Cheez Final Report

COMS4115 Programming Language and Translators

Ruize Li (rl3250)
Ken Xiong (kx2175)
Jianyang Duan (jd3794)
Jingyi Wang (jw4204)

Department of Computer Science
Columbia University
Spring 2022

Contents

1 Introduction

The purpose of Cheez is to mimic some of the object oriented features as well as supporting data manipulation. Cheez is a strongly typed, static language with a syntax that is close to mainstream programming languages like Java and C's. The motivation of creating Cheez is to incorporate some object-oriented feature, so that experts in non-cs domain can use them to facilitate their research and area of interest. In Cheez, struct and array are two non-primitive data structures that are powerful in simulating OO events.

Cheez supports basic primitive data types, including int, bool, float, and string. It also supports two non-primitive data-types, including array and struct. Array is fixed length while struct has similar functions and use in C, such as defining field. Cheez also provides functions in printing.

2 Language Tutorial

2.1 Lexical Conventions

2.1.1 Punctuators and Comments

The punctuators are used to satisfy the rules of syntax while parsing. In Cheez, semicolon denotes an end of a statement. Curly brackets mark blocks of statements. Comma is used to separate and parse function arguments. Dot indicates name referencing of a struct to its field.

```
; {} , .
```

2.2 Types

2.2.1 Primitive Data Types

int
The integer type represents whole number values in 32 bits. For example, 1234.

float
The float type represents fractional number or point number values in 32 bits. For example, 12.34.

bool
The bool type represents a boolean value in 2 bytes. For example, true.

void
The void type is used to declare that a method does not return a value to its caller.

string
The string type represents a chain of character literals surrounded with single quote in 40 bytes. For example, 'Hello World'.

2.2.2 Non-primitive Data Types

Array
Array is a fixed-size structure used to store a collection of data items. Cheez supports int array, bool array, and float array where all elements of the array must have type consistent with the array. Array type is denoted as `int[]`, `bool[]`, and `float[]`.

Each array element is separated by comma, and all elements are enclosed in square brackets. For example, `[1, 2, 3]`.

Array is zero-based indexing in Cheez. Array element can be accessed by its position index enclosed by square brackets. For example, we can access the element at index 0 by `arr[0]`.

Array element can be reassigned with valid value after declaration. For example, `arr[1] = 5;` modifies the element at index 1 by assigning it with integer 5.

Array must be declared with initial value assigned, and the size of array is fixed once it is declared.

Struct
In Cheez, struct is a user-defined data type consisting of a collection of variables.

2.3 Operators

Cheez supports the four standard arithmetic operators for int and float types. Cheez also have several comparison operators; two expressions connected by a comparison operator return a boolean value. Lastly, Cheez also encompasses logical operators.

1. Addition (`+`)
2. Subtraction (`-`)
3. Division (`/`)
4. Multiplication (`*`)
5. Equal to (`x == y`)
6. Not equal to (`x != y`)
7. Greater than (`x > y`)
8. Less than (`x < y`)
9. Greater than or equal to (`x >= y`)
10. Less than or equal to (`x <= y`)
11. and operator (`&&`)
12. or operator (`||`)
13. not operator (`!`)

2.4 Statements and Expressions

2.4.1 Struct Declaration, Assignment, and Access

Struct Declaration

```
    struct Person {
    int age;
    string name;
};
```

Struct Assignment and Access

```
    int main()
{
    Person person1;
    person1.name = 'Ken';
    person1.age = 22;
    prints(person1.name);
    printi(person1.age);

    return 0;
}
```

2.4.2 Function Declaration

2.4.3 Primitive Data Type Expressions

```
int num = 1;                        //Integer
float num = 1.5;                    //Float
learnedocaml = true;                //Boolean
void main() { prints('hello'); }    //Void Return Type
string name = 'ken';                //String
```

2.4.4 Array Declaration, Assignment, and Access

Array Declaration and Assignment

```
int[] arri = [1, 2, 3];
bool[] arrb = [true, false, true];
float[] arrf = [2.3, 4.5, 1.0];
```

Array Access

```
int[] a = [1,2,3];
int r = 2;
int temp = a[0];
a[0] = a[r];
a[r] = temp;
```

2.4.5 Control Flow

If/Else Block

```
if (a > b)
    a = a-b;
else
    b = a-b;
```

While Loop

```
while (a != b)
    //do something
```

For Loop

```
int n = 10;
int result = 1;
for (int i = 0; i < n - 1; i = i + 1)
{
    result = result + 1;
}
```

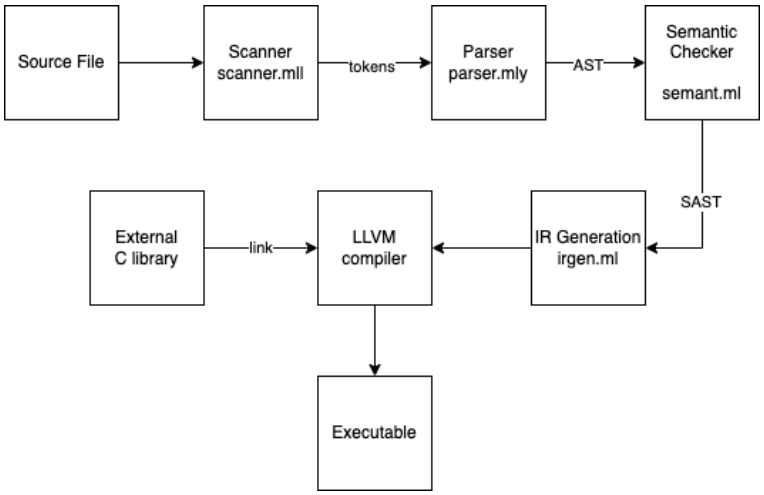
2.5 Built-in Functions

2.5.1 Printing functions

We support print in different type of string literals, including boolean, integer, string and float

```
prints("hello world");
printi(1);
printf(3.14);
printb(true);
```

3 Architectural Design



3.1 Interfaces

Scanner combines symbols from the source program files into list of tokens. It throws error when facing unrecognizable symbol or invalid syntax.

Parser uses production rules (CFGs) to combine the symbols into abstract syntax tree. It throws error when no derivation rules can be found for some stream of tokens.

In the next component, semantic checker uses symbol table and checks if the program is semantically correct, e.g. whether types are matching for some operators or whether function calls received correct arguments.

IR generation is where it converts the semantically checked program and use LLVM module to build IR code, where api calls are made to allocate memory space and manipulate pointers to store/access data.

The last piece is to link external library files where library function definitions are linked and incorporated in LLVM's code.

Finally, the executable is produced and user can run the program.

3.2 Implementation

Scanner : Ken Xiong, Jianyang Duan, Jingyi Wang, Ruize Li
Parser : Jianyang Duan, Ken Xiong
Semantic Checker : Jianyang Duan, Ken Xiong, Jingyi Wang Ruize Li
IR Generation : Ruize Li, Jianyang Duan, Ken Xiong
External Lib : Ruize Li, Jingyi Wang

4 Test Plan

4.1 Example 1

```
/* source program */
int main()
{
    printi(gcd(10,100));
}

int gcd(int a, int b) {
    while (a != b)
        if (a > b) a = a - b;
        else b = b - a;
    return a;
}
/* translated program */

; ModuleID = 'Cheez'
source_filename = "Cheez"

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

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


define i32 @gcd(i32 %b, i32 %a) {
entry:
  %b1 = alloca i32, align 4
  store i32 %b, i32* %b1, align 4
  %a2 = alloca i32, align 4
  store i32 %a, i32* %a2, align 4
  br label %while

while:                                            ; preds = %merge, %entry
  %a11 = load i32, i32* %a2, align 4
  %b12 = load i32, i32* %b1, align 4
  %"general op13" = icmp ne i32 %a11, %b12
  br i1 %"general op13", label %while_body, label %merge14

while_body:                                       ; preds = %while
  %a3 = load i32, i32* %a2, align 4
  %b4 = load i32, i32* %b1, align 4
  %"general op" = icmp sgt i32 %a3, %b4
  br i1 %"general op", label %then, label %else

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

```
then:                                             ; preds = %while_body
  %a5 = load i32, i32* %a2, align 4
  %b6 = load i32, i32* %b1, align 4
  %"general op7" = sub i32 %a5, %b6
  store i32 %"general op7", i32* %a2, align 4
  br label %merge

else:                                             ; preds = %while_body
  %b8 = load i32, i32* %b1, align 4
  %a9 = load i32, i32* %a2, align 4
  %"general op10" = sub i32 %b8, %a9
  store i32 %"general op10", i32* %b1, align 4
  br label %merge

merge14:                                          ; preds = %while
  %a15 = load i32, i32* %a2, align 4
  ret i32 %a15
}

define i32 @main() {
entry:
  %gcd_result = call i32 @gcd(i32 10, i32 100)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr
    inbounds ([4 x i8], [4 x i8]* @4, i32 0, i32 0), i32 %gcd_result)
  ret i32 0
}
```

4.2 Example 2

```
/* source program */
int main(){
int[] a = [1,2,3];
// two ptr reverse
int l = 0;
int r = 2;


while (l < r)
{
    int temp = a[l];
    a[l] = a[r];
    a[r] = temp;
    l = l + 1;
    r = r - 1;
}

// show array

for (int i = 0; i < 3; i = i + 1)
{
    printi(a[i]);
}


return 0;
}
```

```
/*translated program*/
define i32 @main() {
entry:
  %a = alloca { i32*, i32, i32 }, align 8
  %alloc = alloca { i32*, i32, i32 }, align 8
  %data_location = getelementptr inbounds { i32*, i32, i32 },
    { i32*, i32, i32 }* %alloc, i32 0, i32 0
  %0 = getelementptr inbounds { i32*, i32, i32 },
    { i32*, i32, i32 }* %alloc, i32 0, i32 1
  %data_loc = alloca i32, i32 3, align 4
  %item_loc = getelementptr i32, i32* %data_loc, i32 0
  store i32 1, i32* %item_loc, align 4
  %item_loc1 = getelementptr i32, i32* %data_loc, i32 1
  store i32 2, i32* %item_loc1, align 4
  %item_loc2 = getelementptr i32, i32* %data_loc, i32 2
  store i32 3, i32* %item_loc2, align 4
  store i32* %data_loc, i32** %data_location, align 8
  store i32 3, i32* %0, align 4
  %value = load { i32*, i32, i32 }, { i32*, i32, i32 }* %alloc, align 8
  store { i32*, i32, i32 } %value, { i32*, i32, i32 }* %a, align 8
  %l = alloca i32, align 4
  store i32 0, i32* %l, align 4
  %r = alloca i32, align 4
  store i32 2, i32* %r, align 4
  br label %while

while:                                            ; preds = %while_body, %entry
  %l11 = load i32, i32* %l, align 4
  %r12 = load i32, i32* %r, align 4
  %"general op13" = icmp slt i32 %l11, %r12
  br i1 %"general op13", label %while_body, label %merge

while_body:                                       ; preds = %while
  %temp = alloca i32, align 4
  %1 = getelementptr inbounds { i32*, i32, i32 },
    { i32*, i32, i32 }* %a, i32 0, i32 0
  %2 = load i32*, i32** %1, align 8
  %l3 = load i32, i32* %l, align 4
  %3 = getelementptr i32, i32* %2, i32 %l3
  %4 = load i32, i32* %3, align 4
  store i32 %4, i32* %temp, align 4
  %5 = getelementptr inbounds { i32*, i32, i32 },
    { i32*, i32, i32 }* %a, i32 0, i32 0
  %6 = load i32*, i32** %5, align 8
  %r4 = load i32, i32* %r, align 4
  %7 = getelementptr i32, i32* %6, i32 %r4
  %8 = load i32, i32* %7, align 4
  %9 = getelementptr inbounds { i32*, i32, i32 },
    { i32*, i32, i32 }* %a, i32 0, i32 0
  %10 = load i32*, i32** %9, align 8
  %l5 = load i32, i32* %l, align 4
  %11 = getelementptr i32, i32* %10, i32 %l5
  store i32 %8, i32* %11, align 4
  %temp6 = load i32, i32* %temp, align 4
  %12 = getelementptr inbounds { i32*, i32, i32 },
```

```
    { i32*, i32, i32 }* %a, i32 0, i32 0
  %13 = load i32*, i32** %12, align 8
  %r7 = load i32, i32* %r, align 4
  %14 = getelementptr i32, i32* %13, i32 %r7
  store i32 %temp6, i32* %14, align 4
  %18 = load i32, i32* %l, align 4
  %"general op" = add i32 %18, 1
  store i32 %"general op", i32* %l, align 4
  %r9 = load i32, i32* %r, align 4
  %"general op10" = sub i32 %r9, 1
  store i32 %"general op10", i32* %r, align 4
  br label %while

merge:                                            ; preds = %while
  %i = alloca i32, align 4
  store i32 0, i32* %i, align 4
  br label %while14

while14:                                          ; preds = %while_body15, %merge
  %i19 = load i32, i32* %i, align 4
  %"general op20" = icmp slt i32 %i19, 3
  br i1 %"general op20", label %while_body15, label %merge21

while_body15:                                     ; preds = %while14
  %15 = getelementptr inbounds { i32*, i32, i32 },
    { i32*, i32, i32 }* %a, i32 0, i32 0
  %16 = load i32*, i32** %15, align 8
  %i16 = load i32, i32* %i, align 4
  %17 = getelementptr i32, i32* %16, i32 %i16
  %18 = load i32, i32* %17, align 4
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([4 x i8], [4 x i8]* @1, i32 0, i32 0), i32 %18)
  %i17 = load i32, i32* %i, align 4
  %"general op18" = add i32 %i17, 1
  store i32 %"general op18", i32* %i, align 4
  br label %while14

merge21:                                          ; preds = %while14
  ret i32 0
}
```

5 Summary

Ruize Li I participated in high-level language design, wrote the semantic checker, scanner, parser, and contributed to the IR generation of array and struct part. I implemented the automatic testing suite, and wrote unit test cases for each functionality in our language. The testing suite also comes with manual analytic that allows user to dig into failed test cases.

This is my first time implementing a programming language translator. My biggest take away is the importance of job assignment and progressive development. In early stage of the project, the team was not fully aware of the difficulty of implementation for some features, and when the feature carries over scanner, paser, and even sast, where we collectively realize that we are unable to implement those, it takes us even more time to revert back and cut off features. No need to mention that things are more difficult when each of us manages different components. We utilized version control tools decently, and were able to recover from failures when accidents happen, and that was pretty good. Overall I realized how crucial incremental development is when working with large-scale project and collaborating with other people.

Ken Xiong I participated in high-level language design and contributed to the scanner, parser, semantic checker, and IR generation. Specifically, I help resolving bugs that we have seen in the semantic checker and IR generation; I also implement the struct from scanner to IR generation.

The experience of designing a language is valuable and the biggest takeaway is that the key success of teamwork is communication. As others have mentioned, version controls have been an issue. Due to schedule conflicts, each of us works independently and the communication was not transparent. Thus, this leads to numerous merge conflicts since we constantly modify codes that other people are also working on. At certain point, we realized to take advantage of github sub-branch to ensure that each of us can work independently and we also assigned a manager to resolve merge conflicts. In terms of Ocaml, due to the nature of functional programming, it's hard to learn and utilize at the beginning. As I gradually become familiar with the usage of the language, I find that this language is extremely powerful and efficient.

In sum, this is a fruitful semester and this course have taught me dense materials about functional programming and how compilers execute programs. I hope to further learn more about functional programming in the future.

Jianyang Duan I participated in high-level language design and contributed to the scanner, parser, semantic checker, and IR generation. Specifically, I implemented most of the scanner and parser, and I implemented the array part in semantic checker and the array part in IR generation.

The most important takeaways from working on the project are documentation review and how to synchronously and collaboratively work on the same code base. Since this is the first time I learn to design a programming language and implement a translator myself, I'm quite unfamiliar with how to implement certain features using OCamllex, OCamlyacc, and LLVM, and also I'm not an expert in OCaml. Thus, I read a lot of documentations when I was doing the coding work and I kept learning. It was extremely important to get used to reading tons of docs and quickly adapt myself to the unknown domain of knowledge. Furthermore, since this is a group project and we work on the same code base, we need to pay special attention to the sequence of implementing different parts and need to handle the case of version conflict. We take full advantage of git version control tool to ensure code consistency and resolve the version conflict as we frequently update on the same code base with different modifications. We also try to organize our work with reasonable plans so that each later component is built on the previous one perfectly.

To sum up, this is a wonderful course that unfolds the secret of programming language and how machines execute the program. Working collaboratively on the project is

challenging yet categorically rewarding.

Jingyi Wang I participated in front-end language design and contributed to scanner, parser, semantic checker, and builtin function partk of the IR generation. Specifically, I tested and debugged the front-end and also implemented built-in libraries.

The most important takeways is learning how to work on a project like this that many part will dependent on each other, so it's hard to work asynchronouly without necessary communication with other members. Also learning how everything work together and make everything work together is challenging, like how the IR generation is associated with the front-end we already have. Also it takes quite amount of time to dive deep into how everything actually works and make small tweaks. If I would do this project again, I would really make our first big picture solid and know what's the expected final product, and list the details of each step and break into small tasks that everyone can do on their own.

5.1 Advices

1. include test cases that focus on memory management, and detect any inappropriate memory usage errors
2. proceed with caution when linking external library files, and make sure llvm value types are consistent
3. start early and code incrementally!