CHZAP

Introduction

CHZAP is a general-purpose, imperative programming language intended to find a middle-ground between complexity, usability, and reliability. Features like type inference make CHZAP easy to use while static scoping and strict evaluation semantics maintain the robustness of C-like languages. CHZAP also aligns with modern programming standards by supporting anonymous and higher-order functions. In all, CHZAP aims to give programmers confidence in their programs through powerful capabilities, thorough semantics, and convenient features.

Language Tutorial

CHZAP In One Slide

```
/* CHZAP in one-slide */

int collatz(int x) {
  if(x == 1) { return 1; }
  if((x & 1) == 0) {
    return x / 2;
  }
  else {
    return 3 * x + 1;
  }
}

// anonymous & higher-order functions & type inference
auto twice = function(function(int)->int f, int x) -> int { return f(f(x)); };
print(twice(collatz, 3));   // 5
```

Compiling CHZAP

The CHZAP compiler can be built by running the “make” command. Programs can then be compiled using the chzap.native executable.

```
> ./chzap.native <-a|-s|-l> <file.chzap>
```

The `-l` flag will compile a .chzap file into LLVM IR code while the `-a` and `-s` flags can be used to construct the abstract syntax tree and semantically-checked abstract syntax tree respectively. CHZAP programs can also be compiled directly into executables using the *generate.sh* script.

```
> ./generate.sh <file> <CPU architecture: arm64, x86_64>
```

Note: OCaml and LLVM must be installed.

CHZAP Programs

CHZAP supports four basic datatypes: `int`, `float`, `char` and `bool`. A `void` type is also supported for functions with no return value. Variables can be declared anywhere in a CHZAP program, or organized into arrays.

Once declared, a variable may be redefined but may never change types. The `const` identifier can be inserted before any basic datatypes and yield a const type.

```
bool my_bool = true;
char my_char = 'a';

int[3] my_int_array;
my_int_array = [1,2,3];



int x = 1;
int y = 2;
x = y + x;

int z;
z = x;
```

```
bool my_bool = true;

my_bool = false;      // OK
my_bool = 2;          // NOT OK

// comments can be left like this
/* or this */

const int i = 1;     // define and initialize
i = 0;               // error: illegal assignment
```

Operators and control flows can also be used to define the behavior of a program. Here, a `roll` statement takes the place of a C-style `for` statement in the definition of loops.

```
int x = 0;
int y = 8;

// equivalent to for in C
roll(int i = 0; i < 10; i = i + 1) {
    if(x > y) {
        bark("CHZAP!", "i =", i);  // you can print to the terminal like this
        break;
    }
    else {
        x = x + 1;
    }
}
```

Functions can be defined and used for repetitive tasks.

```
int power(int x, int y) {
    int tot = 1;
    roll(int i = 0; i < y; i = i + 1) {
        tot = tot * x;
    }
        return tot;
}

print(power(2, 3));                // 8
```

In CHZAP, functions are also treated as datatypes (just like `int` or `float`), which means that they can be declared and assigned to like normal variables.

Function type can be expressed using the `function` keyword followed by `()`-enclosed type-list (corresponding to the argument types), and an `->` with its return type.

```
// declare a function that takes a character and returns an integer
function(char) -> int ctoi;

// type-list must be non-empty for declarations
// void can be used as a placeholder
function(void) -> float rand;
```

Anonymous functions are also supported. The body of anonymous functions should be enclosed by `{` and `}` and may contain multiple statements.

```
// this is an anonymous function (expression)
function(int x) -> int { return x + 1; }
```

Functions can also be passed and taken as arguments.

```
// the 2nd arugument is a function
// it takes an integer and returns a boolean
bool check_number(int x, function(int) -> bool f){
    return(f(x));
}

assert check_number(1, function(int x) -> bool { return x < 10; });  // true
```

Type inference is supported **on declaration** for any expression types including functions. It is indicated with the `auto` keyword.

```
/* CHZAP inference */

auto c = 'c';                    // char
auto s = 2 + 3;                  // int
auto f = function(int x) {       // function(int)->void
    bark("This is an int:", x);
};
```

Other Notes

- Identifiers must begin with a letter and may only contain letters, numbers, and underscores.
- Keywords are all lowercase.
- Chars are enclosed by `''` , strings by `""`
- CHZAP supports the following operators
    - arithmetic operations: `+` `-` `*` `/`
    - comparisons: `==` `!=` `<` `>` `<=` `>=`
    - logical operators: `&&` `||` `!`
    - bitwise operators: `&` `|`
    - assignment: `=`

Architectural Design

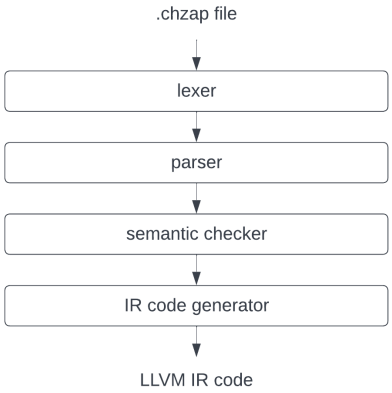


Fig. 1: CHZAP translator block diagram

Lexer

The lexer takes a *.chzap* file as an input and tokenizes the contents. These tokens are then passed to the parser.

Parser

The parser takes tokens generated by the lexer and organizes them into an abstract syntax tree (AST). The tree construction is based on grammar rules which use tokens as terminals. Provided no parsing errors occur the *.chzap* file is reduced to a `program`, the top level non-terminal symbol.

The program structure of CHZAP is simply a list of `statements`, which mainly consists of:

- Bindings: declarations
- Expressions and assertion
- Control flows: conditional clause, loops and return
- Function definition: regular (named) C-style functions

The bottom level non-terminals are `expressions`:

- Literals: integers, floats, characters, strings and identifiers
- Operations: arithmetic, logical, bitwise and subscription
- Assignments
- Function calls
- Anonymous function definitions

The program will be passed to the semantic checker for semantic analysis.

Semantic Checker

The semantic checker takes an AST as an input and semantically checks the program. This involves checking variable types and enforcing CHZAP's strict, static scoping, and it also verifies that operators are used properly, functions arguments and identifiers match expected types. The scoping is kept C-style by maintaining global and local symbol tables for every statement and expression.

The semantic checker outputs a semantically-checked AST (SAST) which matches the structure of the original AST but replaces all nodes with a tuple containing in the following format: `(type, node)`. The SAST will be used as input for the IR generator.

LLVM IR Generator

The IR generator takes the SAST generated by the semantic checker and produces LLVM IR 3-address code. This stage covers from LLVM function definitions and control flow instantiations, to variable allocation and execution of expressions.

Due to the program structure, CHZAP is developed to be used without the need to define a top-level `main` like traditional C-language, with support of arrays and function types. Major architectural designs that aim to achieve this include:

- Allocate all variables including arrays on stack frames. Dynamic memory allocation is not supported at this moment.
- Arrays are constructed using LLVM array types. All array sizes must be predefined.
- An *implicit* entry point is added to the generated IR enclosing all statements to enable scripting style coding.
- All user defined functions are referenced by their pointers. All function calls happen after dereferencing them.
- Function pointers and variables are stored in the same symbol table. There are no difference between them for the programmer so it's easier to manipulate.

Contributions

The lexer and parser were developed by Angela Peng. Chengbo Zang and Jiayang Hu created the AST structure and contributed to the parser as well. The semantic checker was developed by Emre Adabag. IR generation was handled by Jiayang Hu and Chengbo Zang. Jiyao Chen created the testing scripts and framework. It's important to note that all sections were largely collaborative efforts with contributions from all team members.

Test Plan

Array Test

- Source code:

```
/* CHZAP arrays */

int[3] arr;
arr = [1, 2, 3];
assert arr[0] != 1;
```

- Parser output:

```
---------parser_output---------

Parsed program:
arr: int[3]
arr = [1,2,3];
assert (arr[0] != 1);
```

- Semantic output:

```
---------semantic_output---------

Sementically checked program:

arr: int[3];
(int[3] : arr = (int[3] : [(int : 1),(int : 2),(int : 3)]));
assert (bool : (int : arr[(int : 0)]) != (int : 1));
```

- IR output:

```
---------ir_output---------

; ModuleID = 'CHZAP'
source_filename = "CHZAP"

@fmt = private unnamed_addr constant [21 x i8] c"assertion failed: %d\00", align 1

declare i32 @printf(i8*, ...)
declare i32 @exit(i32)
declare i32 @bitwiseAnd(i32, i32)
declare i32 @bitwiseOr(i32, i32)

define i32 @main() {
entry:
  %arr = alloca [3 x i32], align 4
  %tmp = alloca [3 x i32], align 4
  %tmp1 = getelementptr inbounds [3 x i32], [3 x i32]* %tmp, i32 0, i32 0
  store i32 1, i32* %tmp1, align 4
  %tmp2 = getelementptr inbounds [3 x i32], [3 x i32]* %tmp, i32 0, i32 1
  store i32 2, i32* %tmp2, align 4
  %tmp3 = getelementptr inbounds [3 x i32], [3 x i32]* %tmp, i32 0, i32 2
  store i32 3, i32* %tmp3, align 4
  %tmp4 = load [3 x i32], [3 x i32]* %tmp, align 4
  store [3 x i32] %tmp4, [3 x i32]* %arr, align 4
  %tmp5 = getelementptr [3 x i32], [3 x i32]* %arr, i32 0, i32 0
  %elem = load i32, i32* %tmp5, align 4
  %tmp6 = icmp ne i32 %elem, 1
  br i1 %tmp6, label %end_assert, label %assert_fail

assert_fail:                                      ; preds = %entry
  %tmp7 = getelementptr [3 x i32], [3 x i32]* %arr, i32 0, i32 0
  %elem8 = load i32, i32* %tmp7, align 4
  %tmp9 = icmp ne i32 %elem8, 1
  %print = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
             ([21 x i8], [21 x i8]* @fmt, i32 0, i32 0), i1 %tmp9)
  %exit = call i32 @exit(i32 1)
  br label %end_assert

end_assert:                                       ; preds = %assert_fail, %entry
  ret i32 0
}
```

- Program output:

```
---------program_output---------

assertion failed: 0
```

Type Inference & Anonymous Function Test

- Source code:

```
/* CHZAP inference */

auto c = 'c';                       // char
auto s = 2 + 3;                     // int
auto f = function(int x) {          // function(int)->void
    bark("This is an int:", x);
};
```

- Parser output:

```
---------parser_output---------

Parsed program:

c: auto c = c
s: auto s = (2 + 3)
```

```
f: auto f = lambda: void (x: int) {{
    bark(This is an int:, x);
}}
```

- Semantic output:

```
---------semantic_output---------

Sementically checked program:

c: char (char : c = (char : c = (char : c)))
s: int (int : s = (int : s = (int : (int : 2) + (int : 3))))
f: function (int) -> void (function (int) -> void : f = (function (int) -> \
    void : f = (function (int) -> void : lambda: void (x: int) {{
    (int : bark((string : This is an int:), (int : x)));
}}))))
```

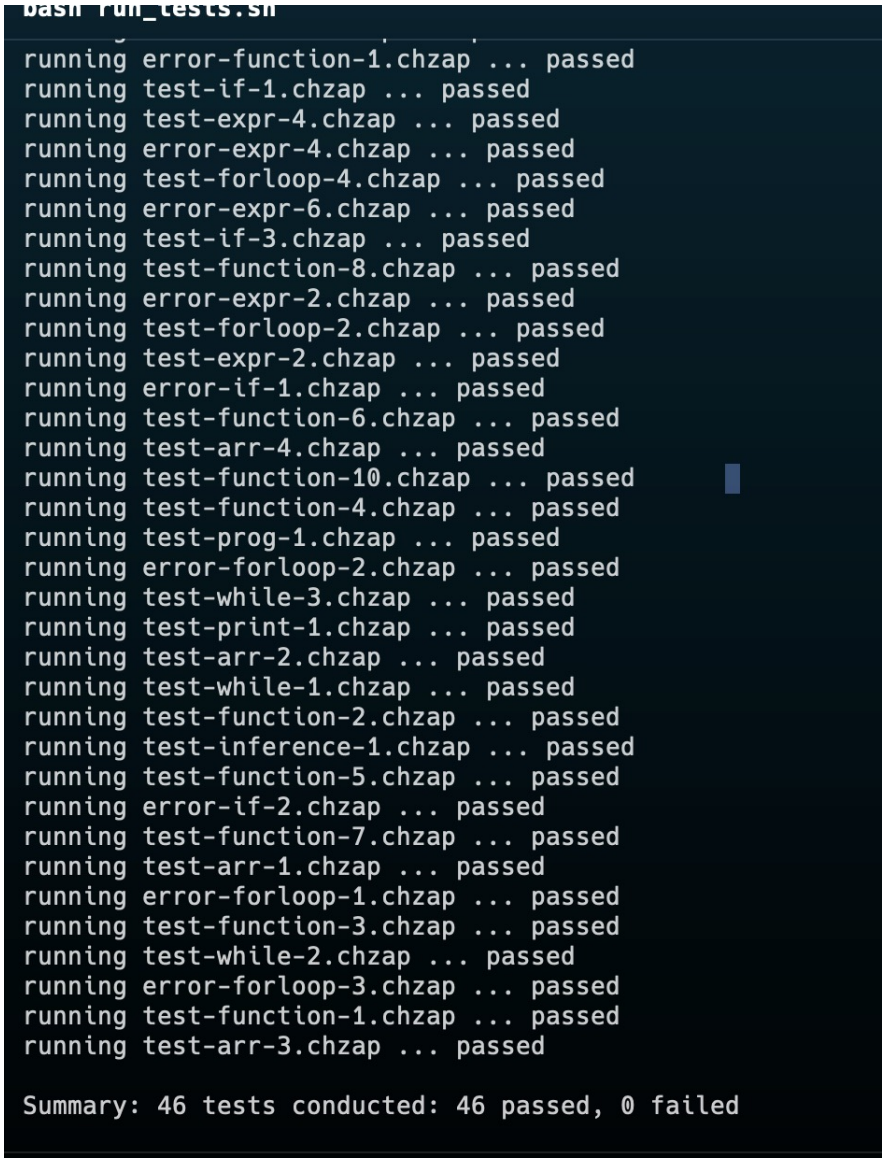
- IR output:

```
---------ir_output---------

; ModuleID = 'CHZAP'
source_filename = "CHZAP"

@fmt = private unnamed_addr constant [3 x i8] c"%s", align 1
@_string = private unnamed_addr constant [16 x i8] c"This is an int:", align 1
@fmt.1 = private unnamed_addr constant [3 x i8] c"%c", align 1
@fmt.2 = private unnamed_addr constant [3 x i8] c"%d", align 1
@fmt.3 = private unnamed_addr constant [3 x i8] c"%c", align 1
@fmt.4 = private unnamed_addr constant [3 x i8] c"%c", align 1

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

declare i32 @exit(i32)

declare i32 @bitwiseAnd(i32, i32)

declare i32 @bitwiseOr(i32, i32)

define i32 @main() {
entry:
  %c = alloca i8, align 1
  store i8 99, i8* %c, align 1
  %s = alloca i32, align 4
  store i32 5, i32* %s, align 4
  %f = alloca void (i32)*, align 8
  %_anonymous = alloca void (i32)*, align 8
  store void (i32)* @_anonymous, void (i32)** %_anonymous, align 8
  store void (i32)* @_anonymous, void (i32)** %f, align 8
  ret i32 0
}

define void @_anonymous(i32 %0) {
entry:
  %x = alloca i32, align 4
  store i32 %0, i32* %x, align 4
  %bark = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([16 
  %bark1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @fmt.1, i32 0, i32 0), i8 32)
  %x2 = load i32, i32* %x, align 4
  %bark3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @fmt.2, i32 0, i32 0), i32 %x2)
  %bark4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @fmt.3, i32 0, i32 0), i8 32)
  %bark5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([3 x i8], [3 x i8]* @fmt.4, i32 0, i32 0), i8 10)
  ret void
}
```

Automation

We first clean up the log directory in case there was any left over from the previous iteration of tests. Then we go through each test which is stored under the `/test` directory with `.chazp` as the extension of the files. For each test file we would store the parser, semantic and intermediate code generation outputs in the corresponding log file under the `/log` directory with the same name as the test file but with a `.log` extension, while also storing the IR code in a separate `.ll` file as well.

In addition to storing the output of our compiler, we also keep track of the number of succeed and failed test cases. If the test was suppose to fail, then it is acceptable for the compiler to produce an error, and we would categorize that test as “passed”. However if the compiler generated “success” when it was suppose to fail, or “fail” when it was suppose to “succeed” we would consider that particular test as failing. Here is an screenshot of all our test cases passing :



Summary

Emre Adabag

I worked mostly on the semantic checker but contributed to other sections as well. This project was enriching in many ways but I was most impressed by the huge amount of work required to provide even basic error checks. I was humbled, both by the wisdom of my teammates and by the inscrutable calculations that occur behind the scenes to produce clear and specific error messages. My advice for future students: don't take compilers for granted.

Jiayang Hu

I worked mostly on implementing the types, arithmetic and control flow support for AST, parser, semantic checker and IR generator. It is my first time writing in OCaml which requires a very different mindset than the ones I've written before. Using the theoretical knowledge I learned from class to solve the problems I encountered while writing translator helped me to really understand how everything is connected together. My advice for future students would be: Plan early and give yourself sufficient time because it always turns out to take more.

Angela Peng

I worked mostly on the Abstract syntax tree and and the IR generation. OCaml is definitely very different from what i was coding in before, and I'm glad to be exposed to a new programming paradigm (learn that word in class). I have learn so much from the final project and also from this class in general. I'm truly grateful i stepped out of my comfort zone and experienced this class! And Super thankful for all my teammates for being super awesome and understanding and collaborative!

Chengbo Zang

I worked mostly on the structures and function features including anonymous and higher-order functions across AST, parser and IR generator. This project was an interesting experience to me for providing detailed insights into how the codes are actually handled. Having the chance to work with a group of productive teammates to implement our own translator definitely took things to another level. My advice for future students: be brave to make changes and be careful to test them.

Jiyao Chen

I worked on different sections of the project to help implement features such as supporting more types in AST and parser and writing tests for our language. The project was rewarding. It allowed me to dive into another kind of programming languages and explore the foundation of languages. Watching the lectures and doing the project were good practice to grind the concepts. I really appreciate my teammates for their fantastic work. My advice for future students is always reach out and communicate.