Final Project Report

The Socket-Oriented Concurrency Kit Unix combines Unix socket programming with bash-like syntax for safe and easy concurrency with the robustness of a C program. With improvements like warning compiler warnings for potential race conditions and deadlocks, SOCKit aims to make concurrent programming safer and more accessible leveraging source and destination data type alongside a pipelining syntax. The language facilitates (pseudo-)multi-threaded and asynchronous inter-client communication and simplifies the flow of data across devices. SOCKit is an ideal choice for developers focused on efficient data transfer and real-time communication. It also closely follows the same general paradigms that would be found in C, which include imperative, statically scoped, strict evaluation order, and its a weakly statically typed language.

Language Tutorial:

Sockit is designed to resemble C in its implementation, so a user with experience in C should not find it very hard to transition to using Sockit, with some slight differences that should be kept in mind when working with it. More specifically, it extends the already existing C functionalities to add support for concurrency protocols and related features.

There are some things to keep in mind however, that are relevant and important to remind users so that their experience can be smooth and as seamless as possible. Just like most languages, Sockit supports block comments using (/*...*/) as well as line comments using //. The identifier names for the variables need to begin with a letter or an underscore for them to be deemed appropriate.

Reserved keywords in Sockit are used to denote control structures, data types, and other language constructs. Sockit enhances type simplicity and introduces scripting-like conveniences. Sockit supports integral constants in various bases: decimal, hexadecimal, and also binary. You can specify binary by adding 0b in front of the value, while 0x can be added to signify hexadecimal notation. It supports background process execution to facilitate the use of daemon processes without having to stop the main code from running.

The same syntax representing mathematical as well as logical operations present in C are carried over to Sockit. The same control structure present in C can also be found in

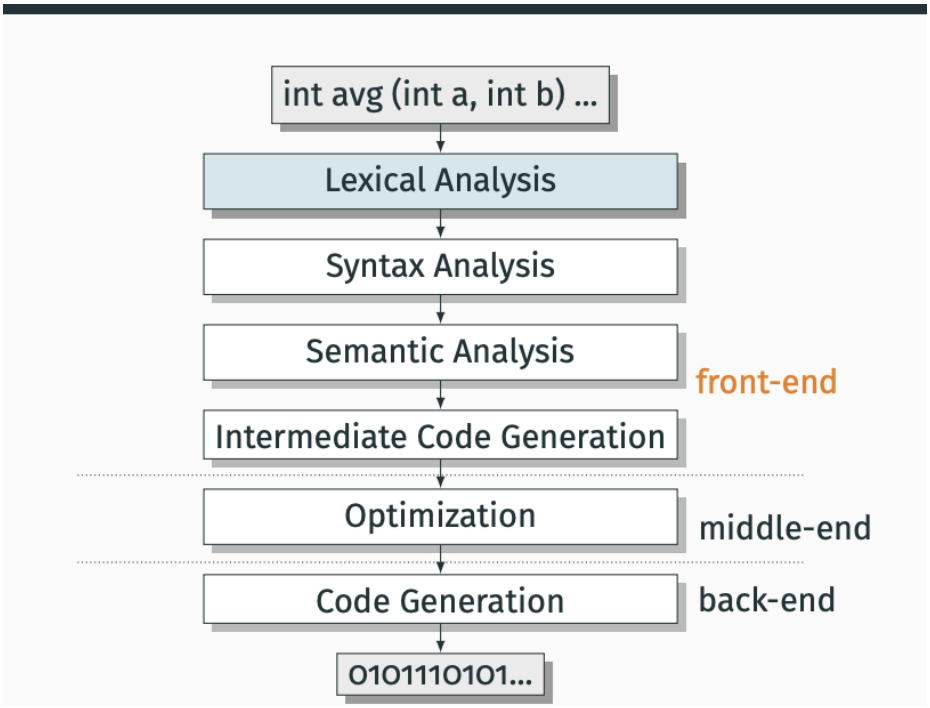
Sockit. The if-else, while, and switch-case statements work using the same logic. Meanwhile, the supported types are int, float, bool, and char. There is an additional type introduced to Sockit for data handling purposes called data. Sockit also introduces constructs for managing concurrent execution flows and simplifies the use of POSIX threads.

The unix-like features are also supported, such as data pipelining found in document1 | document2, redirection, and executing background processes.

It is meant to be easy to transition from C, and a preferable alternative when having to deal with multithreading and socket programming.

Architectural design:

Our architectural design closely follows the diagram that is presented in the course.



Test plan:

```
staceyyao@Staceys-MacBook-Pro-2 SOCKit % dune exec -- sockit -l test/etc/example.mc
; ModuleID = 'SOCKit'
source_filename = "SOCKit"

@a = global i32 0
@b = global i32 0
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1

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

define i32 @gcd(i32 %a, i32 %b) {
entry:
  %a1 = alloca i32, align 4
  store i32 %a, i32* %a1, align 4
  %b2 = alloca i32, align 4
  store i32 %b, i32* %b2, align 4
  br label %while

while:                                            ; preds = %merge, %entry
  %a11 = load i32, i32* %a1, align 4
  %b12 = load i32, i32* %b2, align 4
  %tmp13 = icmp ne i32 %a11, %b12
  br i1 %tmp13, label %while_body, label %merge14

while_body:                                       ; preds = %while
  %b3 = load i32, i32* %b2, align 4
  %a4 = load i32, i32* %a1, align 4
  %tmp = icmp slt i32 %b3, %a4
  br i1 %tmp, label %then, label %else

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

```
then:                                             ; preds = %while_body
  %a5 = load i32, i32* %a1, align 4
  %b6 = load i32, i32* %b2, align 4
  %tmp7 = sub i32 %a5, %b6
  store i32 %tmp7, i32* %a1, align 4
  br label %merge

else:                                             ; preds = %while_body
  %b8 = load i32, i32* %b2, align 4
  %a9 = load i32, i32* %a1, align 4
  %tmp10 = sub i32 %b8, %a9
  store i32 %tmp10, i32* %b2, align 4
  br label %merge

merge14:                                          ; preds = %while
  %a15 = load i32, i32* %a1, align 4
  ret i32 %a15
}
```

```
define i32 @main() {
entry:
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  store i32 18, i32* @a, align 4
  store i32 9, i32* @b, align 4
  store i32 2, i32* %x, align 4
  store i32 14, i32* %y, align 4
  %y1 = load i32, i32* %y, align 4
  %x2 = load i32, i32* %x, align 4
  %gcd_result = call i32 @gcd(i32 %x2, i32 %y1)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.1, i32 0, i32 0), i32 %gcd_result)
  %gcd_result3 = call i32 @gcd(i32 3, i32 15)
  %printf4 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.1, i32 0, i32 0), i32 %gcd_result3)
  %gcd_result5 = call i32 @gcd(i32 99, i32 121)
  %printf6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.1, i32 0, i32 0), i32 %gcd_result5)
  %b = load i32, i32* @b, align 4
  %a = load i32, i32* @a, align 4
  %gcd_result7 = call i32 @gcd(i32 %a, i32 %b)
  %printf8 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt.1, i32 0, i32 0), i32 %gcd_result7)
  ret i32 0
}
```

The kind of automation used in testing Sockit was OCaml's OUnit testing framework to automate unit testing of the compiler's functionality, including lexer, parser, and code generator. 'dune run test' automatically compiles and runs specified test cases. The test scenarios used were:

test1.ml: Automated Parser tests to ensure AST is generated as expected.

test2.ml: Automated Semantic Analysis tests to ensure SAST validates semantic rules.

test_eval.ml: Automated evaluation tests of expressions and statements to ensure matching of behavior for runtime vs. output

Summary:

Ryan was the team leader and manager who did the heavy lifting when it came to organizing and implementing the ideas we drafted when we brainstormed what functionalities we wanted Sockit to include. Most important takeaway was how to organize people in the best way possible to complete a project.

Hakim was the “float goat”, in charge of hovering around multiple areas and contributing where help was needed. Most important takeaway was the importance of starting early to avoid issues.

Stacey was the language tester who made sure the code ran and debugged any issues that we had as we went. Most important takeaway was the usefulness of OCaml.

Quentin was the system architect who designed how each separate feature would link together and operate. Most important takeaway was how to work as a team and organize all the different pieces at play in a language.

Misha was the language guru in charge of making sure the language’s different functionalities were implemented correctly while seeing what was not feasible. Most important takeaway was to do a better job of scoping what can realistically be accomplished given the estimated timeframe.

We would recommend starting the project as early as possible and trying to work ahead of the deadlines. It is a big endeavor that requires a lot of time and testing.