ezap Final Report

Ryan Lee

January 5, 2022

C-like language designed to bridge the gap between Java and procedural system programming in COMS 3157 by providing primitive socket-networking functionality with minimal overhead to explicitly manager resources utilized by strings and sockets

Professor Stephen Edwards
Columbia University
Fall 2021 Programming Languages and Translators

WHITE PAPER

1 Introduction

COMS 3157 serves as the bridge between object oriented introductory coursework and the world of system programming through the lens of C within Columbia's CS curriculum. The goal of *ezap* is to provide abstractions which while different then the syntax employed in Java allow for the *creation* of sockets in a manner that is more familiar than the purely procedural approach C takes which requires a working understanding of lower-level concepts such as memory management and accessing variables through pointers. In essence the goal of *ezap* is to motivate the course by allowing the students to code up constructs similar to the later deliverables in the course and excite them about learning the lower level language constructs that allow the larger deliverables to be feasible later on in the course.

2 Development Plan

The intent of motivating the language within the constructs of a course is to define narrowly the functionality I wish to achieve and build upon. Given the timeline of the project the codebase will be built on top of MicroC such that the entire time of the project is not spent frantically attempting to achieve feature parity with MicroC before preforming original, interesting work.

3 Language Features

The syntax of *ezap* while align closely with MicroC and will have a superset of MicroC's language features. The critical feature to the language is its context manager which is similar to the context manager found in Python.

3.1 DATA TYPES

Name	description	size	declaration
int	primitive integer	4-bytes	int x = 4;
float	primitive float	8-bytes	float f = 7.5;
bool	primitive boolean	1-byte	bool x = true;
string	string literal array of chars	dynamic	str s = "hello world";
socket	socket for primitive TCP networking	dynamic	socket s = ['c', 1300];
char	primitive ASII char	1-byte	char c = 'c';

3.2 KEYWORDS

Name	Usage
if	conditional test
else	alternative if nearest conditional test fails
while	loop that checks conditional before each iterative execution
for	loop that checks conditional execution before each iteration
return	return the evaluated expression to the caller

4 Goals

Given the compressed timeline I have narrowed my goals for the project.

1. Build in socket language features that allow for basic chatting
2. Build in socket language features that automatically allocate/dealloate the resources
    - A natural way to display this would be with the messaging programming such that upon receiving a termination "message" a control flow loop for listening is exited and resources are deallocated from the socket accordingly
3. If time permits building in a full on client/server example would naturally be desirable

5 Socket Data type

5.1 Socket Creation

```
socket S = [char mode, int port];
```

- creates a new socket
- handles "memsetting" and configuring low-level structs used in configuring the sockets
- allows to be configured as a **client** or **server**. This piece of metadeta will be tracked to make sure that calls for each socket are appropriate.
- handles binding to the specified port

5.2 Socket Connection

```
void connect(socket s, str address, int remoteport);
```

- handles configuring lower level structs about the remote address
- used only by "client" sockets server sockets are automatically setup to begin listing and accepting at the port passed in to the creation of a new socket
- in the case of failure the program will simply terminate

5.3 Sending/Receiving over Sockets

```
str receive(socket s);
```

- receive will simply block and return back to the caller an immutable, null-terminated, string literal that was received over the socket.
- the buffering implementation will not be visible to the ezap programmer

```
void send(socket s, str data);
```

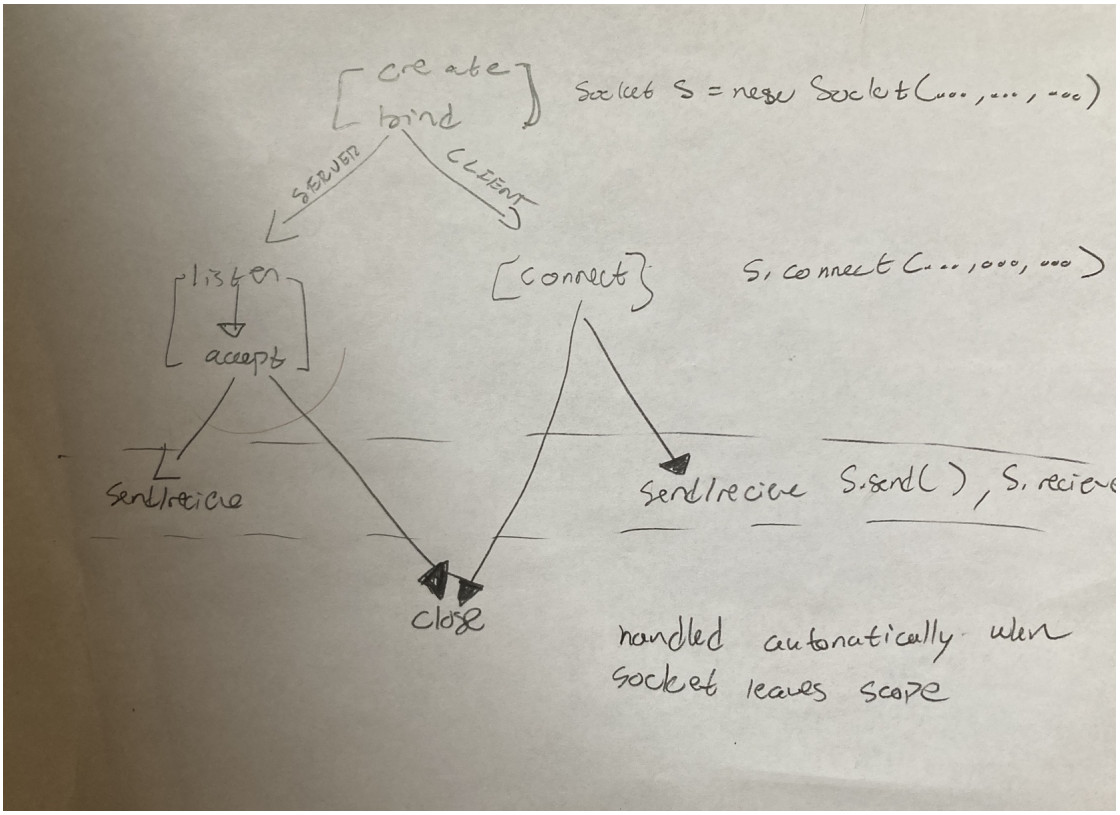
- a string will be sent over the socket and the function will return void. In the case of an error called by the underlying call to send() in the C standard library runtime-error messages will be printed out to the stdout.

5.4 Closure of Sockets

- Sockets will be closed as soon as they leave the scope in which they were created. All necessary dynamic memory cleanup will be done on behalf of the programmer and the socket will be closed without an explicit call to any sort of a "close" function as would be necessary when operating with true "C" sockets.

I have included a hand-written diagram for the constructs I have discussed below.

The end product reflects the right hand side of the diagram. The time was primarily spent on adding support for strings as well as the context manager instead of beefing up the standard library at the expense of these language features such that server side operations are supported. This highlights a somewhat low-hanging fruit for future opportunities to develop the language.



6 Operators

ezap will leverage all of the Micro C operators with the addition of an ==, @, +=, + operators for Strings.

7 Compiler Overview

At the *top-level* of the ezap directory type **make**. This will build the *./ezap.exe* compiler binary by leveraging *dune* in the *src* folder as well as an additional Makefile in the *stdlib* folder to compile and link in the standard library. The compiler executable will be symlinked to the top-level of the directory for easy access and the regression test suite will run. If you just wish to generate the compiler without running the regression test suite simply run **make compiler**.

The *ezap* compiler makes passes at a source file repeatedly before outputting an LLVM module to stdout. If you wish to compile a source file all the way down to a binary file leverage the **compiler.sh** shell script with usage:
./compiler.sh source.ez.

The tooling for the *ezap* compiler is nearly identical to MicroC and was developed inside of an Ubuntu 20.04.3 virtual machine instance. Notable differences include that the OCaml version is **4.08.1**, the OPam LLVM version is **13.0.0** as well as the system-level LLVM version. For straightforward installation instructions please refer to the README available on my Github in order to get started with the *ezap* language.

LANGUAGE REFERENCE MANUAL

8 Structure of the LRM

This LRM will seek to primarily highlight the additions/changes to the MicroC language instead of rehashing the syntax and semantics of the MicroC language except when it is illuminating to do so.

9 Lexical Conventions

9.1 Reserved Keywords

ezap has the following reserved keywords across its control flow and types:

```
if, else, for, while, int, float, bool, str, socket, char, return, as, with
```

ezap has the following built-in functions with reserved keywords

```
recv, send, printb, prints, print, printf, read, connect
```

9.2 Comments

Comments in ezap allow for multi-line comments and add in support for single-line comments. Single line comments are initiated with an opening // and are terminated upon reaching either newline or newline with a carriage return.

```
/* Multi-line comments can continue
over multiple lines */

//Single line comments can only span one line
```

9.3 Chars

ezap adds in support for ASCII chars. There are chars which are not printable but can still be used by preceding the char with a
. The following escape sequences are supported by ezap.

```
'\n' '\t' '\r' '\b' '\f' '\\'
```

An ezap char consists of exactly one ASCII char no more no less. Chars are preceded with and followed by an apostrophe.

9.4 Strings

Strings are a heap-allocated array of null-terminated ASCII characters. They support the same escape sequences as singular chars. Strings begin and end with a ". The String data type in ezap is strictly immutable. This means that the built in operations for Strings which would manipulate the underlying structure instead copy any would-be modification to a new buffer and return this new heap-allocated string to the user.

9.5 Socket

Sockets are stack-allocated structs with a file descriptor associated with the socket not visible to the user. A socket consists of a **char** which specifies the mode of the socket (either client or server) as well as an **int** which defines the port number the socket will be bound to when it is created. The syntax for defining a socket is:

9.6 Punctuation

ezap adds in the use of left and right "square" brackets during the definition of a struct literal these brackets are as follows [ ]

9.7 Operators

ezap adds in support for the following operators for the String data type. These operators are

`@, +, +=, ==`

10 Syntax

10.1 Program Structure

The program structure of an ezap program is identical to that of MicroC. It is notable that variables must be declared at the top-level of either a file or function before they are defined.

10.2 Functions

ezap programs require that a **main** function is defined. In addition to this mandatory function there is also requirement that each function which returns a non-void type actually return said non-void type. This is different than in C where the result is simply undefined behavior.

```
int main(){
    int i;
    i = 5;
    //syntax error must return an int
}
```

```
int main(){
    int i;
    i = 5;
    return i;
    //compiles -- main returns an int as it says it will in its definition
}
```

```
void main(){
    int i;
    i = 5;
    //compiles -- void functions do not have to have a return statement
}
```

11 Deep-Dive on String Operations

11.1 Charat

The charat operator takes in a string literal or an identifier which maps to a string as well as an integer which specifies the index of the string to return with the first char of a string being indexed from 0. The syntax for the operator is:

`string@int`

In the event that operators do not match the specifications an error is thrown at compile-time. It is not possible to deduce at compile time the value of the index on the RHS so we throw an error at run-time in the event that the index specified is out of the bounds of the length of the string passed on the LHS.

11.2 Equality

The equality operator takes in two strings and returns whether the lexicographical contents of the strings are identical (not their locations in memory). In the case that the strings are identical a **boolean** literal with value **true** is returned. In the event the strings are not equivalent a **boolean** literal equating to false is returned to the user. The syntax for the equality operator on strings is as follows.

```
str s1;
str s2;
bool cmp;

s1 = "ez";
s2 = "ap";
//will return false boolean literal
cmp = s1 == s2
```

11.3 Concatenation

The concatenation operator is +. Concatenation takes in two strings on either side of the operator. Neither of the underlying strings is modified due to the strict immutability of the string type discussed earlier. Instead a new heap-allocated string is returned back to the user that is the result of placing the string on the LHS first and then following it with the string on the RHS. The syntax for the concatenation operator is as follows.

```
str s1;
str s2;
str s3;

s1 = "hello ";
s2 = "world";

//s3 will be assigned "hello world"
s3 = s1 + s2;
```

11.4 Plus Assignment

Plus assignment leverages the concatenation routine and desugars to s1 = s1 + s2; in the event that s1 += s2; is present int he program and s1 and s2 are both strings. The operation returns a new heap allocated string and assigns it to s1. The syntax for the plus assignment operator is as follows.

```
str s1;
str s2;

s1 = "hello";
s2 = " world";

//the expression assigns "hello world" to s1
s1 += s2;
```

11.5 Precedence of String Operators

1. Plus assignment sits at the lowest level of precedence among the string operators and at the same level of regular assignment. It is a right-associative operator.
2. Equality is already an operator present in the grammar and so its precedence was not modified. It is a left associative operator.
3. Concatenation is already present in the grammar in the form of the "+" operator for ints and floats. Its precedence is unchanged and it is a left-associative operator.
4. Charat sits at the same level of precedence as "+" and "-" binary operators. It is a left associative operator.

12 Built-in Socket Functions

12.1 Connect

When a socket is specified via the form [**sock˙type, port˙number**] a file descriptor gets associated with the socket. In addition the socket is bound to the port number specified in **port˙number**. If the **port˙number** is invalid an exception will be thrown at run-time and execution will terminate. In addition there is the connect function specified by.

```
void connect(socket s, str address, int port);
```

In the event that the address or port number are invalid an exception is thrown at run-time. Otherwise a transient TCP connection is established between the socket s and the remote server at address, **address** listening on port, **port**.

12.2 Send

The send call takes a connected socket and attempts to send a string over the connection. The declaration of the send function is as follows:

```
void send(socket s, str data);
```

In the event that the socket is not connected or the send of the data is unsuccessful an error is thrown at run-time.

12.3 Recv

The recv call takes a connected socket and attempts to receive a string over the connection until either EOF is reached or the buffer overflows. The underlying buffer is of size 16k but only a string of the size of the length of the received string is returned to the user. If this is unsatisfactory it is suggested to pursue alternative methods for receiving from a connected socket. In the event a string is successfully received the string is returned to the user. The declaration of the receive call is as follows.

```
str recv(socket s);
```

13 Additional Built-in Functions

13.1 Print Functions

There are print functions for printing both strings and chars they are called prints and printc respectively and their syntax mirrors that of the print and printf functions. Their definions are as follows.

```
void prints(str s);
void printc(char c);
```

13.2 Read

Read reads stdin until either a newline or EOF is reached. The underlying buffer is of size 16K but only a string with size of the data read is allocated and returned to the user. The declaration of read is as follows:

```
str read();
```

14 Context Manager

The context manager is a statement in the ezap language which abstracts away resource management of strings and sockets. Context Managers can be nested within one another and contain any additional control flow statements inside of them. The syntax for a context manager is as follows:

```
str s;
socket sock;

with s as ("hello"){
   //body statements
}
with sock as (['c',1300]){
    //body statements
}
```

More generally context managers are specified by: **with expr1 as (expr2) stmt**. expr1 must evaluate to an **id** in order for the syntax of the context manager to be valid. expr2 must evaluate to either a **string** or a **socket**. Finally there is no casting between strings and sockets. The data type associated with the id of expr1 must match the data type expr2 evaluates to. When control moves past the context manager's scope in the event the type of expr2 is a string the heap allocation is freed and no leak will occur. In the event the type of expr2 is a socket the file descriptor associated with the socket will be closed for both reading and writing.

Context managers do not user-defined functions for cleaning up resources held by **expr1** and as a result only support string and socket data types.
Additionally the **id** which expr1 evaluates to cannot be reassigned in the case of both a socket and string. This mirrors the PEP 343 specification of Python's *with* statement.

15 Demo Program

A demo program is provided below which leveages several of the features of the ezap language described in this language reference manual. It offers an opportunity to highlight style for coding in ezap.
The one notable difference between writing standard C-style code and ezap code is the indentation of context manager statements. Nested context managers do not get indented further. It is only in the event of an expression or statement that is not a context manager that indentation continues. This allows the programmer to easily visualize the resources being managed by context managers and what scope they are bound to.

```
//a simple chat application that interfaces with netcat

int main(){
        //declarations
        socket s;
        str terminate;
        char c;
        bool b;
        int cmp;
        str inbound;
        str outbound;
        str addr;
        cmp = 1;

        with s as  (['c',1304]){
        with addr as ("127.0.0.1"){
        connect(s,addr, 1250);

        with terminate as ("end\n"){
        while(cmp == 1){
                with outbound as (read()){
                send(s, outbound);
                with inbound as (recv(s)){
                prints(inbound);
```

```
                b = ( inbound == terminate);
                if(b == false){
                        cmp = 1;
                }
                else{
                        cmp = 0;
                }
                }
                }

        }
        }
        }
        }
        return 0;
}
```

PROJECT PLAN

16 Project Plan

16.1 Planning/Development Process

Planning occurred during finals week of Fall 2021. I opted towards motivating the language with socket program and TA John Hui met in order to discuss the stages of development. John suggested developing support for strings first as way to become familiar with implementing additional features to the MicroC code base. After this I focused intently on developing the context manager for strings. Next the socket data type was created and I focused on implementing this structure and manipulating the underlying pointer to the socket using the OCaml LLVM bindings. Finally, I wrote the socket standard library in C.

16.2 Style Guide

The style of the code base is intended to mirror the conventions used in the MicroC code base.

16.3 Project Timeline/Log

```
commit bcd3b91ba3f13eb2714b3d74c7fe7f0f16f6e46d
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Jan 4 16:14:07 2022 -0800

    renaming test1.sh to compile.sh to be consistent with README

commit 41084098dfae08853b7cd1b7c9afca896d632079
Merge: bd1ea45 5d5d418
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Jan 4 16:11:55 2022 -0800

    Merge branch 'main' of github.com:RyanLee64/ez-AP into main

commit 5d5d418427dc73298013992ae67d69b615c7e142
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Tue Jan 4 18:09:35 2022 -0600

    Added using the ezap compiler section to README

commit c9ef45c7a964e75663b0c2cf7e84d6c988a67dce
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Tue Jan 4 18:01:46 2022 -0600

    Final version of README

commit 51bea30bf5fd7a04561e2b0047c23a447c292f97
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Tue Jan 4 17:31:46 2022 -0600

    Update README.md

commit 7b740041f2953d89bd99fefd2e818fb4fc12eec4
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Tue Jan 4 17:30:01 2022 -0600

    Update README.md

commit d1069d664710b8d880d63824ad88030183c1b863
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Tue Jan 4 17:29:42 2022 -0600

    Update README.md

commit bd1ea456345eedcea345320e0eeb7b983ae5b363
Merge: 315b455 c8e28da
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Jan 4 15:03:29 2022 -0800

    Merge branch 'main' of github.com:RyanLee64/ez-AP into main

commit c8e28dadf5bdb2644b329e99132c93f0a613a56b
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Tue Jan 4 17:01:01 2022 -0600

    updated testall.sh to access lli/llc
```

```
commit 315b455e01dd752dd224e45726ea4ab23b387687
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Jan 4 13:50:54 2022 -0800

    rename complete compilation shell script to match final report

commit 9ed0fe27c696e540a9bb930a773806f4599f3baf
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Tue Jan 4 14:20:14 2022 -0600

    top-level Makefile target name tweaks

commit 12e5cdad058cfac4f415f8b6aab4a04bd7156bb8
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 21:57:59 2022 -0800

    temporary commit of demo files for safekeeping -- will remove if project spec requires it

commit 99d8682c58408a42c2371e255927f96f68f2a80d
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 21:17:35 2022 -0800

    context manager fully compatible with closing sockets -- source is ready for demo

commit 44427cdf0b98c99ff2a3720079b24d2e484ddfc4
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 20:43:10 2022 -0800

    source code cleanup and working demo program ready on local machine for tomorrow -- semanti

commit 9df1c3d1e4ef167df617242f68d647b5f45593e1
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 19:48:38 2022 -0800

    reading from stdin working

commit 85f6c46a025f9a992cfb7c848779eb627f584c93
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 18:12:55 2022 -0800

    recv working for sockets

commit e370b9b6cc7c29045b1a9f7fa29484cac85b6a09
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 18:08:43 2022 -0800

    adding support for non-void return types from built-in functions

commit 06175579f99366624c22790a7df1182a42d9018e
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 17:21:50 2022 -0800

    send working for sockets

commit acc8fc1f1e549c3e26222e6fbc87f4f6e427b21a
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 16:48:31 2022 -0800

    connect up and running for sockets
```

```
commit 9e9175cd1474a4a31333cbb275a1a5fda30cfe63
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 14:01:16 2022 -0800

    bind up and running for sockets

commit 91247aefe30fbf39c8ecc4d933dfa67502039630
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 13:42:01 2022 -0800

    creation of a socket file descriptor up and running

commit 2ea28d9ad1b360bd0a0212d506b88fc023c4fcad
Merge: 923b0d6 2f93260
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 11:34:31 2022 -0800

    Merge branch 'main' of github.com:RyanLee64/ez-AP into main

commit 923b0d6af9cff0f3c2190066565961333ea48151
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Jan 3 11:34:20 2022 -0800

    bare socket files added to stdlib/makefile

commit 2f93260f91a595396f2882d76e53afc6c63ac5e7
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Mon Jan 3 11:32:26 2022 -0800

    Delete font2c

commit aa96423567ae0f7d0799ab556b3a3bd0e5344a73
Author: Ryan Lee <65369992+RyanLee64@users.noreply.github.com>
Date:   Mon Jan 3 11:32:14 2022 -0800

    Delete llvm.sh

commit 1c034a0aeb8186753112a6bae9c04588e8ccc606
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Jan 2 20:47:08 2022 -0800

    socket struct type is generated in LLVM IR and members are being correctly set to evaluated

commit 3a87961f56e23e0411452fda4a12989b6ad472c8
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Jan 2 13:15:23 2022 -0800

    socket type declaration and empty assignment. Need to mess with gep instruction to fill out

commit 46dd4a5dd021a3f68ffedc0c268b30c526c88de4
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Jan 1 20:30:09 2022 -0800

    added support for char literls that are not generated from a function i.e. of the form char

commit c229976084e3076ccb778478bbb51dde06162563
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Jan 1 14:20:58 2022 -0800
```

```
    just moved test files around -- test also shows charat working with new char data type

commit 99e6b4e9de10a1f57f1ac7bd80765f5b37e1cc33
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Jan 1 14:09:32 2022 -0800

    test case for nested conext managers

commit 4c1766a80f991ecc23fc7792d3a32e3ece73ecfb
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Jan 1 14:05:05 2022 -0800

    added support for nested context managers

commit 38097097bb366c7ad12ecb9ec93c7294d4c79218
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Jan 1 12:11:36 2022 -0800

    char data type --> going to switch charat operator to return a char now

commit ba6ae1b878cab24c0e361000124dce5b00ecdcec
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Fri Dec 31 14:35:01 2021 -0800

    failure tests for context manager semantic checks 3rd needed for sockets once created

commit affca917e5a158d0960290c020f8cb46d5576471
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Fri Dec 31 14:33:32 2021 -0800

    semantic checking for context manager just need to add socket data type once created

commit 703695c0f2b81352fd2663ce405bcfec53ff4451
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Thu Dec 30 19:17:23 2021 -0800

    first context manager test

commit 0d8b934e1db3744434b6ed6c23b7e8c51afdfb68
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Thu Dec 30 19:16:18 2021 -0800

    context manager working for strings and correctly freeing heap allocations

commit 620e7ee33a175dac8d807c1110ab32c3fdd514ed
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Wed Dec 29 19:52:31 2021 -0800

    cleaning up partial leak with +=

commit 6f6d7f21f234ca889417ddaaf3eb544e795e045d
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Wed Dec 29 14:24:45 2021 -0800

    seperate header file for stringops

commit cd03f369878ed1c05d50f2bf40cd3cc0084c7c2d
Author: Ryan Lee <dbl2127@columbia.edu>
```

```
Date:   Wed Dec 29 14:18:30 2021 -0800

    string equality no additional heap allocation necessary

commit ddc945bbfc35c853118d558a8dcd9fd13e560345
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Wed Dec 29 12:57:00 2021 -0800

    charat tests

commit e90cb4ea194eef71c7370d57f744ca451b37c915
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Dec 28 13:59:49 2021 -0800

    charat operator  with heap allocation/leak

commit 3bb646944e7ff329733dc8f762695f496666be12
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Dec 28 11:51:38 2021 -0800

    += opeartor for strings tested and working with leaks

commit d3bf1a073a54dbb8fe23b9ea8651037c92a78347
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Dec 28 08:50:38 2021 -0800

    Makefile tweak to remove ezap.exe on new  call

commit cb928a425d4a789f4af000d2fdcba28937b1dfdf
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Tue Dec 28 08:50:08 2021 -0800

    test cases for new return semantics

commit fa436935e0be134bd0fa673698ade0918f625cb0
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Mon Dec 27 11:18:31 2021 -0800

    added semantic check to make sure return types match up and that a return statement is pres

commit 4a540bcc40d0e43a5ce85bc3b7e312109eed54f8
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Dec 25 21:27:44 2021 -0800

    removing printbig.c from top level directory

commit f73057493ba23ba34442108e21413bf74fccb58d
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Dec 25 21:04:10 2021 -0800

    renamed .mc tests to .ez and added todays features to regression test suite

commit 67b3dfd4309e6d2769194c9dee5aa32c594b057f
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Dec 25 20:45:05 2021 -0800

    working hello world with + as strcat operator

commit 1be2902102f3b8bc72f57315aaed240dec135d9b
```

```
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Dec 25 19:54:31 2021 -0800

    working heap allocate strings with printing down to .exe with some makefile cleanup

commit c772c009aa5d04445006ec27b2d2193109c41f9c
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sat Dec 25 07:32:58 2021 -0800

    working str assignment

commit e8fd39f825f01cc620c8013e1a2958ca8b8e3f1c
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Fri Dec 24 09:39:36 2021 -0800

    end to end working microc fusion with scanner

commit 23241b957d3abd295419bd75236ed054ed9e8304
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 19 16:05:32 2021 -0800

    parser added just for setup

commit bac9b57ff5b8096dbd3ded86b45bae00b7c674de
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 19 16:05:12 2021 -0800

    initial dunefile

commit c40bfa94c1cadc38dd0d34ee74fba5e932c69f53
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 19 16:05:02 2021 -0800

    started on scanner

commit 5f938426de3fe930bec2aae7d9b0373e2b1c1f4d
Author: Ryan Lee <dbl2127@columbia.edu>
Date:   Sun Dec 19 15:00:10 2021 -0800

    first commit
```

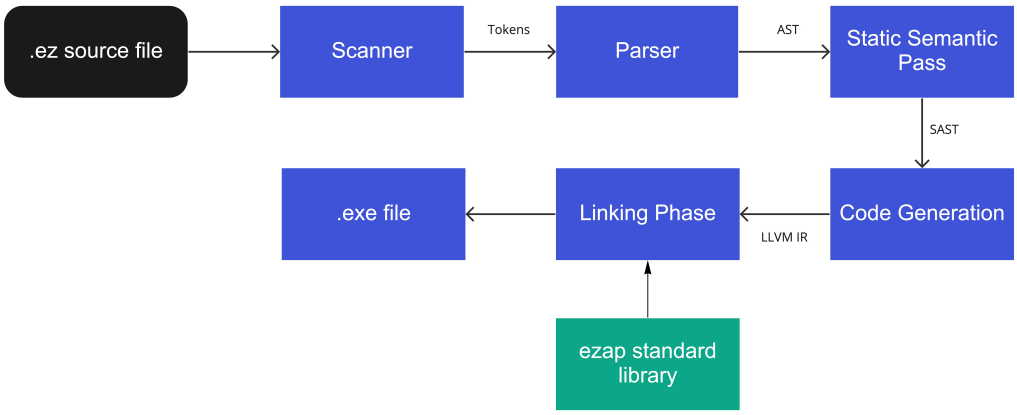
16.4 Roles/Environment

The development environment was hosted on an Ubuntu VM run through VMWare on a Intel based Macbook Pro. The editor of choice was VSCode. Within VSCode I made use of the OCaml and Reason IDE plugin which interfaces with Merlin as well as OCaml's LSP server. Makefile and Dune were utilized to build the code base. Finally, valgrind was used to check for leaking memory. Git and Github were used for version control and source code hosting respectively. The code base is written in OCaml for the compiler, C for the standard library, Makefile/Dune for build instructions and Shell script for automating compilation of source code/running the test suite.

Naturally I fulfilled all of the responsibilities with the help and guidance of TA John Hui to orient my language towards some attainable goals.

ARCHITECTURAL DESIGN

17 Block Diagram



18 Interface

ezap.ml sequentially calls the phases of the block diagram described above. It calls the lexer to generate a stream of tokens from the source file as input. This list is passed to the grammar output by ocamlyacc which creates an AST tree corresponding to the syntax recognized by the stream of tokens via an LR(1) parsing algorithm. This AST is then passed to the Static Semantic checker which associates a type with each expression. This "SAST" is passed as a list of globals and functions to the codegen.ml file which generates an LLVM module using the OCaml MOE bindings. When using compiler.sh this is passed to llc which generates assembly code for the architecture of the computer the code is running on. Finally this assembly file is run through the system's c compiler and linked with the standard library for the ezap language. The final output is a binary executable.

TESTING PLAN

19 Representative Programs

19.1 Simple Chat

SOURCE

```
//simple chat application that interfaces with netcat
int main(){
        //declarations
        socket s;
        str terminate;
        char c;
        bool b;
        int cmp;
        str inbound;
        str outbound;
        str addr;

        cmp = 1;

        with s as  (['c',1304]){
        with addr as ("127.0.0.1"){

        connect(s,addr, 1250);

        with terminate as ("end\n"){
        while(cmp == 1){
                with outbound as (read()){
                send(s, outbound);
                with inbound as (recv(s)){
                prints(inbound);
                b = ( inbound == terminate);
                if(b == false){
                        cmp = 1;
                }
                else{
                        cmp = 0;
                }
                }
                }

        }
        }
        }
        }
        return 0;
}
```

LLVM IR

```
; ModuleID = 'MicroC'
source_filename = "MicroC"

%sock_struct = type { i8, i32, i32 }

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

```
@temp_assign_ptr = private unnamed_addr constant [10 x i8] c"127.0.0.1\00", align 1
@temp_assign_ptr.4 = private unnamed_addr constant [5 x i8] c"end\0A\00", align 1

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

declare i32 @printbig(i32)

declare i8* @createstr(i8*)

declare i8* @concatstrs(i8*, i8*)

declare i8 @charatstr(i8*, i32)

declare i1 @checkstreq(i8*, i8*)

declare void @ez_create(%sock_struct*)

declare void @ez_connect(%sock_struct*, i8*, i32)

declare void @ez_close(%sock_struct*)

declare void @ez_send(%sock_struct*, i8*)

declare i8* @ez_recv(%sock_struct*)

declare void @writestr(i8*)

declare i8* @readstr()

define i32 @main() {
entry:
  %s = alloca %sock_struct*, align 8
  %terminate = alloca i8*, align 8
  %c = alloca i8, align 1
  %b = alloca i1, align 1
  %cmp = alloca i32, align 4
  %inbound = alloca i8*, align 8
  %outbound = alloca i8*, align 8
  %addr = alloca i8*, align 8
  store i32 1, i32* %cmp, align 4
  %socket = alloca %sock_struct, align 8
  %conn_ptr = getelementptr %sock_struct, %sock_struct* %socket, i32 0, i32 0
  %port_ptr = getelementptr %sock_struct, %sock_struct* %socket, i32 0, i32 1
  %file_descrip = getelementptr %sock_struct, %sock_struct* %socket, i32 0, i32 2
  store i8 99, i8* %conn_ptr, align 1
  store i32 1304, i32* %port_ptr, align 4
  store i32 0, i32* %file_descrip, align 4
  call void @ez_create(%sock_struct* %socket)
  %contextptr = alloca %sock_struct*, align 8
  store %sock_struct* %socket, %sock_struct** %contextptr, align 8
  store %sock_struct* %socket, %sock_struct** %s, align 8
  br label %body

clean:                                            ; preds = %clean1
  %cleanup_load = load %sock_struct*, %sock_struct** %contextptr, align 8
  call void @ez_close(%sock_struct* %cleanup_load)
  ret i32 0

body:                                             ; preds = %entry
```

```
  %strlit = call i8* @createstr(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @temp_assign_
  %contextptr3 = alloca i8*, align 8
  store i8* %strlit, i8** %contextptr3, align 8
  store i8* %strlit, i8** %addr, align 8
  br label %body2

clean1:                                           ; preds = %clean7
  %cleanup_load4 = load i8*, i8** %contextptr3, align 8
  tail call void @free(i8* %cleanup_load4)
  br label %clean

body2:                                            ; preds = %body
  %addr5 = load i8*, i8** %addr, align 8
  %s6 = load %sock_struct*, %sock_struct** %s, align 8
  call void @ez_connect(%sock_struct* %s6, i8* %addr5, i32 1250)
  %strlit9 = call i8* @createstr(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @temp_assign_p
  %contextptr10 = alloca i8*, align 8
  store i8* %strlit9, i8** %contextptr10, align 8
  store i8* %strlit9, i8** %terminate, align 8
  br label %body8

clean7:                                           ; preds = %merge29
  %cleanup_load11 = load i8*, i8** %contextptr10, align 8
  tail call void @free(i8* %cleanup_load11)
  br label %clean1

body8:                                            ; preds = %body2
  br label %while

while:                                            ; preds = %clean12, %body8
  %cmp27 = load i32, i32* %cmp, align 4
  %tmp28 = icmp eq i32 %cmp27, 1
  br i1 %tmp28, label %while_body, label %merge29

while_body:                                       ; preds = %while
  %readstr = call i8* @readstr()
  %contextptr14 = alloca i8*, align 8
  store i8* %readstr, i8** %contextptr14, align 8
  store i8* %readstr, i8** %outbound, align 8
  br label %body13

clean12:                                          ; preds = %clean18
  %cleanup_load15 = load i8*, i8** %contextptr14, align 8
  tail call void @free(i8* %cleanup_load15)
  br label %while

body13:                                           ; preds = %while_body
  %outbound16 = load i8*, i8** %outbound, align 8
  %s17 = load %sock_struct*, %sock_struct** %s, align 8
  call void @ez_send(%sock_struct* %s17, i8* %outbound16)
  %s20 = load %sock_struct*, %sock_struct** %s, align 8
  %recvd_data = call i8* @ez_recv(%sock_struct* %s20)
  %contextptr21 = alloca i8*, align 8
  store i8* %recvd_data, i8** %contextptr21, align 8
  store i8* %recvd_data, i8** %inbound, align 8
  br label %body19

clean18:                                          ; preds = %merge
  %cleanup_load22 = load i8*, i8** %contextptr21, align 8
```

```
  tail call void @free(i8* %cleanup_load22)
  br label %clean12

body19:                                           ; preds = %body13
  %inbound23 = load i8*, i8** %inbound, align 8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.2,
  %inbound24 = load i8*, i8** %inbound, align 8
  %terminate25 = load i8*, i8** %terminate, align 8
  %equality = call i1 @checkstreq(i8* %inbound24, i8* %terminate25)
  store i1 %equality, i1* %b, align 1
  %b26 = load i1, i1* %b, align 1
  %tmp = icmp eq i1 %b26, false
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  br label %clean18

then:                                             ; preds = %body19
  store i32 1, i32* %cmp, align 4
  br label %merge

else:                                             ; preds = %body19
  store i32 0, i32* %cmp, align 4
  br label %merge

merge29:                                          ; preds = %while
  br label %clean7
}

declare void @free(i8*)
```

19.2 SIMPLE BROWSER

```
int main(){
        //declarations
        socket s;
        str terminate;
        str url;
        str request;
        char c;
        str inbound;

        with url as (read()){
        with request as ("GET /index.html HTTP/1.0\n\n"){
        with s as  (['c',1305]){

                connect(s, url, 80);


                send(s, request);
                with inbound as (recv(s)){

                        prints(inbound);

        }
        }
        }
        }
```

```
        return 0;
}
```

LLVM IR

```
; ModuleID = 'MicroC'
source_filename = "MicroC"

%sock_struct = type { i8, i32, i32 }

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@fmt.3 = private unnamed_addr constant [4 x i8] c"%c\0A\00", align 1
@temp_assign_ptr = private unnamed_addr constant [27 x i8] c"GET /index.html HTTP/1.0\0A\0A\00"

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

declare i32 @printbig(i32)

declare i8* @createstr(i8*)

declare i8* @concatstrs(i8*, i8*)

declare i8 @charatstr(i8*, i32)

declare i1 @checkstreq(i8*, i8*)

declare void @ez_create(%sock_struct*)

declare void @ez_connect(%sock_struct*, i8*, i32)

declare void @ez_close(%sock_struct*)

declare void @ez_send(%sock_struct*, i8*)

declare i8* @ez_recv(%sock_struct*)

declare void @writestr(i8*)

declare i8* @readstr()

define i32 @main() {
entry:
  %s = alloca %sock_struct*, align 8
  %terminate = alloca i8*, align 8
  %url = alloca i8*, align 8
  %request = alloca i8*, align 8
  %c = alloca i8, align 1
  %inbound = alloca i8*, align 8
  %readstr = call i8* @readstr()
  %contextptr = alloca i8*, align 8
  store i8* %readstr, i8** %contextptr, align 8
  store i8* %readstr, i8** %url, align 8
  br label %body

clean:                                            ; preds = %clean1
  %cleanup_load = load i8*, i8** %contextptr, align 8
  tail call void @free(i8* %cleanup_load)
```

```
  ret i32 0

body:                                             ; preds = %entry
  %strlit = call i8* @createstr(i8* getelementptr inbounds ([27 x i8], [27 x i8]* @temp_assign_
  %contextptr3 = alloca i8*, align 8
  store i8* %strlit, i8** %contextptr3, align 8
  store i8* %strlit, i8** %request, align 8
  br label %body2

clean1:                                           ; preds = %clean5
  %cleanup_load4 = load i8*, i8** %contextptr3, align 8
  tail call void @free(i8* %cleanup_load4)
  br label %clean

body2:                                            ; preds = %body
  %socket = alloca %sock_struct, align 8
  %conn_ptr = getelementptr %sock_struct, %sock_struct* %socket, i32 0, i32 0
  %port_ptr = getelementptr %sock_struct, %sock_struct* %socket, i32 0, i32 1
  %file_descrip = getelementptr %sock_struct, %sock_struct* %socket, i32 0, i32 2
  store i8 99, i8* %conn_ptr, align 1
  store i32 1305, i32* %port_ptr, align 4
  store i32 0, i32* %file_descrip, align 4
  call void @ez_create(%sock_struct* %socket)
  %contextptr7 = alloca %sock_struct*, align 8
  store %sock_struct* %socket, %sock_struct** %contextptr7, align 8
  store %sock_struct* %socket, %sock_struct** %s, align 8
  br label %body6

clean5:                                           ; preds = %clean13
  %cleanup_load8 = load %sock_struct*, %sock_struct** %contextptr7, align 8
  call void @ez_close(%sock_struct* %cleanup_load8)
  br label %clean1

body6:                                            ; preds = %body2
  %url9 = load i8*, i8** %url, align 8
  %s10 = load %sock_struct*, %sock_struct** %s, align 8
  call void @ez_connect(%sock_struct* %s10, i8* %url9, i32 80)
  %request11 = load i8*, i8** %request, align 8
  %s12 = load %sock_struct*, %sock_struct** %s, align 8
  call void @ez_send(%sock_struct* %s12, i8* %request11)
  %s15 = load %sock_struct*, %sock_struct** %s, align 8
  %recvd_data = call i8* @ez_recv(%sock_struct* %s15)
  %contextptr16 = alloca i8*, align 8
  store i8* %recvd_data, i8** %contextptr16, align 8
  store i8* %recvd_data, i8** %inbound, align 8
  br label %body14

clean13:                                          ; preds = %body14
  %cleanup_load17 = load i8*, i8** %contextptr16, align 8
  tail call void @free(i8* %cleanup_load17)
  br label %clean5

body14:                                           ; preds = %body6
  %inbound18 = load i8*, i8** %inbound, align 8
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.2,
  br label %clean13
}

declare void @free(i8*)
```

20 Test Suites

The testing process consisted of adding one fail and one successful test for each feature implemented into the language. In addition larger more complex source code was passed to the language which was designed to interweave different language features in order to make sure no undesirable side effects were introduced.
The testall.sh script was modified and used for the purposes of the project. In addition the compiler.sh script was used to test individual language features as they were built up.

LESSONS LEARNED

21 Advice and Such

- Above all else communication is paramount in a semester long group project. This is a different flavor than something like a "lab" or semester long project with tight specifications broken into very small chunks. This affords you the freedom to design a language with all the features **you** want but also naturally the flexibility and only having one firm due date/deliverable introduces opportunities to err farther off course than would be possible during smaller projects.
- If you are unfamiliar with OCaml/functional programming in general like I was I would suggest this playlist.
- Read up on LLVM IR/Static Single Assignment ahead of lecture. This is the target representation of your compiler so you are going to want to be very familiar with and start visualizing how to implementing your language constructs within this IR.
- Ask for help as early as you think you even *might* need it. The teaching staff is there to support you and **you** can do this so long as you leverage the support system in place and work diligently.

SOURCE CODE APPENDIX

Listing 1: scanner.mll

```
(* Scanner for ezap language*)
{
open Parser
exception SyntaxError of string
}

let digit = ['0' - '9']
let digits = digit+
let whitespace_chars = [' ' '\t']
let newline = '\r' | '\n' | "\r\n"
let whitespace = whitespace_chars | newline
rule token = parse
    whitespace {token lexbuf} (*eat whitespace*)
(*comments/strings*)
| "//"      {s_comment lexbuf}
| "/*"      {mult_comment lexbuf}
| '"'       {read_string (Buffer.create 17) lexbuf}

| "socket" {SOCKET}

| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LSQUARE }
| ']'       { RSQUARE }
| '''       { read_first_char (Buffer.create 1) lexbuf }
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| "+="      { ADDASSIGN }
| "@"       { CAT }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| "!"       { NOT }
| "str"     { STRING }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "return"  { RETURN }
| "int"     { INT }
| "char"    { CHAR }
| "bool"    { BOOL }
| "float"   { FLOAT }
| "void"    { VOID }
| "true"    { BLIT(true)  }
```

```
| "false"  { BLIT(false) }
| "with"   { WITH }
| "as"     { AS }

| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

and s_comment = parse
    "\n"    {token lexbuf}
|   "\r\n"  {token lexbuf} (*windows support*)
|   _       {s_comment lexbuf}



and read_string buf =
  parse
  | '"'       { STRLIT (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")) }

and read_first_char buf =
  parse
   '''       {raise (SyntaxError ("No char provided"))} (* char is EXACTLY one char no more no
  | '\\' '/'  { Buffer.add_char buf '/'; read_apostrophe buf lexbuf }
  | '\\' '\\' { Buffer.add_char buf '\\'; read_apostrophe buf lexbuf }
  | '\\' 'b'  { Buffer.add_char buf '\b'; read_apostrophe buf lexbuf }
  | '\\' 'f'  { Buffer.add_char buf '\012';read_apostrophe buf lexbuf }
  | '\\' 'n'  { Buffer.add_char buf '\n'; read_apostrophe buf lexbuf }
  | '\\' 'r'  { Buffer.add_char buf '\r'; read_apostrophe buf lexbuf }
  | '\\' 't'  { Buffer.add_char buf '\t'; read_apostrophe buf lexbuf }
  | [^ '\\']  { Buffer.add_char buf (Lexing.lexeme_char lexbuf 0);
                read_apostrophe buf lexbuf }
  | eof       { raise (SyntaxError ("Char is not terminated"))}

and read_apostrophe buf =
  parse
    '''       {CHARLIT (Buffer.nth buf 0)}
  | _         {raise (SyntaxError ("Extraneous extra chars provided to a char literal"))}
```

Listing 2: ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

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

type uop = Neg | Not

type typ = Int | Bool | Float | Void | String | Char | Socket

type bind = typ * string

type expr =
    Literal of int
  | Fliteral of string
  | StrLiteral of string
  | BoolLit of bool
  | CharLiteral of char
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | PAssign of string * expr
  | Sock of  expr * expr
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Context of expr * expr * stmt


type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)

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

```
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"
  | Charat -> "@"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Fliteral(l) -> l
  | StrLiteral(s) -> s
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | PAssign(v, e) -> v ^ " += " ^ string_of_expr e
  | CharLiteral(c) ->  String.make 1 c
  | Sock(e1, e2) -> string_of_expr e1 ^ "type socket. With desired port number
  : " ^ string_of_expr e2

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\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
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Context(e1,e2,s) ->
    "With "^ string_of_expr e1 ^ "as "^string_of_expr e2 ^ string_of_stmt s



let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | String -> "string"
  | Char  ->  "char"
  | Socket -> "socket"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
```

```
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

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

Listing 3: parser.mly

```
/* Ocamlyacc parser for ezap*/

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR ADDASSIGN CAT APOSTROPHE
%token RETURN IF ELSE FOR WHILE INT WITH AS LSQUARE RSQUARE SOCKET
/*TYPES*/
%token BOOL FLOAT VOID STRING CHAR
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT STRLIT
%token <char> CHARLIT
%token EOF

%start program
%type <Ast.program> program
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN ADDASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS CAT
%left TIMES DIVIDE
%right NOT

%%

program:
  decls EOF { $1 }

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

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

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

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

typ:
```

```
    INT   { Int    }
  | BOOL  { Bool   }
  | FLOAT { Float  }
  | VOID  { Void   }
  | STRING{ String }
  | CHAR  { Char   }
  | SOCKET{ Socket }
  /*| SOCKET{ Sock   }*/


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

vdecl:
   typ ID SEMI { ($1, $2) }

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

stmt:
    expr SEMI                                  { Expr $1               }
  | RETURN expr_opt SEMI                       { Return $2             }
  | LBRACE stmt_list RBRACE                    { Block(List.rev $2)    }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt     { If($3, $5, $7)        }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                             { For($3, $5, $7, $9)   }
  | WHILE LPAREN expr RPAREN stmt            { While($3, $5)         }
  | WITH expr AS LPAREN expr RPAREN stmt     { Context($2, $5, $7)   }



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

expr:
    LITERAL          { Literal($1)            }
  | FLIT                  { Fliteral($1)           }
  | BLIT             { BoolLit($1)            }
  | ID               { Id($1)                 }
  | CHARLIT          { CharLiteral($1)        }
  | STRLIT           { StrLiteral ($1)        }
  | expr CAT    expr { Binop($1, Charat,$3)   }
  | 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 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 AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | MINUS expr %prec NOT { Unop(Neg, $2)      }
```

```
  | NOT expr          { Unop(Not, $2)          }
  | ID ASSIGN expr    { Assign($1, $3)         }
  | ID LPAREN args_opt RPAREN { Call($1, $3)   }
  | LPAREN expr RPAREN { $2                    }
  | ID ADDASSIGN expr { PAssign($1, $3)        }
  | LSQUARE expr COMMA expr RSQUARE { Sock($2, $4)    }




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

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

Listing 4: semant.ml

```
(* Semantic checking for the ezap compiler *)

open Ast
open Sast
open String
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_binds (kind : string) (binds : bind list) =
    List.iter (function
        (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) binds;
    let rec dups = function
        [] -> ()
      | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
          raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in

  (**** Check global variables ****)

  check_binds "global" globals;

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

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =
    let add_bind map (name, ty) =
      let formals =
        (*janky pattern match will fix time permitting*)
        match name with
        "createstr" | "charat" -> [(String,"x1");(String,"x2")]
        |"connect"-> [(Socket,"x1");(String,"x2");(Int,"x3")]
        |"send"   -> [(Socket,"x1");(String,"x2")]
        |"recv"   -> [(Socket, "x1")]
        |"read"   -> [ ]
        |_          -> [(ty,"x")] in
      StringMap.add name {
      typ =
      (match name with
      "createstr" | "read" | "recv" -> String
      |"charat" ->   Char
      |"checkstreq" -> Bool
      | _ -> Void);
      fname = name;
      formals = formals;
      locals = []; body = [] } map
    in List.fold_left add_bind StringMap.empty [ ("print", Int);
```

```
                                        ("printb", Bool);
                                        ("prints", String);
                                        ("printf", Float);
                                        ("printc", Char);
                                        ("printbig", Int);
                                        ("createstr", String);
                                        ("charat",   Char);
                                        ("checkstreq", Bool);
                                        ("connect", Void);
                                        ("ez_create", Void);
                                        ("send", Void);
                                        ("recv", String);
                                        ("read", String);
                                        ("close", Socket);
                                        ("write", String)]
                                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 -> raise (Failure ("unrecognized function " ^ s))
in

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




let check_function func =
  (* Make sure no formals or locals are void or duplicates *)
  check_binds "formal" func.formals;
  check_binds "local" func.locals;

  (* 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 symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
                    StringMap.empty (globals @ func.formals @ 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
    Literal  l -> (Int, SLiteral l)
  | Fliteral l -> (Float, SFliteral l)
  | BoolLit l  -> (Bool, SBoolLit l)
  | StrLiteral s ->(String, SStrLiteral s)
  | Noexpr     -> (Void, SNoexpr)
  | Id s       -> (type_of_identifier s, SId s)
  | CharLiteral c -> (Char, SCharLiteral c)
  | Sock(e1, e2) ->
    let (t1, _) = expr e1 and
    (t2, _) = expr e2 in
    if(t1=Char && t2=Int) then (Socket, SSock(expr e1,expr e2))
    else(raise(Failure("Invalid socket " ^ "["^string_of_typ t1^", "^
    string_of_typ t2^"]" ^ "Should be: [Char, Int]")))
  | Assign(var, e) | PAssign(var,e) as ex->
      let lt = type_of_identifier var
      and (rt, e') = expr e in
      let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
        string_of_typ rt ^ " in " ^ string_of_expr ex
      in (check_assign lt rt err,
      match ex with
        PAssign(_, _) -> SPAssign(var, (rt, e'))
        |Assign(_,_) -> SAssign(var, (rt, e'))
        | _ -> raise (Failure ("bad assignment"))
      )
  | Unop(op, e) as ex ->
      let (t, e') = expr e in
      let ty = match op with
        Neg when t = Int || t = Float -> t
      | Not when t = Bool -> Bool
      | _ -> raise (Failure ("illegal unary operator " ^
                             string_of_uop op ^ string_of_typ t ^
                             " in " ^ string_of_expr ex))
      in (ty, SUnop(op, (t, e')))
  | 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 = Int   -> Int
      | Add | Sub | Mult | Div when same && t1 = Float -> Float
      | Add                    when same && t1 = String -> String
      | Equal                  when same && t1 = String -> Bool
      | Equal | Neq            when same               -> Bool
      | Less | Leq | Greater | Geq
                 when same && (t1 = Int || t1 = Float) -> Bool
      | And | Or when same && t1 = Bool -> Bool
      (*charat has a strlit/id on LHS and an int on RHS *)
```

```
      | Charat   when t1 = String && t2 = Int  -> Char
      | _ -> 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')))
  | 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'))
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
let check_return (ret_type: typ) (body_statements: stmt list) =
  match ret_type with
    Void ->  () (*no return statement needed for void function *)
   | _ ->
    let return_statement = List.hd (List.rev body_statements) in
    match return_statement with
     Return e ->
      let evaluated = expr e in
      if fst evaluated != ret_type then raise(Failure("return type does not match")) else (
    | _ -> raise (Failure ("non-void function without return statement"))
     in
check_return func.typ func.body;
(* Return a semantically-checked statement i.e. containing sexprs *)
let rec check_stmt = function
    Expr e -> SExpr (expr e)
  | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
  | For(e1, e2, e3, st) ->
      SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
  | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
  (*context semantic check*)
  | Context(e1,e2, s) ->
    let (t1, e1') = expr e1
    and (t2, _) = expr e2 in
    let same = t1 = t2 in
      (match (t1,e1') with
      (String, (SId _) ) | (Socket, (SId _) ) ->
        (match (t2) with
        (*TODO ADD SOCKET ONCE SOCKET IS BUILT OUT*)
          String | Socket ->  if same then SContext(expr e1, expr e2, check_stmt s) else
            raise(Failure "type of resource and variable do not match")
          |_-> raise(Failure "resource expression must evaluate to a socket or str"))
        |_-> raise(Failure "must assign expression to an id of type socket or str"))
```

```
      | 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))
            (* 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)

    in (* body of check_function *)
    { styp = func.typ;
      sfname = func.fname;
      sformals = func.formals;
      slocals  = func.locals;
      sbody = match check_stmt (Block func.body) with
        SBlock(sl) -> sl
      | _ -> raise (Failure ("internal error: block didn't become a block?"))
    }
  in (globals, List.map check_function functions)
```

Listing 5: codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)

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 "ezap" in

  let sock_t = L.named_struct_type context "sock_struct" in
  let sock_t_ptr = L.pointer_type sock_t 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 str_t      = L.pointer_type (L.i8_type context)
  and void_t     = L.void_type   context in

  (* Return the LLVM type for a ezap type *)
  let ltype_of_typ = function
      A.Int   -> i32_t
    | A.Bool  -> i1_t
    | A.Float -> float_t
    | A.Void  -> void_t
    | A.String -> str_t
    | A.Char  -> i8_t
    | A.Socket -> sock_t_ptr
  in

  (*fill out the body of our socket struct type*)
  ignore(L.struct_set_body sock_t [|i8_t; i32_t; i32_t|] false);


  (* Create a map of global variables after creating each *)
  let global_vars : L.llvalue StringMap.t =
```

```
  let global_var m (t, n) =
    let init = match t with
        A.Float -> 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

(* External function declarations *)
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

let printbig_t : L.lltype =
    L.function_type i32_t [| i32_t |] in
let printbig_func : L.llvalue =
    L.declare_function "printbig" printbig_t the_module in

let createstr_t: L.lltype =
    L.function_type str_t [| str_t |] in
let createstr_func: L.llvalue  =
    L.declare_function "createstr" createstr_t the_module in

let add_strs_t: L.lltype =
  L.function_type str_t [|str_t; str_t|] in
let add_strs_func: L.llvalue =
  L.declare_function "concatstrs" add_strs_t the_module in

let char_at_t: L.lltype =
  L.function_type i8_t [|str_t; i32_t|] in
let char_at_func: L.llvalue =
  L.declare_function "charatstr" char_at_t the_module in

let check_str_eq_t: L.lltype =
  L.function_type i1_t [|str_t; str_t|] in
let check_str_func: L.llvalue =
  L.declare_function "checkstreq" check_str_eq_t the_module in

let create_t: L.lltype =
  L.function_type void_t [|sock_t_ptr|] in
let create_func: L.llvalue =
  L.declare_function "ez_create" create_t the_module in

let connect_t: L.lltype =
  L.function_type void_t [|sock_t_ptr; str_t; i32_t|] in
let connect_func: L.llvalue =
  L.declare_function "ez_connect" connect_t the_module in

let close_t: L.lltype =
  L.function_type void_t [|sock_t_ptr|] in
let close_func: L.llvalue =
  L.declare_function "ez_close" close_t the_module in

let send_t: L.lltype =
  L.function_type void_t [|sock_t_ptr; str_t|] in
let send_func: L.llvalue =
  L.declare_function "ez_send" send_t the_module in

let recv_t: L.lltype =
```

```
  L.function_type str_t [|sock_t_ptr|] in
let recv_func: L.llvalue =
  L.declare_function "ez_recv" recv_t the_module in

let write_t: L.lltype =
  L.function_type void_t [|str_t|] in
let write_func: L.llvalue =
  L.declare_function "writestr" write_t the_module in

let read_t: L.lltype =
  L.function_type str_t [| |] in
let read_func: L.llvalue =
  L.declare_function "readstr" read_t the_module 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 (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 int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
  and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
  and str_format_str = L.build_global_stringptr   "%s\n" "fmt" builder
  and char_format_str = L.build_global_stringptr  "%c\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 (t, n) 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 (t, n) =
      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
        SLiteral i  -> L.const_int i32_t i
      | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
      | SFliteral l -> L.const_float_of_string float_t l
      | SStrLiteral s ->
        let temp = L.build_global_stringptr s "temp_assign_ptr" builder in
        L.build_call createstr_func [| temp |] "strlit" builder
      | SNoexpr     -> L.const_int i32_t 0
      | SCharLiteral c -> L.const_int i8_t (int_of_char c)
      | SSock(e1, e2) -> let sock = L.build_alloca sock_t "socket" builder in
          let e1' = expr builder e1 in
          let e2' = expr builder e2 in
          (*get pointer to first/second/third element of socket struct
          a.k.a. connection type/port #/socket file descriptor (null intitially)*)
          let connection_typ_ptr = L.build_gep sock [|L.const_int i32_t 0; L.const_int i32_t 0
          let port_number_ptr = L.build_gep sock [|L.const_int i32_t 0; L.const_int i32_t 1|] "
          let file_descriptor_ptr = L.build_gep sock [|L.const_int i32_t 0; L.const_int i32_t 2
          (*store our calculated values in the allocd struct via the ptrs*)
          ignore(L.build_store e1' connection_typ_ptr builder);
          ignore(L.build_store e2' port_number_ptr builder);
          ignore(L.build_store (L.const_int i32_t 0) file_descriptor_ptr builder);
          ignore(L.build_call create_func[|sock|] "" builder);
          sock
          (*return the filled out struct*)
      | SId s       -> L.build_load (lookup s) s builder
      | SAssign (s, e) -> let e' = expr builder e in
                          ignore(L.build_store e' (lookup s) builder); e'
      | SPAssign (s, e) ->
        (* leverage concat logic for SPAssign*)
        let old_str = L.build_load (lookup s) s builder in
        let e' = expr builder e in
        let new_str  = L.build_call add_strs_func[|old_str; e'|] "strcat" builder in
        ignore(L.build_free old_str builder);
        L.build_store new_str (lookup s) builder

      | SBinop ((A.Float,_ ) as 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
          | A.And | A.Or | A.Charat->
              raise (Failure "internal error: semant should have rejected and/or on float")
```

```
      ) e1' e2' "tmp" builder
  | SBinop ((A.String,_) as e1, op, e2) ->
let e1' = expr builder e1
and e2' = expr builder e2 in
(match op with
  A.Add     -> L.build_call add_strs_func [|e1'; e2'|] "strcat" builder
 | A.Charat -> L.build_call char_at_func [|e1'; e2'|] "charat" builder
 | A.Equal  -> L.build_call check_str_func [|e1'; e2'|] "equality" builder

| _ ->      raise (Failure "unsupported string operation")
)
  | SBinop (e1, op, e2) ->
      let e1' = expr builder e1
      and e2' = expr builder e2 in
      (match op with
        A.Add     -> L.build_add
      | A.Sub     -> L.build_sub
      | A.Mult    -> L.build_mul
      | A.Div     -> L.build_sdiv
      | A.And     -> L.build_and
      | A.Or      -> L.build_or
      | A.Equal   -> L.build_icmp L.Icmp.Eq
      | A.Neq     -> L.build_icmp L.Icmp.Ne
      | A.Less    -> L.build_icmp L.Icmp.Slt
      | A.Leq     -> L.build_icmp L.Icmp.Sle
      | A.Greater -> L.build_icmp L.Icmp.Sgt
      | A.Geq     -> L.build_icmp L.Icmp.Sge
| _ ->      raise (Failure "unsupported int operation semant should have rejected")
      ) e1' e2' "tmp" builder
  | SUnop(op, ((t, _) as e)) ->
      let e' = expr builder e in
      (match op with
        A.Neg when t = A.Float -> L.build_fneg
      | A.Neg                  -> L.build_neg
      | A.Not                  -> L.build_not) e' "tmp" builder
  | SCall ("print", [e]) | SCall ("printb", [e]) ->
      L.build_call printf_func [| int_format_str ; (expr builder e) |]
        "printf" builder
  | SCall ("printbig", [e]) ->
      L.build_call printbig_func [| (expr builder e) |] "printbig" builder
  | SCall ("printf", [e]) ->
      L.build_call printf_func [| float_format_str ; (expr builder e) |]
        "printf" builder
  | SCall ("prints", [e]) ->
L.build_call printf_func [| str_format_str ; (expr builder e) |]
  "printf" builder
  | SCall ("printc", [e]) ->
L.build_call printf_func [| char_format_str; (expr builder e) |]
  "printf"  builder
  | SCall ("connect", lst) ->
L.build_call connect_func [|(expr builder (List.nth lst 0));
(expr builder (List.nth lst 1));(expr builder (List.nth lst 2)) |]
  "" builder
  | SCall ("send", lst) ->
L.build_call send_func [|(expr builder (List.nth lst 0));
(expr builder (List.nth lst 1))|]
    "" builder
  | SCall ("recv", [e]) ->
L.build_call recv_func [|(expr builder e)|]
```

```
    "recvd_data" builder
  | SCall ("write", [e]) ->
L.build_call write_func [|expr builder e|] "" builder
  | SCall ("read", _ ) ->
L.build_call read_func [| |] "readstr" 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
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. *)
   (*NOTE: stmt add_terminal different that function add_terminal*)
let add_terminal builder instr =
  match L.block_terminator (L.insertion_block builder) with
    Some _ -> ()
  | None -> ignore (instr builder) 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
  | SExpr e -> ignore(expr builder e); builder
  | SReturn e -> ignore(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
  | SContext(((typ, v)), resource,  body) ->
    (match v with
    SId s ->
      (*bb for cleaning up resources*)
      let cleanup_bb = L.append_block context "clean" the_function in

      (*bb for exucuting the statements in the body of the with statement*)
```

```
      let body_bb = L.append_block context "body" the_function in


      let evaluated_resource = expr builder resource in
      (*so while the value s maps to might change this pointer will not as we control it*)
      let pointer = L.build_alloca (ltype_of_typ typ) "contextptr" builder in
      (*store the evaluated resource in our pointer for later and in our symbol tabl*)
      ignore(L.build_store evaluated_resource pointer builder);
      ignore(L.build_store evaluated_resource (lookup s) builder);
      ignore( L.build_br body_bb builder);

      (*cleanup routine*)
      let cleanup_builder = L.builder_at_end context cleanup_bb in

      let lookup = L.build_load pointer "cleanup_load" cleanup_builder in
      (*here is where we will add conditional behavior for sockets
      vs strings currently only configured for strings*)

      (*we have to free strings*)
      ignore(
        if typ = String then L.build_free lookup cleanup_builder
        else L.build_call close_func [|lookup|] "" cleanup_builder);

      (*body routine*)
      let body_builder = L.builder_at_end context body_bb in
      ignore(add_terminal (stmt body_builder body) (L.build_br cleanup_bb));

      (*return the cleanup builder to continue building the module*)
      cleanup_builder

    |_-> raise(Failure "semant should have caught that this is not assignable"))


  | 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

  (* Implement for loops as while loops *)
  | SFor (e1, e2, e3, body) -> stmt builder
        ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
in

(* Build the code for each statement in the function *)
let builder = stmt builder (SBlock fdecl.sbody) in

(* Add a return if the last block falls off the end *)
add_terminal builder (match fdecl.styp with
    A.Void -> L.build_ret_void
```

```
      | A.Float -> L.build_ret (L.const_float float_t 0.0)
      | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body functions;
  the_module
```

Listing 6: ezap.ml

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

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: ./ezap.exe [-a|-s|-l|-c] [file.mc]" 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)
```

Listing 7: TOP LEVEL Makefile

```
# "make" Compiles everything and runs the regression tests
LIB = ./stdlib
TEST = ./tests
SRC = ./src


.PHONY : test
test : compiler
	cd $(TEST) && ./testall.sh

# "make compiler" builds the executable as well as the "stdlib" library

.PHONY : compiler
compiler : clean
	cd $(SRC) && dune build
	ln -s ./_build/default/src/ezap.exe ezap.exe
	cd $(LIB) && make



#
# 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


# "make clean" removes all generated files

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

# Testing the "printbig" example

printbig : printbig.c
	cc -o printbig -DBUILD_TEST printbig.c
```

Listing 8: Compile.sh

```
#!/bin/sh
#script for compiling just a singular .ez source file down to a binary
# Path to the LLVM interpreter
LLI="/usr/bin/lli-13"
# Path to the LLVM compiler
LLC="/usr/bin/llc-13"
CC="cc"
EZAP="./ezap.exe"
SignalError() {
    if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
    fi
    echo "  $1"
}


Run() {
    echo $* 1>&2
    eval $* || {
        SignalError "$1 failed on $*"
        return 1
    }
}
    basename=`echo $1 | sed 's/.*\\///
                             s/.ez//'`

    Run "$EZAP" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "./stdlib/ezaplib-main.a"
```

Listing 9: STDLIB Makefile

```
CC := gcc
CFLAGS := -Wall -g
OBJS := printbig.o stringops.o socketops.o
DEPENDS := stringops.h socketops.h

# clean up object files cause we don't need them
ezaplib-main.a: clean stdlib
	ar -cq ezaplib-main.a *.o
	rm -f *.o

ezaplib.a: stdlib
	ar -cq ezaplib.a *.o
	rm -f *.o

stdlib: $(OBJS)

%.o: %.c $(DEPENDS)
	$(CC) $(CFLAGS) -c $<

clean:
	rm -f *.o *.a

all: clean ezaplib-main.a

.PHONY: clean all
```

Listing 10: socketops.h

```
#ifndef _SOCKETOPS_H_
#define _SOCKETOPS_H_

struct sock{
    char conn_type;
    int  port_num;
    int  fd;
};

void ez_create(struct sock *);

void ez_connect(struct sock *unconnected_socket, char *address, int port);

void ez_send(struct sock *connected_socket, const char *message);

char *ez_recv(struct sock *connected_socket);

void ez_close(struct sock *connected_socket);

#endif
```

Listing 11: socketops.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include "socketops.h"

static void die(const char *s) { perror(s); exit(1); }

void ez_create(struct sock *bare_socket){
    int fd;
    if ((fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
            die("failure to create a socket file descriptor");
    }
    bare_socket->fd = fd;
    struct addrinfo hints, *res;

    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me
    int port = bare_socket->port_num;
    char port_str[7];
    sprintf(port_str, "%d", port);
    if(getaddrinfo(NULL, port_str, &hints, &res) != 0){
        die("invalid port passed");
    }
    // bind it to the port we passed in to getaddrinfo():
    if(bind(fd, res->ai_addr, res->ai_addrlen) < 0) die("bind failed");
    freeaddrinfo(res);


}

void ez_connect(struct sock *unconnected_socket, char *address, int port){
    struct addrinfo hints, *res;
    int sockfd;

    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me


    char remote_port_str[7];
```

```
    sprintf(remote_port_str, "%d", port);
    getaddrinfo(address, remote_port_str, &hints, &res);

    sockfd = unconnected_socket->fd;

    // connect!

    if(connect(sockfd, res->ai_addr, res->ai_addrlen) < 0) die("connect failed");
    freeaddrinfo(res);
}

void ez_send(struct sock *connected_socket, const char *message){
    int length = strlen(message);
    if(send(connected_socket->fd, message, length, 0) < 0) die("send failed");
}

char *ez_recv(struct sock *connected_socket){
    char buf[4096];
    memset(buf,0,sizeof(buf));
    recv(connected_socket->fd, buf, sizeof(buf), 0);
    int recieved_length = strlen(buf);
    char *ret = (char *) malloc(recieved_length+1);
    strcpy(ret, buf);
    return ret;
}

void ez_close(struct sock *connected_socket){
    int fd = connected_socket->fd;
    close(fd);

}
```

Listing 12: stringops.h

```
#ifndef _STRINGOPS_H_
#define _STRINGOPS_H_

char *createstr(char *input);

char *concatstrs(const char *str1, const char* str2);

char charatstr(const char* str1, int i);

char checkstreq(const char *str1, const char *str2);

char *readstr();

void writestr(const char *);

#endif
```

Listing 13: stringops.c

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include "stringops.h"

char *createstr(char *input){
    int len = strlen(input);
    char *allocd_ptr = (char*)malloc(sizeof(char)*len+1);
    sprintf(allocd_ptr, "%s", input);
    return allocd_ptr;
}

char *concatstrs(const char *str1, const char *str2){
    int len = strlen(str1) + strlen(str2);
    char *allocd_ptr = (char *)malloc(sizeof(char)*len+sizeof(char));
    sprintf(allocd_ptr,"%s%s",str1, str2);
    return allocd_ptr;
}

//Return a null-terminated string of length 2 with index 0 equal to the char
char charatstr(const char *str1, int i ){
    int len = strlen(str1);
    //in the event the user passs an out of bounds integer
    if(i > len){
        errno = EPERM;
        perror("invalid index:");
        return 0;
    }
    else
    {
        char c = *(str1+i);
        return c;
    }
}

char checkstreq(const char *str1, const char *str2){
    if(strcmp(str1,str2) != 0) return 0;
    else return 1;
}

char *readstr(){
    /*artificial limit of 16k impossed on reading time permitting add
    more flexible read support*/
    char buf[16384];
    memset(buf,0,sizeof(buf));
    fgets(buf, 16384, stdin);
    char *ret = (char *)malloc(strlen(buf)+1);
    strcpy(ret, buf);
    return ret;

}

void writestr(const char *towrite){
    //implement later if time permits
}
```