Chase Feng and Jake Lee
Email: cfeng18@jh.edu
Course: EN.601.429 – Functional Programming
Instructor: Professor Scott Smith

Project Design

Term: Fall 2024
Final Project
Due Date: November 13th, 2024

Motivation

For our final project, we will write a simple C compiler in OCaml. The compiler will take flags and input code written in a subset of C and create an executable derived from LLVM intermediate representation. We will write our own lexer for tokenization, a Menhir-based parser to create an AST, our own translator to perform semantic analysis and generate LLVM IR, and, finally, we'll use the LLVM tool to create an optimized executable file with the specified compilation flags. While our base implementation will take only a single C file as its input, time permitting, we will extend our compiler to handle linking and preprocessing, enabling support for multi-file input.

Specifications

We plan to support the following features of C in our source language

1. **Data Types:** We will support all notable C data types – including int, floats, char, long, and so forth. We will exclude quantifiers (i.e. const, volatile, etc.), and, for our base implementation, we will limit pointers to a depth of 1.
2. **Control Flow:** We plan to support all notable control flow features, such as if-else statements, loops (i.e. for, while, and do), and switch statements.
3. **Functions:** We will support all basic function definitions/calls, and we'll implement the proper stack-handling to support recursion. We will exclude variadic functions (i.e., functions with ... in arguments).
4. **Basic Expressions:** We will support all basic operations, including arithmetic, relational, logical, bitwise and assignment operations.
5. **Memory Management:** We will support static memory and stack usage – including local and global variables – but will exclude dynamic memory usage.
6. **Aggregate Types:** We will support structs, but, in our base implementation, will exclude union types. This is for the reason that while structs involve only non-overlapping memory addresses, unions require a single address capable of being one of two types. As far as we can tell, this is a more challenging task to handle in LLVM; though it seems may be do-able, we will only work on adding this once the rest of our implementation is near-complete.
7. **Preprocessor Directives:** In our base implementation, we will exclude the handling of preprocessor directives such as include, define, etc. With time permitting, we will implement a basic preprocessor to handle these to expand our compiler to take inputs of multiple C files.

In addition to standard C/LLVM flags, we'll also add two custom flags for enhanced compiler functionality:

- `--to-llvm-ir`: Outputs only the LLVM IR, skipping executable generation. This flag cannot be combined with other flags.

- `--preprocess-only`: Outputs the preprocessed code, showing the result after handling preprocessor directives like `#define` and `#include`.

Our main project goal is functionality – delivering an accurate compiler with comprehensive error handling. Because our backend is powered by LLVM, we expect we should be able to deliver comparable performance to GCC/Clang in terms of compilation time, runtime, and memory usage, though, initially, this will not be an area of focus of ours. With time permitting, we may explore these areas in more detail.

Design

To implement our compiler, we'll perform the following steps

1. **Preprocessor (time permitting):** Implement a basic preprocessor in OCaml, which will search for directives such as define, ifdef, ifndef, else, endif, and include. This will allow us to handle inputs of more than a single .c file.
   - #define: Track each macro definition in a map and expand macros by substituting defined values wherever they appear
   - #ifdef, #ifndef, #else and #endif: Include or exclude code blocks based on macro definitions
   - #include: Open the specified file and insert its contents. We'll track the set of already-included files to ensure we don't add a file more than once. We'll track any header guards we come across (see above) to prevent the same file being included multiple times
2. **Lexer:** Using the built-in Lexing module, create lexer.ml to read and classify characters. We'll use the Lexing.lexbuf type from the module to track the input's current position and help generate potential error messages. The main function will be one to identify and produce tokens (e.g., identifiers, numbers, operators, etc.), taking a lexbuf and returning the next token.
3. **Parser:** Using Menhir, create menhir_parser.mly to define our grammar. When run, this will then create parser.ml and parser.mli OCaml files, which will then use our lexer and perform both lexing and parsing sequentially, with the lexer piece passing on one token at a time, and the parser using it to build the abstract syntax tree (AST), then requesting the next token. Altogether, the parser will verify that the inputted code has no errors with respect to our grammar tree, and will pass along an AST once finished.
4. **Semantic Analysis and Translation:** Given an AST, implement translator.ml to perform semantic analysis and "annotate" the AST to confirm its validity, as well as translate the AST into LLVM IR code.
   - Semantic Analysis: Check the input for more "logical" errors that go beyond the grammar. Whereas the Parser might check for syntax errors like forgotten symbols, typos or similar, we'd now check for things like identifiers being declared before they are used, correctly-declared type and type-consistent computations, and more.
   - Translation: Take the annotated AST and perform the actual translation to LLVM IR. To do this, we'll extend the existing ollvm library in OCaml.
5. **LLVM Manipulation and Exectuable File Creation:** Lastly, apply necessary manipulations to the IR before generating the final executable for the target architecture. In this step, we'll handle

the logic inputted by the initial compilation flags (i.e. linking, if time permits) and perform some simple optimizations, including getting rid of redundant code, loop unrolling and inlining (also time permitting). We'll use our extended ollvm library to manipulate the IR, and use LLVM to create the executable, likely via an OCaml call to a predefined shell script.

Libraries and Tools

Our program will use the core and ppx_jane libraries, and Menhir and LLVM as additional tools. We will also extend the existing ollvm library for our needs.

Appendix

LLVM

LLVM provides a framework for translating high-level code into optimized machine code via an architecture-independent intermediate representation (IR). Our compiler generates LLVM IR from an annotated abstract syntax tree (AST) derived from C code, which is then transformed by LLVM's backend into machine code for a platform-specific executable.

LLVM IR is a typed, static single assignment (SSA) language, meaning each variable is assigned exactly once before its use. Values are passed through a series of named registers (i.e. `%`-prefixed identifiers) in basic blocks, and control flow is managed by branches between blocks, allowing LLVM to efficiently implement optimizations like constant propagation and dead code elimination. LLVM IR follows a three-address code format, where each instruction generally involves an operator and up to two operands. It distinguishes between pointer types (e.g., `i8*`, `i32*`) and primitive types (`i32`, `float`), providing explicit typing for operations.

```
; Sample: @add is a function that takes two i32 parameters (%a and %b)
; The entry block performs an add operation on these arguments
; and returns the result.

define i32 @add(i32 %a, i32 %b) {
entry:
  %sum = add i32 %a, %b
  ret i32 %sum
}
```

During optimization, LLVM IR allows target-independent passes, such as constant folding and loop unrolling, as well as target-specific optimizations tailored to the machine architecture. The `opt` tool applies a series of passes, each represented as a transformation over the IR's control-flow graph (CFG). These passes reduce redundancy and enhance performance by analyzing dependencies, merging duplicate computations, and restructuring loops and branches for efficiency.

For code generation, LLVM's backend maps the optimized IR onto machine-specific assembly, addressing details like register allocation and instruction scheduling. This stage involves mapping SSA values to registers and handling architecture-specific constraints, which are critical for efficient execution.