "main" then
                (print_string "public static void main(String[] args) {\n";
                List.iter print_var_decl (List.rev f.checked_locals);
                List.iter print_stmt (List.rev f.checked_body);
                List.iter print_unit_decl (List.rev f.checked_units);
                print_string "}")
        else
                (
                        print_string " static ";
                        print_var_types f.ftype;
                        print_string f.fname;
                        print_string "(";
                        print_param_list (List.rev f.checked_formals);
                        print_string ") {\n";
                        List.iter print_var_decl (List.rev f.checked_locals);
                        List.iter print_stmt (List.rev f.checked_body);
                        List.iter print_unit_decl (List.rev f.checked_units);
                        print_string "}\n"
                )
```

```
let code_gen j =
      let _ = print_string "public class Program {\n\n\t" in
      let (structs, vars, funcs, unts) = j in
                  List.iter print_struct_decl (List.rev structs);
                  List.iter print_var_decl (List.rev vars);
                  List.iter print_func_decl (List.rev funcs);
                  List.iter print_unit_decl (List.rev unts);
                  print_string "\n}\n"

let _ =
      code_gen jastjast.mli
-----------------

open Sast


type j_var_struct_decl = {
      name: string;
      the_variable: variable_decl; (* int a *)
      mutable asserts: (expression * stmt list) list; (* @ (bar > 1) { ... } *)
}

type j_struct_decl = {
      sname: string; (* Name of the struct *)
      variable_decls: j_var_struct_decl list; (* list of asserts/shared
variables *)
      original_struct: Sast.struct_decl;
      mutable j_name: string;
}

(* type j_func_decl = {
      f_decl: Sast.function_decl;
      mutable j_name: string;
}

type variable_decl
 *)
type program = j_struct_decl list * variable_decl list * function_decl list *
unit_decl listpretty_printer.ml
-----------------

open Ast
open Lexing

let print_op = function
      Add -> print_string "+ "
      | Sub -> print_string "- "
      | Mult -> print_string "* "
```

```
	| Div -> print_string "/ "
	| Mod -> print_string "% "
	| Equal -> print_string "= "
	| Neq -> print_string "!= "
	| Less -> print_string "< "
	| Leq -> print_string "<= "
	| Greater -> print_string "> "
	| Geq -> print_string ">= "
	| Or -> print_string "| "
	| And -> print_string "& "
	| Not -> print_string "! "

let rec print_expr = function
	Noexpr -> print_string ""
	| Id(id) -> Printf.printf "%s " id
	| Integer_literal(i) -> Printf.printf "%d " i
	| String_literal(str) -> Printf.printf "%s " str
	| Boolean_literal(b) -> Printf.printf "%B " b
	| Array_access(str, expr) -> Printf.printf "%s[" str; print_expr expr;
print_string "]"
	| Assign(str, expr) -> Printf.printf "%s = " str; print_expr expr
	| Uniop(op, expr) -> print_op op; print_expr expr
	| Binop(expr1, op, expr2) -> print_expr expr1; print_op op; print_expr
expr2
	| Call(str, expr_list) -> Printf.printf "%s(" str; let rec
print_expr_list_comma = function
	[] -> print_string ""
	| hd::[] -> print_expr hd
	| hd::tl -> print_expr hd; print_string ", "; print_expr_list_comma tl
	in print_expr_list_comma expr_list; print_string ") "
	| Access(str1, str2) -> Printf.printf "%s.%s " str1 str2
	| Struct_Member_Assign(str1, str2, expr) -> Printf.printf "%s" str1;
print_string "."; Printf.printf "%s = " str2; print_expr expr
	| Array_Member_Assign(str1, expr1, expr2) -> Printf.printf "%s" str1;
print_string "["; print_expr expr1; print_string "] = "; print_expr expr2

let rec print_expr_list_comma = function
	[] -> print_string ""
	| hd::[] -> print_expr hd
	| hd::tl -> print_expr hd; print_string ", "; print_expr_list_comma tl

(* and print_expr_comma expr =
	print_expr expr; print_string ", "
	 *)
let print_expr_semi e =
	print_expr e; print_string ";\n"

let rec print_expr_list = function
```

```
      [] -> print_string ""
      | hd::[] -> print_expr hd
      | hd::tl -> print_expr hd; print_string "; "; print_expr_list tl

let rec print_stmt = function
      Block(stmt_list) -> print_string "{"; List.iter print_stmt stmt_list;
print_string "}\n"
      | Expr(expr) -> print_expr_semi expr
      | Return(expr) -> print_string "return "; print_expr_semi expr
      | If(expr, stmt1, stmt2) -> print_string "if ("; print_expr_semi expr;
print_string ")"; print_stmt stmt1; print_stmt stmt2
      | For(expr1, expr2, expr3, stmt) -> print_string "for ("; print_expr_semi
expr1; print_string ";"; print_expr_semi expr2; print_string ";"; print_expr
expr3; print_stmt stmt
      | While(expr, stmt) -> print_string "while ("; print_expr_semi expr;
print_string ")"; print_stmt stmt

let rec print_var_types = function
      Void -> print_string "void "
      | Int -> print_string "int "
      | String -> print_string "str "
      | Boolean -> print_string "bool "
      | Struct(str) -> Printf.printf "struct %s " str
      | Array(var_types, expr) -> print_var_types var_types; print_string "[";
print_expr expr; print_string "] "

let rec print_var_decl = function
      Variable(var_types, str) -> print_var_types var_types; print_string (str
^ ";\n")
      | Variable_Initialization(var_types, str, expr) -> print_var_types
var_types; Printf.printf "%s = " str; print_expr_semi expr
      | Array_Initialization(var_types, str, expr_list) -> print_var_types
var_types; Printf.printf "[]%s = { " str; print_expr_list_comma expr_list;
print_string "};\n"
      | Struct_Initialization(var_types, str, expr_list) -> print_var_types
var_types; Printf.printf "%s = { " str; List.iter print_expr expr_list;
print_string "};\n"

let print_asserts a =
      let (expr, stmt_list) = a in
      print_string "@("; print_expr expr; print_string ") "; List.iter
print_stmt stmt_list

(* FIX THIS *)
let print_struct_decl s =
      print_string "struct ";
      print_string s.sname;
      print_string " {\n";
```

```
      List.iter print_var_decl s.variable_decls;
      List.iter print_asserts s.asserts;
      print_string "}"


let print_unit_decl = function
      Local_udecl(udecl_params, udecl_check_val, true) -> print_string "unit(";
print_expr_list_comma udecl_params; print_string "):equals("; print_expr
udecl_check_val; print_string "):accept;\n"
      | Local_udecl(udecl_params, udecl_check_val, false) -> print_string
"unit("; print_expr_list_comma udecl_params; print_string "):equals(";
print_expr udecl_check_val; print_string "):reject;\n"
      | Outer_udecl(str, udecl_params, udecl_check_val, true) -> print_string
"unit:"; print_string (str ^ "(");  print_expr_list_comma udecl_params;
print_string "):equals("; print_expr udecl_check_val; print_string
"):accept;\n"
      | Outer_udecl(str, udecl_params, udecl_check_val, false) -> print_string
"unit:"; print_string (str ^ "(");  print_expr_list_comma udecl_params;
print_string "):equals("; print_expr udecl_check_val; print_string
"):reject;\n"

let print_param = function
      Param(var_types, str) -> print_var_types var_types; print_string (str)

let rec print_param_list = function
      [] -> print_string "";
      | hd::[] -> print_param hd;
      | hd::tl -> print_param hd; print_string ", "; print_param_list tl

let print_func_decl f =
      print_var_types f.ftype;
      print_string f.fname;
      print_string "(";
      print_param_list f.formals;
      print_string ") {\n";
      List.iter print_var_decl f.locals;
      List.iter print_stmt f.body;
      List.iter print_unit_decl f.units;
      print_string "}\n"

let print_program p =
      let (structs, vars, funcs, unts) = p in
            List.iter print_struct_decl structs;
            List.iter print_var_decl vars;
            List.iter print_func_decl (List.rev funcs);
            List.iter print_unit_decl unts

let print_position outx lexbuf =
```

```
  let pos = lexbuf.lex_curr_p in
  Printf.fprintf outx "%s:%d:%d" pos.pos_fname
    pos.pos_lnum (pos.pos_cnum - pos.pos_bol + 1)

let _ =
      let lexbuf = Lexing.from_channel stdin in
      let program = try
      Parser.program Scanner.token lexbuf
      with _ -> Printf.fprintf stderr "%a: syntax error\n" print_position
lexbuf; exit (-1) in
      print_program program
```

parser.mly

```
%{ open Ast %}

%token SEMI COLON LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA
%token MINUS TIMES DIVIDE MOD STRING INT EOF OR AND NOT PLUS
%token ASSIGN EQ NEQ LT LEQ GT GEQ RETURN IF ELSE FOR WHILE BOOL
%token ACCESS STRUCT ASSERT UNIT THIS VOID EQUALS ACCEPT REJECT
%token <string> ID
%token <int> INT_LITERAL
%token <string> STRING_LITERAL
%token <bool> BOOL_LITERAL

%nonassoc ID
%nonassoc NOELSE /* Precedence and associativity of each operator */
%nonassoc ELSE
%nonassoc LBRACK RBRACK
%left ASSERT
%left ACCESS
%right ASSIGN
%left OR AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT

%start program /* Start symbol */
%type <Ast.program> program /* Type returned by a program */

%%

program:
      /* nothing */      { [], [], [], [] }
      | program sdecl { let (str, var, func, unt) = $1 in $2::str, var, func,
unt }
```

```
      | program vdecl { let (str, var, func, unt) = $1 in str, $2::var, func,
unt } /* int world = 4; */
      | program fdecl { let (str, var, func, unt) = $1 in str, var, $2::func,
unt }
      | program udecl { let (str, var, func, unt) = $1 in str, var, func,
$2::unt }

fdecl:
      the_type ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list
udecl_list RBRACE
      { { ftype   = $1;
            fname   = $2;
            formals = $4;
            locals  = List.rev $7;
            body    = List.rev $8;
            units = List.rev $9; } }

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

formal_list:
      the_type ID              { [Param($1, $2)] }
      | formal_list COMMA the_type ID      { Param($3, $4) :: $1 }

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

vdecl:
      the_type LBRACK RBRACK ID ASSIGN LBRACE expr_list RBRACE SEMI {
Array_Initialization(Array($1, Noexpr), $4, List.rev $7) }
      | the_type ID SEMI { Variable($1, $2) }
      | the_type ID ASSIGN expr SEMI { Variable_Initialization($1, $2, $4) }
      | the_type ID ASSIGN LBRACE expr_list RBRACE SEMI {
Struct_Initialization($1, $2, List.rev $5) }

/* ------------- udecl stuff --------------*/


udecl_list:
      /* nothing */            { [] }
      | udecl_list udecl       { $2 :: $1 }

udecl:
      UNIT LPAREN actuals_opt RPAREN COLON EQUALS LPAREN expr RPAREN COLON
ACCEPT SEMI { Local_udecl($3, $8, true) }
```

```
      | UNIT LPAREN actuals_opt RPAREN COLON EQUALS LPAREN expr RPAREN COLON
REJECT SEMI { Local_udecl($3, $8, false) }
      | UNIT COLON ID LPAREN actuals_opt RPAREN COLON EQUALS LPAREN expr RPAREN
COLON ACCEPT SEMI { Outer_udecl($3, $5, $10, true) }
      | UNIT COLON ID LPAREN actuals_opt RPAREN COLON EQUALS LPAREN expr RPAREN
COLON REJECT SEMI { Outer_udecl($3, $5, $10, false) }

/* ------------- end udecl stuff --------------*/

assert_list:
      /* nothing */ { [] }
      | assert_list asrt { $2 :: $1 }

asrt:
      ASSERT LPAREN expr RPAREN stmt_list { $3, List.rev $5 }

expr_list:
      expr { [$1] }
      | expr_list SEMI expr { $3 :: $1 }
      | expr_list COMMA expr { $3 :: $1 } /*will this work for udecl? */

sdecl:
      STRUCT ID LBRACE vdecl_list assert_list RBRACE
      { { sname = $2;
            variable_decls = List.rev $4;
            asserts = List.rev $5; } }

the_type:
      INT { Int }
      | VOID { Void }
      | STRING { String }
      | BOOL { Boolean }
      | STRUCT ID { Struct($2) }
      | the_type LBRACK expr RBRACK { Array($1, $3) }

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

init:
      ID LBRACK RBRACK ASSIGN block SEMI { Array_Initialization($1, $5) }*/

stmt:
      expr SEMI
                  { Expr($1) }
      | RETURN expr SEMI
            { Return($2) }
```

```
	| block
				{ $1 }
	/*| IF LPAREN expr RPAREN stmt %prec NOELSE
{ If($3, $5, Block([])) } */
	| IF LPAREN expr RPAREN stmt ELSE stmt
	{ If($3, $5, $7) }
	| FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt	{
For($3, $5, $7, $9) }
	| WHILE LPAREN expr RPAREN stmt
	{ While($3, $5) }

block:
	LBRACE stmt_list RBRACE { Block(List.rev $2) }

expr_opt:
	/* nothing */	{ Noexpr }
	| expr			{ $1 }


expr:
	ID							{ Id($1) }
	| INT_LITERAL						{ Integer_literal($1) }
	| STRING_LITERAL					{ String_literal($1) }
	| BOOL_LITERAL						{ Boolean_literal($1) }
	| NOT expr						{ Uniop(Not, $2) }
	| expr PLUS expr					{ Binop($1, Add, $3) }
	| expr MINUS expr					{ Binop($1, Sub, $3) }
	| expr TIMES expr					{ Binop($1, Mult, $3) }
	| expr DIVIDE expr					{ Binop($1, Div, $3) }
	| expr MOD expr						{ Binop($1, Mod, $3) }
	| expr EQ expr						{ Binop($1, Equal, $3) }
	| expr NEQ expr						{ Binop($1, Neq, $3) }
	| expr LT expr						{ Binop($1, Less, $3) }
	| expr LEQ expr						{ Binop($1, Leq, $3) }
	| expr GT expr						{ Binop($1, Greater, $3)
}
	| expr GEQ expr						{ Binop($1, Geq, $3) }
	| expr OR expr						{ Binop ($1, Or, $3) }
	| expr AND expr						{ Binop ($1, And, $3) }
	| ID ACCESS ID						{ Access ($1, $3) }
	| ID ASSIGN expr					{ Assign ($1, $3) }
	| ID LPAREN actuals_opt RPAREN				{ Call ($1, $3) }
	| ID ACCESS ID ASSIGN expr		{ Struct_Member_Assign($1, $3, $5) }
	| ID LBRACK expr RBRACK ASSIGN expr	{ Array_Member_Assign($1, $3, $6) }
	| LPAREN expr RPAREN					{ $2 }
	| ID LBRACK expr RBRACK			{ Array_access($1, $3) }

actuals_opt:
```

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

actuals_list:
      expr                                  { [$1] }
      | actuals_list COMMA expr      { $3 :: $1 }run.sh
-----------------

#/bin/bash

./code_gen < "$1" > "Program.java" 2>&1
javac "Program.java"
java "Program" | tee  "output_java_$1.txt" 2>&1
sast.mli
-----------------

open Ast

type var_types =
      Void
      | Int
      | String
      | Boolean
      | Struct of struct_decl
      | Array of var_types * expression
and checked_var_decl =
      Variable of var_types * string
      | Variable_Initialization of var_types * string * expression
      | Array_Initialization of var_types * string * expression list
      | Struct_Initialization of var_types * string * expression list
and variable_decl = checked_var_decl * var_types
and function_decl = {
      ftype: var_types;
      fname : string; (* Name of the function *)
      checked_formals : variable_decl list; (* Formal argument names *)
      checked_locals : variable_decl list; (* Locally defined variables *)
      checked_body : stmt list;
      checked_units : unit_decl list;
}
and unit_decl =
      Local_udecl of expression list * expression * bool
      | Outer_udecl of function_decl * expression list * expression * bool
and struct_decl = {
      sname: string; (* Name of the struct *)
      variable_decls: variable_decl list; (* int foo *)
      asserts: (expression * stmt list) list; (* @ (bar > 1) { ... } *)
}
and expr_detail =
```

```
      Noexpr
      | IntConst of int
      | StrConst of string
      | BoolConst of bool
      | ArrayAccess of checked_var_decl * expression
      | Id of checked_var_decl
      | Call of function_decl * expression list
      | Access of struct_decl * checked_var_decl * checked_var_decl
      | Uniop of op * expression
      | Binop of expression * op * expression
      | Assign of checked_var_decl * expression
      | Struct_Member_Assign of struct_decl * checked_var_decl * variable_decl
* expression
      | Array_Member_Assign of checked_var_decl * expression * expression
and expression = expr_detail * var_types
and stmt =
      Block of stmt list (* { ... } *)
      | Expr of expression (* foo = bar + 3; *)
      | Return of expression (* return 42; *)
      | If of expression * stmt * stmt (* if (foo == 42) {} else {} *)
      | For of expression * expression * expression * stmt (* for
(i=0;i<10;i=i+1) { ... } *)
      | While of expression * stmt

type program = struct_decl list * variable_decl list * function_decl list *
unit_decl list
test
-----------------

undefinedsast_to_jast.ml
-----------------

open Sast
open Jast
open Semantic_checker
open Lexing
(* open Map
 *)
let find_decl (var_decl : Sast.checked_var_decl) (var_list :
Jast.j_var_struct_decl list) =
      List.find (fun v -> let (v, _) = v.the_variable in v = var_decl) var_list

let rec check_j_in_a (j : Sast.variable_decl) (e : Sast.expression) =
      let (the_variable, _) = j in
      let (expr_detail, _) = e in match expr_detail with
            ArrayAccess(var, expr) -> if var = the_variable then true else
check_j_in_a j expr
            | Id(var) -> if var = the_variable then true else false
```

```
            | Call(_, expr_list) -> let rec check_list = function
                  | [] -> false
                  | hd::tl -> if check_j_in_a j e then true else check_list tl
in
                  check_list expr_list
            | Access(_, _,var) -> if var = the_variable then true else false
            | Uniop (_, expr) -> check_j_in_a j expr
            | Binop (expr1, _, expr2) -> check_j_in_a j expr1 || check_j_in_a j
expr2
            | Assign (var, expr) -> if var = the_variable then true else
check_j_in_a j expr
            | _ -> false

let rec check_assert_expr assert_list (var_decl : Sast.variable_decl) a (e :
Sast.expression) =
      try
            let _ = List.find(fun other_assert -> other_assert = a) assert_list
in false
      with Not_found -> if check_j_in_a var_decl e then true else false

(* iterate over s.variable_decls to make
      corresponding j_var_struct_decls intially with empty asserts*)
let process_struct_decl (s : Sast.struct_decl) =
      let j_var_decls = List.fold_left (
            fun a v ->
                  let (decl, _) = v in
                  let id = match decl with
                        Variable(_, id) -> id
                        | Variable_Initialization(_, id, _) -> id
                        | Array_Initialization(_, id, _) -> id
                        | Struct_Initialization(_, id, _) -> id in
                  let asserts = List.fold_left (
                        fun a the_assert ->
                              let (e, _)  = the_assert in
                              if check_assert_expr a v the_assert e then
the_assert :: a
                              else a
                  ) [] s.asserts in
                  {the_variable = v; asserts = asserts; name = id} :: a
      ) [] s.variable_decls in
      { sname = s.sname; variable_decls = j_var_decls; original_struct = s;
j_name = "" }

let sast_to_jast p =
      let (structs, vars, funcs, units) = p in
      let structs = List.fold_left (fun a s -> process_struct_decl s :: a) []
structs in
      (structs, vars, funcs, units)
```

```
(*
let _ =
      let lexbuf = Lexing.from_channel stdin in
      let ast = try
      Parser.program Scanner.token lexbuf
      with _ -> Printf.fprintf stderr "%a: syntax error\n" print_position
lexbuf; exit (-1) in
      let sast = check_program ast in
      sast_to_jast sast  *)scanner.mll
-----------------

{ open Parser } (* Get the token types *)

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
  | "/*" { comment lexbuf } (* Comments *)
  | '(' { LPAREN }  | ')' { RPAREN } (* Punctuation *)
  | '{' { LBRACE }  | '}' { RBRACE }
  | '[' { LBRACK }  | ']' { RBRACK }
  | ';' { SEMI }    | ',' { COMMA }
  | '+' { PLUS }    | '-' { MINUS }
  | '*' { TIMES }   | '/' { DIVIDE }
  | '%' { MOD }     | ':' { COLON }
  | '=' { ASSIGN }
  | '<' { LT }      | '>' { GT }
  | "==" { EQ }     | "!=" { NEQ }
  | "<=" { LEQ }    | ">=" { GEQ }
  | '!' { NOT }
  | '|' { OR }      | '&' { AND } (* Short circuits *)
  | "accept" { ACCEPT } | "reject" { REJECT } (*test functions*)
  | "@" { ASSERT }  | "unit" { UNIT } | '.' { ACCESS }
  | "else" { ELSE } | "if" { IF } (* Keywords *)
  | "while" { WHILE } | "for" { FOR }
  | "return" { RETURN } | "accept" { ACCEPT }
  | "struct" { STRUCT } | "reject" { REJECT }
  | "void" { VOID }
  | "bool" { BOOL } | "int" { INT } | "str" { STRING }
  | "equals" { EQUALS }
  | '"'('\\'_ |[^'"'])*'"' as str { STRING_LITERAL(str) }  (* Strings *)
  | ['0'-'9']+ as lxm { INT_LITERAL(int_of_string lxm) } (* Integers *)
  | "true" | "false" as boolean { BOOL_LITERAL(bool_of_string boolean) }
  | eof { EOF } (* End-of-file *)
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | _ as char { raise (Failure("illegal character " ^
      Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf } (* End-of-comment *)
```

```
  | _ { comment lexbuf } (* Eat everything else *)
semantic_checker.ml
-----------------

open Ast
open Sast
open Lexing
open Map

(* check return exists *)
(* have to reverse lists lololol *)

type function_table = {
      funcs : func_decl list
}

type symbol_table = {
      mutable parent : symbol_table option;
      mutable variables : (string * checked_var_decl * var_types) list;
      mutable functions : function_decl list;
      mutable structs : struct_decl list;
      mutable return_found : bool;
}

type translation_environment = {
      mutable scope : symbol_table; (* symbol table for vars *)
      mutable found_main: bool;
}

let the_print_function = {
      ftype = Sast.Void;
      fname = "print";
      checked_formals = [];
      checked_locals = [];
      checked_body = [];
      checked_units = []
}

let the_exit_function = {
      ftype = Sast.Void;
      fname = "exit";
      checked_formals = [];
      checked_locals = [];
      checked_body = [];
      checked_units = []
}

let find_struct (s : struct_decl list) stru =
```

```
      List.find(fun c -> c.sname = stru) s

let find_func (l : function_decl list) f =
      List.find(fun c -> c.fname = f) l

let rec check_id (scope : symbol_table) id =
      try
            (* let _ = print_string ("check_id called, legnth of
scope.variables is " ^ string_of_int (List.length scope.variables) ^ "\n") in
*)
            (* let _ = List.iter (fun (n, _, _) -> print_string ("try printing
in check_id: " ^ n ^ "\n")) scope.variables in *)
            let (_, decl, t) = List.find(fun (n, _, _) -> n = id )
scope.variables in
            decl, t
      with Not_found -> match scope.parent with
            Some(parent) -> check_id parent id
            | _ -> raise Not_found

 let rec check_expr (scope : symbol_table) (expr : Ast.expr) = match expr with
      (* let _ = print_string ("try printing at top of process_var_decl, length
of scope.variables is " ^ string_of_int (List.length scope.variables) ^ "\n")
in match expr with *)
      Noexpr -> Sast.Noexpr, Void
      | Id(str) ->
            (try
                  let (decl, t) = check_id scope str in Sast.Id(decl), t
            with Not_found -> raise (Failure ("Id named " ^ str ^ " not
found")))
      | Integer_literal(i) -> Sast.IntConst(i), Sast.Int
      | String_literal(str) -> Sast.StrConst(str), Sast.String
      | Boolean_literal(b) -> Sast.BoolConst(b), Sast.Boolean
      | Array_access(_, _) as a -> check_array_access scope a
      | Assign(_, _) as a -> check_assign scope a
      | Uniop(op, expr) as u -> check_uni_op scope u
      | Binop(_, _, _) as b -> check_op scope b
      | Call(_, _) as c -> check_call scope c
      | Access(_, _) as a -> check_access scope a
      | Struct_Member_Assign(_, _, _) as a -> check_struct_assignment scope a
      | Array_Member_Assign(_, _, _) as a -> check_array_assignment scope a

and check_array_assignment (scope : symbol_table) a = match a with
      Ast.Array_Member_Assign(arr, expr, expr2) ->
            (
                  try
                        let (original_decl, var_type) = check_id scope arr in
match var_type with
                              Sast.Array(decl, _) ->
```

```
                                        (
                                                let access_expr = check_expr scope
expr in
                                                let (_, t) = access_expr in
                                                if t <> Sast.Int then
                                                      raise (Failure "Array access
must be type int")
                                                else
                                                      (let assign_expr = check_expr
scope expr2 in
                                                      let (_, t2) = assign_expr in
                                                      if decl <> t2 then raise
(Failure "type assignment is wrong")
                                                      else
Sast.Array_Member_Assign(original_decl, access_expr, assign_expr), t2)
                                        )
                                  | _ -> raise (Failure (arr ^ " is not an
array."))
                  with Not_found -> raise (Failure ("Variable " ^ arr ^ " not
declared."))
            )
      | _ -> raise (Failure "Not an array assignment")

and check_struct_assignment (scope : symbol_table) a = match a with
      Ast.Struct_Member_Assign(stru, mem, expr) ->
            (
                  try
                        let (original_decl, var_type) = check_id scope stru in
match var_type with
                        | Sast.Struct(decl) ->
                              (
                                    try
                                          let v = List.find(
                                                fun (v, _) -> match v with
                                                Variable(_, s) -> s = mem
                                                | Variable_Initialization(_,
s, _) -> s = mem
                                                | Array_Initialization(_, s,
_) -> s = mem
                                                | Struct_Initialization(_, s,
_) -> s = mem
                                          ) decl.variable_decls in
                                          let expr = check_expr scope expr in
                                          let (_, t) = v in
                                          let (_, t2) = expr in
                                          if t <> t2 then raise (Failure "type
assignment is wrong")
```

```
                                          else Sast.Struct_Member_Assign(decl,
original_decl, v, expr), var_type
                                    with Not_found -> raise (Failure (mem ^ "
not found in struct " ^ stru))
                              )
                        | _ -> raise (Failure (stru ^ " is not a struct."))
                  with Not_found -> raise (Failure ("Variable " ^ stru ^ " not
declared."))
            )
      | _ -> raise (Failure "Not a struct assignment")

and check_op (scope : symbol_table) binop = match binop with
      Ast.Binop(xp1, op, xp2) ->
            let e1 = check_expr scope xp1 and e2 = check_expr scope xp2 in
            let (_, t1) = e1 and (_, t2) = e2 in
            let t = match op with
                  Add ->
                        if (t1 <> Int || t2 <> Int) then
                              if (t1 <> String || t2 <> String) then raise
(Failure "Incorrect types for +")
                              else String
                        else Int
                  | Sub -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for - ") else Sast.Int
                  | Mult -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for * ") else Sast.Int
                  | Div -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for / ") else Sast.Int
                  | Mod -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for % ") else Sast.Int
                  | Equal -> if (t1 <> t2) then raise (Failure "Incorrect types
for = ") else Sast.Boolean
                  | Neq -> if (t1 <> t2) then raise (Failure "Incorrect types
for != ") else Sast.Boolean
                  | Less -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for < ") else Sast.Boolean
                  | Leq -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for <= ") else Sast.Boolean
                  | Greater -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for > ") else Sast.Boolean
                  | Geq -> if (t1 <> Int || t2 <> Int) then raise (Failure
"Incorrect types for >= ") else Sast.Boolean
                  | Or -> if (t1 <> Boolean || t2 <> Boolean) then raise
(Failure "Incorrect types for | ") else Sast.Boolean
                  | And -> if (t1 <> Boolean || t2 <> Boolean) then raise
(Failure "Incorrect types for & ") else Sast.Boolean
                  | Not -> raise (Failure "! is a unary operator.")
            in Sast.Binop(e1, op, e2), t
```

```
      | _ -> raise (Failure "Not an op")

and check_array_access (scope : symbol_table) a = match a with
      Ast.Array_access(id, expr) ->
            let (decl, t) = check_id scope id in (match t with
            Sast.Array(t, _) ->
                  let e1 = check_expr scope expr in
                  let (_, t2) = e1 in
                  if t2 <> Int then raise (Failure "Array access must be
integer.") else
                  Sast.ArrayAccess(decl, e1), t
            | _ -> raise (Failure "this id is not an array"))
      | _ -> raise (Failure "Not an array access")

and check_assign (scope : symbol_table) a = match a with
      Ast.Assign(id, expr) ->
            let (decl, t) = check_id scope id in
            let e = check_expr scope expr in
            let (_, t2) = e in
            if t <> t2 then raise (Failure "Incorrect type assignment.") else
Sast.Assign(decl, e), t
      | _ -> raise (Failure "Not an assignment")

and check_call (scope : symbol_table) c = match c with
      Ast.Call(id, el) ->
            (
                  try
                        let f = find_func scope.functions id in
                        let exprs = List.fold_left2 (
                                    fun a b c ->
                                          let (_, t) = b in
                                          let expr = check_expr scope c in
                                          let (_, t2) = expr in
                                          if t <> t2
                                          then raise (Failure "wrong type")
                                          else expr :: a
                              ) [] f.checked_formals el in
                        Sast.Call(f, exprs), f.ftype
                  with Not_found ->
                        if id = "print" then match el with
                              | hd :: []-> let expr = check_expr scope hd in
                                    let (_, t) = expr in
                                    if (t = Sast.String || t = Sast.Int) then
Sast.Call(the_print_function, [expr]), Sast.Void else raise (Failure "Print
takes only type string or int")
                              | _ -> raise (Failure "Print only takes one
argument")
                        else if id = "exit" then match el with
```

```
                              | hd :: []-> let expr = check_expr scope hd in
                                    let (_, t) = expr in
                                    if t = String then
Sast.Call(the_exit_function, [expr]), Sast.Void else raise (Failure "Exit takes
only type string")
                              | _ -> raise (Failure "Exit only takes one
argument")
                        else if id = "main" then
                              raise (Failure "Cannot fall main function")
                        else raise (Failure ("Function not found with name " ^
id))
            )
      | _ -> raise (Failure "Not a call")

and check_access (scope : symbol_table) a = match a with
      Ast.Access(id, id2) ->
            (let (original_decl, t) = check_id scope id in match t with
                  Struct(decl) ->
                        (try
                              let var = List.find (
                                    fun (t, _) -> match t with
                                    Variable(_, n) -> n = id2
                                    | Variable_Initialization(_, n, _) -> n =
id2

                                    | Array_Initialization(_, n, _) -> n = id2
                                    | Struct_Initialization(_, n, _) -> n = id2
                              ) decl.variable_decls in
                              let (var, _) = var in
                              let t = match var with
                                    Variable(t, _) -> t
                                    | Variable_Initialization(t, _, _) -> t
                                    | Array_Initialization(t, _, _) -> t
                                    | Struct_Initialization(t, _, _) -> t
                              in Sast.Access(decl, original_decl, var), t
                        with Not_found -> raise (
                              Failure (id ^ " is type struct " ^ decl.sname ^ "
which does not have a member named " ^ id2)
                        ))
                  | _ -> raise (Failure (id ^ " is not a struct."))
            )
      | _ -> raise (Failure "Not an access")

and check_uni_op (scope : symbol_table) uniop = match uniop with
      Ast.Uniop(op, expr) -> (
            match op with
                  Not ->
                        let e = check_expr scope expr in
                        let (_, t) = e in
```

```
                         if (t <> Boolean) then raise (Failure "Incorrect type
for ! ") else Sast.Uniop(op, e), Boolean
                   | _ -> raise (Failure "Not a unary operator")
             )
      | _ -> raise (Failure "Not a uniop")
(*
let process_func_formals (env : translation_environment) f =
      let scope' = { env.scope with parent = Some(env.scope); variables = [] }
in
      let scope' = List.iter (fun var -> scope.variables:: head) *)

let rec check_stmt (scope : symbol_table) (stmt : Ast.stmt) = match stmt with
      Block(sl) -> Sast.Block(List.fold_left ( fun a s -> (check_stmt scope s)
:: a) [] sl)
      | Expr(e) -> Sast.Expr(check_expr scope e)
      | Return(e) -> Sast.Return(check_expr scope e)
      | If(expr, stmt1, stmt2) ->
            let new_expr = check_expr scope expr in
            let (_, t) = new_expr in
            if t <> Sast.Boolean then
                  raise (Failure "If statement must have a boolean expression")
            else
                  let new_stmt1 = check_stmt scope stmt1 in
                  let new_stmt2 = check_stmt scope stmt2 in
                  Sast.If(new_expr, new_stmt1, new_stmt2)
      | For(expr1, expr2, expr3, stmt) ->
            let expr = check_expr scope expr1 in
            let expr2 = check_expr scope expr2 in
            let (_, t) = expr2 in
            if t <> Sast.Boolean then
                  raise (Failure "If statement must have a boolean expression")
            else
                  let expr3 = check_expr scope expr3 in
                  let stmt = check_stmt scope stmt in
                  Sast.For(expr, expr2, expr3, stmt)
      | While(expr, stmt) ->
            let expr = check_expr scope expr in
            let (_, t) = expr in
            if t <> Sast.Boolean then
                  raise (Failure "If statement must have a boolean expression")
            else
                  let stmt = check_stmt scope stmt in
                  Sast.While(expr, stmt)

let rec check_var_type (scope : symbol_table) (v : Ast.var_types) = match v
with
      Ast.Void -> Sast.Void
      | Ast.Int -> Sast.Int
```

```
	| Ast.String -> Sast.String
	| Ast.Boolean -> Sast.Boolean
	| Ast.Struct(id) ->
		(try
			let s = find_struct scope.structs id in
			Sast.Struct(s)
		with Not_found -> raise (Failure ("Struct " ^ id ^ " not found.")))
	| Ast.Array(v, expr) ->
		let v = check_var_type scope v in
		let expr = check_expr scope expr in
		let (_, t) = expr in
		if t <> Int then raise (Failure "Array size must have integer.")
		else Sast.Array(v, expr)

let process_var_decl (scope : symbol_table) (v : Ast.var_decl) =
	(* let _ = print_string ("try printing at top of process_var_decl, length
of scope.variables is " ^ string_of_int (List.length scope.variables) ^ "\n")
in *)
	let triple = match v with
		Variable(t, name) ->
			let t = check_var_type scope t in
			(name, Sast.Variable(t, name), t)
		| Variable_Initialization(t, name, expr) ->
			let t = check_var_type scope t in
			let expr = check_expr scope expr in
			let (_, t2 ) = expr in
			if t <> t2 then raise (Failure "wrong type for variable
initialization") else (name, Sast.Variable_Initialization(t, name, expr), t)
		| Array_Initialization(t, name, el) -> (match t with
			Ast.Array(v, expr) ->
				let t = check_var_type scope v in
				let el = List.fold_left (
					fun a elem ->
					let expr = check_expr scope elem in
					let (_, t2 ) = expr in
					if t <> t2 then raise (Failure "wrong type for
array initilization") else expr :: a
				) [] el in (name,
Sast.Array_Initialization(Sast.Array(t, (Noexpr, Void)), name, List.rev el),
Sast.Array(t, (Noexpr, Void)))
			| _ -> raise (Failure "Not an arrary!"))
		| Struct_Initialization(t, name, el) ->
			let t = check_var_type scope t in match t with
				Struct(decl) ->
					(* DO NOT THROW AWAY RESPONSE`*)
					let el = List.fold_left2 (
						fun a b c -> let t =
						let (b, _ ) = b in match b with
```

```
                                    Variable(t, _) -> t
                                    | Variable_Initialization(t, _, _) -> t
                                    | Array_Initialization(t, _, _) -> t
                                    | Struct_Initialization(t, _, _) -> t in
                                    let e = check_expr scope c in
                                    let (_, t2) = e in
                                    if t <> t2 then raise (Failure "types are
not the same in struct initialization") else e :: a
                              ) [] decl.variable_decls el in (name,
Sast.Struct_Initialization(t, name, el), t)
                        | _ -> raise (Failure "Not a struct") in (*test?*)
      let (_, decl, t) = triple in
      if t = Void then
            raise (Failure "Variable cannot be type void.")
      else
            scope.variables <- triple :: scope.variables; (* List.iter (fun (n,
_, _) -> print_string ("try printing in process_var_decl:" ^ n ^ "\n"))
scope.variables; *) (* Update the scope *)
            (decl, t)

let rec check_func_stmt (scope : symbol_table) (stml : Sast.stmt list) (ftype :
Sast.var_types) =
      List.iter (
            fun s -> match s with
            Sast.Block (sl) ->
                  check_func_stmt scope sl ftype
            | Sast.Return(e) ->
                  let (_, t) = e in
                  if t <> ftype then raise (Failure "func return type is
incorrect") else ()
            | Sast.If(_, s1, s2) ->
                  check_func_stmt scope [s1] ftype; check_func_stmt scope [s2]
ftype
            | Sast.For(_, _, _, s) ->
                  check_func_stmt scope [s] ftype
            | Sast.While(_, s) ->
                  check_func_stmt scope [s] ftype
            | _ -> ()
      ) stml

let process_func_stmt (scope : symbol_table) (stml : Ast.stmt list) (ftype :
Sast.var_types) =
      List.fold_left (
            fun a s -> let stmt = check_stmt scope s in
            match stmt with
            Sast.Block (sl) ->
                  check_func_stmt scope sl ftype; stmt :: a
            | Sast.Return(e) ->
```

```
                let (_, t) = e in
                if t <> ftype then raise (Failure "while processing func
statement, return type incorrect") else
                scope.return_found <- true; stmt :: a
            | Sast.If(_, s1, s2) ->
                check_func_stmt scope [s1] ftype; check_func_stmt scope [s2]
ftype; stmt :: a
            | Sast.For(_, _, _, s) ->
                check_func_stmt scope [s] ftype; stmt :: a
            | Sast.While(_, s) ->
                check_func_stmt scope [s] ftype; stmt :: a
            | _ -> stmt :: a
      ) [] stml

let process_func_units (scope : symbol_table) (u : Ast.unit_decl) (formals :
Sast.variable_decl list) (ftype : Sast.var_types) = match u with
      Local_udecl (el, e, b) ->
      let exprs = List.fold_left2 (
                  fun a b c ->
                        let (_, t) = b in
                        let expr = check_expr scope c in
                        let (_, t2) = expr in
                        if t <> t2
(*stopped tests here going *)
                        then raise (Failure "while processing func units,
wrong type")
                        else expr :: a
                  ) [] formals el in
            let expr = check_expr scope e in
            let (_, t ) = expr in
            if t <> ftype then raise (Failure "while processing func units,
incorrect return type") else
            Sast.Local_udecl(exprs, expr, b)
      | Outer_udecl (f, el, e, b) ->
      (try
            let f = find_func scope.functions f in
            let exprs = List.fold_left2 (
                  fun a b c ->
                        let (_, t) = b in
                        let expr = check_expr scope c in
                        let (_, t2) = expr in
                        if t <> t2
                        then raise (Failure "wrong type")
                        else expr :: a
                  ) [] f.checked_formals el in
            let expr = check_expr scope e in
            let (_, t ) = expr in
            if t <> f.ftype then raise (Failure "Incorrect return type") else
```

```
            Sast.Outer_udecl(f, exprs, expr, b)
      with Not_found -> raise (Failure ("Function not found with name " ^ f)))

let check_func_decl (env : translation_environment) (f : Ast.func_decl) =
      let scope' = { env.scope with parent = Some(env.scope); variables = [];
return_found = false } in
      let t = check_var_type env.scope f.ftype in
      let formals = List.fold_left (
            fun a f -> match f with
            Ast.Param(t, n) ->
                  let t = check_var_type scope' t in
                  scope'.variables <- (n, Sast.Variable(t, n), t) ::
scope'.variables; (Sast.Variable(t, n), t) :: a
      ) [] f.formals in
      let locals = List.fold_left ( fun a l -> process_var_decl scope' l :: a )
[] f.locals in
      let statements = process_func_stmt scope' f.body t in
      let units = List.fold_left ( fun a u -> process_func_units scope' u
formals t :: a) [] f.units in
      if scope'.return_found then
            let f = { ftype = t; fname = f.fname; checked_formals = formals;
checked_locals = locals; checked_body = statements; checked_units = units } in
            env.scope.functions <- f :: env.scope.functions; (* throw away
scope of function *) f
      else (if f.ftype = Void then
            let f = { ftype = t; fname = f.fname; checked_formals = formals;
checked_locals = locals; checked_body = statements; checked_units = units } in
                  env.scope.functions <- f :: env.scope.functions; (* throw
away scope of function *) f
      else
            raise (Failure ("No return for function " ^ f.fname ^ " when return
expected.")))

let process_func_decl (env : translation_environment) (f : Ast.func_decl) =
      try
            let _ = find_func env.scope.functions f.fname in
                  raise (Failure ("Function already declared with name " ^
f.fname))
      with Not_found ->
            if f.fname = "print" then raise (Failure "A function cannot be
named 'print'")
            else
                  if f.fname = "main" then
                        (
                              if f.ftype <> Void || (List.length f.formals) <>
0 then
                              raise (Failure "main function must be type void
with no parameters")
```

```
                                else
                                      let func = check_func_decl env f in
                                      env.found_main <- true; func
                                )
                    else
                          check_func_decl env f

let rec check_struct_stml (stml : Sast.stmt list) =
      List.iter (
            fun s -> match s with
            Sast.Block (sl) ->
                  check_struct_stml sl
            | Sast.Return(_) -> raise (Failure "No returns are allowed in
asserts")
            | Sast.If(_, s1, s2) ->
                  check_struct_stml [s1]; check_struct_stml [s2]
            | Sast.For(_, _, _, s) ->
                  check_struct_stml [s]
            | Sast.While(_, s) ->
                  check_struct_stml [s]
            | _ -> ()
      ) stml

let process_assert (scope: symbol_table) a =
      let (expr, stml) = a in
      let expr  = check_expr scope expr in
      let (_, t) = expr in
      if t <> Sast.Boolean then (raise (Failure "assert expr must be boolean"))
else
      let stml = List.fold_left ( fun a s -> check_stmt scope s :: a) [] stml
in
      let _ = check_struct_stml stml in (expr, stml)

(* let check_struct (scope : symbol_table) s =
      let scope' = { scope with parent = Some(scope); variables = [] } in
      let vars = List.fold_left ( fun a s -> process_var_decl scope' :: a) []
s.variable_decls in
      (* should we keep result of process_var_decl? *)
      List.iter process_assert scope' s.asserts *)

let process_struct_decl (env : translation_environment) (s : Ast.struct_decl) =
      try
            let _ = find_struct env.scope.structs s.sname in
                  raise (Failure ("struct already declared with name " ^
s.sname))
      with Not_found ->
            let scope' = { env.scope with parent = Some(env.scope); variables =
[] } in
```

```
            let vars = List.fold_left ( fun a v -> process_var_decl scope' v ::
a ) [] s.variable_decls in
            let asserts = List.fold_left ( fun a asrt -> process_assert scope'
asrt :: a ) [] s.asserts in
            let s = { sname = s.sname; variable_decls = vars; asserts =
asserts; } in
            env.scope.structs <- s :: env.scope.structs; s

let process_global_decl (env : translation_environment) (g : Ast.var_decl) =
      try
            let name = match g with
                  Variable(_, id) -> id
                  | Variable_Initialization(_, id, _) -> id
                  | Array_Initialization(_, id, _) -> id
                  | Struct_Initialization(_, id, _) -> id in
            let _ = check_id env.scope name in
            raise (Failure ("Variable already declared with name " ^ name))
      with Not_found ->
            (* let _ = print_string ("p_global_decl called, this id not found,
legnth of env.scope.variables is " ^ string_of_int (List.length
env.scope.variables) ^ "\n") in *)
            process_var_decl env.scope g

let process_outer_unit_decl (env : translation_environment) (u : Ast.unit_decl)
= match u with
      Local_udecl (el, _, _) ->  raise (Failure "Can not define unit of this
type in global scope ")
      | Outer_udecl (f, el, e, b) ->
      (try
            let f = find_func env.scope.functions f in
            let exprs = List.fold_left2 (
                        fun a b c ->
                              let (_, t) = b in
                              let expr = check_expr env.scope c in
                              let (_, t2) = expr in
                              if t <> t2
                              then raise (Failure "wrong type while processing
outer unit declaration")
                              else expr :: a
                  ) [] f.checked_formals el in
            let expr = check_expr env.scope e in
            let (_, t ) = expr in
            if t <> f.ftype then raise (Failure "Incorrect return type in outer
unit test") else
            Sast.Outer_udecl(f, exprs, expr, b)
      with Not_found -> raise (Failure ("Function not found with name " ^ f)))

let check_program (p : Ast.program) =
```

```
      (* let _ = print_string ("check_program called \n") in *)
      let s = { parent = None; variables = []; functions = []; structs = [];
return_found = false } in
      let env = { scope = s; found_main = false } in
      let (structs, vars, funcs, units) = p in
      let structs =
            List.fold_left (
                  fun a s -> process_struct_decl env s :: a
            ) [] structs in
      let globals =
            List.fold_left (
                  fun a g -> process_global_decl env g :: a
            ) [] (List.rev vars) in
      let funcs =
            List.fold_left (
                  fun a f -> process_func_decl env f :: a
            ) [] (List.rev funcs) in
      let units =
            List.fold_left (
                  fun a u -> process_outer_unit_decl env u :: a
            ) [] units in
(*    try *)
      (* let _ = print_string ("length of env.scope.functions is " ^
string_of_int (List.length env.scope.functions) ^ "\n") in *)
(*     let rec findMain = function
      [] -> false
      | hd::tl ->
            if hd.fname = "main" then
                  (if (hd.ftype <> Void || (List.length hd.checked_formals) <>
0) then (raise (Failure "main function must be type void with no arguments"))
else true)
            else findMain tl
    in let foundMain  = findMain env.scope.functions in *)
    (if env.found_main then structs, globals, funcs, units else (raise (Failure
"No main function defined.")))
            (* let _ = List.iter( fun f -> if f.fname = "main" then
print_string "Found main" else(*  print_string ("did not find main, found " ^
f.fname ^ "\n")) env.scope.functions in  *)
            let _ = List.find( fun f -> f.fname = "main" ) env.scope.functions
in
            structs, globals, funcs, units
      with Not_found -> raise (Failure "No main function defined.") *)

let print_position outx lexbuf =
  let pos = lexbuf.lex_curr_p in
  Printf.fprintf outx "%s:%d:%d" pos.pos_fname
    pos.pos_lnum (pos.pos_cnum - pos.pos_bol + 1)
```

```
(* let _ =
      let lexbuf = Lexing.from_channel stdin in
      let program =
            try Parser.program Scanner.token lexbuf
            with _ -> Printf.fprintf stderr "%a: syntax error\n" print_position
lexbuf; exit (-1) in
      check_program program *)
```

TESTS

```
arr_access.k
-----------------

int main(){
      int[1] a;
      a[0] = 1;
      return 0;
}

array_decl.k
-----------------

void main(){
      int[1] arr;
}

array_expr_decl.k
-----------------

void main(){
      int x = 1;
      int[x+1] arr;
}

array_init.k
```

```
int main () {
	int[] a = {1, 2, 3};
}
```

array_list_decl.k

```
void main(){
	int[] array= {1,2,3,4,5,6};
}
```

array_size_not_int.k

```
void main(){
	int["1"] a;
}
```

assert.k

```
struct potato {
int size;
int potat;
@(size > 1) {}
}
```

bad_params_to_fn.k

```
int foo(int x, int y){
	return x;
}

int main(){
	int a;
	int b;

	a = 5;
	b = foo(a);
```

```
     return 0;
}

call_function_w_args.k
-----------------

int foo(int a, int b){
     return 0;
}
int main(){
     foo(2,3);
}

call_2_int.k
-----------------

int foo(int x, int y){
     return x+y;
}

void main(){
     foo(3,5);
}

call_function.k
-----------------

int foo(){
     return 0;
}
void main(){
     int a;
     a = 3;
     foo();
}


codegen_arr_access.k
-----------------

void main(){
     int[1] a;
     a[0] = 2;
```

```
}
```

decl_unit_outsideOfMethod.k

```
int foo(int i){
	int a;
	a = i;
	return a;
}

void main(){
	foo();
	return 0;
}

unit:foo():equals(0):accept;
```

fn_reclare_and_use.k

```
int foo(){
	return 0;
}

int foo(){
	return 1;
}

int main(){
	int a = foo();
	return a;
}
```

comment_reject.k

```
void main(){
	int a;
	/* comment, lol */
	int b;
}
```

```
function_w_args_reject.k
-----------------

void main(){
	int foo(int a, int b){
		return 0;
	}
}

exit_test.k
-----------------

void main(){
	int a;
	exit("exited.");
}

function_w_arg_reject.k
-----------------

void main(){
	int foo(int a){
		return 0;
	}
}

garbage_reject.k
-----------------

sdlkflkajsdflkjasdlfkjasd

int_minus_str_rejectsem.k
-----------------

int main(){
	int a = 1;
	str b = "1";
	int c = a - b;
	return 0;
}

inline_assign.k
-----------------
```

```
void main(){
	int a = 1;
}
```

int_mod_str_reject_sem_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	int c = a % b;
	return 0;
}
```

int_divided_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	int c = a / b;
	return 0;
}
```

multiple_funcs.k

```
int foo(){return 0;}
void main(){}
```

printing_test.k

```
int main() {
	print("hello, world!");
}
```

semantic_accessing_non_array_w_brack_rejectsem.k

```
int main(){
	int a = 1;
	a[1.2] = 2;
	return 0;
}
```

semantic_bad_fn_in_top_outer_unit_rejectsem.k

```
void main(){

}

unit:foo(0,0):equals(0):accept;
```

semantic_arr_wrong_type_rejectsem.k

```
void main(){
	int[] a = {"1"};
}
```

semantic_arr_access_non_int_rejectsem.k

```
int main(){
	int [1] a;
	a[1.2] = 2;
	return 0;
}
```

semantic_array_size_not_int_rejectsem.k

```
void main(){
	int["1"] a;
}
```

semantic_bad_func_return_type_rejectsem.k

```
int foo(){
	return "1";
```

```
}

int main(){
     foo();
}
```

semantic_bad_func_ret_type_inner_rejectsem.k

```
int foo(){
     if(1) {return "1";}
}

int main(){
     foo();
}
```

semantic_bad_params_to_fn_reject_rejectsem.k

```
int foo(int x, int y){
     return x;
}

void main(){
     int a;
     int b;

     a = 5;
     b = foo(a);
}
```

semantic_call_id_not_found_rejectsem.k

```
void main(){
     int a;
     b = 10;
 }
```

semantic_call_mismatch_argnum_rejectsem.k

```
int foo(int x, int y){
	return x+y;
}

void main(){
	foo(3);
}
```

semantic_call_mismatch_type_rejectsem.k

```
int foo(int x, int y){
	return x+y;
}

void main(){
	foo(Hello, World);
}
```

semantic_decl_mismatchtype_rejectsem.k

```
void main(){
	int hi;
	hi = "Hello";
}
```

semantic_fill_one_of_two_members_struct_rejectsem.k

```
struct foo {
	int x;
	int y;
}

void main(){
	struct foo f = {10};
}
```

semantic_func_declared_after_main_rejectsem.k

```
void main(){
```

```
	foo();
}

int foo(){
	return 4;
}
```

semantic_inner_unit_decl_top_rejectsem.k

```
int foo(int a, int b){
	return 0;
}

void main(){

}

unit(5,6):equals(0):accept;
```

semantic_fn_named_print_rejectsem.k

```
void print(){

}

void main(){

}
```

semantic_int_and_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a & b)
		return 1;
	return 0;
}
```

semantic_int_geq_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a >= b)
		return 1;
	return 0;
}
```

semantic_int_gt_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a > b)
		return 1;
	return 0;
}
```

semantic_int_equals_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if( a = b)
		return 1;
	return 0;
	}
}
```

semantic_int_dollarSign_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a $ b)
		return 1;
	return 0;
}
```

semantic_int_leq_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a <= b)
		return 1;
	return 0;
}
```

semantic_int_neq_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a != b)
		return 1;
	return 0;
}
```

semantic_int_lt_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a < b)
		return 1;
	return 0;
}
```

semantic_int_not_str_rejectsem.k

```
int main(){
	int a = 1;
	str b = "1";
	if (a ! b)
		return 1;
	return 0;
```

```
}

semantic_int_or_str_rejectsem.k
-----------------

int main(){
     int a = 1;
     str b = "1";
     if (a | b)
          return 1;
     return 0;
}

semantic_mem_name_two_structs.k
-----------------

struct a {
     int mem;
}

struct b {
     int mem;
}

void main(){
     struct a x;
     struct b y;
}

semantic_main_with_param_rejectsem.k
-----------------

void main(int a){

}

semantic_main_with_return_rejectsem.k
-----------------

void main(){
     return 1;
}
```

semantic_mism_unit_ret_type_rejectsem.k

```
int foo(int a){
     return 0;
}

void main(){

}

unit:foo(0):equals("1"):accept;
```

semantic_mism_unit_ret_type2_rejectsem.k

```
int foo(int a){
     return 0;
}

void main(){

}

unit:foo("1"):equals(0):accept;
```

semantic_mismatch_stringint_rejectsem.k

```
str foo(){
     return 1;
}
void main(){
     foo();
}
```

semantic_no_main_rejectsem.k

```
void min(){
```

```
}

semantic_no_ret_when_expected_rejectsem.k
-----------------

int foo(){

}

void main(){

}

semantic_nonunique_global_var_rejectsem.k
-----------------

int a;
int b;
int a;

void main(){

}

semantic_non_bool_in_assert_rejectsem.k
-----------------

struct foo {
     int a;
     @("1"){}
}

void main(){

}

semantic_nonunique_struct_name_rejectsem.k
-----------------

struct foo{
     int a;
}
```

```
struct foo{
	int b;
}

void main(){

}
```

semantic_nonvoid_main_decl_rejectsem.k

```
int main(){

}
```

semantic_reject_undeclared_fn_use_rejectsem.k

```
void main(){
	int a;
	a = foo();
}
```

semantic_return_in_assert_rejectsem.k

```
struct foo {
	int a;
	@(true){return 1;}
}

void main(){

}
```

semantic_single_unittest_rejectsem.k

```
void main(){
	unit(0):equals(15):accept;
}
```

semantic_same_varname_dif_scope.k

```
void foo(){
	int a;
	a = 10;
}

void main (){
	int a;
	a = 6;
	foo();
}
```

semantic_return_in_block_rejectsem.k

```
int foo(){
	if(true) {return 1;}
}

void main(){
	foo();
}
```

semantic_return_string_rejectsem.k

```
str foo(){
	return "Hello World!";
}
str main(){
	return foo();
}
```

semantic_str_plus_int_rejectsem.k

```
void main(){
	int a = 1;
	str b = "1";
	int c = a + b;
}
```

semantic_test_if_stmt_bool_type_rejectsem.k

```
void main(){
	if("true"){
		exit("bad.");
	}
}
```

semantic_undeclared_fn_use_rejectsem.k

```
void main(){
	int a;
	a = foo();
}
```

semantic_struct_id_not_found_rejectsem.k

```
struct s {
	int a;
}
void main(){
	struct z j;
}
```

semantic_type_mm_struct_init_rejectsem.k

```
struct foo {
	int a;
}
void main(){
	struct foo f = {"1"};
}
```

semantic_unit_outer.k

```
int foo(int a){
	return a;
```

```
}

void main(){
      int hi = 1;
      unit:foo(hi):equals(1):accept;
      unit:foo(hi):equals(0):reject;
}

semantic_using_global_in_fn.k
-----------------

int a;

void foo(){
      a = 10;
}

void main(){
      foo();
}

semantic_undeclared_var_rejectsem.k
-----------------

int main(){
      a = a + 1;
      return 0;
}

semantic_void_var_rejectsem.k
-----------------

void main(){
      void a;
}

struct_member_access.k
-----------------

struct test{
           int testInt;
}
void main(){
```

```
	struct test t;
	t.testInt = 20;
}

struct_init.k
-----------------

int main() {
struct potato potate  = { size };
}

semantic_wrong_type_init_rejectsem.k
-----------------

void main(){
	int i = "1";
}

str_initialization.k
-----------------

void main(){
	str x = "Hello World";
}

unit_multiple_args.k
-----------------

int hello(int a, int b){
	return a+b;
	unit(20,5):equals(25):accept;
}

void main(){
	hello(1,2);
}

test_suite.sh
-----------------

#/bin/bash
echo "-------------------------------------------"
echo "-------------------------------------------"
```

```
echo "----------BEGIN PRETTY PRINT-------------"
echo "-----------------------------------------"
echo "-----------------------------------------"

cd ../
make clean
if make pretty
     then
          cd test/
          for filename in *.k; do
               ../pretty < "$filename" > "output_$filename" 2>&1
               python test_logic.py pretty "$filename"
"output_$filename"
          done
else
     exit
fi
cd ../
if make
     then
          cd test/
          echo "-----------------------------------------"
          echo "-----------------------------------------"
          echo "-----------BEGIN SEMANTIC CHK------------"
          echo "-----------------------------------------"
          echo "-----------------------------------------"
          rm -f test/output_semantic*.k
          for filename in semantic*.k; do
               ../code_gen < "$filename" > "output_$filename" 2>&1
               python test_logic.py semantic "$filename"
"output_$filename"

          done
          echo "-----------------------------------------"
          echo "-----------------------------------------"
          echo "-----------------END---------------------"
          echo "-----------------------------------------"
          echo "-----------------------------------------"

          echo "-----------------------------------------"
          echo "-----------------------------------------"
          echo "---------BEGIN Java Compilation----------"
          echo "-----------------------------------------"
```

```
          echo "-----------------------------------------"
          for filename in *.k; do
               ../../code_gen < "$filename" > "Program.java" 2>&1
               javac "Program.java"
               java "Program" > "output_java_$filename.txt" 2>&1
          done
          echo "-----------------------------------------"
          echo "-----------------------------------------"
          echo "-----------------END---------------------"
          echo "-----------------------------------------"
          echo "-----------------------------------------"

else
     exit
fi
```

var_decl_after_assert_reject.k

```
struct potato {
int size;
int potat;
@(size > 1) {}
int j;
}

void main(){
}
```

test_logic.py

```
#A simple test script to test for equality of
#the output after parsed by the trees and printed by the pretty
printer
#This will be expanded to accomodate more tests in the future

import sys
mode = sys.argv[1]
inputFileName = sys.argv[2]
outputFileName = sys.argv[3]
semanticReject = False
semantic = False
```

```
syntax = True
syntaxReject = False
if (inputFileName[:8] == "semantic"):
     semantic = True
     syntax = False
     if (inputFileName[-11:] == "rejectsem.k"):
          semanticReject = True
else:
     print inputFileName[-8:] == "reject.k"
     if (inputFileName[-8:] == "reject.k"):
          syntaxReject = True

shouldReject = False

#convert the string by taking out spaces, newlines, and tabs
def convert_str(thestring):
     testStr = ''.join(thestring.split())
     return testStr

#read the files
file1 = open(inputFileName, 'r')
inputFileStr = file1.read()
file2 = open(outputFileName, 'r')
outputFileStr = file2.read()

#convert to string
inString = str(inputFileStr)
outString = str(outputFileStr)

inputTestStr = convert_str(inString)
outputTestStr = convert_str(outString)

if (syntax == True and mode == "pretty"):
     print inputFileName
     print outputFileName
     if (outputTestStr == inputTestStr and syntaxReject == False):
          print "Syntax test ACCEPTED\n"
     elif (outputTestStr != inputTestStr and syntaxReject == True):
          print "Syntax test ACCEPTED\n"
     else:
          print "Syntax test REJECTED"
          print "The input file:"
          print inputFileStr
```

```
            print "The output file:"
            print outputFileStr
            print "\n"

elif (semantic == True and mode == "semantic" ):
      print inputFileName
      print outputFileName

      #print outputTestStr
      try:
            outputTestStr.index("Fatalerror:")
            if (semanticReject == True):
                  print "Semantic test ACCEPTED\n"
            elif (semanticReject == False):
                  print "Semantic Test REJECTED"
                  print "The input file:"
                  print inputFileStr
                  print "The error:"
                  print outputFileStr
                  print "\n"
      except ValueError:
            if (semanticReject == False):
                  print "Semantic test ACCEPTED\n"
            elif (semanticReject == True):
                  print "Semantic Test REJECTED"
                  print "The input file:"
                  print inputFileStr
                  print "The error:"
                  print outputFileStr
                  print "\n"
```

int_times_str_rejectsem.k

```
int main(){
      int a = 1;
      str b = "1";
      int c = a * b;
      return 0;
}
```

var_eq_var.k

```
void main(){
	int a;
	int b;
	a=1;
	b=a;
}
```

var_reassign.k

```
void main(){
		int a;
		a=1;
		a=2;
}
```