Typescript-on-LLVM

Statically-typed scripting without the browser

Ratheet Pandya
UNI: rp2707

COMS 4115 H01 (CVN)

Table of Contents

1. Introduction

Typescript is gaining popularity as a statically-typed alternative to Javascript since it affords compile-time type checks and therefore reduces the likelihood of reading or writing data incorrectly and allows for better code optimization since the compiler can use the type system to eliminate a certain subset of runtime decisions that must be considered without types. Historically, Javascript is notorious for happily executing code that misuses data and provides virtually no pre-runtime checks for the developer, resulting in strange behavior and script errors that can only be caught through thorough runtime testing.

Typescript compiles down to Javascript that is executed in the browser. In contrast, the aim of the Typescript-on-LLVM project is to remove the browser and the JS engine from the equation and provide a means to use Typescript syntax to build programs that can run server-side, without a browser. This is accomplished by translating a subset of Typescript code to LLVM, a platform-independent intermediate representation that may execute on any modern operating system. As a result, Typescript programmers who would like to write Typescript-like code for building server-side applications may use Typescript-on-LLVM.

2. Language Tutorial

Typescript-on-LLVM uses a subset of the Typescript syntax to express statically-typed programs (see Language Manual section below for what is currently supported). Unlike Typescript, each program must have a `main()` function that is used as the entry-point for program execution. A simple example of a working program is the following:

```
function main() : number {
  printn(10);
  return 0;
}
```

This program prints out the number 10 to standard output. `printn()` is a built-in function for printing out a number. The `main()` function designates a return type of `number` and returns 0 to indicate success.

A slightly more interesting example is to allocate an array of numbers and print out one of them, as in the following:

```
function main() : number {
  let values: number[3] = [1, 2, 3];
  printn(values[1]);
  return 0;
}
```

Here, the output of the program will be the number 2, since that is the second element of the values array and is indexed using `values[1]`.

3. Language Manual

3.1 Lexical Conventions

3.1.1 Tokens

There are four kinds of tokens:

- Identifiers

- Keywords
- Operators
- Other separators

Whitespace (defined here) and comments (described below) are ignored except as they separate tokens:

- Blanks
- Horizontal and vertical tabs
- Newlines

Some whitespace is required in order to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

3.1.2 Comments

Comments begin with the characters `/*` and end with the characters `*/`. Comments do not nest.

3.1.3 Identifiers

An identifier (or name) is a sequence of letters and digits with the following properties:

- The first character must be a letter or an underscore (‘_’).
- Identifiers are case-sensitive.
- Identifiers may have any length.

Generally, identifiers are names that are bound to functions or variables in a scoped namespace, as explained below.

3.1.4 Reserved Keywords

The following identifiers are used as keywords, and may not be used otherwise:

```
boolean
else
false
function
if
let
number
return
```

```
true
void
```

3.1.5 Operators

Typescript-on-LLVM only supports binary operators. The table below summarizes the operators supported:

Operator	Use
+	Arithmetic addition
-	Arithmetic subtraction
*	Arithmetic multiplication
/	Arithmetic division
=	Assignment
!=	Not-equal boolean comparison
<	Less-than boolean comparison
<=	Less-than-or-equal-to boolean comparison
>	Greater-than boolean comparison
>=	Greater-than-or-equal-to boolean comparison
==	Equals boolean comparison

3.1.5.1 Associativity

The assignment operator, '=', is right-associative. The remaining operators are left associative, with the following rules:

- = takes precedence over !=
- The following operators are in precedence order, left-to-right: < > <= >=
- + takes precedence over -
- * takes precedence over /

3.1.6 Other Separators

The other separator tokens include:

- the semicolon '`;`' for sequencing of statements
- the comma, ',' for separating items in an array and a function parameter list
- curly braces, '`{`' and '`}`' for block-scoping
- parentheses, '`(`' and '`)`' for parameter lists in function definitions

3.2 Expressions

Expressions refer to program code that may be evaluated in a particular scope. There are two two kinds of top-level expressions: **primary** and **postfix**.

Primary expressions have the following form:

primary-expression:
  identifier
  boolean-value
  number-value

Postfix expressions group operators left-to-right:

postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `(` *argument-expression-list*$_{opt}$ `)`

argument-expression-list:
  assignment-expression-list
  argument-expression-list `,` *assignment-expression*

Assignment expressions are covered in Section 3.2.1.

As shown here, array indexing is done via a postfix expression in which the expression inside brackets resolves to the integer index of the array.

Similarly, function calls are postfix expressions that contain zero or more argument expressions within parentheses.

3.2.1 Operator Expressions

For the non-assignment operators described in Section 3.1.5, the following forms are used:

binary-expression:
expression + *expression*
expression - *expression*
expression * *expression*
expression / *expression*

conditional-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression == *expression*

Assignments take the following form:

assignment-expression:
identifier = *primary-expression* ;
identifier : *type-specifier* = *primary-expression* ;

Assignments are described in more detail in Section 3.3.5.

3.3 Statements

Statements are sequences of expressions, and have the following form:

statement:
expression-statement
compound-statement

expression-statement:
$expression_{opt}$;

compound-statement:
{ $statement\text{-}list_{opt}$ }

statement-list:

statement
statement-list statement

Conditional and loop statements are defined in Section 3.3.1 and Section 3.3.2, respectively.

3.3.1 Conditionals

Conditional statements allow for flow control based on the boolean value of a given conditional expression:

conditional-statement:
`if` `(` *conditional-expression* `)` *compound-statement*
`if` `(` *conditional-expression* `)` *compound-statement* `else` *compound-statement*

For example:

```
if (x < y) {
  printn(x);
}

if (x > y) {
  printn(y);
} else {
  printn(x);
}
```

3.3.2 Looping

Looping is supported using the `while` construct, and has the following form:

loop-statement:
while `(` *conditional-expression* `)` *compound-statement*

For example:

```
while (x < 100) {
  printAndIncrement(x);
}
```

3.4. Functions

Functions are optionally parameterized scoped blocks of code that are given identifiers in the global namespace.

3.4.1 Function Definition

Function definitions have the form `function D : T` *compound-statement*, where

- `D` has the form `D'(` *parameter-list* `)`
    - `D'` denotes the identifier for the function
- `T` denotes the return type of the function

The syntax of parameters is:

parameter-list:
parameter-list
parameter-list , parameter-declaration

parameter-declaration:
identifier `:` *type-specifier*

For example:

```
function add(x : number, y : number) : number {
    return x + y;
}
```

Here:

- `add` corresponds to D' above
- `x : number, y : number` is the *parameter-list*
- `{ return x + y; }` is the *compound-statement*

3.5 The `let` keyword

The `let` keyword is used to define a scope for an expression and assign that expression to a variable.

`let` assignment has the following form:

let-expression:
`let` *assignment-expression*

For example:

```
let x : number = 42;
```

3.6 Types

A type constrains the value of an expression to adhere to a particular space of values. There are four fundamental types in Typescript-on-LLVM, namely: `boolean`, `number`, `void`, and `Array`.

The type-specifiers are:

type-specifier:
`boolean`
`number`
`number[]`
`void`

These types are defined below.

3.6.1 `boolean`

A value has `boolean` type if it is either `true` or `false`.

3.6.2 `number`

All numbers in Typescript-on-LLVM are floating-point values consisting of decimal literals, represented as a sequence of digits, optionally followed by a single '.' and a trailing sequence of digits. Hexadecimal values are not supported.

Examples of numbers are `1`, `3.14`, etc.

3.6.3 `void`

The `void` type is used to indicate the absence of a value being returned from a function.

For example:

```
function sleep(): void {
    /* ... */
}
```

3.6.4 `Array`

An `Array` is an indexable collection of values of the same type.

It may be declared using `let`, as described in Section 3.3.5.

For example, to declare an array of numbers:

```
let values: number[3] = [1, 2, 3];
```

4. Project Plan

4.1 Planning

Since this project was completed by a single developer, planning consisted of setting goals from week-to-week without a formal process. Once the boilerplate translator was written, features were added in a test-first approach: add tests for a new feature, update the translator to include necessary lexing, parsing, semantic checking, and code generation, add edge test cases, and iterate.

4.2 Style Guide

The style guide used was loosely that of the OCaml guidelines - specifically, lines of code are kept to under 80 characters and variable naming uses underscores, e.g. `array_expr`.

4.3 Project Timeline

September 19	Submitted Project Proposal

October 15	Submitted Language Reference Manual, Scanner, and Parser
October 23	Initial commit for Github repo
November 14	Submitted “Hello World” demonstrating Scanner, Parser, Semantic Checks and Successful Code Generation
December 13 - 17	Added critical features, including arrays
December 17 - 19	Wrote Final Project Report and added more test cases

4.4 Roles and Responsibilities

As I am a CVN student working alone, I wrote all components and tests (largely adapted from the MicroC source code written by Professor Edwards).

4.5 Software Development Environment

The software development environment consisted of:

- Github Version Control (in a private repo)
- OCaml 4.05.0 with `llvm` and `llvm.analysis` packages
- Clang 6.0.0-1ubuntu2 (for testing/inspecting LLVM output via `-S -emit-llvm` flags)

4.6 Project Log

The following log was obtained using `git log` on December 18, 2018:

```
commit 5752f906c1459af6c0cdecd76e6160224ca88a5b
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 16:02:20 2018 -0500

    Add failure case for multiplication of booleans, and some
cleanup.

commit d06ee9728c90b8641746abfafc83708fdb47b52d
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 13:37:09 2018 -0500

    Add test cases for multiplication and division.

commit 10c7223408effefe9ffb299c7e2d7f23fdcc3d6e
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 13:12:41 2018 -0500
```

```
    Add failure test cases for functions.

commit 1725fd7fc39c6389949271584541 26fb817cab6f
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 11:23:47 2018 -0500

    Add return failure test cases.

commit fce4a8f00a88e3b745e36c2d8aca5c422f612994
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 10:38:27 2018 -0500

    Add failure test cases for redefining builtins.

commit c3a3492f1eab62d3910a79ee6b0b8d5064fc0866
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 10:11:01 2018 -0500

    Add failure test cases for while loops.

commit bba96b053c40884ca618ea56e2433f64e7881797
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 09:54:06 2018 -0500

    Add tests for while loops.

commit 7a11a2a8ec6744b8c6466802006703666807d499
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 09:24:35 2018 -0500

    Add test cases for array addition and GCD.

commit 1dd26c43f00024affb2e438f10f860cbb8faa68b
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 06:42:18 2018 -0500

    Add authorship to file comments.

commit 12b342bc92322a16821140b86e01bd36ce13453c
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Dec 18 06:32:57 2018 -0500

    Add a test case for array addition.

commit 7d88346716d6c6e56210a5a2726052cd22b3fd95
Author: rp2707 <rp2707@columbia.edu>
```

```
Date:   Mon Dec 17 14:58:06 2018 -0500

    Update subtraction tests.

commit 0bf4c8e1f697adb08c5cc5ddbe7af48bc50b89ba
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 12:54:43 2018 -0500

    Distinguish between if-else true and false cases.

commit 41357e9d531fa75bb5baa6d065e18001d11c2b35
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 11:53:24 2018 -0500

    Rename existing tests to use self-documenting names.

commit 6b2970d3c9815503fa122079975a409fcba2f0d2
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 11:22:40 2018 -0500

    Use self-documenting test names for if statements.

commit cf4344c599fbf2df97ecd6e6a5979499840dd359
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 09:09:01 2018 -0500

    Add false case for if statement.

commit e47f1c0c1ed536d070f8256cea45d6f28298f4a9
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 09:04:36 2018 -0500

    Update error message for empty arrays.

commit 1b7b728e38235a69ccc1d88f2ac4c39f02352cf3
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 09:03:57 2018 -0500

    Add failure case for negative size arrays.
commit e955457c38acb6b96ab6fc6a54889d975855aeef
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 08:29:59 2018 -0500

    Add failure case for mixed types in arrays.

commit 900ffe51b0045bca5a0a9bbfe79dd2cf39352744
Author: rp2707 <rp2707@columbia.edu>
```

```
Date:   Mon Dec 17 08:25:42 2018 -0500

    Add test case for mismatch in array type.

commit 7f82a2f54bd776b619e15c6ffda967fd21c9ea65
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 08:18:43 2018 -0500

    Add failure case for array size mismatch.

commit fb45d9f3f9a0fc182a57988ddc0e2525bcf8e037
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Dec 17 08:15:46 2018 -0500

    Add test case for changing an array.

commit fcd82dd8c3438836543f228b26191e4f0b720b9c
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 19:27:09 2018 -0500

    Add if/else test.

commit bfdf457c69724ff1cf1e2accdd18fdcd65040e62
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 19:02:49 2018 -0500

    Add test case for multiple subtractions.

commit a148cb1acfaefc583c34f4d2960a7fb1e869059d
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 18:48:09 2018 -0500

    Add test case for adding 3 numbers.

commit 133c8bde779676c9382b5fad5762724cc25c2070
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 18:44:43 2018 -0500

    Add failure case for missing main.
commit d438f6c2e47cc2497d8da30535172b27d3fbbcb5
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 18:32:39 2018 -0500

    Add failure case for number assignment to boolean.

commit 6dc6bcc7fd16b07bcf7bfc4619bee248d3763c08
Author: rp2707 <rp2707@columbia.edu>
```

```
Date:   Sun Dec 16 18:28:23 2018 -0500

    Add failure case for number assignment to boolean.

commit a95421e6cc31252509e08bda0900531e584052e3
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 18:20:38 2018 -0500

    Added empty array test.

commit 2e64d30d5d700e5fd2680d7106d6834f6dffe130
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 17:14:44 2018 -0500

    Add array index range checks.

commit 68ceb03a228c8f7af1cd50ee753cadc86fb68806
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 16:59:02 2018 -0500

    Array assignment and indexing works!

commit 7e858783cbf461929c7faada55ed2b41bd0cf465
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 16:32:58 2018 -0500

    Assignment of booleans works.

commit 17ed6d04974fb8217236956b55fc8edc2e66188f
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 16:00:31 2018 -0500

    Assignment working e2e.
commit c2919c110eeda37855ac433c06ad21b1943503c0
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 13:41:32 2018 -0500

    Add assignment expressions, part two.

commit 603f4aef56ac92d8a287f1d9686042c6d8ae24e2
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 12:00:07 2018 -0500

    Add assignment expressions, part one.

commit f4068e7c3eced8975d3b731982d990c7750ac83f
Author: rp2707 <rp2707@columbia.edu>
```

```
Date:   Sun Dec 16 08:28:47 2018 -0500

    Integer indexing.

commit 6ae1e977fc9c1f4e1ca065c5bc65b22d76f46b68
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Dec 16 07:57:34 2018 -0500

    Set array size exactly.

commit b91e33daae13b727f888bd8f6cf76cd7abdd4507
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Dec 15 18:25:12 2018 -0500

    Added array indexing.

commit 88dd7bc4eed0778c4138e4c4ef94af36afa75e12
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Dec 15 16:34:25 2018 -0500

    Added support for local vars.

commit 17d36c76fed115cab4bd58d46113ce5443416539
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Dec 15 16:02:28 2018 -0500

    Array declaration working.

commit 9c91f04ff9b08df948cb021c734955b834acdc6c
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Dec 15 11:15:13 2018 -0500

    All tests pass with special return case.
commit c855ff4cac8648714a84f90ab058a88b250f2d29
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Dec 15 09:14:01 2018 -0500

    Add support for adding numbers, and formal args.

commit b860ce8a37d4c856734da0738b9f94d5800d42c4
Author: rp2707 <rp2707@columbia.edu>
Date:   Thu Dec 13 05:56:22 2018 -0500

    Function call matches type signature.

commit 595e3eefc75b6705aa62746f4a5d56074921f421
Author: rp2707 <rp2707@columbia.edu>
```

```
Date:   Sun Nov 25 19:01:12 2018 -0500

    A new error - code gen for add op.

commit 3f755339bdaac6b5d3d23bb042993e60f51c17ad
Author: rp2707 <rp2707@columbia.edu>
Date:   Wed Nov 14 07:56:51 2018 -0500

    make (testall in particular) works completely.

commit 2aa8d2c6ad5d6a6e5ea9a246dc5bd79e49041953
Author: Ratheet Pandya <ratheet@google.com>
Date:   Tue Nov 13 16:10:57 2018 -0500

    return zero for all main calls

commit 631b731c34e454aa61048c8e1c596dacc62f7345
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Nov 13 08:26:29 2018 -0500

    More cleanup, getting ready for hello-world submission.

commit cbb6b17ea94173302ddfd03dcdfb7735340f67c0
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Nov 13 06:36:08 2018 -0500

    Some cleanup.

commit a897608ffe80636cbb9a34033d52b856e8b33369
Author: rp2707 <rp2707@columbia.edu>
Date:   Tue Nov 13 06:07:32 2018 -0500

    Built-in function calls can compile to LLVM.
commit 5a4dcdfc9649705f1e15dcecc596720eacf7119d
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Nov 11 16:05:46 2018 -0500

    Compilation of basic functions works.

commit 4e668ae3e94ebb333667dc5483f70be35c5000e5
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Nov 11 15:35:44 2018 -0500

    Added return expressions.

commit b428d9d7f2ac1254d23e5f82aebf2cd45057f069
Author: rp2707 <rp2707@columbia.edu>
```

```
Date:   Sun Nov 11 11:24:59 2018 -0500

    Very simple function with simplest possible body - [semantic]
parsing works.

commit df82ae308a0d80f2b391a6641a1a896ed22394fd
Author: rp2707 <rp2707@columbia.edu>
Date:   Sun Nov 11 11:03:11 2018 -0500

    Very simple function [semantic] parsing works.

commit 819939fa6e8472055e2f283d5577418d38e2a397
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 17:25:09 2018 -0500

    Working on fdecl.

commit 77197a1bfadc96e7f81388139e09c245561d4952
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 16:14:35 2018 -0500

    Compilation of global definitions works.

commit 94204fbdd0faf1c34bf1a0f3bd8c4b96767cf778
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 15:42:35 2018 -0500

    Codegen of global variable definition works.

commit 85871b6e9fc21a67c4b5778134fd97d0df9e6af3
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 14:53:18 2018 -0500

    Boilerplate codegen working for global declaration, but not
initialization.
commit 8e3cb29819ddc2a6761f1b7311cdb546e442da8c
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 12:38:41 2018 -0500

    Very simple semantic checking of global bindings.

commit 9ae7500fd93640e5e8306d9de3adabf5345ba18d
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 11:30:04 2018 -0500

    Literal string assignment can be scanned and parsed.
```

```
commit 64cb13b03d3ca600380466d427d8de88a8812415
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 11:17:17 2018 -0500

    Number assignment can be scanned and parsed.

commit 0e788a3a520b662aebfcfebe04dc3a869c7ba8c4
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 11:05:51 2018 -0500

    Boolean let expressions with assignment works.

commit c3f31505906d27d347f41c22b42d22712e902c02
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 10:39:38 2018 -0500

    Let syntax can be scanned, without assignment.

commit 9cc575c322093125b0b2037519a56b810c642d1a
Author: rp2707 <rp2707@columbia.edu>
Date:   Sat Nov 10 10:33:10 2018 -0500

    The simplest possible program can be scanned and parsed.

commit 6ed6696cbe27f7301683b61e6beaa774b3165c61
Author: rp2707 <rp2707@columbia.edu>
Date:   Fri Nov 9 21:27:59 2018 -0500

    replace lxm

commit f52012d374085ac0ba045a3c542ed27427744ca8
Author: rp2707 <rp2707@columbia.edu>
Date:   Fri Nov 9 21:19:39 2018 -0500

    Simple test case has a parse error.
commit c03cdf99a2d397a129d1d5af7516a6d2a402080f
Author: rp2707 <rp2707@columbia.edu>
Date:   Fri Nov 9 20:00:09 2018 -0500

    Assignment/let flattened into vdecl.

commit 5d9a113eb82d3bbb67069f5de41f585278f6a69f
Author: rp2707 <rp2707@columbia.edu>
Date:   Fri Nov 9 18:45:20 2018 -0500

    Save before cleanup.
```

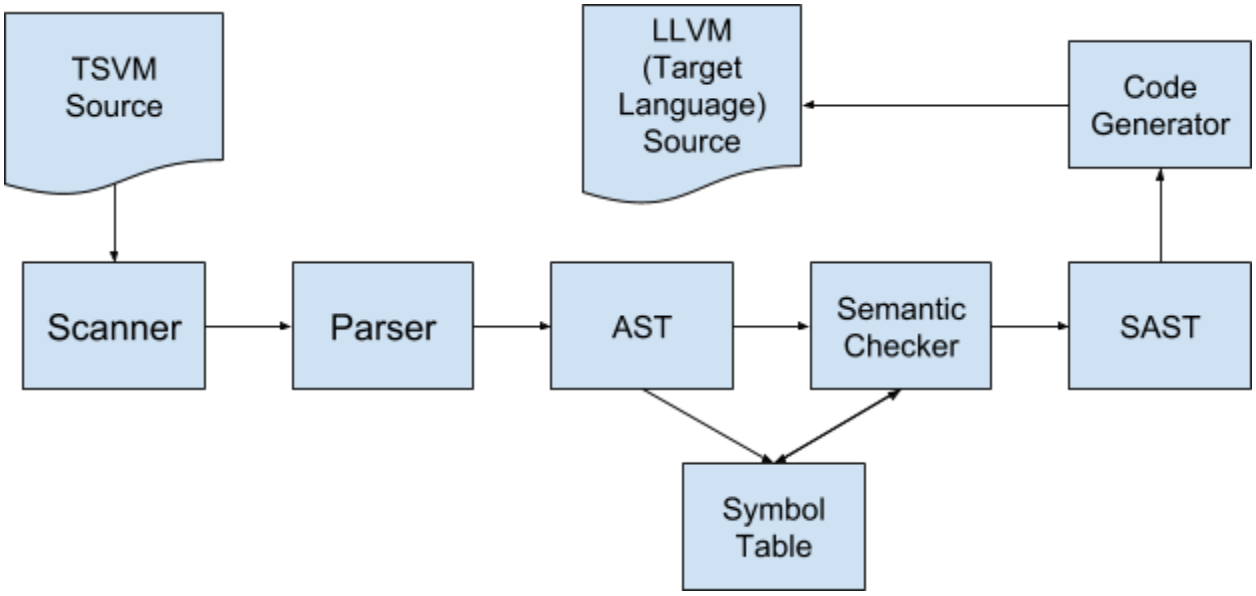
```
commit 797cd8559523b691716938c037df08c3e88c0be2
Author: rp2707 <rp2707@columbia.edu>
Date:   Fri Nov 9 14:10:26 2018 -0500

    Add SSH instructions.

commit f58b16c86255afca93bd476d0648e7865ae76429
Author: rp2707 <rp2707@columbia.edu>
Date:   Thu Nov 8 09:08:41 2018 -0500

    NumLit seems to be working.

commit 97dfeccc639b4747083e14f8f3c2d589504572ea
Author: rp2707 <rp2707@columbia.edu>
Date:   Thu Nov 8 08:52:12 2018 -0500

    Add stock codegen and comment out a bunch of code, in order to
simple expressions working first.

commit db846d7fba2f7198b56c9e3fa826a5bf3bfed1a4
Author: rp2707 <rp2707@columbia.edu>
Date:   Wed Nov 7 08:34:02 2018 -0500

    Add some instructions for building.

commit 437401d325bd6eb9b39063f855535142beacfbeb
Author: rp2707 <rp2707@columbia.edu>
Date:   Wed Nov 7 08:10:52 2018 -0500

    Add boilerplate semant.ml, just to get all files in place.

commit b776114c1b26dae09bb40aa8fd5c6388033e784e
Author: rp2707 <rp2707@columbia.edu>
Date:   Wed Nov 7 07:46:13 2018 -0500

    Add vanilla SAST from MicroC

commit 0a02088645c6c846f9da835baf262f1e5a3ab911
Author: rp2707 <rp2707@columbia.edu>
Date:   Mon Nov 5 08:29:41 2018 -0500

    Add skeleton AST and TSVM files.

commit 32e550318c2337757e6dabff6e70239ff7e126dd
Author: rp2707 <rp2707@users.noreply.github.com>
Date:   Tue Oct 23 19:13:05 2018 -0400
```

```
    Add initial scanner and parser.

commit 5e2853e5151e8508bc15c7661672c7a46c384009
Author: rp2707 <rp2707@users.noreply.github.com>
Date:   Tue Oct 23 19:00:24 2018 -0400

    Initial commit
```

5. Architectural Design

5.1 Block Diagram

The following block diagram shows the major components of the Typescript-on-LLVM translator:



The architecture works as follows:

- **Tyepscript-on-LLVM (TSVM) source**, written in a .ts file, is fed into the **Scanner**
- The **Scanner** tokenizes input source and feeds this into the **Parser**
- The **Parser** builds an **Abstract Syntax Tree (AST)**
- The **AST** represents the structure of the program and is fed into the **Semantic Checker**
- The **Semantic Checker** builds a **Semantically-annotated Abstract Syntax Tree (SAST)** using the **AST** and populates a **Symbol Table** for object lookups
- The **SAST** is used by the **Code Generator** to build **LLVM** instructions for execution of the program

More detail on each component is elaborated below.

5.2 Scanner (`scanner.mll`)

The Scanner is written in `ocamllex`, a lexer tool that is used for converting program source into a stream of tokens which can be used by a parser to assemble the Abstract Syntax Tree. Specifically, it takes program source and converts it into identifiers (described in Section 3.1.3) keywords (described in Section 3.1.4), and literals (described in Section 3.2), strips out comments and whitespace, and ensures that all input can be tokenized (if not, a `Parse_error` is thrown during lexing).

5.3 Parser (`parser.mly`)

The Parser is written in `ocamlyacc`, a tool for creating an executable OCaml parser from a declarative grammar. The Parser's job is to take a stream of tokens and output an Abstract Syntax Tree, which represents the program structure. It contains the grammar for the program, represented as a set of production rules for program constituents. If the input program passes the parsing stage, then it is a syntactically correct program, even if there are semantic errors.

5.4 Abstract Syntax Tree (`ast.ml`)

The Abstract Syntax Tree is a data structure that represents the syntactic structure of the program, and is generated using `ocamlyacc` in conjunction with the Parser and the `ast.ml` file, which encodes the relationships between OCaml types used to represent the tree. The `ast.ml` file also contains a library of pretty-printing functions used for examining how the program was parsed into a tree, which can be helpful for debugging.

5.5 Semantic Checker (`semant.ml`)

The Semantic Checker, written in OCaml, transforms the generated AST into a Semantically-annotated AST, or SAST, and builds a symbol table containing identifiers and their types. The symbol table is used to resolve references to symbols when building the SAST.

The Semantic Checker uses several `check()` functions to perform semantic validation on the generated AST, e.g., by checking that there are no duplicate identifiers in a given scope, that

5.6 Semantically-annotated Abstract Syntax Tree (`sast.ml`)

The Semantically-annotated Abstract Syntax Tree is built by the Semantic Checker with reference to an OCaml file (`sast.ml`) that contains semantically-annotated types and their relationships.

5.7 Code Generator (`codegen.ml`)

Finally, the code generator is used to translate the checked SAST into an LLVM program by traversing the SAST and generating LLVM code for the entirety of the semantic structure of the program. This is also written in OCaml (using the `llvm` and `llvm.analysis` packages) and is found in the `codegen.ml` file.

6. Test Plan

6.1 Representative Programs

6.1.1 Example: Storing values in an array and adding them

```
function add(x : number, y : number, z : number) : number {
  return x + y + z;
}

function main() : number {
  let vals : number[3] = [33, 33, 34];
  printn(add(vals[0], vals[1], vals[2]));
  return 0;
}
```

This will print out the value 100. The LLVM code generated is:

```
; ModuleID = 'TSVM'
source_filename = "TSVM"

@fmt = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %vals = alloca [3 x double]
  store [3 x double] [double 3.300000e+01, double 3.300000e+01, double
```

```
3.400000e+01], [3 x double]* %vals
  %tmp = getelementptr [3 x double], [3 x double]* %vals, i32 0, i32 2
  %tmp1 = load double, double* %tmp
  %tmp2 = getelementptr [3 x double], [3 x double]* %vals, i32 0, i32 1
  %tmp3 = load double, double* %tmp2
  %tmp4 = getelementptr [3 x double], [3 x double]* %vals, i32 0, i32 0
  %tmp5 = load double, double* %tmp4
  %add_result = call double @add(double %tmp5, double %tmp3, double %tmp1)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), double %add_result)
  ret i32 0
}

define double @add(double %x, double %y, double %z) {
entry:
  %x1 = alloca double
  store double %x, double* %x1
  %y2 = alloca double
  store double %y, double* %y2
  %z3 = alloca double
  store double %z, double* %z3
  %x4 = load double, double* %x1
  %y5 = load double, double* %y2
  %tmp = fadd double %x4, %y5
  %z6 = load double, double* %z3
  %tmp7 = fadd double %tmp, %z6
  ret double %tmp7
}
```

6.1.2 Example: Greatest Common Divisor

```
function gcd(a : number, b : number) : number {
  while (a != b) {
    if (a > b)
      a = a - b;
      else b = b - a;
    }
    return a;
```

```
}

function main() : number {
  printn(gcd(45,54));
  printn(gcd(125,15130));
  printn(gcd(11,121));
  return 0;
}
```

This will print:

```
9
5
11
```

The LLVM code generated is:

```
; ModuleID = 'TSVM'
source_filename = "TSVM"

@fmt = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main() {
entry:
  %gcd_result = call double @gcd(double 4.500000e+01, double 5.400000e+01)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), double %gcd_result)
  %gcd_result1 = call double @gcd(double 1.250000e+02, double 1.513000e+04)
  %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), double %gcd_result1)
  %gcd_result3 = call double @gcd(double 1.100000e+01, double 1.210000e+02)
  %printf4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), double %gcd_result3)
  ret i32 0
```

```
}

define double @gcd(double %a, double %b) {
entry:
  %a1 = alloca double
  store double %a, double* %a1
  %b2 = alloca double
  store double %b, double* %b2
  br label %while

while:                                            ; preds = %merge, %entry
  %a11 = load double, double* %a1
  %b12 = load double, double* %b2
  %tmp13 = fcmp one double %a11, %b12
  br i1 %tmp13, label %while_body, label %merge14

while_body:                                       ; preds = %while
  %a3 = load double, double* %a1
  %b4 = load double, double* %b2
  %tmp = fcmp ogt double %a3, %b4
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  br label %while

then:                                             ; preds = %while_body
  %a5 = load double, double* %a1
  %b6 = load double, double* %b2
  %tmp7 = fsub double %a5, %b6
  store double %tmp7, double* %a1
  br label %merge

else:                                             ; preds = %while_body
  %b8 = load double, double* %b2
  %a9 = load double, double* %a1
  %tmp10 = fsub double %b8, %a9
  store double %tmp10, double* %b2
  br label %merge

merge14:                                          ; preds = %while
  %a15 = load double, double* %a1
  ret double %a15
```

```
}
```

6.1.3 Example: Greatest Common Divisor (Recursive)

Here is the recursive version of the GCD algorithm:

```
function gcd(a: number, b: number): number {
  if (a == b) {
    return a;
  }
  if (a > b) {
    return gcd(a - b, b);
  } else {
    return gcd(a, b - a);
  }
}

function main(): number {
  printn(gcd(45, 54));
  printn(gcd(125, 15130));
  printn(gcd(11, 121));
  return 0;
}
```

It will print the same output as the non-recursive version since it uses the same inputs.

Here is the generated LLVM code:

```
; ModuleID = 'TSVM'
source_filename = "TSVM"

@fmt = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

declare i32 @printf(i8*, ...)
```

```
define i32 @main() {
entry:
  %gcd_result = call double @gcd(double 4.500000e+01, double 5.400000e+01)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), double %gcd_result)
  %gcd_result1 = call double @gcd(double 1.250000e+02, double 1.513000e+04)
  %printf2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), double %gcd_result1)
  %gcd_result3 = call double @gcd(double 1.100000e+01, double 1.210000e+02)
  %printf4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt, i32 0, i32 0), double %gcd_result3)
  ret i32 0
}

define double @gcd(double %a, double %b) {
entry:
  %a1 = alloca double
  store double %a, double* %a1
  %b2 = alloca double
  store double %b, double* %b2
  %a3 = load double, double* %a1
  %b4 = load double, double* %b2
  %tmp = fcmp oeq double %a3, %b4
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else
  %a6 = load double, double* %a1
  %b7 = load double, double* %b2
  %tmp8 = fcmp ogt double %a6, %b7
  br i1 %tmp8, label %then10, label %else15

then:                                             ; preds = %entry
  %a5 = load double, double* %a1
  ret double %a5

else:                                             ; preds = %entry
  br label %merge

merge9:                                           ; No predecessors!
  ret double 0.000000e+00
```

```
then10:                                           ; preds = %merge
  %b11 = load double, double* %b2
  %a12 = load double, double* %a1
  %b13 = load double, double* %b2
  %tmp14 = fsub double %a12, %b13
  %gcd_result = call double @gcd(double %tmp14, double %b11)
  ret double %gcd_result

else15:                                           ; preds = %merge
  %b16 = load double, double* %b2
  %a17 = load double, double* %a1
  %tmp18 = fsub double %b16, %a17
  %a19 = load double, double* %a1
  %gcd_result20 = call double @gcd(double %a19, double %tmp18)
  ret double %gcd_result20
}
```

6.2 Test Suite

The test suite used is adapted from the one used in the MicroC source. These tests are scripted via Bash and are automatically invoked when using `make` or `testall.sh`.

Test cases were written and chosen to exercise features - naturally there are more failure cases for each feature on average because there are several edge cases to test. I wrote all of the test cases, with reference to the MicroC test suite.

The following shows the full set of 57 tests (test code can be found in the Test Code Listing):

```
$ make clean && make
ocamlbuild -clean
Finished, 0 targets (0 cached) in 00:00:00.
00:00:00 0    (0   ) STARTING
-------- |rm -rf testall.log ocamlllvm *.diff
opam config exec -- \
ocamlbuild -use-ocamlfind tsvm.native -package llvm,llvm.analysis
Finished, 25 targets (0 cached) in 00:00:11.
./testall.sh
test-add-2-numbers...OK
test-add-3-numbers...OK
test-array-addition...OK
test-array-alloc...OK
test-assign-bool-false...OK
```

```
test-assign-bool-true...OK
test-assign-number...OK
test-divide-2-numbers...OK
test-divide-3-numbers...OK
test-func-noargs-nobody...OK
test-func-noargs-simplebody...OK
test-gcd...OK
test-hello...OK
test-if-else-false...OK
test-if-else-true...OK
test-if-false...OK
test-if-true...OK
test-multiply-2-numbers...OK
test-multiply-3-numbers...OK
test-rec-gcd...OK
test-sub-2-numbers...OK
test-sub-3-numbers...OK
test-while-dec...OK
test-while-func...OK
fail-add-boolean...OK
fail-array-change...OK
fail-array-empty...OK
fail-array-index-over...OK
fail-array-index-under...OK
fail-array-mixed-types...OK
fail-array-negative-size...OK
fail-array-size-mismatch...OK
fail-array-type-mismatch...OK
fail-assign-bool-rval...OK
fail-assign-number-rval...OK
fail-code-after-nested-return...OK
fail-code-after-return...OK
fail-divide-boolean...OK
fail-func-dupe-formal...OK
fail-func-dupe...OK
fail-func-redefine-printb...OK
fail-func-redefine-printn...OK
fail-func-too-few-args...OK
fail-func-too-many-args...OK
fail-func-void-formal...OK
fail-func-void-local...OK
fail-func-wrong-formal-types-number...OK
fail-func-wrong-formal-types-void...OK
fail-let-dupe...OK
fail-multiply-boolean...OK
fail-no-main...OK
fail-redefine-printb...OK
```

```
fail-redefine-printn...OK
fail-return-bool-instead-of-number...OK
fail-return-number-instead-of-void...OK
fail-sub-boolean...OK
fail-while-bad-call...OK
fail-while-nonbool...OK
```

7. Lessons Learned

There are two areas where lessons were learned: technical learnings and project/product management.

Technical

- **Array allocation/local storage:** initially when implementing the array construct I thought I could simply use `L.const_array` and LLVM would understand that I wanted to store the values being declared; during testing I discovered a bug - I was getting values back when indexing into an array using `L.build_gep`, but they did not match what I expected. It turned out that I was not properly storing local variables using `L.build_alloca` and `L.build_store` - which extended to arrays and any other type.
    - I immediately added integration tests to exercise local variable allocation and storage.
- **Entry point:** Typescript-on-LLVM requires a `main()` function that acts as an entry point for the program. This was done as a simplification; given more time, it would be interesting to put in compile-time checks that ensure that at least one function is called within the global scope.

Project and Product Management

- Make tradeoffs on feature support: learn to “legislate away” features that add unnecessary complexity during implementation and are not critical when there’s a feasible workaround.
- A useful practice in software development is to “launch-and-iterate” - that is, get something up and running end-to-end as soon as possible (and ideally, being used by real users), and add features incrementally. In my daily work I am constantly emphasizing this to my team. For this solo project, I did this to a certain extent but in truth there was a mad dash in the last week in order to provide a more robust set of integration tests that exercised the feature set adequately.
- Test-first development works well for compilers: once all the boilerplate was set up, adding new features became an iterative (and enjoyable) process with relatively fast turnaround time.
    - **Corollary:** You’re never really “finished” - Typescript-on-LLVM does not have feature parity with Typescript, but is in a place where getting better feature parity is possible: adding features now has a blueprint:
        - Add an integration test for the simplest program that exercises the feature
        - Update the Scanner to correctly tokenize the input program
        - Update the Parser and AST correctly parse the tokenized input into a proper AST
        - Update the SAST and Semantic Checker (with TODOs for checks) to transform the AST into a SAST

        - Update the Translator with to generate LLVM for the feature
        - Fix TODOs until the test case works as intended
        - Iterate on the integration test suite to exercise edge cases and add appropriate checks as needed.
- Don’t name a language until you’ve gone through the exercise of implementing some parts of it. Typescript-on-LLVM is a hideous name; given that the emphasis is server-side (rather than browser) execution, a more fashionable name might have been “Node.ts”

8. Appendix

8.1 Translator Code Listing

8.1.1 `ast.ml`

```
(*
 * Abstract Syntax Tree and functions for printing it
 * Author: Ratheet Pandya (rp2707)
 *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq

type typ = Bool | Num | NumArray of int | Str | Void

type expr =
    BoolLit of bool
  | NumLit of float
  | NumLitArray of float list
  | StrLit of string
  | Id of string
  | Index of string * int
  | Call of string * expr list
  | Binop of expr * op * expr
  | Noexpr

type stmt =
    Block of stmt list
  | ReAssign of string * expr
  | Expr of expr
  | If of expr * stmt * stmt
  | Return of expr
  | While of expr * stmt

type assignment_expr =
  Assign of string * typ * expr

type func_decl = {
    fname : string;
```

```
    formals : (string * typ) list;
    typ : typ;
    locals : assignment_expr list;
    body : stmt list;
}

type program = assignment_expr list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="

let string_of_typ = function
    Bool -> "boolean"
  | Num  -> "number"
  | NumArray(n) -> "number[" ^ string_of_int n ^ "]"
  | Str  -> "string"
  | Void -> "void"

let rec string_of_expr = function
    BoolLit(true)  -> "true"
  | BoolLit(false) -> "false"
  | Id(s)          -> s
  | NumLit(n)      -> string_of_float n
  | NumLitArray(el)   ->
      "[" ^ String.concat ", " (List.map string_of_float el) ^ "]"
  | StrLit(s)      -> "\"" ^ s ^ "\""
  | Index(s, i)    -> s ^ "[" ^ string_of_int i ^ "]"
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
```

```
  | Noexpr         -> ""

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

let string_of_assignment_expr = function
    Assign(id, t, expr) ->
      "let " ^ id ^ " : " ^ string_of_typ t ^ " = " ^ string_of_expr expr ^
";\n"

let string_of_fdecl fdecl =
  "function " ^ fdecl.fname ^ "(" ^ String.concat ", " (List.map fst
fdecl.formals) ^
  ") : " ^ string_of_typ fdecl.typ ^ " " ^ "\n{\n" ^
  String.concat "" (List.map string_of_assignment_expr fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

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

8.1.2 `codegen.ml`

```
(*
 * Code generation: translate takes a semantically checked AST and
 * produces LLVM IR.
 * Author: Ratheet Pandya (rp2707)
 *)
```

```
module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "TSVM" in

  (* LLVM insists each basic block end with exactly one "terminator"
     instruction that transfers control.  This function runs "instr
builder"
     if the current block does not already have a terminator.  Used,
     e.g., to handle the "fall off the end of the function" case. *)
  let add_terminal builder instr =
      match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()
      | None -> ignore (instr builder) in

  (* Get types from the context *)
  let i32_t         = L.i32_type    context
  and i8_t          = L.i8_type     context
  and i1_t          = L.i1_type     context
  and float_t       = L.double_type context
  and void_t        = L.void_type   context in

  (* Return the LLVM type for a Typescript-on-LLVM type *)
  let ltype_of_typ = function
    | A.Bool  -> i1_t
    | A.Num -> float_t
    | A.NumArray(size) -> L.array_type float_t size
    | A.Void  -> void_t
    | _ -> raise (Failure ("ltype_of_typ: Case not implemented"))
  in

  let printf_t : L.lltype =
```

```
    L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func : L.llvalue =
    L.declare_function "printf" printf_t the_module in

  (* Create a map of global variables after creating each *)
  let global_vars : L.llvalue StringMap.t =
    let global_var m a = match a with
      SAssign(n, t, _) ->
        let init = match t with
          A.Num  -> L.const_float (ltype_of_typ t) 0.0
        | _      -> L.const_int (ltype_of_typ t) 0
     in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in

  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
    let function_decl m fdecl =
      let name = fdecl.sfname
      and formal_types = Array.of_list (List.map (fun (_,t) -> ltype_of_typ
t)
                                         fdecl.sformals)
      in let ftype = L.function_type (match fdecl.sfname with
                              "main" -> i32_t
                                      | _ -> ltype_of_typ fdecl.styp)
formal_types
    in
      StringMap.add name (L.define_function name ftype the_module, fdecl) m
in
    List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.sfname function_decls in
    let builder = L.builder_at_end context (L.entry_block the_function) in

    let float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
    and bool_format_str = L.build_global_stringptr "%d\n" "fmt" builder in

    (* Construct the function's "locals": formal arguments and locally
       declared variables.  Allocate each on the stack, initialize their
       value, if appropriate, and remember their values in the "locals" map
```

```
*)
    let local_vars =
      let add_formal m (n, t) p =
        L.set_value_name n p;
      let local = L.build_alloca (ltype_of_typ t) n builder in
        ignore (L.build_store p local builder);
        StringMap.add n local m

      (* Allocate space for any locally declared variables and add the
       * resulting registers to our map *)
      and add_local m = function
        SAssign(n, t, _) ->
        let local_var = L.build_alloca (ltype_of_typ t) n builder
        in StringMap.add n local_var m
      in

      let formals = List.fold_left2 add_formal StringMap.empty
fdecl.sformals
          (Array.to_list (L.params the_function)) in
      List.fold_left add_local formals fdecl.slocals
    in

    (* Return the value for a variable or formal argument.
       Check local names first, then global names *)
    let lookup n = try StringMap.find n local_vars
                   with Not_found -> StringMap.find n global_vars
    in

    (* Construct code for an expression; return its value *)
    let rec expr builder ((_, e) : sexpr) = match e with
        SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
      | SNumLit  n  -> L.const_float float_t n
      | SNumLitArray vals ->
          let llvals = List.map (L.const_float float_t) vals
          in L.const_array float_t (Array.of_list llvals)
      | SNoexpr     -> L.const_int i32_t 0
      | SId s       -> L.build_load (lookup s) s builder
      | SIndex (s, i) ->
          let index = L.build_add (L.const_int i32_t i)
                        (L.const_int i32_t 0) "tmp" builder
          in let value = L.build_gep (lookup s)
                           [| (L.const_int i32_t 0) ; index ; |] "tmp"
```

```
builder
                        in L.build_load value "tmp" builder
      | SBinop (e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
           A.Add     -> L.build_fadd
         | A.Sub     -> L.build_fsub
         | A.Mult    -> L.build_fmul
         | A.Div     -> L.build_fdiv
         | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
         | A.Neq     -> L.build_fcmp L.Fcmp.One
         | A.Less    -> L.build_fcmp L.Fcmp.Olt
         | A.Leq     -> L.build_fcmp L.Fcmp.Ole
         | A.Greater -> L.build_fcmp L.Fcmp.Ogt
         | A.Geq     -> L.build_fcmp L.Fcmp.Oge
        ) e1' e2' "tmp" builder
      | SCall ("printb", [e]) ->
          L.build_call printf_func [| bool_format_str ; (expr builder e) |]
        "printf" builder
      | SCall ("printn", [e]) ->
          L.build_call printf_func [| float_format_str ; (expr builder e)
|]
        "printf" builder
      | SCall (f, args) ->
         let (fdef, fdecl) = StringMap.find f function_decls in
       let llargs = List.rev (List.map (expr builder) (List.rev args)) in
       let result = (match fdecl.styp with
                       A.Void -> ""
                     | _ -> f ^ "_result") in
        L.build_call fdef (Array.of_list llargs) result builder
      | _ -> raise (Failure ("expr: Case not implemented"))
    in

    (* Construct code for an assignment expression; return the builder for
       the next assignment expression. *)
    let build_assignments al builder =
      let assignment_expr builder = function
          SAssign (s, t, e) -> let e' = expr builder (t, e) in
                               ignore(L.build_store e' (lookup s) builder);
                               builder
      in List.fold_left assignment_expr builder al
```

```
    in

    (* Build the code for the given statement; return the builder for
       the statement's successor (i.e., the next instruction will be built
       after the one generated by this call). *)
    let rec stmt builder = function
        SBlock sl -> List.fold_left stmt builder sl
      | SReAssign (s, (t, e)) -> let e' = expr builder (t, e) in
                                 ignore(L.build_store e' (lookup s)
builder);
                                 builder
      | SExpr e -> ignore(expr builder e); builder
      | SReturn e -> ignore(match fdecl.sfname with
                            (* main should always return an int *)
                            "main" -> L.build_ret (L.const_int i32_t 0)
builder
                          | _ -> match fdecl.styp with
                           (* Special "return nothing" instr *)
                                 A.Void -> L.build_ret_void builder
                                 (* Build return statement *)
                               | _ -> L.build_ret (expr builder e)
builder);
                    builder
      | SIf (predicate, then_stmt, else_stmt) ->
        let bool_val = expr builder predicate in
       let merge_bb = L.append_block context "merge" the_function in
        let build_br_merge = L.build_br merge_bb in (* partial function *)

       let then_bb = L.append_block context "then" the_function in
       add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
         build_br_merge;

       let else_bb = L.append_block context "else" the_function in
       add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
         build_br_merge;

       ignore(L.build_cond_br bool_val then_bb else_bb builder);
       L.builder_at_end context merge_bb
      | SWhile (predicate, body) ->
          let pred_bb = L.append_block context "while" the_function in
          ignore(L.build_br pred_bb builder);
```

```
          let body_bb = L.append_block context "while_body" the_function in
          add_terminal (stmt (L.builder_at_end context body_bb) body)
            (L.build_br pred_bb);

          let pred_builder = L.builder_at_end context pred_bb in
          let bool_val = expr pred_builder predicate in

          let merge_bb = L.append_block context "merge" the_function in
          ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
          L.builder_at_end context merge_bb
    in

    (* Build the code for each assignment in the function *)
    let assignment_builder = build_assignments fdecl.slocals builder in
    (* Build the code for each statement in the function *)
    let builder = stmt assignment_builder (SBlock fdecl.sbody) in
    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.sfname with
        (* main should always return an int *)
        "main" -> L.build_ret (L.const_int i32_t 0)
      | _ -> match fdecl.styp with
              A.Void -> L.build_ret_void
            | _ -> L.build_ret (L.const_float float_t 0.0))
  in
  List.iter build_function_body functions;
  the_module
```

8.1.3 Makefile

```
# Builds the translator and executes the test suite.
# Author: Ratheet Pandya (rp2707)
# "make test" Compiles everything and runs the regression tests

.PHONY : test
test : all testall.sh; ./testall.sh

# "make all" builds the executable

.PHONY : all
```

```
all : tsvm.native

# "make tsvm.native" compiles the compiler
#
# The _tags file controls the operation of ocamlbuild, e.g., by including
# packages, enabling warnings
#
# See https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc

tsvm.native :
      opam config exec -- \
      ocamlbuild -use-ocamlfind tsvm.native -package llvm,llvm.analysis

# "make clean" removes all generated files

.PHONY : clean
clean :
      ocamlbuild -clean
      rm -rf testall.log ocamlllvm *.diff

# Building the tarball

TESTS = \
  add-2-numbers add-3-numbers array-alloc array-addition \
  assign-number assign-bool-true assign-bool-false \
  divide-2-numbers divide-3-numbers \
  func-noargs-nobody func-noargs-simplebody \
  gcd hello \
  if-else-false if-else-true if-false if-func \
  multiply-2-numbers multiply-3-numbers \
  rec-gcd \
  sub-2-numbers sub-3-numbers \
  while-dec while-func

FAILS = \
  add-boolean array-empty array-change array-index-over array-index-under
array-size-mismatch \
  array-mixed-types array-negative-size array-type-mismatch \
  assign-number-rval assign-bool-rval \
  code-after-nested-return code-after-return \
  func-dupe func-dupe-formal func-redefine-printn func-redefine-printb \
  func-too-few-args func-too-many-args func-void-formal func-void-local \
```

```
  func-wrong-formal-types-number func-wrong-formal-types-void \
  let-dupe multiply-boolean no-main \
  redefine-printb redefine-printn return-bool-instead-of-number
return-number-instead-of-void \
  sub-boolean \
  while-bad-call while-nonbool

TESTFILES = $(TESTS:%=test-%.ts) $(TESTS:%=test-%.out) \
          $(FAILS:%=fail-%.ts) $(FAILS:%=fail-%.err)

TARFILES = ast.ml sast.ml codegen.ml Makefile _tags tsvm.ml parser.mly \
      README scanner.mll semant.ml testall.sh \
      $(TESTFILES:%=tests/%)

tsvm.tar.gz : $(TARFILES)
      cd .. && tar czf tsvm/tsvm.tar.gz \
            $(TARFILES:%=tsvm/%)
```

8.1.4 `parser.mly`

```
/*
 * Ocamlyacc parser for Typescript-on-LLVM
 * Author: Ratheet Pandya (rp2707)
 */

%{
open Ast
%}

%token ASSIGN PLUS MINUS TIMES DIVIDE
%token EQ NEQ LT LEQ GT GEQ
%token LET FUNCTION RETURN
%token IF ELSE WHILE
%token BOOL NUMBER STRING VOID
%token LEFT_BRACE RIGHT_BRACE
%token LEFT_BRACKET RIGHT_BRACKET
%token LEFT_PAREN RIGHT_PAREN
%token COLON SEMICOLON COMMA
%token <bool> BOOL_LIT
```

```
%token <float> NUM_LIT
%token <string> ID STR_LIT
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { ([], [])                 }
   | decls assignment_expr { (($2 :: fst $1), snd $1) }
   | decls fdecl { (fst $1, ($2 :: snd $1)) }

assignment_expr_list:
    /* nothing */    { [] }
    | assignment_expr_list assignment_expr { $2 :: $1 }

assignment_expr:
  LET ID COLON type_specifier ASSIGN expr SEMICOLON { Assign($2, $4, $6) }

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

stmt:
    LEFT_BRACE stmt_list RIGHT_BRACE { Block(List.rev $2) }
  | IF LEFT_PAREN expr RIGHT_PAREN stmt %prec NOELSE { If($3, $5,
Block([])) }
  | IF LEFT_PAREN expr RIGHT_PAREN stmt ELSE stmt    { If($3, $5, $7)
}
```

```
  | ID ASSIGN expr SEMICOLON { ReAssign($1, $3) }
  | expr SEMICOLON { Expr $1 }
  | RETURN expr_opt SEMICOLON { Return $2 }
  | WHILE LEFT_PAREN expr RIGHT_PAREN stmt { While($3, $5) }

type_specifier:
    BOOL      { Bool }
  | NUMBER LEFT_BRACKET NUM_LIT RIGHT_BRACKET { NumArray(int_of_float $3) }
  | NUMBER    { Num  }
  | STRING    { Str  }
  | VOID      { Void }

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

expr:
    BOOL_LIT      { BoolLit($1) }
  | LEFT_BRACKET nums_opt RIGHT_BRACKET { NumLitArray($2) }
  | NUM_LIT       { NumLit($1)  }
  | STR_LIT       { StrLit($1)  }
  | ID LEFT_PAREN args_opt RIGHT_PAREN  { Call($1, $3)  }
  | ID LEFT_BRACKET NUM_LIT RIGHT_BRACKET { Index($1, int_of_float $3) }
  | ID { Id($1) }
  | expr PLUS  expr  { Binop($1, Add,  $3)     }
  | expr MINUS expr  { Binop($1, Sub,  $3)     }
  | expr TIMES expr  { Binop($1, Mult, $3)     }
  | expr DIVIDE expr { Binop($1, Div,  $3)     }
  | expr 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 EQ     expr { Binop($1, Equal,   $3)  }
  | expr NEQ    expr { Binop($1, Neq,     $3)  }

nums_opt:
    /* nothing */ { [] }
  | nums_list  { List.rev $1 }

nums_list:
    NUM_LIT                 { [ $1 ] }
  | nums_list COMMA NUM_LIT { $3 :: $1 }
```

```
args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

args_list:
    expr                 { [ $1 ] }
  | args_list COMMA expr { $3 :: $1 }

fdecl:
  FUNCTION ID LEFT_PAREN formals_opt RIGHT_PAREN COLON type_specifier
  LEFT_BRACE assignment_expr_list stmt_list RIGHT_BRACE
    { {
        fname = $2;
      formals = List.rev $4;
      typ = $7;
      locals = List.rev $9;
      body = List.rev $10; } }

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

formal_list:
    ID COLON type_specifier                    { [($1,$3)]     }
  | formal_list COMMA ID COLON type_specifier { ($3,$5) :: $1 }
```

8.1.5 `sast.ml`

```
(*
 * Semantically-checked Abstract Syntax Tree and functions for printing it
 * Author: Ratheet Pandya (rp2707)
 *)

open Ast

type sexpr = typ * sx
and sx =
    SBoolLit of bool
  | SNumLit of float
```

```
  | SNumLitArray of float list
  | SStrLit of string
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SIndex of string * int
  | SCall of string * sexpr list
  | SNoexpr

type sassignment_expr =
    SAssign of string * typ * sx

type sstmt =
    SBlock of sstmt list
  | SReAssign of string * sexpr
  | SExpr of sexpr
  | SReturn of sexpr
  | SWhile of sexpr * sstmt
  | SIf of sexpr * sstmt * sstmt

type sfunc_decl = {
    sfname : string;
    sformals : (string * typ) list;
    styp : typ;
    slocals : sassignment_expr list;
    sbody : sstmt list;
  }

type sprogram = sassignment_expr list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SBoolLit(true)   -> "true"
  | SBoolLit(false)  -> "false"
  | SNumLit(f)       -> string_of_float f
  | SNumLitArray(el) -> "[" ^ String.concat ", " (List.map string_of_float
el) ^ "]"
  | SStrLit(s)       -> s
  | SId(s) -> s
  | SIndex(s, i)  -> s ^ "[" ^ string_of_int i ^ "]"
  | SBinop(e1, o, e2) ->
```

```
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SNoexpr -> ""
                          ) ^ ")"

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SReAssign(s, expr) -> s ^ " = " ^ string_of_sexpr expr ^ ";\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n"
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n"
  | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt
s

let string_of_sassignment_expr = function
    SAssign(n, t, e) ->
      "let " ^ n ^ " : " ^ string_of_sexpr (t, e) ^ ";\n"

let string_of_sfdecl fdecl =
  "function " ^ fdecl.sfname ^ "(" ^ String.concat ", " (List.map fst
fdecl.sformals) ^ ") : " ^
  string_of_typ fdecl.styp ^ "\n{\n"   ^
  String.concat "" (List.map string_of_sassignment_expr fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

let string_of_sprogram (vars, funcs) =
  String.concat "" (List.map string_of_sassignment_expr vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

8.1.6 scanner.mll

```
(*
 * Ocamllex scanner for Typescript-on-LLVM
 * Author: Ratheet Pandya (rp2707)
```

```
 *)

{
open Parser

exception SyntaxError of string
}

let digit = ['0' - '9']
let frac = '.' digit*
let exp = ['e' 'E'] ['-' '+']? digit+
let number = '-'? digit* frac? exp?

rule token = parse
    [' ' '\t' '\r' '\n'] { token lexbuf }   (* Whitespace *)
  | "/*"                 { comment lexbuf } (* Comments *)
  | "if"          { IF }
  | "else"        { ELSE }
  | "true"        { BOOL_LIT(true)  }
  | "false"       { BOOL_LIT(false) }
  | "boolean"     { BOOL   }
  | "number"      { NUMBER }
  | "string"      { STRING }
  | "void"        { VOID   }
  | "function"    { FUNCTION }
  | "let"         { LET }
  | "while"       { WHILE }
  | "return"      { RETURN }
  | '='           { ASSIGN }
  | "=="          { EQ }
  | "!="          { NEQ }
  | '<'           { LT }
  | "<="          { LEQ }
  | ">"           { GT }
  | ">="          { GEQ }
  | '+'           { PLUS   }
  | '-'           { MINUS  }
  | '*'           { TIMES }
  | '/'           { DIVIDE }
  | ':'           { COLON     }
  | ';'           { SEMICOLON }
  | ','           { COMMA     }
```

```
  | '{'           { LEFT_BRACE }
  | '}'           { RIGHT_BRACE }
  | '['           { LEFT_BRACKET }
  | ']'           { RIGHT_BRACKET }
  | '('           { LEFT_PAREN }
  | ')'           { RIGHT_PAREN }
  | '"'           { read_string (Buffer.create 17) lexbuf }
  | ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | number as lxm { NUM_LIT (float_of_string lxm) }
  | _             { raise (SyntaxError ("Unexpected char: " ^ Lexing.lexeme
lexbuf)) }
  | eof           { EOF }

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

and read_string buf = parse
    '"'       { STR_LIT (Buffer.contents buf) }
  | '\\' '/'  { Buffer.add_char buf '/'; read_string buf lexbuf }
  | '\\' '\\' { Buffer.add_char buf '\\'; read_string buf lexbuf }
  | '\\' 'b'  { Buffer.add_char buf '\b'; read_string buf lexbuf }
  | '\\' 'f'  { Buffer.add_char buf '\012'; read_string buf lexbuf }
  | '\\' 'n'  { Buffer.add_char buf '\n'; read_string buf lexbuf }
  | '\\' 'r'  { Buffer.add_char buf '\r'; read_string buf lexbuf }
  | '\\' 't'  { Buffer.add_char buf '\t'; read_string buf lexbuf }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      read_string buf lexbuf
    }
  | _   { raise (SyntaxError ("Illegal string character: " ^ Lexing.lexeme
lexbuf)) }
  | eof { raise (SyntaxError ("String is not terminated")) }
```

8.1.7 `semant.mll`

```
(*
 * Semantic checking for the Typescript-on-LLVM compiler
 * Author: Ratheet Pandya (rp2707)
 *)
```

```
open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =
  (* Verify a list of bindings has no void types or duplicate names *)
  let check_assignment_exprs (kind : string) (binds : assignment_expr list)
=
    List.iter (function Assign(n, Void, b) ->
                  raise (Failure ("illegal void " ^ kind ^ " (" ^ n ^ ") "
^
                                  string_of_expr b))
      | _ -> ()) binds;
    let rec dups = function
        [] -> ()
      |     ((n1,_,_) :: (n2,_,_) :: _) when n1 = n2 ->
        raise (Failure ("duplicate " ^ kind ^ ": " ^ n1))
      | _ :: t -> dups t
    in
    let tuple_of = function
      Assign(a, b, c) -> (a, b, c)
    in dups (List.sort compare (List.map tuple_of binds));
    let check_assignment_type (t, expected, ex) =
      if t = expected then t
      else raise (Failure ("Invalid type "     ^ string_of_typ t ^
                      " for assignment: " ^ string_of_expr ex))
    in
    let check_assignment_expr = function
      Assign(s, t, e) -> match e with
          NumLit  n  -> SAssign(s, check_assignment_type(t, Num, e),
SNumLit n)
        | BoolLit b  -> SAssign(s, check_assignment_type(t, Bool, e),
SBoolLit b)
      | NumLitArray v -> (let sz = List.length v in
                          if sz < 1
```

```
                            then raise (Failure ("Array size " ^
                                        string_of_int sz ^
                                        " must be greater than zero in " ^
                                        string_of_expr e))
                            else
                              SAssign(
                                s,
                                check_assignment_type(t, NumArray(sz), e),
                                SNumLitArray v))
        | _           -> raise (Failure ("Unsupported expression for
assignment: " ^
                                        string_of_expr e))
    in List.map check_assignment_expr binds
  in

  (**** Check global variables ****)
  let check_globals g =
    check_assignment_exprs "global" g
  in

  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = Void;
      fname = name;
      formals = [("x", ty)];
      locals = [];
      body = [] } map
    in List.fold_left add_bind StringMap.empty [ ("printn", Num);
                                                 ("printb", Bool) ]
  in

  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *)
    in match fd with (* No duplicate functions or redefinitions of
built-ins *)
```

```
        _ when StringMap.mem n built_in_decls -> make_err built_in_err
      | _ when StringMap.mem n map -> make_err dup_err
      | _ ->  StringMap.add n fd map
  in

  (* Collect all function names into one symbol table *)
  let function_decls = List.fold_left add_func built_in_decls functions
  in

  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> if s = "main"
                      then raise (Failure ("main must be defined"))
                      else raise (Failure ("unrecognized function " ^ s))
  in

  let _ = find_func "main" in (* Ensure "main" is defined *)

  let check_function func =
    let check_formals formals =
      List.iter (function
             (n, Void) -> raise (Failure("illegal void parameter (" ^ n ^")
"))
      | _ -> ()) formals;
      let rec dups = function
          [] -> ()
        | ((n1,_) :: (n2,_) :: _) when n1 = n2 ->
          raise (Failure ("duplicate formal " ^ n1))
        | _ :: t -> dups t
      in dups (List.sort (fun (a,_) (b,_) -> compare a b) formals)
    in

    (* Make sure formals have valid types, are unique, and
       have valid assignments. *)
    check_formals func.formals;

    (* Raise an exception if the given rvalue type cannot be assigned to
       the given lvalue type *)
    let check_assign lvaluet rvaluet err =
       if lvaluet = rvaluet then lvaluet else raise (Failure err)
    in
```

```
    (* Build local symbol table of variables for this function *)
    let extract_binds = List.map (fun a -> match a with Assign(n, t, _) ->
(n, t)) in
    let symbols = List.fold_left (fun m (name, ty) -> StringMap.add name ty
m)
                  StringMap.empty (extract_binds globals @
                    func.formals @ extract_binds func.locals)
    in

    (* Return a variable from our local symbol table *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in

    (* Return a semantically-checked expression, i.e., with a type *)
    let rec expr = function
        NumLit  n  -> (Num, SNumLit n)
      | BoolLit b  -> (Bool, SBoolLit b)
      | Noexpr     -> (Void, SNoexpr)
      | Id s       -> (type_of_identifier s, SId s)
      | Binop(e1, op, e2) as e ->
          let (t1, e1') = expr e1
          and (t2, e2') = expr e2 in
          (* All binary operators require operands of the same type *)
          let same = t1 = t2 in
          (* Determine expression type based on operator and operand types
*)
          let ty = match op with
            Add | Sub | Mult | Div when same && t1 = Num -> Num
        | Equal | Neq            when same             -> Bool
          | Less | Leq | Greater | Geq
                     when same && t1 = Num    -> Bool
          | _ -> raise (
            Failure ("illegal binary operator " ^
                      string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                      string_of_typ t2 ^ " in " ^ string_of_expr e))
          in (ty, SBinop((t1, e1'), op, (t2, e2')))
      | Index(s, i) ->
          let vtype = match type_of_identifier s with
        NumArray(sz) -> if sz < 1 then raise (Failure("Array (" ^ s ^ ") "
```

```
^
                                        "size (" ^
                                        string_of_int sz ^
                                        ") must be a positive " ^
                                        "integer."))
                                 else if 0 <= i && i < sz then Num
                                    else raise (Failure (
                                "Array (" ^ s ^ ") index " ^
                                string_of_int i ^
                                " out of range: index must be in range [0, " ^
                                string_of_int (sz) ^ ")"))
          | x -> raise (Failure ("Invalid array type: " ^ string_of_typ x))
        in (vtype, SIndex(s, i))
      | Call(fname, args) as call ->
          let fd = find_func fname in
        let param_length = List.length fd.formals in
          if List.length args != param_length then
            raise (Failure ("expecting " ^ string_of_int param_length ^
                            " arguments in " ^ string_of_expr call))
        else
        let check_call (_, ft) e =
            let (et, e') = expr e in
            let err = "illegal argument: found " ^ string_of_typ et ^
              "; expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
            in (check_assign ft et err, e')
          in
          let args' = List.map2 check_call fd.formals args
        in
        (fd.typ, SCall(fname, args'))
      | e -> raise (Failure ("expr: Case not implemented - " ^
string_of_expr e))
    in

    let check_bool_expr e =
      let (t', e') = expr e
      and err = "expected Boolean expression in " ^ string_of_expr e
      in if t' != Bool then raise (Failure err) else (t', e')
    in

    (* Return a semantically-checked statement i.e. containing sexprs *)
    let rec check_stmt = function
      (* A block is correct if each statement is correct and nothing
```

```
         follows any Return statement.  Nested blocks are flattened. *)
        Block sl ->
          let rec check_stmt_list = function
              [Return _ as s] -> [check_stmt s]
            | Return _ :: _   -> raise (Failure "nothing may follow a
return")
            | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten
blocks *)
            | s :: ss         -> check_stmt s :: check_stmt_list ss
            | []              -> []
          in SBlock(check_stmt_list sl)
      | ReAssign(s, e) -> SReAssign(s, expr e)
      | Expr e -> SExpr (expr e)
      | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt
b2)
      | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
      | Return e -> let (t, e') = expr e in
        if t = func.typ then SReturn (t, e')
        else raise (
        Failure ("return gives " ^ string_of_typ t ^ " expected " ^
              string_of_typ func.typ ^ " in " ^ string_of_expr e))


    in (* body of check_function *)
    {
      sfname = func.fname;
      sformals = func.formals;
      styp = func.typ;
      (* Make sure locals have valid types, are unique, and
         have valid assignments. *)
      slocals = check_assignment_exprs "local" func.locals;
      sbody = match check_stmt (Block func.body) with
        SBlock(sl) -> sl
        | _ -> raise (Failure ("internal error: block didn't become a
block?"))
    }

 in (check_globals globals, List.map check_function functions)
```

8.1.8 testall.sh

```
#!/bin/sh

# Regression testing script for Typescript-on-LLVM
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test
# Author: Ratheet Pandya (rp2707)

# Path to the LLVM interpreter
LLI="lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the tsvm compiler.  Usually "./tsvm.native"
# Try "_build/tsvm.native" if ocamlbuild was unable to create a symbolic
link.
TSVM="./tsvm.native"
#TSVM="_build/tsvm.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.ts files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
```

```
}

SignalError() {
    if [ $error -eq 0 ] ; then
      echo "FAILED"
      error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to
difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
      SignalError "$1 differs"
      echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
      SignalError "$1 failed on $*"
      return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
      SignalError "failed: $* did not report an error"
      return 1
    }
    return 0
}
```

```
Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.ts//'`
    reffile=`echo $1 | sed 's/.ts$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s
${basename}.exe ${basename}.out" &&
    Run "$TSVM" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s"
&&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" &&
    Run "./${basename}.exe" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
      if [ $keep -eq 0 ] ; then
          rm -f $generatedfiles
      fi
      echo "OK"
      echo "###### SUCCESS" 1>&2
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.ts//'`
```

```
    reffile=`echo $1 | sed 's/.ts$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$TSVM" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
      if [ $keep -eq 0 ] ; then
          rm -f $generatedfiles
      fi
      echo "OK"
      echo "###### SUCCESS" 1>&2
    else
      echo "###### FAILED" 1>&2
      globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
      k) # Keep intermediate files
          keep=1
          ;;
      h) # Help
          Usage
          ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
```

```
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in
testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.ts tests/fail-*.ts"
fi

for file in $files
do
    case $file in
      *test-*)
          Check $file 2>> $globallog
          ;;
      *fail-*)
          CheckFail $file 2>> $globallog
          ;;
      *)
          echo "unknown file type $file"
          globalerror=1
          ;;
    esac
done

exit $globalerror
```

8.1.9 `tsvm.ml`

```
(* Top-level of the Typescript-on-LLVM compiler: scan & parse the input,
 * check the resulting AST and generate an SAST from it, generate LLVM IR,
 * and dump the module
 * Author: Ratheet Pandya (rp2707)
 *)
```

```
type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./tsvm.native [-a|-s|-l|-c] [file.ts]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename)
usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
    | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
sast))
    | Compile -> let m = Codegen.translate sast in
      Llvm_analysis.assert_valid_module m;
      print_string (Llvm.string_of_llmodule m)
```

8.2 Test Code Listing

```
==> tests/fail-add-boolean.ts <==
function main() : number {
  let x : number = 3;
  let y : boolean = false;
  printn(x + y);
```

```
  return 0;
}
==> tests/fail-array-change.ts <==
function main() : number {
  let values: number[3] = [1, 2, 3];
  let values: number[3] = [4, 5, 6];
  printn(values[1]);
  return 0;
}

==> tests/fail-array-empty.ts <==
function main() : number {
  let values: number[0] = [];
  printn(values[0]);
  return 0;
}

==> tests/fail-array-index-over.ts <==
function main() : number {
  let values: number[3] = [1, 2, 3];
  printn(values[3]);
  return 0;
}

==> tests/fail-array-index-under.ts <==
function main() : number {
  let values: number[3] = [1, 2, 3];
  printn(values[-3]);
  return 0;
}

==> tests/fail-array-mixed-types.ts <==
function main() : number {
  let values: number[3] = [1, false, 3];
  printn(values[1]);
  return 0;
}

==> tests/fail-array-negative-size.ts <==
function main() : number {
  let values: number[-3] = [1, 2, 3];
  printn(values[1]);
```

```
  return 0;
}

==> tests/fail-array-size-mismatch.ts <==
function main() : number {
  let values: number[10] = [1, 2, 3];
  printn(values[1]);
  return 0;
}

==> tests/fail-array-type-mismatch.ts <==
function main() : number {
  let values: number[3] = [true, false, true];
  printn(values[1]);
  return 0;
}

==> tests/fail-assign-bool-rval.ts <==
function main() : number {
  let value: boolean = 5; /* Fail: assign number to a boolean. */
  printb(value);
  return 0;
}

==> tests/fail-assign-number-rval.ts <==
function main() : number {
  let value: number = true; /* Fail: assigning boolean to a number. */
  printn(value);
  return 0;
}

==> tests/fail-code-after-nested-return.ts <==
function main() : void
{
  let i : number = 3;

  {
     i = 15;
     return i;
  }
  i = 32; /* Error: code after a return */
}
```

```
==> tests/fail-code-after-return.ts <==
function main() : void
{
  let i : number = 15;
  return i;
  i = 32; /* Error: code after a return */
}

==> tests/fail-divide-boolean.ts <==
function main() : number {
  let x : number = 3;
  let y : boolean = false;
  printn(x / y);
  return 0;
}
==> tests/fail-func-dupe-formal.ts <==
function foo(a:number, b:number, c:number) : void { }

function bar(a:number, b:boolean, a:number) : void {} /* Error: duplicate
formal a in bar */

function main() : number
{
  return 0;
}
==> tests/fail-func-dupe.ts <==
function foo() : void {}

function bar() : number {}

function baz() : number {}

function bar() : void {} /* Error: duplicate function bar */

function main() : number
{
  return 0;
}

==> tests/fail-func-redefine-printb.ts <==
function foo() : void {}
```

```
function bar() : void {}

function printb() : number {} /* Should not be able to define print */

function baz() : void {}

function main() : number
{
  return 0;
}
==> tests/fail-func-redefine-printn.ts <==
function foo() : void {}

function bar() : void {}

function printn() : number {} /* Should not be able to define print */

function baz() : void {}

function main() : number
{
  return 0;
}
==> tests/fail-func-too-few-args.ts <==
function foo(a:number, b:boolean) : void
{
}

function main() : number
{
  foo(42, true);
  foo(42); /* Wrong number of arguments */
  return 0;
}

==> tests/fail-func-too-many-args.ts <==
function foo(a:number, b:boolean) : void
{
}

function main() : number
```

```
{
  foo(42, true);
  foo(42, true, false); /* Wrong number of arguments */
  return 0;
}
==> tests/fail-func-void-formal.ts <==
function foo(a:number, b:number, c:number) : void { }

function bar(a:number, b:void, c:number) : void {} /* Error: duplicate
formal a in bar */

function main() : number
{
  return 0;
}
==> tests/fail-func-void-local.ts <==
function foo() : void {}

function bar() : number {
  let a : number = 0;
  let b : void = 3; /* Error: illegal void local b */
  let c : boolean = false;
  return 0;
}

function main() : number
{
  return 0;
}

==> tests/fail-func-wrong-formal-types-number.ts <==
function foo(a:number, b:boolean) : void
{
}

function main() : void
{
  foo(42, true);
  foo(42, 42); /* Fail: number, not boolean */
}
==> tests/fail-func-wrong-formal-types-void.ts <==
function foo(a:number, b:boolean) : void
```

```
{
}

function bar() : void
{
}

function main() : number
{
  foo(42, true);
  foo(42, bar()); /* number and void, not number and boolean */
}

==> tests/fail-let-dupe.ts <==
function main() : void {
  let x : number = 1;
  let x : number = 2;
}
==> tests/fail-multiply-boolean.ts <==
function main() : number {
  let x : number = 3;
  let y : boolean = false;
  printn(x * y);
  return 0;
}
==> tests/fail-no-main.ts <==
function main2() : number {
  return 0;
}

==> tests/fail-redefine-printb.ts <==
/* Should be illegal to redefine */
function printb() : void {}

==> tests/fail-redefine-printn.ts <==
/* Should be illegal to redefine */
function printn() : void {}

==> tests/fail-return-bool-instead-of-number.ts <==
function main() : number
{
  return true; /* Should return a number */
```

```
}

==> tests/fail-return-number-instead-of-void.ts <==
function foo() : void
{
  if (true) return 42; /* Should return void */
  else return;
}

function main() : number
{
  return 42;
}

==> tests/fail-sub-boolean.ts <==
function main() : number {
  let x : number = 3;
  let y : boolean = false;
  printn(x - y);
  return 0;
}
==> tests/fail-while-bad-call.ts <==
function main() : void
{
  let i : number = 0;

  while (true) {
    i = i + 1;
  }

  while (true) {
    i = i + 1;
    foo(); /* foo undefined */
  }
}

==> tests/fail-while-nonbool.ts <==
function main() : void
{
  let i : number = 0;

  while (true) {
```

```
    i = i + 1;
  }

  while (1) { /* Should be boolean */
    i = i + 1;
  }
}

==> tests/test-add-2-numbers.ts <==
function add(x : number, y : number) : number {
  return x + y;
}

function main() : number {
  printn(add(17, 25));
  return 0;
}
==> tests/test-add-3-numbers.ts <==
function add(x : number, y : number, z : number) : number {
  return x + y + z;
}

function main() : number {
  printn(add(17, 25, 32));
  return 0;
}

==> tests/test-array-addition.ts <==
function add(x : number, y : number, z : number) : number {
  return x + y + z;
}

function main() : number {
  let vals : number[3] = [33, 33, 34];
  printn(add(vals[0], vals[1], vals[2]));
  return 0;
}

==> tests/test-array-alloc.ts <==
function main() : number {
  let values: number[3] = [1, 2, 3];
  printn(values[1]);
```

```
  return 0;
}

==> tests/test-assign-bool-false.ts <==
function main() : number {
  let value: boolean = false;
  printb(value);
  return 0;
}

==> tests/test-assign-bool-true.ts <==
function main() : number {
  let value: boolean = true;
  printb(value);
  return 0;
}

==> tests/test-assign-number.ts <==
function main() : number {
  let value: number = 3.14;
  printn(value);
  return 0;
}

==> tests/test-divide-2-numbers.ts <==
function divide(x : number, y : number) : number {
  return x / y;
}

function main() : number {
  printn(divide(35, 5));
  return 0;
}


==> tests/test-divide-3-numbers.ts <==
function divide(x : number, y : number, z : number) : number {
  return x / y / z;
}

function main() : number {
  printn(divide(100000, 100, 10));
```

```
  return 0;
}

==> tests/test-func-noargs-nobody.ts <==
function main() : void {}
==> tests/test-func-noargs-simplebody.ts <==
function main() : number {
  return 0;
}

==> tests/test-gcd.ts <==
function gcd(a : number, b : number) : number {
  while (a != b) {
    if (a > b)
      a = a - b;
      else b = b - a;
    }
    return a;
}

function main() : number {
  printn(gcd(45,54));
  printn(gcd(125,15130));
  printn(gcd(11,121));
  return 0;
}

==> tests/test-hello.ts <==
function main() : number {
  printn(42);
  return 0;
}

==> tests/test-if-else-false.ts <==
function main() : number {
  if (false) printn(42); else printn(8);
  printn(17);
  return 0;
}

==> tests/test-if-else-true.ts <==
function main() : number {
```

```
  if (true) printn(42); else printn(8);
  printn(17);
  return 0;
}

==> tests/test-if-false.ts <==
function main() : number {
  if (false) printn(42);
  printn(17);
  return 0;
}

==> tests/test-if-true.ts <==
function main() : number {
  if (true) printn(42);
  printn(17);
  return 0;
}

==> tests/test-multiply-2-numbers.ts <==
function multiply(x : number, y : number) : number {
  return x * y;
}

function main() : number {
  printn(multiply(17, 25));
  return 0;
}


==> tests/test-multiply-3-numbers.ts <==
function multiply(x : number, y : number, z : number) : number {
  return x * y * z;
}

function main() : number {
  printn(multiply(17, 25, 32));
  return 0;
}

==> tests/test-rec-gcd.ts <==
function gcd(a: number, b: number): number {
```

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

function main(): number {
  printn(gcd(45, 54));
  printn(gcd(125, 15130));
  printn(gcd(11, 121));
  return 0;
}

==> tests/test-sub-2-numbers.ts <==
function subtract(x : number, y : number) : number {
  return x - y;
}

function main() : number {
  printn(subtract(25, 17));
  return 0;
}


==> tests/test-sub-3-numbers.ts <==
function subtract(x : number, y : number, z : number) : number {
  return x - y - z;
}

function main() : number {
  printn(subtract(17, 25, 32));
  return 0;
}


==> tests/test-while-dec.ts <==
function main() : number
{
```

```
  let i : number = 5;
  while (i > 0) {
    printn(i);
    i = i - 1;
  }
  printn(42);
  return 0;
}

==> tests/test-while-func.ts <==
function foo(a : number) : number
{
  let j : number = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

function main() : number
{
  printn(foo(7));
  return 0;
}
```