Final CMSC388Z Project: Abhay Patel and Seyed Mohammad Ghaemi

Github Repo

https://github.com/apatelcs/CMSC388Z-Final-Project

Summary of Proposal and Goals

Our goal for this project was to create a compiler written in Rust for a subset of the Racket programming language (which we called MiniRacket). Our compiler converts MiniRacket source code into x86 assembly instructions. We also used the programming language C as our runtime environment. This project used University of Maryland's CMSC430 notes as a guide in implementing the various features.

Our 75% Goals

- Create a tokenizer and a parser which can accept code of our source language.
- Create a compiler which can convert our target language into x86 Assembly
- Create a runtime system in C to run outputted assembly files
- Support for integers, booleans, and characters
- Simple arithmetic operations such as incrementing, decrementing, and two integer addition and subtraction
- Conditional expressions (if-else statements)
- Support for variable bindings and lexical scoping

Our 100% Goals

- Support for lists and the binary `cons` operator for list construction
- Support for boxed values and the unary operators `box` and `unbox`
- Error checking to provide detailed error messages and prevent program crashing without explanation

Our 125% Goals

- Support for vectors and strings using the developed heap
- Support for top-level function declarations and function calls

---

Our Accomplishments

We were able to finish:

- We created a lexer/tokenizer that translates MiniRacket code into tokens
  - Our lexer also gives descriptive error messages based on where it failed to tokenize the code
- We created a parser which takes tokens from the lexer and parses them into an Abstract Syntax Tree
- We created an assembly library called a86 that provides an abstraction for writing sequences of x86 code in Rust
- We created a compiler that reads the AST and outputs x86 code

- Features our lexer/parser/compiler support:
    - Parsing integers, booleans, and characters
    - `add1`/`sub1`
    - `+`/`-`
        - Only accept 2 parameters
    - `if` statements
    - `let`
        - Only binds one variable at a time currently

We could not finish:

- Unfortunately we could not meet our 100% and 125% goals
- Support for the `cons` operator
- Support for `box` and `unbox`
- Compile-time error checking
- Vectors and strings
- Functions

Usage Instructions

Requirements:

- x86-64 ABI conforming OS
- Latest version of the rust stable build
- NASM version 2.15.05
- gcc

Instructions:

1. Write Racket code into a file, let's call it test.rkt
    - For a list of Racket features, see the **Our Accomplishments** section
    - Example file can be found in `mini_racket/src/test.rkt`
2. To get the assembly instructions for the file, there are two options
    - Run the command `cargo run test.rkt` in the `mini_racket/src` folder. This will create a file called `test.s` with the compiled assembly instructions for your `test.rkt` file.
    - Run the command `make test.s` in the `mini_racket/src` folder. This will also create a file called `test.s` with the compiled assembly instructions for your `test.rkt` file.
3. To compile and run your code, simply do as follows
    - Run the command `make clean`
    - Run the command `make test.run`
    - Run the command `./test.run`

Challenges and Surprises

One aspect that made this project much more difficult was tokenizing and especially parsing our MiniRacket source code. Suprisingly, in most cases, handling the parsing was more difficult than writing the compiler

code for a feature. In CMSC430, our compiler was written in Racket, and the language had a built in feature for parsing raw source code into S-expressions, which were easier to parse. These S-expressions were nested data structures which could be matched to easily. This eliminated the need for `lookahead()` and `match_token()` and also completely eliminated the tokenizing process. Since Rust did not have this (or we couldn't find a crate to do this in time), we decided to write a lexer and parser from scratch which became very tedious.

Another challenging aspect was creating a library that would allow us to combine together x86 instructions with ease. If we had not written this library, our compiler would be performing many string manipulations since the output is a string containing x86 instructions. This way, we were able to create abstractions for each type of assembly instruction and enforce syntax for instruction types. Creating the library required a lot of planning because we needed a way to package multiple instructions together and a way to package sequences of instructions together to form assembly programs.

Rust Observations

The first observation we made was the file system was odd in Rust. We decided that it would be best for our project to be divided up into multiple files since each file contained large chunks of code. We used the module system along with the `use` keyword to create links between files and import functions between files in our system. We also found interesting ways to use Rust I/O to read in and out from files to get the `main.rs` file to work with our runtime system to compile a file and produce the assembly instructions for some MiniRacket code. Finally, the robust type system in Rust proved very useful in our project along with the pattern matching because they provided ways for us to easily represent our AST, match expressions in our AST type, and build the entire compiler. We leveraged the type system to create representations for more robust errors and used built in types such as the `Result` and `Option` types to handle errors. Overall, Rust was a complicated but very useful language to work in, and once we layed out the foundation for our project, it was a very robust language for developing a compiler.