CS 454/554 – Project 1
Team A, aka Team Really Really Long Compiler Errors

Our team is composed of Alecto Perez, Jorge Loredo, and David Arredondo. Alecto is most proficient in C++. David and Jorge are most proficient in Python. We used C++ for a majority of the project up to and including the creation of the intermediate language. Python was used for the final translation of the intermediate language to RISC-V assembly. Our team met every week since the project was assigned. Alecto was the project lead, since she was most proficient in C++ and had written the noam parser combinator library prior to this class, which she convincingly proposed as the ideal solution for parsing and creating the abstract syntax tree. She set up the skeleton for our project, using her own personal cmake and build scripts that she has developed in her previous C++ career. For the first many meetings, she taught us (David and Jorge) C++ and how to use the libraries. We became proficient enough to contribute on our own, although often needed guidance and help debugging. Near the end of the project, we decided to implement the intermediate representation to RISC-V script in python, to more evenly divide the labor and since it required less help from Alecto to debug. The project in total required a lot of effort and many hours to complete, including devising a strategy, planning and delegating tasks, coding and testing.

The noam parser combinator library, tuplet library, and recursive variant library were written by Alecto Perez outside the scope of this project; the motivation and use cases for these libraries are described in their respective repositories here: https://github.com/codeinred?tab=repositories

Using parser combinators, we are able to scan a .imp file and generate the abstract syntax tree in one pass. The AST is represented as a vector of commands, and can be passed to a script that generates a .dot file for graphviz, or to the assembler for further compilation into our intermediate representation. Finally the intermediate representation is read by the python script 'print_risc.py' which produces the final RISC-V assembly code.

Our compilation process is as follows:

1. Read a .imp file in as a string.
2. Use the comment parser to remove all comments from the string
3. Parse the rest of the string using parser combinators to generate an AST directly.
4. Translate the AST into either a .dot file for visual representation, or into a .txt file containing the intermediate representation.
5. Read the intermediate .txt file with a python script that translates to the final RISC-V assembly code.

The rest of this document is a comprehensive list of our files, tests, and examples; organized generally in the order of our compilation process. A high-level description of each folder and file is followed by the code itself.

/include/imp/
The files in this folder include all functions used for parsing, printing, generating the AST, printing the AST representation as a graphviz .dot file, and generating the intermediate representation

/include/imp/util/
This subdirectory contains utilities for AST node representation for graphviz, and the scope guard, which is used when reading external files.

/include/imp/util/node_id.hpp

The node_id struct is used to generate a unique hash for any given expression based on its type and value. This hash is used as the node id in graphviz, because the nodes can not share a name.

```
#pragma once

#include <fmt/format.h>
#include <rva/variant.hpp>

namespace imp {
struct node_id {
    size_t type_size;
    size_t node_location;
    template <class T>
    constexpr node_id(T const& node) noexcept
      : type_size(sizeof(T))
      , node_location((size_t)&node) {}
};

template <class T>
node_id get_id(const T& anything) {
    // This creates a unique identifier by combining the location of the
    // variable in memory with the size of the type. This ensures that types
    // that contain other types produce a unique identifier.
    return node_id(anything);
}
template <class... T>
node_id get_id(const rva::variant<T...>& anything) {
    auto visitor = [](auto const& x) { return get_id(x); };
    return rva::visit(visitor, anything);
}
} // namespace imp

// Having a formatter is like providing an overload of ostream. This will allow
// you to use imp::node_id in fmt expressions
```

```cpp
template <>
struct fmt::formatter<imp::node_id> {
    constexpr auto parse(format_parse_context& ctx) -> decltype(ctx.begin()) {
        return ctx.begin();
    }

    template <typename FormatContext>
    constexpr auto format(imp::node_id n, FormatContext& ctx) {
        return fmt::format_to(
            ctx.out(),
            "s{:02x}x{:x}",
            n.type_size,
            n.node_location);
    }
};
```

/include/imp/util/scope_guard.hpp

This is a utility to close a file when done reading.

```cpp
#pragma once

#include <type_traits>

namespace imp {

// A scope_guard contains one member, on_exit. on_exit will be invoked when the
// scope_guard is destroyed (aka, when exiting the scope in which the
// scope_guard was declared.)
template <class F>
struct scope_guard {
    // on_exit will be invoked when the scope_guard is destroyed
    F on_exit;

    // ~scope_guard() will be noexcept if invoking on_exit() won't throw an
    // exception
    constexpr ~scope_guard() noexcept(noexcept(on_exit())) { on_exit(); }
};

template <class F>
scope_guard(F) -> scope_guard<typename std::unwrap_ref_decay<F>::type>;
} // namespace imp
```

/include/read_file.hpp

This file simply takes the path of a file as input and returns a string containing the contents of that file, using /util/scope_guard to close the input file when done reading.

```cpp
#pragma once

#include <filesystem>
#include <system_error>

#include <stdexcept>
#include <string>
#include <system_error>

#include <imp/util/scope_guard.hpp>

namespace imp {
namespace fs = std::filesystem;

std::string read_file(const fs::path& path) {
    FILE* input_file = fopen(path.c_str(), "r");

    // When on_exit gets destroyed, it will close the input file
    scope_guard on_exit {[=] {
        if (input_file)
            fclose(input_file);
    }};

    // If input_file is nullptr, we need to throw an exception
    if (input_file == nullptr) {
        throw std::system_error(errno, std::system_category(), path);
    }

    // Get the size of the file
    size_t file_size = fs::file_size(path);

    std::string res(file_size, '\0');
    ssize_t bytes_read = fread(res.data(), 1, file_size, input_file);

    if (bytes_read < res.size()) {
        if (int at_eof = feof(input_file)) {
            clearerr(input_file);
            throw std::runtime_error(
                "Unable to read all bytes in" + path.string()
                + ". EOF reached early.");
        }
        if (int error_code = ferror(input_file)) {
            clearerr(input_file);
            throw std::system_error(
                error_code,
                std::system_category(),
                "Error when reading " + path.string());
        }
    }

    return res;
}
} // namespace imp
```

/include/syntax_types.hpp

This file contains the structs for each of the abstract syntax types. The structs are then grouped into three high level categories: arithmetic expressions, Boolean expressions, and commands. The high level syntax types are of the type 'rva::variant' or recursive variants. Recursive variants are recursive sum types; for example, we define an arithmetic expression to be either a constant, a variable, or a binary expression. The binary expression itself has three components – a left-hand side, operator, and a righthand side. The left- and righthand side can each be a constant, variable, or binary expression, thus the type is recursive. Each struct may contain auxiliary functions that make it easier to retrieve the struct elements. The rest of the compilation process will handle objects of the the rva::variant type.

```
#pragma once

#include <memory>
#include <noam/combinators.hpp>
#include <rva/variant.hpp>
#include <string_view>

#include <imp/copyable_ptr.hpp>

namespace imp {
struct constant {
    long long value;

    constexpr void for_each(auto&& func) const noexcept {}
};
struct bool_const {
    bool value;

    constexpr void for_each(auto&& func) const noexcept {}
```

```cpp
};
struct variable {
    std::string_view name;
    std::string_view get_name() const { return name; }

    constexpr void for_each(auto&& func) const noexcept {}
};
struct comment : std::string_view {

    constexpr void for_each(auto&& func) const noexcept {}
};

template <class T>
struct unary_expr {
    struct expr_data {
        T value;
        char op;
    };
    copyable_ptr<expr_data> data_ptr;
    T const& get_input() const { return data_ptr->value; }
    char get_op() const { return data_ptr->op; }

    constexpr void for_each(auto&& func) const noexcept { func(get_input()); }
};
template <class T>
struct binary_expr {
    struct data {
        T left;
        T right;
    };
    copyable_ptr<data> data_ptr;
    char op;
    binary_expr(T&& left, T&& right, char op)
      : data_ptr(data {std::move(left), std::move(right)})
      , op(op) {}

    T const& get_left() const { return data_ptr->left; }
    T const& get_right() const { return data_ptr->right; }
    char get_op() const { return op; }

    constexpr void for_each(auto&& func) const noexcept {
        func(get_left());
        func(get_right());
    }
};
struct skip_command {
    constexpr void for_each(auto&& func) const noexcept {}
};
```

```cpp
template <class BExpr, class Cmd>
struct if_command {
    struct data {
        BExpr condition;
        Cmd when_true;
        Cmd when_false;
    };
    copyable_ptr<data> data_ptr;
    BExpr const& get_condition() const { return data_ptr->condition; }
    Cmd const& when_true() const { return data_ptr->when_true; }
    Cmd const& when_false() const { return data_ptr->when_false; }

    constexpr void for_each(auto&& func) const noexcept {
        func(get_condition());
        func(when_true());
        func(when_false());
    }
};
template <class BExpr, class Cmd>
struct while_loop {
    struct data {
        BExpr condition;
        Cmd body;
    };
    copyable_ptr<data> data_ptr;
    BExpr const& get_condition() const { return data_ptr->condition; }
    Cmd const& get_body() const { return data_ptr->body; }

    constexpr void for_each(auto&& func) const noexcept {
        func(get_condition());
        func(get_body());
    }
};
template <class Expr>
struct assignment {
    variable dest;
    Expr value;

    constexpr void for_each(auto&& func) const noexcept {
        func(dest);
        func(value);
    }
};

template <class T, class F>
void for_each(T const& item, F&& func) {
    item.for_each(func);
}

template <class T, class F>
void for_each(std::vector<T> const& vect, F&& func) {
    for (auto& item : vect) {
        func(item);
    }
}

template <class... T, class F>
void for_each(rva::variant<T...> const& v, F&& func) {
    rva::visit([&](auto const& node) { for_each(node, func); }, v);
}



// clang-format off
/**
 * Arithmetic expressions a ::= x | n | a1 o_a a2
 * Boolean expressions b ::= true | false | not b1 | b1 o_b b2 | a1 o_r a2
 * Commands c ::= x := a | skip | c1 ; c2 | if b then c1 else c2 fi | while b do c1 od
 *
 * where x ranges over (scalar) variable names; n ranges over integer literals;
 * o_a ranges over binary arithmetic operators, namely +, −, ×;
 * true and false are boolean literals; not is a unary
 * boolean operator; o_b ranges over binary boolean operators, namely ∧, ∨;
 * o_r ranges over binary relational operators, namely =, <, ≤, >, ≥.
 */


// clang-format on
using arith_expr = rva::variant<
    constant,                  // Integral constant
    variable,                  // Variable
    binary_expr<rva::self_t>>; // Binary operation on to arithmatic expressions

using bool_expr = rva::variant<
    bool_const,               // Boolean constant
    unary_expr<rva::self_t>,  // Unary operation on boolean expression
    binary_expr<rva::self_t>, // Binary operation on boolean expression
    binary_expr<arith_expr>>; // Comparison on arithmetic expressions

using command = rva::variant<
    assignment<arith_expr>,              // Assignment command
    skip_command,                        // Skip command
    if_command<bool_expr, rva::self_t>, // If statement
    while_loop<bool_expr, rva::self_t>, // while loop
    std::vector<rva::self_t>>;           // List of commands separated by ;
```

```cpp
static_assert(std::is_copy_constructible_v<arith_expr>);
static_assert(std::is_copy_constructible_v<bool_expr>);
static_assert(std::is_copy_constructible_v<command>);
static_assert(std::is_move_constructible_v<arith_expr>);
static_assert(std::is_move_constructible_v<bool_expr>);
static_assert(std::is_move_constructible_v<command>);
static_assert(std::is_copy_assignable_v<arith_expr>);
static_assert(std::is_copy_assignable_v<bool_expr>);
static_assert(std::is_copy_assignable_v<command>);
static_assert(std::is_move_assignable_v<arith_expr>);
static_assert(std::is_move_assignable_v<bool_expr>);
static_assert(std::is_move_assignable_v<command>);
} // namespace imp
```

/include/copyable_ptr.hpp
This file defines the class ‘copyable_ptr’, which is an owning pointer to an object (it will deallocate the object at the end of it's lifetime). When copyable_ptr is copied, the new copyable_ptr will contain a copy of the object pointed to by the old copyable_ptr.

```cpp
#pragma once
#include <type_traits>
#include <utility>

namespace imp {
template <class T>
class copyable_ptr {
    T* pointer = nullptr;
    constexpr explicit copyable_ptr(T* pointer) noexcept
      : pointer(pointer) {}

   public:
    static constexpr copyable_ptr aquire(T* pointer) {
        return copyable_ptr(pointer);
    }
    constexpr copyable_ptr() = default;
    constexpr copyable_ptr(T&& value)
      : pointer(new T(static_cast<T&&>(value))) {}
    constexpr copyable_ptr(T const& value)
      : pointer(new T(value)) {}
    copyable_ptr(copyable_ptr&& c) noexcept
      : pointer(std::exchange(c.pointer, nullptr)) {}
    copyable_ptr(copyable_ptr const& c)
      : pointer(new T(*c.pointer)) {}

    constexpr auto& operator=(copyable_ptr other) noexcept {
        std::swap(pointer, other.pointer);
        return *this;
    }

    constexpr T* operator->() noexcept { return pointer; }
    constexpr T const* operator->() const noexcept { return pointer; }

    constexpr T& operator*() & noexcept { return *pointer; }
    constexpr T const& operator*() const& noexcept { return *pointer; }
    constexpr T&& operator*() && noexcept { return static_cast<T&&>(*pointer); }
    constexpr T const&& operator*() const&& noexcept {
        return static_cast<T const&&>(*pointer);
    }

    constexpr operator bool() const noexcept {
        return bool(pointer);
    }

    constexpr ~copyable_ptr() {
        delete pointer;
    }
};
} // namespace imp
```

/include/printing.hpp

This file defines ‘print_expr’ which is simply used to print any expression for debugging/visualization. We define the overloaded function ‘print_expr’ for each of our syntax types. The visit function will match the correct overload for the expression and print out the relevant information for that expression. For example, a constant will have its value printed, or a binary expression will have its left- and righthand side printed as well as the operator used.

```
#pragma once

#include <imp/syntax_types.hpp>
#include <fmt/core.h>

namespace imp {

// Declare it ahead of time so we can use it in other functions
// The function itself could just go here b/c templates
void print_expr(auto const& expr);

void print_expr(constant const& c) {
    fmt::print("{}", c.value);
}
void print_expr(variable const& v) {
    fmt::print("{}", v.get_name());
}
void print_expr(binary_expr<arith_expr> const& expr) {
    print_expr(expr.get_left());
    fmt::print(" {} ", expr.get_op());
    print_expr(expr.get_right());
}

////// BOOL EXPR //////////

void print_expr(bool_const const& bc) {
    fmt::print("{}", bc.value);
}

void print_expr(unary_expr<bool_expr> const& ue) {
    print_expr(ue.get_input());
    fmt::print("{}", ue.get_op());
}

//same as print_expr(binary_expr<arith_expr>)
void print_expr(binary_expr<bool_expr> const& be) {
    print_expr(be.get_left());
    fmt::print(" {} ", be.get_op());
    print_expr(be.get_right());
}

//print_expr(binary_expr<arith_expr>) already exists -
//need another print_expr for the fourth case in bool_expr (binary_expr<arith_expr>)
??

//////// COMMAND //////////

void print_expr(assignment<arith_expr> const& a) {
    print_expr(a.dest);
    fmt::print(" = ");
    print_expr(a.value);
    fmt::print(";");
}

void print_expr(skip_command const& s) {
    fmt::print("skip;");
}

void print_expr(if_command<bool_expr, command> const& ic) {
    fmt::print("if ");
    print_expr(ic.get_condition());
    fmt::print(" then\n");
    print_expr(ic.when_true());
    fmt::print("\nelse\n");
    print_expr(ic.when_false());
    fmt::print("fi\n");
}

void print_expr(while_loop<bool_expr, command> const& wl) {
    fmt::print("while ");
    print_expr(wl.get_condition());
    fmt::print(" do \n");
    print_expr(wl.get_body());
    fmt::print("\ndone\n");
}
void print_expr(std::vector<command> const& commands) {
    for(auto const& cmd : commands) {
        print_expr(cmd);
        fmt::print("\n");
    }
}

// Now we actually declare it
void print_expr(auto const& expr) {
    // This lambda is templated so it can accept any type that has a print_expr
    // function. It calls the correct version of print_expr by finding the right
    // overload
    auto visitor = [](auto const& v) { print_expr(v); };

    // We use rva::visit since we're using rva::variant
    rva::visit(visitor, expr);
}

} //namespace imp
```

/include/parsers/
This folder contains the parsers that are used by the assembler (described in a later section) to generate the AST. Because we are using parser combinators from the noam library, this will directly produce the AST instead of lexing and parsing in two steps.

/include/parsers/basic_parsers.hpp
This file includes parsers for constants (long long values in C++), boolean constants (true, false), variables (a string of various length fulfilling the requirements in the project description), arithmetic operators (*,+,-), boolean operators (‘and’, ‘or’), and comparison operators (=,>,<,>=,<=).

```
#pragma once
#include <imp/syntax_types.hpp>
```

```cpp
#include <noam/combinators.hpp>
#include <noam/intrinsics.hpp>
#include <noam/result_types.hpp>
#include <noam/type_traits.hpp>

namespace imp {
constexpr auto parse_constant = noam::make<constant>(noam::parse_long_long);
constexpr auto parse_bool_const = noam::make<bool_const>(noam::parse_bool);

constexpr auto parse_variable = noam::parser {
    [](noam::state_t st) -> noam::result<variable> {
        std::string_view sv = st;

        size_t len = 0;
        while (len < sv.length() && isalnum(sv[len])) {
            len++;
        }
        if (len > 0) {
            return {
                st.substr(len),
                variable {std::string_view {sv.substr(0, len)}}};
        } else {
            return {};
        }
    }};
constexpr auto parse_op = noam::parser {
    [](noam::state_t st) -> noam::result<char> {
        if (st.empty()) {
            return {};
        }
        char ch = st[0];
        switch (ch) {
            case '+': return {st.substr(1), ch};
            case '-': return {st.substr(1), ch};
            case '*': return {st.substr(1), ch};
        }
        return {};
    }};
constexpr auto parse_bool_op = noam::parser {
    [](noam::state_t st) -> noam::result<char> {
        if (st.empty()) {
            return {};
        }
        if (st.starts_with("and")) {
            return {st.substr(3), '&'};
        }
        if (st.starts_with("or")) {
            return {st.substr(2), '|'};
        }
        return {};
    }};
constexpr auto parse_comparison = noam::parser {
    [](noam::state_t st) -> noam::result<char> {
        if (st.empty()) {
            return {};
        }
        if (st.starts_with("=")) {
            return {st.substr(1), '='};
        }
        if (st.starts_with("<=")) {
            // L represents less than
            return {st.substr(2), 'L'};
        }
        if (st.starts_with(">=")) {
            // G represents greater than
            return {st.substr(2), 'G'};
        }
        if (st.starts_with('<')) {
            return {st.substr(1), '<'};
        }
        if (st.starts_with('>')) {
            return {st.substr(1), '>'};
        }
        return {};
    }};
constexpr auto parse_cons_or_var = noam::either<arith_expr>(
    parse_constant,
    parse_variable);
}
```

/include/parsers/arith_expr.hpp
This file includes the parser for the arithmetic expression syntax type. This parser is composed of the basic parsers, and includes additional parsing for left- and righthand sides of the expression separated by whitespace.

```
#pragma once
#include <imp/parsers/basic_parsers.hpp>

#include <noam/combinators.hpp>
#include <noam/intrinsics.hpp>
#include <noam/result_types.hpp>
#include <noam/type_traits.hpp>

namespace imp {
```

```
using arith_expr = rva::variant<
    constant,                  // Integral constant
    variable,                  // Variable
    binary_expr<rva::self_t>>; // Binary operation on to arithmatic expressions

constexpr auto parse_arith_expr = noam::recurse<arith_expr>(
    [](auto parse_arith_expr) {
        return noam::parser {[=](noam::state_t st) -> noam::result<arith_expr> {
            st = noam::whitespace.parse(st).get_state();
            if (noam::result<arith_expr> lx = parse_cons_or_var.parse(st)) {
                st = lx.get_state();
                // Read all whitespace
                st = noam::whitespace.parse(st).get_state();
                if (auto op = parse_op.parse(st)) {
                    st = op.get_state();
                    st = noam::whitespace.parse(st).get_state();

                    if (noam::result<arith_expr> rx = parse_arith_expr.parse(
                            st)) {
                        st = rx.get_state();
                        return noam::result {
                            st,
                            arith_expr {binary_expr<arith_expr> {
                                std::move(lx).get_value(),
                                std::move(rx).get_value(),
                                op.get_value()}}};
                    }
                } else {
                    return lx;
                }
            }
            return {};
        }};
    });
}
```

/include/parsers/bool_expr.hpp

This file includes the parser for Boolean expressions, which may be a Boolean constant, or expression of an operation that returns a Boolean. This parser is composed again of the basic parsers with additional pattern recognition based on whitespace and the number of expected elements for an expression. For example, a unary expression expects two elements, the operator and the operant, where a binary operation expects a left- and righthand side as well as an operator in the middle.

```
#pragma once
#include <imp/parsers/arith_expr.hpp>
```

```cpp
namespace imp {
using bool_expr = rva::variant<
    bool_const,                // Boolean constant
    unary_expr<rva::self_t>,  // Unary operation on boolean expression
    binary_expr<rva::self_t>, // Binary operation on boolean expression
    binary_expr<arith_expr>>; // Comparison on arithmetic expressions


constexpr auto parse_bool_expr = noam::recurse<
    bool_expr>([](auto parse_bool_expr) {
    return noam::parser {[=](noam::state_t st) -> noam::result<bool_expr> {
        st = noam::whitespace.parse(st).get_state();
        if (st.starts_with("not")) {
            st.remove_prefix(3);
            st = noam::whitespace.parse(st).get_state();
            if (noam::result<bool_expr> expr = parse_bool_expr.parse(st)) {
                return noam::result<bool_expr> {
                    expr.get_state(),
                    unary_expr<bool_expr> {expr.get_value(), '!'}};
            } else {
                return {};
            }
        }
        if (noam::result<bool_const> lx = parse_bool_const.parse(st)) {
            st = lx.get_state();
            // Read all whitespace
            st = noam::whitespace.parse(st).get_state();
            if (auto op = parse_bool_op.parse(st)) {
                st = op.get_state();
                st = noam::whitespace.parse(st).get_state();

                if (noam::result<bool_expr> rx = parse_bool_expr.parse(st)) {
                    st = rx.get_state();
                    return noam::result {
                        st,
                        bool_expr {binary_expr<bool_expr> {
                            std::move(lx).get_value(),
                            std::move(rx).get_value(),
                            op.get_value()}}};
                }
            } else {
                return noam::result<bool_expr> {lx.get_state(), lx.get_value()};
            }
        }
        if (noam::result<arith_expr> lx = parse_arith_expr.parse(st)) {
            st = lx.get_state();
            st = noam::whitespace.parse(st).get_state();
            if (auto op = parse_comparison.parse(st)) {
                st = op.get_state();
                st = noam::whitespace.parse(st).get_state();

                if (noam::result<arith_expr> rx = parse_arith_expr.parse(st)) {
                    st = rx.get_state();
                    // Update lx, then check for a right hand side
                    auto lx2 = bool_expr {binary_expr<arith_expr> {
                        std::move(lx).get_value(),
                        std::move(rx).get_value(),
                        op.get_value()}};
                    st = noam::whitespace.parse(st).get_state();
                    if (auto op = parse_bool_op.parse(st)) {
                        st = op.get_state();
                        st = noam::whitespace.parse(st).get_state();

                        if (noam::result<bool_expr> rx = parse_bool_expr.parse(
                                st)) {
                            st = rx.get_state();
                            return noam::result {
                                st,
                                bool_expr {binary_expr<bool_expr> {
                                    std::move(lx2),
                                    std::move(rx).get_value(),
                                    op.get_value()}}};
                        }
                    } else {
                        return noam::result<bool_expr> {st, lx2};
                    }
                }
            }
        }
        return {};
    }};
});
}
```

/include/parsers/command.hpp

This file contains the command parser, composed of the other parsers as well as additional parsing for loops. Also this file contains the 'parse_program' definition, since our AST will always represent a program as a list of commands. The 'parse_program' function will be called by the assembler.

```
#pragma once
#include <imp/parsers/basic_parsers.hpp>
#include <imp/parsers/arith_expr.hpp>
```

```cpp
#include <imp/parsers/bool_expr.hpp>

namespace imp {
using command = rva::variant<
    assignment<arith_expr>,              // Assignment command
    skip_command,                        // Skip command
    if_command<bool_expr, rva::self_t>, // If statement
    while_loop<bool_expr, rva::self_t>, // while loop
    std::vector<rva::self_t>>;           // List of commands separated by ;

constexpr auto parse_one_command = [](auto parse_command) {
    return noam::parser {
        [parse_sub_cmd = noam::whitespace_enclose(parse_command)](
            noam::state_t st) -> noam::result<command> {
            constexpr auto parse_condition = noam::whitespace_enclose(
                parse_bool_expr);
            const auto empty_command = command {std::vector<command> {}};

            // Parse a command
            if (auto res = noam::literal<"skip">.read(st)) {
                return {st, skip_command {}};
            }
            if (noam::literal<"if">.read(st)) {
                noam::result<bool_expr> cond = parse_condition.read(st);
                if (!cond)
                    return {};
                if (!noam::literal<"then">.read(st))
                    return {};
                noam::result<command> when_true = parse_sub_cmd.read(st);
                if (!when_true)
                    return {};
                if (!noam::literal<"else">.read(st))
                    return {};
                noam::result<command> when_false = parse_sub_cmd.read(st);
                if (!when_false)
                    return {};
                if (!noam::literal<"fi">.read(st))
                    return {};

                return noam::result<command> {
                    st,
                    if_command<bool_expr, command>(
                        std::move(cond).get_value(),
                        std::move(when_true).get_value(),
                        std::move(when_false).get_value())};
            }
            if (noam::literal<"while">.read(st)) {
                noam::result<bool_expr> cond = parse_condition.read(st);
                if (!cond)
                    return {};
                if (!noam::literal<"do">.read(st))
                    return {};
                noam::result<command> when_true = parse_sub_cmd.read(st);
                if (!when_true)
                    return {};
                if (!noam::literal<"od">.read(st))
                    return {};
                return noam::result<command> {
                    st,
                    while_loop {
                        std::move(cond).get_value(),
                        std::move(when_true).get_value()}};
            }
            if (auto var = parse_variable.read(st)) {
                if (!noam::whitespace_enclose(noam::literal<":=">).read(st))
                    return {};
                auto value = parse_arith_expr.read(st);
                if (!value)
                    return {};
                return noam::result<command> {
                    st,
                    assignment<arith_expr> {
                        std::move(var).get_value(),
                        std::move(value).get_value()}};
            }
            return {};
        }};
};
constexpr auto parse_command = noam::recurse<command>([](auto self) {
    return noam::parser {
        [parse_one = parse_one_command(self)](
            noam::state_t st) -> noam::result<command> {
            auto first_command = parse_one.read(st);
            // If there's no first command, return an empty vector
            if (!first_command)
                return {st, command {std::vector<command> {}}};

            constexpr auto sep = noam::whitespace_enclose(noam::literal<';'>);

            std::vector<command> all;
            all.push_back(std::move(first_command.get_value()));

            auto next_command = noam::join(sep, parse_one);

            while (auto next = next_command.read(st)) {
                all.push_back(next.get_value());
            }

            // Read the separator if it was at the end with no command present
            sep.read(st);
            if (all.size() > 1) {
                return {st, command {all}};
            } else {
                return {st, std::move(all[0])};
            }
        }};
});
constexpr auto parse_program = noam::whitespace_enclose(parse_command);
} // namespace imp
```

/include/ast_printing.hpp
This file contains styling parameters for the output .dot file, as well as the functions used to generate the nodes and their structure/connections. The AST is passed in as a vector of commands, where the commands contain nested expressions or other commands. The AST is traversed twice, once to declare all unique nodes and their names, and again to define the node connections. First, the ‘declare_node’ function is called on the AST, which traverses the tree and declares all unique nodes according to their hash generated by the ‘node_id’ util. Then ‘print_edges’ is called which does a second pass over the AST, calling ‘style_node’ on that node’s hash. The ‘style_node’ function is overloaded for each of our various syntax types, giving it a name according to its value and styling it according to its type.

To produce the .dot file, simply call ‘ast_to_dotfile’, passing in a string which is the file output name (be sure to include the .dot prefix to the filename), and the AST which is a vector of commands.

```
#pragma once

#include <imp/util/node_id.hpp>
#include <fstream>
#include <imp/syntax_types.hpp>
#include <iostream>
#include <string>

namespace imp {
using std::string;
using std::ostream;

// By default, there is no styling
std::string_view style_node(std::vector<command> const&) {
    return R"(label="[list of commands]")";
```

```cpp
}
template <class Expr>
std::string_view style_node(assignment<Expr> const&) {
    return R"(fontname="Hack italic, monospace italic"; label="assignment")";
}
auto style_node(bool_const const& node) {
    return fmt::format(R"(fontcolor="#9ECE6A"; label="{}")", node.value);
}
auto style_node(constant const& node) {
    return fmt::format(R"(fontcolor="#9ECE6A"; label="{}")", node.value);
}
std::string_view style_node(while_loop<bool_expr, command> const&) {
    return R"(fontcolor="#BB9AF7"; label="while loop")";
}
std::string_view style_node(if_command<bool_expr, command> const&) {
    return R"(fontcolor="#BB9AF7"; label="conditional")";
}
std::string_view style_node(skip_command const&) {
    return R"(fontcolor="#BB9AF7"; label="skip")";
}
auto style_node(variable const& node) {
    return fmt::format(R"(fontcolor="#7AA2F7"; label="{}")", node.get_name());
}
template <class Expr>
auto style_node(unary_expr<Expr> const& node) {
    return fmt::format(
        R"(fontname="Hack bold, monospace bold"; label="{}")",
        node.get_op());
}
template <class Expr>
auto style_node(binary_expr<Expr> const& node) {
    return fmt::format(
        R"(fontname="Hack bold, monospace bold"; label="{}")",
        node.get_op());
}
template <class... T>
auto style_node(rva::variant<T...> const& node) {
    return rva::visit([](auto const& item) { return style_node(item); }, node);
}

void declare_nodes(ostream& os, auto const& node) {
    // Declare this node
    os << fmt::format("    {} [{}];\n", get_id(node), style_node(node));
    // Declare the child nodes
    for_each(node, [&](auto const& child) {
        declare_nodes(os, child);
    });
}
```

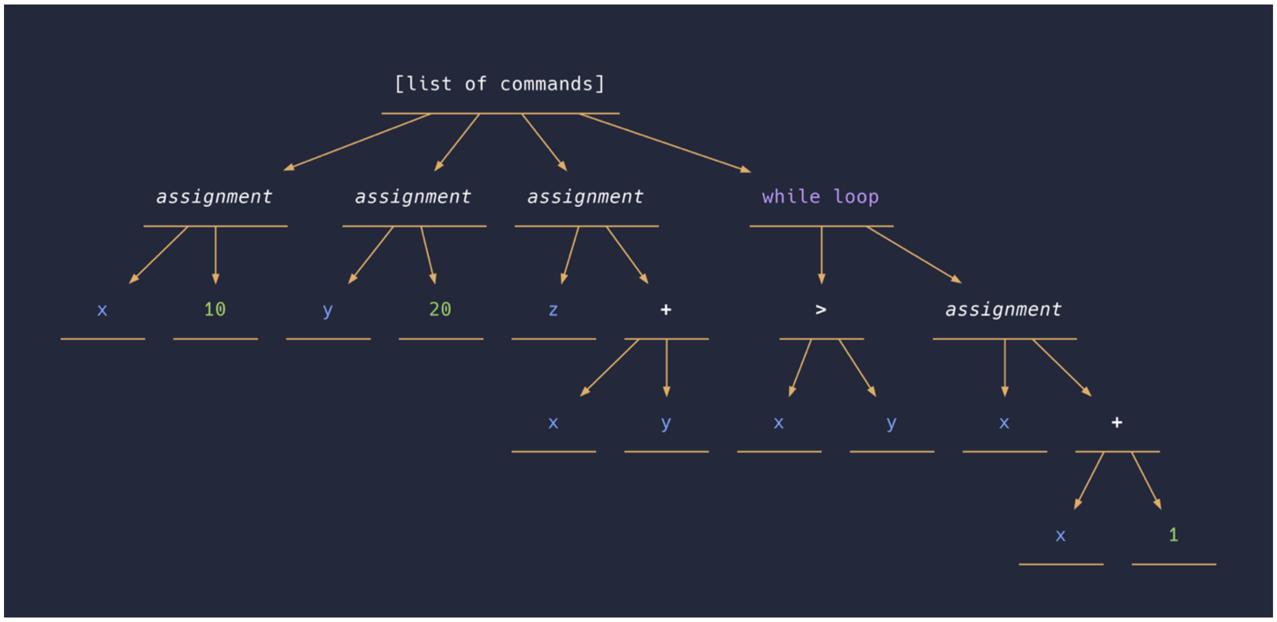
```
void print_edges(ostream& os, auto const& node) {
    // For each child, print an edge from this node to the child
    for_each(node, [&](auto const& child) {
        os <<  fmt::format("    {} -> {};\n", get_id(node), get_id(child));
        print_edges(os, child);
    });
}

void print_graph(ostream& os, arith_expr const& ast) {
    os << R"(digraph {
    graph [
        bgcolor="#24283B"
        pad="0.5"
        dpi=300]
    node [
        fontsize=12
        fontcolor="#ffffff"
        color="#E0AF68"
        shape=underline
        fontname="Hack, monospace"]
    edge [
        color="#E0AF68"
        arrowsize=0.5]
)";
    declare_nodes(os, ast);
    print_edges(os, ast);
    os << "}\n";
}
void print_graph(ostream& os, command const& ast) {
    os << R"(digraph {
    graph [
        bgcolor="#24283B"
        pad="0.5"
        dpi=300]
    node [
        fontsize=12
        fontcolor="#ffffff"
        color="#E0AF68"
        shape=underline
        fontname="Hack, monospace"]
    edge [
        color="#E0AF68"
        arrowsize=0.5]
)";
    declare_nodes(os, ast);
    print_edges(os, ast);
    os << "}\n";
}
// ast_to_dotfile
void ast_to_dotfile(string fname, command const& ast) {
    // Open the file
    std::ofstream dotfile(fname);

    // Print the graph to the dotfile
    print_graph(dotfile, ast);

    // dotfile closed automatically when it falls out of scope
}
} // namespace imp
```

/src/ast_printing_example.cpp

This takes an example AST command vector, containing all possible syntax types, and calls the ‘ast_to_dotfile’ function in the ‘ast_printing.hpp’ file, generating a .dot file that is given to graphviz to generate the visual representation.

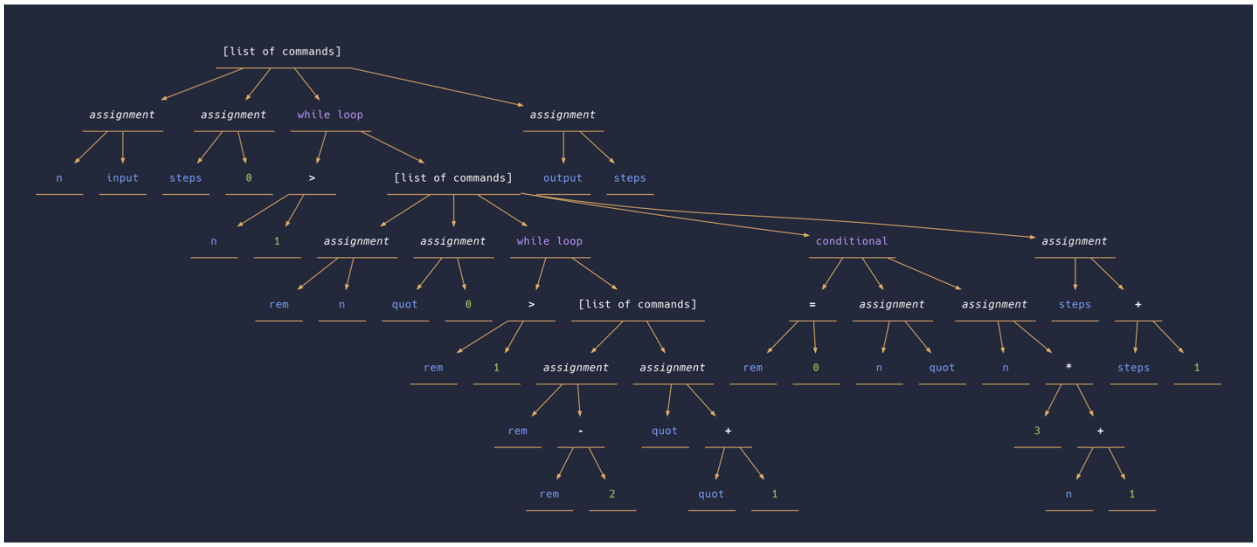
```
#include <fmt/core.h>
#include <fstream>
#include <imp/ast_printing.hpp>
#include <imp/syntax_types.hpp>
#include <iostream>

using namespace imp;

int main(int argc, char const* argv[]) {

    command test_command = command {
        std::vector<command> {
            assignment<arith_expr> {
                variable {"x"}, // <br>
                constant {10}   // <br>
            },                  // <br>
            assignment<arith_expr> {
                variable {"y"}, // <br>
                constant {20}   // <br>
            },                  // <br>
            assignment<arith_expr> {
                variable {"z"}, // <br>
                arith_expr {
                    binary_expr<arith_expr> {
                        variable {"x"}, // <br>
                        variable {"y"}, // <br>
                        '+'             // <br>
                    }                   // <br>
                }                       // <br>
            },                          // <br>
            while_loop<bool_expr, command> {
                while_loop<bool_expr, command>::data {
                    binary_expr<arith_expr> {
                        variable {"x"}, // <br>
                        variable {"y"}, // <br>
                        '>'             // <br>
                    },                  // <br>
                    assignment<arith_expr> {
                        variable {"x"}, // <br>
                        arith_expr {
                            binary_expr<arith_expr> {
                                variable {"x"}, // <br>
                                constant {1},   // <br>
                                '+'             // <br>
                            }                   // <br>
                        }                       // <br>
                    }                           // <br>
                }                               // <br>
            }                                   // <br>
        }                                       // <br>
    };


    // If no file is specified, print to standard output
    if (argc == 1) {
        print_graph(std::cout, test_command);
    } else {
        ast_to_dotfile(
            argv[1], // The filename is stored in argument 1
            test_command);
    }
}
```

The output from graphviz is shown below:



/programs/collatz.imp

The example command vector ‘test_command’ defined in ‘ast_printing_example’ is only visible in human-readable format because it was defined in the program. In reality, our parser will take read a .imp file and produce a vector of commands, which will then be directly passed to either ‘ast_to_dotfile’ for visual representation, or onto the next stage of compilation. As proof that our parser is working, we parsed the collatz.imp file and generated the following graphviz representation:



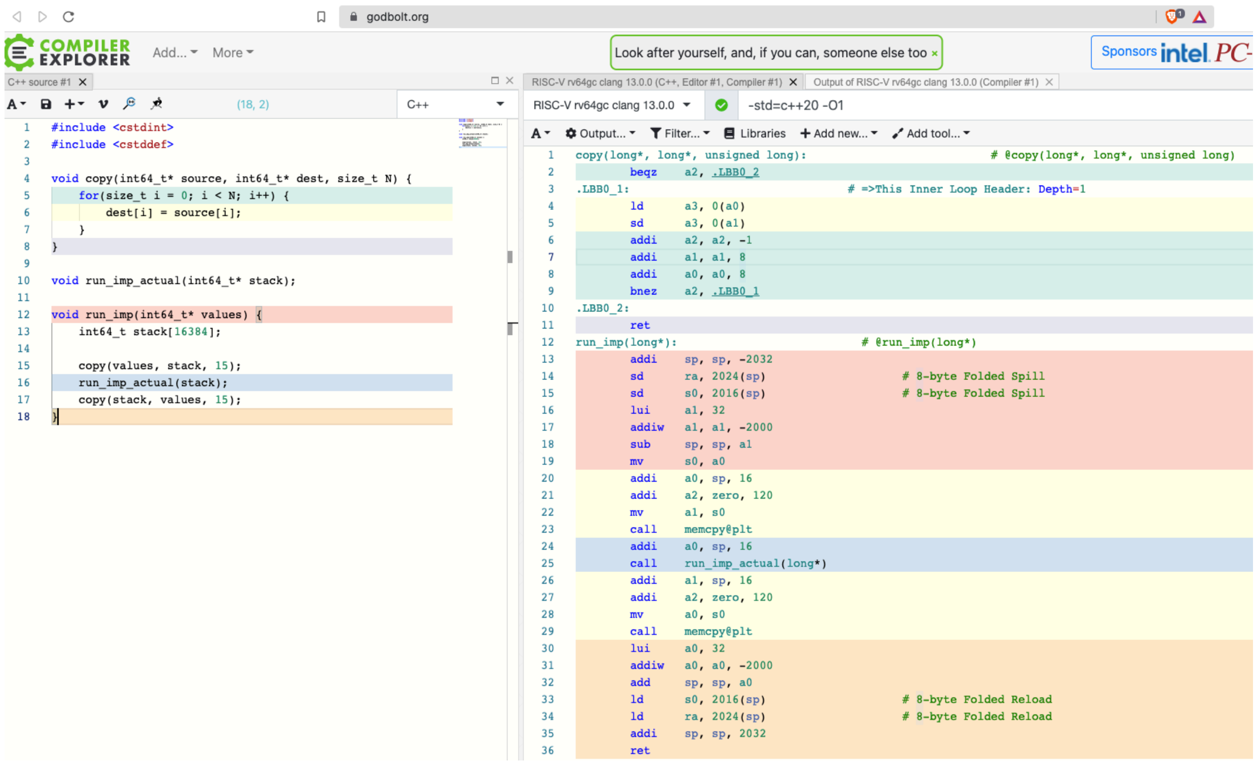
/include/translator.hpp
This file defines the 'assign_addresses' function, which first retrieves a list of all variables and their values. Then it assigns memory locations to the variables, which will have permanent locations on the stack. Then it assigns memory locations on the stack to all expressions in the abstract syntax tree. These addresses will be used in the intermediate representation in place of variable names. The addresses will be expressed in the intermediate representation as an index in the stack, which will then be interpreted in relation to the stack pointer in the final RISC-V assembly code.

Once variables, constants, and subexpressions have been assigned memory locations, the AST is converted into the intermediate representation. Each statement in the AST is translated to a series of instructions in the IR, and then the entirety of the IR is printed to a file to be compiled by the python script into RISC-V assembly.

```cpp
#pragma once
#include <fmt/core.h>
#include <imp/instruction.hpp>
#include <imp/syntax_types.hpp>
#include <imp/util/overload_set.hpp>
#include <set>
#include <unordered_map>

namespace imp {
using arith_expr = rva::variant<
    constant,                  // Integral constant
    variable,                  // Variable
    binary_expr<rva::self_t>>; // Binary operation on to arithmatic expressions

using bool_expr = rva::variant<
    bool_const,               // Boolean constant
    unary_expr<rva::self_t>,  // Unary operation on boolean expression
    binary_expr<rva::self_t>, // Binary operation on boolean expression
    binary_expr<arith_expr>>; // Comparison on arithmetic expressions

using command = rva::variant<
    assignment<arith_expr>,             // Assignment command
    skip_command,                       // Skip command
    if_command<bool_expr, rva::self_t>, // If statement
    while_loop<bool_expr, rva::self_t>, // while loop
    std::vector<rva::self_t>>;          // List of commands separated by ;

// Step 1: Get a list of all the variables in alphabetical order
// Step 2: assign addresses to variables and temporaries

// Step 1 - function to get all the variables
inline std::set<variable> get_variables(command const& program) {
    std::set<variable> variables;
    traverse(
        program,
        overload_set {
            [](auto const&) {},
            [&](variable const& var) { variables.insert(var); }});
    return variables;
}

// Step 2 - class to assign addresses to everything
// Assigns addresses to variables, temporaries, and constants
struct address_assigner {
    std::set<variable> variables;
    std::unordered_map<variable, address_t> var_addresses;
    void assign_address(variable& var, address_t) {
        var.address = var_addresses[var];
    }
    void assign_address(constant& c, address_t addr) { c.address = addr; }
    void assign_address(bool_const& c, address_t addr) { c.address = addr; }

    template <class Expr>
    void assign_address(unary_expr<Expr>& expr, address_t addr) {
        expr.address = addr;
        assign_address(expr.get_input(), addr + 1);
    }
    template <class... T>
    void assign_address(rva::variant<T...>& thing, address_t addr) {
        rva::visit(
            [addr, this](auto& thing) { assign_address(thing, addr); },
            thing);
    }
    template <class Expr>
    void assign_address(binary_expr<Expr>& expr, address_t addr) {
        expr.address = addr;
        assign_address(expr.get_left(), addr + 1);
        assign_address(expr.get_right(), addr + 2);
    }

    void assign_address(assignment<arith_expr>& ass, address_t addr) {
        assign_address(ass.dest, addr);
        assign_address(ass.value, addr);
    }
    void assign_address(skip_command&, address_t) {}
    void assign_address(if_command<bool_expr, command>& if_, address_t addr) {
        assign_address(if_.get_condition(), addr);
        assign_address(if_.when_true(), addr);
        assign_address(if_.when_true(), addr);
    }
    void assign_address(
        while_loop<bool_expr, command>& while_,
        address_t addr) {
        assign_address(while_.get_condition(), addr);
        assign_address(while_.get_body(), addr);
    }
    void assign_address(std::vector<command>& commands, address_t addr) {
        for (command& cmd : commands) {
            assign_address(cmd, addr);
        }
    }
    void assign_address(command& program) {
        variables = get_variables(program);
        var_addresses.reserve(variables.size());

        address_t addr = 0;
        for (auto const& var : variables) {
            var_addresses[var] = addr;
            addr++;
        }
        assign_address(program, addr);
    }
};

struct ir_compiler {
    // The position to jump to when skip occurs. Set to -1 initially, because
    // it can only be used inside a loop body
    int skip_position = -1;
    // The value of the next lael
    int next_label = 0;
    // Gets an unused unique label number
    int get_next_label() { return next_label++; }
    std::vector<instruction> ins;

    // No-op - does nothing
    void compile(variable const& v) {}
    void compile(constant const& c) {
        ins.push_back(instruction {Op::LoadConstant, c.value, 0, c.address});
    }
    void compile(bool_const const& c) {
        ins.push_back(instruction {Op::LoadConstant, c.value, 0, c.address});
    }
    void compile(binary_expr<arith_expr> const& b) {
        compile(b.get_left());
        compile(b.get_right());
        instruction i;
        switch (b.get_op()) {
            case '+':
                i = {
                    Op::Plus,
                    get_address(b.get_left()),
                    get_address(b.get_right()),
                    b.address};
                break;
            case '-':
                i = {
                    Op::Minus,
                    get_address(b.get_left()),
                    get_address(b.get_right()),
                    b.address};
                break;
            case '*':
                i = {
                    Op::Times,
                    get_address(b.get_left()),
                    get_address(b.get_right()),
                    b.address};
                break;
            case '=':
                i = {
                    Op::Equal,
                    get_address(b.get_left()),
                    get_address(b.get_right()),
                    b.address};
                break;
            case '<':
                i = {
                    Op::Greater,
                    // Flip operands: right first
                    get_address(b.get_right()),
                    get_address(b.get_left()),
                    b.address};
                break;
            case '>':
                i = {
                    Op::Greater,
                    get_address(b.get_left()),
                    get_address(b.get_right()),
                    b.address};
                break;
            case 'L':
                i = {
                    Op::GreaterEq,
                    // flip operands: right first
                    get_address(b.get_right()),
                    get_address(b.get_left()),
                    b.address};
                break;
            case 'G':
                i = {
                    Op::GreaterEq,
                    get_address(b.get_left()),
                    get_address(b.get_right()),
                    b.address};
        }
        ins.push_back(i);
    }
    void compile(unary_expr<bool_expr> const& b) {
        compile(b.get_input());
        // The only one right now is not, so I didn't bother writing a switch
        ins.push_back(
            instruction {Op::Not, get_address(b.get_input()), 0, b.address});
    }
    void compile(binary_expr<bool_expr> const& b) {
        compile(b.get_left());
        compile(b.get_right());
        Op op;
        switch (b.get_op()) {
            case '&': op = Op::And; break;
            case '|': op = Op::Or; break;
        }
        ins.push_back(instruction {
            op,
            get_address(b.get_left()),
            get_address(b.get_right()),
            b.address});
    }
    void compile(assignment<arith_expr> const& ass) {
        compile(ass.value);
        // Emit an instruction to move the output of the expression into the
        // destination
        ins.push_back(instruction {
            Op::Move,
            get_address(ass.value),
            0,
            ass.dest.address});
    }
    void compile(skip_command const& cmd) {
        // Jump to the location set for a skip
        // This should be the condition of a loop, because it skips everything
        // else in the body
        ins.push_back(instruction {Op::Jump, skip_position, 0, 0});
    }
    void compile(while_loop<bool_expr, command> const& loop) {
        int loop_condition = get_next_label();
        int loop_body = get_next_label();
        int loop_end = get_next_label();
        // Replace the skip position with the current loop condition. Save it to
        // 'previous_skip_position'. It'll be restored at the end of the
        // function
        int previous_skip_position = std::exchange(
            skip_position,
            loop_condition);

        address_t cond_addr = get_address(loop.get_condition());

        // Add the label for the start of the condition
        ins.push_back(instruction {Op::Label, loop_condition, 0, 0});
        compile(loop.get_condition());
        // Jump to the end of the loop if the condition is false
        ins.push_back(instruction {Op::JumpIfZero, cond_addr, loop_end, 0});

        // Compile the ody of the loop
        // First: Add a label indicating the start of the body
        ins.push_back(instruction {Op::Label, loop_body, 0, 0});
        // Then compile the body of the loop itself
        compile(loop.get_body());

        // Jump back to the start of the loop
        ins.push_back(instruction {Op::Jump, loop_condition, 0, 0});
        // Add a label for the instruction after the end of the loop
        ins.push_back(instruction {Op::Label, loop_end, 0, 0});

        // Restore the skip position
        skip_position = previous_skip_position;
    }
    void compile(if_command<bool_expr, command> const& if_) {
        int else_label = get_next_label();
        int end_of_if = get_next_label();

        address_t cond_addr = get_address(if_.get_condition());

        compile(if_.get_condition());
        // Jump to the else block if the condition was false
        ins.push_back(instruction {Op::JumpIfZero, cond_addr, else_label, 0});
        compile(if_.when_true());
        // Jump to the end of the if statement after finishing the 'then' block
        ins.push_back(instruction {Op::Jump, end_of_if, 0, 0});
        // Add the label for the start of the else block, then compile the else
        // block
        ins.push_back(instruction {Op::Label, else_label, 0, 0});
        compile(if_.when_false());
        // Add a label for the instruction after the end of the if statement
        ins.push_back(instruction {Op::Label, end_of_if, 0, 0});
    }
    void compile(std::vector<command> const& commands) {
        for (command const& c : commands) {
            compile(c);
        }
    }
    template <class... T>
    void compile(rva::variant<T...> const& v) {
        rva::visit([this](auto const& expr) { compile(expr); }, v);
    }
    // Step 3: Translate to Intermediate Representation
    void print(command cmd) {
        address_assigner scope;
        // Give everything an address
        scope.assign_address(cmd);

        // Compile the command into the instruction vector
        compile(cmd);
        for(instruction const& i : ins) {
            fmt::print("{}\n", i);
        }
    }
};
} // namespace imp
```

/intermediate_to_riscv/
This folder contains python code which reads in a text file containing the intermediate representation of the AST, translates this and prints the entire RISC-V code. The .ipynb file is a jupyter notebook that was used for quick iteration, which was then finalized as a .py file that takes a command line argument that is the name of the intermediate representation text file, and prints the RISC-V code. Our strategy was to give all input variables a permanent location on the stack. For each operation, we load the necessary values from the stack to registers, then perform the operation, and push the result back to the stack.

The ‘run_imp’ string was generated with the godbolt.org compiler explorer. The input to godbolt is a simple C++ script which takes a vector of 64-bit integers, loads them to the stack by calling the ‘copy’ function, runs the imp program on these values, and then loads the new values back to the stack. In the middle of the ‘run_imp’ program is a call to ‘run_imp_actual’, which is a placeholder for the risc-v instructions generated from our AST.


godbolt.org
COMPILER EXPLORER
Add...
More
Look after yourself, and, if you can, someone else too
Sponsors intel PC-
C++ source #1
(18, 2)
C++
RISC-V rv64gc clang 13.0.0 (C++, Editor #1, Compiler #1)
Output of RISC-V rv64gc clang 13.0.0 (Compiler #1)
RISC-V rv64gc clang 13.0.0
-std=c++20 -O1
Output...
Filter...
Libraries
Add new...
Add tool...

```cpp
#include <cstdint>
#include <cstddef>

void copy(int64_t* source, int64_t* dest, size_t N) {
    for(size_t i = 0; i < N; i++) {
        dest[i] = source[i];
    }
}

void run_imp_actual(int64_t* stack);

void run_imp(int64_t* values) {
    int64_t stack[16384];

    copy(values, stack, 15);
    run_imp_actual(stack);
    copy(stack, values, 15);
}
```

```asm
copy(long*, long*, unsigned long):                         # @copy(long*, long*, unsigned long)
        beqz    a2, .LBB0_2
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        ld      a3, 0(a0)
        sd      a3, 0(a1)
        addi    a2, a2, -1
        addi    a1, a1, 8
        addi    a0, a0, 8
        bnez    a2, .LBB0_1
.LBB0_2:
        ret
run_imp(long*):                         # @run_imp(long*)
        addi    sp, sp, -2032
        sd      ra, 2024(sp)                    # 8-byte Folded Spill
        sd      s0, 2016(sp)                    # 8-byte Folded Spill
        lui     a1, 32
        addiw   a1, a1, -2000
        sub     sp, sp, a1
        mv      s0, a0
        addi    a0, sp, 16
        addi    a2, zero, 120
        mv      a1, s0
        call    memcpy@plt
        addi    a0, sp, 16
        call    run_imp_actual(long*)
        addi    a1, sp, 16
        addi    a2, zero, 120
        mv      a0, s0
        call    memcpy@plt
        lui     a0, 32
        addiw   a0, a0, -2000
        add     sp, sp, a0
        ld      s0, 2016(sp)                    # 8-byte Folded Reload
        ld      ra, 2024(sp)                    # 8-byte Folded Reload
        addi    sp, sp, 2032
        ret
```

/intermediate_to_riscv/print_risc.py

This file contains the python functions that translate our intermediate representation to a RISC-V implementation. Each instruction in the intermediate representation contains four elements: op, i0, i1, and output. Not all elements are used for all instructions, and are described in greater detail in the next section.

The binaryops and jumpops dictionaries store the corresponding RISC-V instruction name for the given intermediate representation operations.

The 'stack' function takes an integer index and returns the respective index in the stack from the a0 register (stack pointer).

The 'load_stack' and 'save_stack' functions return load and save RISC-V instructions which will move values to/from an input register and stack location.

The 'binary_op' function will take as arguments three registers and an operation name, and returns the RISC-V instruction to perform a binary operation on the values held in two registers ai and aj, and put the output in the output register.

The 'unary_op' function returns the RISC-V operation name with two arguments. We just call them arg1 and arg2 since unary ops can include jumps or load_immediate whose arguments do not refer to registers.

To generate the RISC-V code, go to the '/intermediate_to_riscv' directory and call './print_risc.py your_program_path.imp'

```
import sys

run_imp = '''
addi    sp, sp, -2032
sd      ra, 2024(sp)                    # 8-byte Folded
sd      s0, 2016(sp)                    # 8-byte Folded
lui     a1, 32
addiw   a1, a1, -2000
sub     sp, sp, a1
mv      s0, a0
addi    a0, sp, 16
addi    a2, zero, 120
mv      a1, s0
call    memcpy@plt
addi    a0, sp, 16
call    run_imp_actual(long*)
addi    a1, sp, 16
addi    a2, zero, 120
mv      a0, s0
call    memcpy@plt
lui     a0, 32
addiw   a0, a0, -2000
add     sp, sp, a0
ld      s0, 2016(sp)                    # 8-byte Folded Reload
ld      ra, 2024(sp)                    # 8-byte Folded Reload
addi    sp, sp, 2032
ret
'''

def print_riscv_instruction(instruction):
    op,i0,i1 = instruction[:3] #assigns the instruction components
    output = instruction[3].strip('\n\t')
    binaryops =
{'Plus':'ADD','Minus':'SUB','Times':'MUL','Greater':'SGT','And':'AND','Or':'OR'}
    jumpops = {'JumpIfZero':'BEQZ','JumpIfNonzero':'BNEZ'}

    stack = lambda i : str(8*int(i)) + '(a0)' #converts stack index
    load_stack = lambda i, a: "LD "+a+", "+ stack(i) + "\n\t"  #load ith value from
stack to register a
    save_stack = lambda i, a: "SD "+a+", "+ stack(i) + "\n\t"  #save register a to ith
index in stack
    binaryop = lambda out_reg, ai, aj, opname: opname + " " + out_reg + ", " + ai +",
" + aj + "\n\t" #perform operation a1 op a2, put result in output register
```

```
    unaryop = lambda arg1, arg2, opname: opname + " " + arg1 + ", " + arg2 + '\n\t'

    if op in binaryops:
        riscv = load_stack(i0,'a1') +
        load_stack(i1,'a2') +
        binaryop('a1','a1','a2',binaryops[op]) +
        save_stack(output,'a1')

    elif op in jumpops:
        riscv = load_stack(i0,'a1') +
        unaryop('a1',i1,jumpops[op])

    elif op == 'GreaterEq': #a3 = (a1 < a2), then a1 = not a3, save a1 to stack output
        riscv = load_stack(i0,'a1') +
        load_stack(i1,'a2') +
        binaryop('a3','a1','a2','SLT') +
        unaryop('a1','a3','NOT') +
        save_stack(output,'a1')

    elif op == 'Equal': #a3 = (a1 < a2), a4 = (a1 > a2), a2 = (a3 xor a4), a1 = not
a2, save a1 to stack output
        riscv = load_stack(i0,'a1') +
        load_stack(i1,'a2') +
        binaryop('a3','a1','a2','SLT') +
        binaryop('a4','a1','a2','SGT') +
        binaryop('a2','a3','a4','XOR') +
        unaryop('a1','a2','NOT') +
        save_stack(output,'a1')

    elif op == 'Not':
        riscv = load_stack(i0,'a1') +
        unaryop('a1','a1','NOT') +
        save_stack(output,'a1')

    elif op == 'LoadConstant':
        riscv = unaryop('a1',i0,'LI') +
        save_stack(output,'a1')

    elif op == 'Label':
        riscv = i0 + '\n\t'

    elif op == 'Move':
        riscv = load_stack(i0,'a1') +
        save_stack(output,'a1')

    else: assert False, 'operation not found'

    return riscv
```

```python
def print_run_imp_actual(instructions):
    run_imp_actual = 'run_imp_actual(long*):\n\t'
    for i in instructions:
        run_imp_actual+=print_riscv_instruction(i)
    return run_imp_actual

with open(str(sys.argv[1]), 'r') as f:
    operations = f.readlines()
    instructions = [o.split(' ') for o in operations]

print(print_run_imp_actual(instructions))
print(run_imp)
```

/src/

This folder contains our source files.

/src/assembler.cpp

This file reads in a .imp file and prints the intermediate representation.

```cpp
#include <fmt/core.h>
#include <imp/ast_printing.hpp>
#include <imp/comment.hpp>
#include <imp/parsers/command.hpp>
#include <imp/read_file.hpp>
#include <imp/translator.hpp>

int main(int argc, char** argv) {
    if (argc == 1) {
        fmt::print("Fatal Error: No input files.\n");
        fmt::print("Usage:\n\n\t{} <filename>", argv[0]);
        return 1;
    }

    std::string file = imp::read_file(argv[1]);
    auto program = imp::filter_comments(file);

    auto parse_result = imp::parse_program.parse(program);
    if (!parse_result.get_state().empty()) {
        fmt::print("Unable to finish parsing program.\n");
        fmt::print(
            "Error beginning here: \n\n{}",
            std::string_view(parse_result.get_state()));
        return 1;
    }
    if (parse_result) {
        auto program_ast = parse_result.get_value();
        imp::ir_compiler compiler {};
        compiler.print(program_ast);
    } else {
        fmt::print("failed to parse thing\n");
    }
    return 0;
}
```

/src/print_ast.cpp

This file reads in a .imp file and returns a .dot file for graphviz.

```
#include <fmt/core.h>
#include <imp/ast_printing.hpp>
#include <imp/comment.hpp>
#include <imp/parsers/command.hpp>
#include <imp/read_file.hpp>
#include <imp/translator.hpp>

int main(int argc, char** argv) {
    if (argc == 1) {
        fmt::print("Fatal Error: No input files.\n");
        fmt::print("Usage:\n\n\t{} <filename>", argv[0]);
        return 1;
    }

    std::string file = imp::read_file(argv[1]);
    auto program = imp::filter_comments(file);

    auto parse_result = imp::parse_program.parse(program);
    if (!parse_result.get_state().empty()) {
        fmt::print("Unable to finish parsing program.\n");
        fmt::print(
            "Error beginning here: \n\n{}",
            std::string_view(parse_result.get_state()));
        return 1;
    }
    if (parse_result) {
        auto program_ast = parse_result.get_value();

        imp::print_graph(std::cout, program_ast);
    } else {
        fmt::print("failed to parse thing\n");
    }
    return 0;
}
```

/example_op_file.txt
This is a test .txt file containing each of the possible intermediate language commands. The intended interpretation of the examples are as follows:

Binary operators (lines 1-8) are interpreted as “Perform the given operation on the values in stack indexes 1 and 2, and store the result in stack index 3”.

Line 9 is interpreted as “Perform the NOT operation on the value in stack index 1, and store the result in stack index 3.”

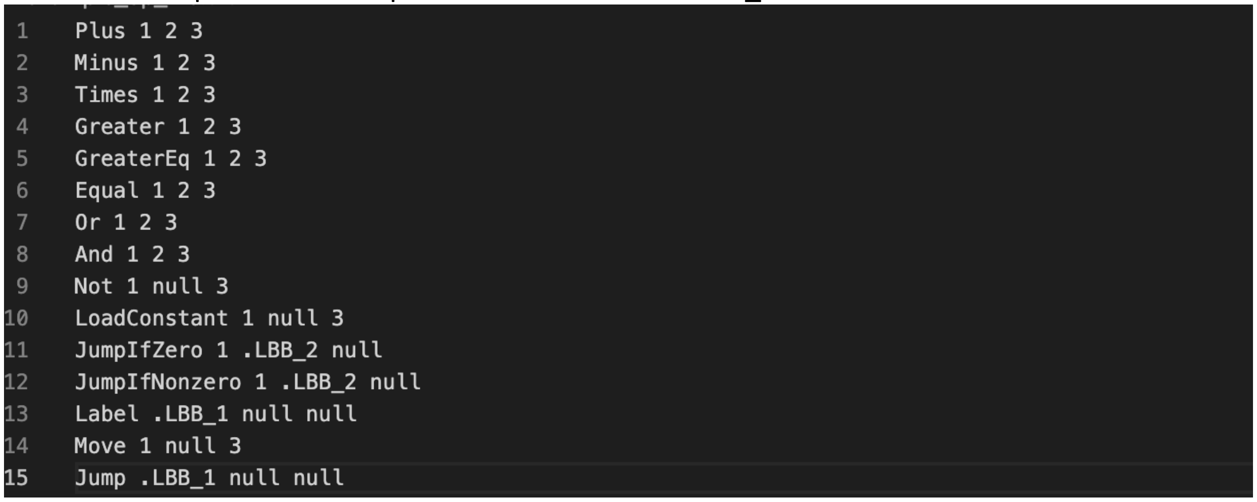
Line 10 is interpreted as “Store the 64-bit value equal to 1 in stack index 3.”

Lines 11, 12 are interpreted as “If the value in stack index 1 is equal/not equal to zero, jump to the line with label .LBB_2”

Line 13 is interpreted as “Insert a jump-to label called .LBB_1”

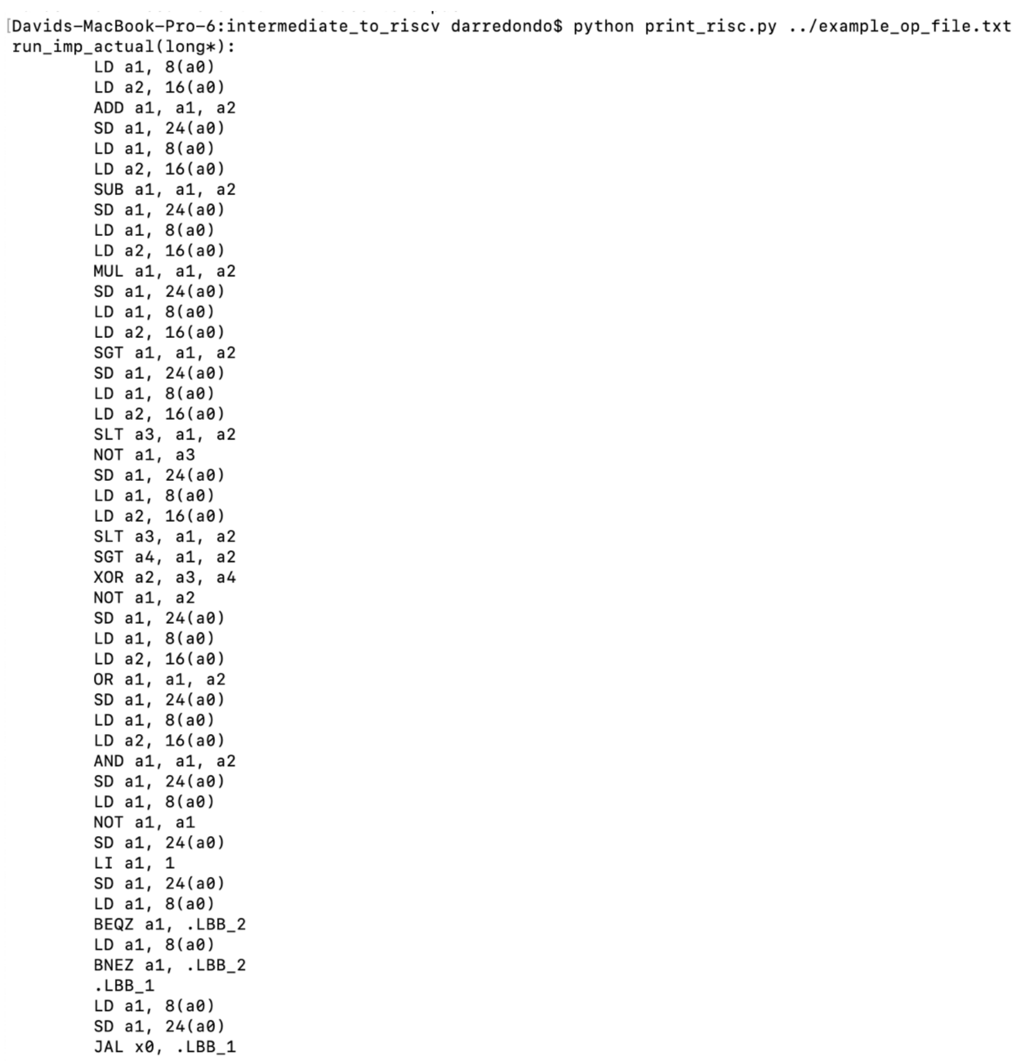
Line 14 is interpreted as “Move the value in stack index 1 to stack index 3”

Line 15 is interpreted as “Jump to the line with label .LBB_1”



```
1   Plus 1 2 3
2   Minus 1 2 3
3   Times 1 2 3
4   Greater 1 2 3
5   GreaterEq 1 2 3
6   Equal 1 2 3
7   Or 1 2 3
8   And 1 2 3
9   Not 1 null 3
10  LoadConstant 1 null 3
11  JumpIfZero 1 .LBB_2 null
12  JumpIfNonzero 1 .LBB_2 null
13  Label .LBB_1 null null
14  Move 1 null 3
15  Jump .LBB_1 null null
```

Running the ‘intermediate_to_riscv.py’ script on the example file gives the following set of instructions as the ‘run_imp_actual’ string.



```
[Davids-MacBook-Pro-6:intermediate_to_riscv darredondo$ python print_risc.py ../example_op_file.txt
 run_imp_actual(long*):
        LD a1, 8(a0)
        LD a2, 16(a0)
        ADD a1, a1, a2
        SD a1, 24(a0)
        LD a1, 8(a0)
        LD a2, 16(a0)
        SUB a1, a1, a2
        SD a1, 24(a0)
        LD a1, 8(a0)
        LD a2, 16(a0)
        MUL a1, a1, a2
        SD a1, 24(a0)
        LD a1, 8(a0)
        LD a2, 16(a0)
        SGT a1, a1, a2
        SD a1, 24(a0)
        LD a1, 8(a0)
        LD a2, 16(a0)
        SLT a3, a1, a2
        NOT a1, a3
        SD a1, 24(a0)
        LD a1, 8(a0)
        LD a2, 16(a0)
        SLT a3, a1, a2
        SGT a4, a1, a2
        XOR a2, a3, a4
        NOT a1, a2
        SD a1, 24(a0)
        LD a1, 8(a0)
        LD a2, 16(a0)
        OR a1, a1, a2
        SD a1, 24(a0)
        LD a1, 8(a0)
        LD a2, 16(a0)
        AND a1, a1, a2
        SD a1, 24(a0)
        LD a1, 8(a0)
        NOT a1, a1
        SD a1, 24(a0)
        LI a1, 1
        SD a1, 24(a0)
        LD a1, 8(a0)
        BEQZ a1, .LBB_2
        LD a1, 8(a0)
        BNEZ a1, .LBB_2
        .LBB_1
        LD a1, 8(a0)
        SD a1, 24(a0)
        JAL x0, .LBB_1
```

example_main.c

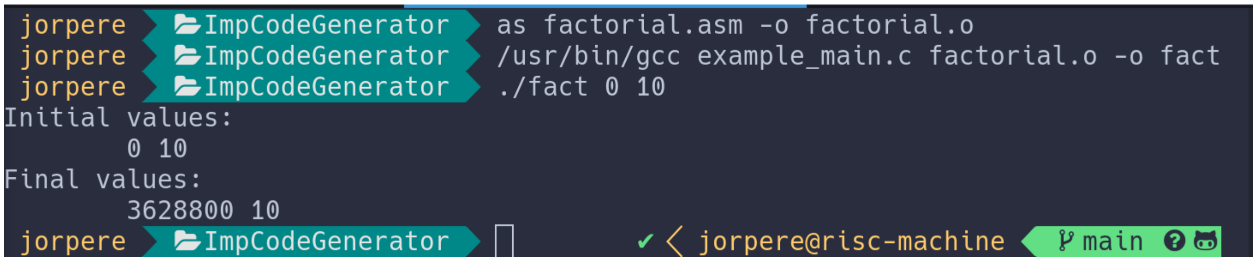
This is the file that links against the assembly code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

void run_imp_actual(int64_t* values);

void print_values(int64_t* arr, size_t N) {
    for(size_t i = 0; i < N; i++) {
        printf("%lli ", (long long)arr[i]);
    }
    printf("\n");
}

int main(int argc, char const* argv[]) {
    // Holds values
    int64_t values[16384] = {0};
    for (int i = 1; i < argc; i++) {
        values[i - 1] = atoll(argv[i]);
    }

    printf("Initial values:\n\t");
    print_values(values, argc - 1);
    run_imp_actual(values);
    printf("Final values:\n\t");
    print_values(values, argc - 1);
    return 0;
}
```

/programs/factorial.imp

```
{-
Computes the factorial of the number stored in x and leaves the result in output:
-}
y := x;
z := 1;
while y > 1 do
  z := z * y;
  y := y - 1
od;
y := 0;
output := z
```

Here is a screenshot of the factorial program running on the risc-v machine, after being assembled by assembler.cpp and linked against example_main.c



```
 jorpere  ImpCodeGenerator  as factorial.asm -o factorial.o
 jorpere  ImpCodeGenerator  /usr/bin/gcc example_main.c factorial.o -o fact
 jorpere  ImpCodeGenerator  ./fact 0 10
Initial values:
        0 10
Final values:
        3628800 10
 jorpere  ImpCodeGenerator            ✔ jorpere@risc-machine  main
```