AU Compilation, 2023

Jul 31, 2024

CONTENTS

Welcome to AU Compilation. This online reference manual contains supplimentary course material.

Part I

Assignments

CHAPTER

ONE

ARITHMETIC EXPRESSIONS IN OCAML AND X86 ASSEMBLY

1.1 Assignment overview

This assignment covers the topics of program representation via Abstract Syntax Trees (ASTs), implementation of an evaluator for arithmetic expressions in OCaml, and pretty printing. There are 3 tasks and 9 questions in this assignment. One of the questions is marked for *glory*. This means it is optional and you should do it if you feel ambitious.

1.1.1 What you need to get started

1. Make sure you have a working installation of OCaml and clang. The *course development container* should have the necessary pre-requisites already installed.
2. We assume that by now you are familiar with OCaml basics.
3. We assume that you are familiar with basic assembly programming.
4. Download the auxiliary file `x86.ml` from Brightspace. Do *not edit* it.

1.1.2 What you need to hand in

Please hand in a `.zip` file containing the following

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each question *and for each task*, briefly (1 – 4 sentences) describe your implementation or answer. Write concisely.
2. All the source files needed to reproduce your solution. Two pieces of code from the assignment description are to be copied into your solution. These are
    1. The OCaml code for the `expr` type declaration below.
    2. `x86.ml` OCaml file. Do not modify this file.

    We do not prescribe how to organize your solution in terms of modules or folder structure, but note that a relatively simple organization should be sufficient for this assignment. Your solution must work with OCaml 5.0.0 on a modern x86-64 Linux system, such as Ubuntu 22.04 LTS (for example in the course's Docker container).

Running `make` in the root folder of your project *must* compile your project successfully.

Important

Make sure to understand all the code you hand in, including what is copied from here.

1.2 Arithmetic expressions

We start with the study representing arithmetic expressions. For this, we introduce the following OCaml declarations.

```
(* -- Use this in your solution without modifications -- *)
(* Defining the type for binary operations *)
type binop = Add | Sub | Mul | Div

(* Defining the type for arithmetic expressions *)
type expr
  = Int of int                        (* Integer constant *)
  | BinOp of binop * expr * expr      (* Binary operation *)
```

Here are a few examples of values of type `expr`.

```
let expression_01 = Int 5                                          (* 5       *)

let expression_02 = BinOp (Add, Int 1, expression_01)              (* 1+5     *)

let expression_03 = BinOp (Mul, BinOp (Add, Int 2, Int 2), Int 2)  (* (2+2)*2 *)

let expression_04 = BinOp (Add, Int 2, BinOp (Mul, Int 2, Int 2)) (* 2+2*2   *)
```

Given just the above, there is not much we can do yet. Our first task is to write an evaluator.

1.2.1 Evaluator

Task 1: Evaluator for arithmetic expressions

Write a function `eval` that has type `expr -> int` for evaluating arithmetic expressions.

As we can see from the type, the function `eval` should take one argument of type `expr` and return an integer. For example, `eval expression_03` should return 8. As you work on this task, answer the following questions:

Question 1
Does this function need to recursively explore its argument, and why (or why not)?

Question 2
Why does this function have the return type int? What other return types may be suitable?

Question 3
How does your evaluation handle the case of division by zero? Note that it may be just fine to not special-treat division by zero, but it is important you understand what actually happens at runtime.

1.2.2 Pretty printer

A *pretty printer* is a function that takes an internal representation of a data structure, e.g., an AST such as `expr` and produces its textual representation, e.g., as a string.

Task 2: Pretty printer for arithmetic expressions

Write a function `string_of_expr` that has type `expr -> string` for pretty printing arithmetic expressions.

The output of the pretty printer should be in the format that is accepted by OCaml's REPL, such as utop, using infix notation. For example, the following are all acceptable results for `string_of_expr expression_04`

```
2+2*2
2+(2*2)
(2+(2*2))
((2)+((2)*(2)))
```

As you work on this task, feel free to define additional functions, if needed. You may use the built-in function `string_of_int` for converting integers to strings, and string concatenation operation `^`, e.g., `"Hello " ^ "world"`. As you work on this task, answer the following questions:

Question 4
Are the functions you have defined recursive, and why (or why not)?

Question 5 (glory)
Make sure you avoid unnecessary parentheses, assuming standard precedence. Explain how you implement this.

1.2.3 Using both the evaluator and the pretty printer

With both the evaluator and the pretty printer ready, they can be put together, for example using the following expression.

```
let expressions
        = [ expression_01 ; expression_02 ; expression_03 ; expression_04] in
let print_expr e = Printf.printf ("%s = %d\n") (string_of_expr e) (eval e) in
List.map print_expr expressions
```

The exact output will depend on your implementation of the printer. Here is one example output.

```
5 = 5
1 + 5 = 6
(2 + 2) * 2 = 8
2 + 2 * 2 = 6
```

1.3 x86 representation

We now switch our attention to a low-level program representation. Our main tool here is an AST for x86 programs, and its associated pretty printer, provided in the file `x86.ml`. We use this AST representation to construct assembly programs in OCaml. There are several advantages to using an AST instead of directly representing assembly programs as strings.

1. *Separation of concerns.* A dedicated representation lets us focus on the semantics of the programs we produce, while the pretty printer handles the syntactic details.
2. *Well-formedness.* A well-designed AST helps in avoiding nonsensical programs, e.g., referring to a non-existing register. Additional well-formedness checks can be programmed as separate passes over the AST.
3. *Opportunities for further analysis.* AST data structures are essential for implementing other analyses or optimization, e.g., dead-code elimination.

Checkpoint

Before proceeding further, please download and study the provided `x86.ml` file.

1.3.1 Using x86 AST

Next, we see how to use the x86 AST. We proceed with the following steps.

1. Construct an OCaml value corresponding to an assembly program, and print it.
2. Compile the resulting assembly program.
3. Run the compiled binary and output the return value.

Producing assembly program programmatically

The following OCaml declaration constructs an AST for an assembly program that returns value 23.

```
let asm_example =
  let open Asm in                    (* Open Asm module locally; this brings
                                        the helper functions from that module
                                        into the local scope *)
   [ gtext "example"                 (* Global label *)
     [ (Movq, [~$ 23; ~% Rax])       (* Store 23 in register %rax.
                                        This is the register we must use
                                        for returning values from a function
                                        according to System V ABI *)
     ; (Retq, [])                    (* Return instruction *)
   ]]
```

As you study the above example, answer the following questions:

Question 6
What is the OCaml type of `asm_example`?

Question 7
How would you rewrite this to use the `Asm` module without the local module open directive?

Question 8
Could we have used `text` instead of `gtext`? Why?

> **Note**
>
> Some platforms, e.g., Intel Macs, require the exported labels to be prefixed with an underscore, e.g., `_example`. This is known as *name mangling*, and has been historically used to prevent collision with reserved names [Lev00]. Linux does not require this.

We can pretty print the above program as follows

```
let _ = Printf.printf "%s\n" (string_of_prog asm_example)
```

This will produce the output

```
        .text
        .globl  example
example:
        movq    $23, %rax
        retq
```

Compiling the assembly program

To run this assembly program, we follow the approach where we link it against a C program that calls the function that we have declared, in this case: `example`. This step depends on the following.

1. The text of the assembly program is saved in a `.s` file so that it can be processed by clang, e.g., `example.s`.
2. There is a C wrapper that calls our example function. We can use a simple program like this

   ```
   /* main.c: C wrapper for our assembly program */
   #include <stdio.h>
   extern int example (); /*  the name of our function */

   int main() {
       int result = example();
       return result;
   }
   ```

   Save this file as `main.c`
3. Call clang to compile and link both files

   ```
   clang main.c example.s
   ```

> **Note**
>
> Arm Mac users (M1/M2), prepend the call to clang with `arch -x86_64`, e.g., `arch -x86_64 clang main.c example.s`. You need Rosetta 2 installed for this. See also the note on mangling above.

This will create a binary with the default name `a.out`. If you want a different name for the output, use the compiler `-o` flag.

Executing the produced binary

We run the produced binary as follows

```
./a.out
```

We can inspect the output by prompting the exit status shell variable `$?`. For example, if we run

```
echo $?
```

immediately after the previous command, we should get the value `23`.

Handling return values greater than 255

If you modify the program to return a value greater than 255, e.g., replace the literal 23 with 2023, you will run into an issue: POSIX reserves only one byte for process return values. This restriction is justified in the context of operating systems because process return values are typically used for distinguishing normal vs. erroneous exits.

Question 9

What output do you get from `echo $?` when returning the value 2023?

To handle results greater than 255, we need a workaround. Instead of returning the value to the OS, we will print it on the console. We extend our C program with a function for printing integers, `print_int`, that can be called from the assembly. The resulting OCaml and C programs look as follows.

```
(* Ocaml expression that constructs the assembly program. *)
let dolphin_main =
  let open Asm in
   [ gtext "example" [
      (Pushq, [~% Rbp])                (* Stack alignment before call *)
   ;  (Movq, [~$ 2023;  ~% Rdi ] )     (* System V ABI requires passing the
                                          first argument via register %rdi *)
   ;  (Callq, [ ~$$ "print_int"])      (* Calling the C function to print *)
   ;  (Movq, [~$ 0; ~% Rax])           (* Return 0 to indicate normal exit *)
   ;  (Popq, [~% Rbp])                 (* Stack re-alignment *)
   ;  (Retq, [])
   ]]
```

```
/* main.c: C wrapper for assembly programs */

#include <stdio.h>
extern int example ();

int main() {
    int result = example();
    return result;
}

void print_int (int x) {
    printf ("%d\n", x);
}
```

If you repeat the steps earlier with pretty printing of the assembly text, saving it as `example.s`, and compiling and running the resulting binary `./a.out`, we will get the output

```
2023
```

It is no longer necessary to `echo $?` other than to ensure that the program exits normally with 0.

Finally, now that we have introduced both the high-level and the low-level representations, we can work with both of them.

Task 3: Translate the following 5 OCaml expressions to assembly

Consider the following OCaml declarations

```
let task3_exp1 = BinOp (Add, Int 20, BinOp (Mul, Int 26, Int 58))
let task3_exp2 = BinOp (Mul, Int 5, BinOp (Div, Int 1, Int 10))
let task3_exp3 = BinOp (Sub, Int 31, Int 870)
let task3_exp4 = BinOp (Mul, BinOp (Add, BinOp (Div, Int 6, BinOp (Add, Int 10, Int␣
 ↪49)), Int 10),
          BinOp (Add, BinOp (Sub, BinOp (Mul, Int 70, Int 77), BinOp (Div, Int 12,␣
 ↪Int 9)), Int 5))
let task3_exp5 = BinOp (Sub, BinOp (Div, Int 34, Int 72), BinOp (Div, Int 17, Int 46))
```

Create a list of OCaml values, named `task3_asm1`, `task3_asm2`, … that are x86 assembly translations of these arithmetic expressions. These translations are to be done manually, without introducing any automation. Your translation should not implement any shortcuts or optimizations, e.g., returning pre-computed constant outputs (we study optimizations later in the course). Neither should it be unnecessarily clunky.

For each expression, cross-reference the output from your evaluator with the output from running the translated assembly. Describe in your report how you implement cross-referencing, and whether you found any inconsistencies during this process.

CHAPTER

TWO

EXPRESSION PROGRAMS AND X86 ASSEMBLY

2.1 Assignment overview

This assignment covers the language of expression programs, their semantic analysis, and compilation to a subset of x86. There are 4 tasks and 12 questions in this assignment. Questions marked for *glory* are optional. Do them only if you feel ambitious and have the time.

2.1.1 What you need to get started

1. Everything from *Assignment 1*, including `x86.ml` file that is the same as before.
2. Programming in OCaml. We recommend you familiarize yourself with everything up to and including Section 5, plus Section 7.1, of the OCaml book. This includes the following concepts.
    1. Programming with environments (aka associative maps) in OCaml. See the following resources in the OCaml book:
        - Chapter 3.8 has a basic example of creating environments using associative lists.
        - Chapter 5.9 explains how to use the Map functionality from the OCaml standard library.
    2. Higher-order programming in OCaml, e.g., using functions such as `List.fold_left`. See Chapter 4 of the OCaml book.
    3. Basics of modular programming in OCaml, in particular the use of functors when working with the standard library. See Chapter 5 of the OCaml book.
    4. Programming with references in OCaml. See Chapter 7.1 of the OCaml book.

2.1.2 What you need to hand in

Please hand in a `.zip` file containing the following

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each question *and for each task*, briefly (1 – 4 sentences) describe your implementation or answer. Write concisely.
2. All the source files needed to reproduce your solution. Three pieces of code from the assignment description are to be copied into your solution. These are
    1. The OCaml code for the `eprog` type declaration below in Section 2.2.
    2. The OCaml code for the `semant_result` type declaration in Section 2.3.
    3. `x86.ml` OCaml file. Do *not modify* this file.

Your project *must compile*.

We recommend that you organize your codebase into several files (remember also that each file is a module in OCaml): one for the definition of `eprog` and its pretty printing, one for example programs, one for the semantic analysis, one for the evaluation, and one for the translation. Please take a look at the Dune build management for OCaml. A relatively simple dune configuration should be sufficient for this project. Your solution must work with OCaml 5.0.0 on a modern x86-64 Linux system, such as Ubuntu 22.04 LTS.

Important

Make sure to understand all the code you hand in, including what is copied from here.

2.2 Expression programs

This assignment revolves around a simple programming language that we call the language of *expression programs*. This language is a direct extension of the language of arithmetic expressions from the first assignment. It has the following AST.

```
(* -- Use this in your solution without modifications *)
(* Defining the type for binary operations *)
type binop =
  | Add | Sub | Mul | Div

type varname = string                (* variable names are strings *)

(* Defining the type for arithmetic expressions *)
type expr =
  | Int of int                       (* Integer constant *)
  | BinOp of binop * expr * expr     (* Binary operation *)
  | Var of varname                   (* Variable lookup  *)


(* Defining the type of statements *)
type estmt =
  | Val of varname * expr            (* Binding variable to a value *)
  | Input of varname                 (* Input statement *)

(* Expression program is a list of statements
   followed by an expression *)
type eprog = estmt list * expr
```

The core change from the arithmetic expressions is the introduction of immutable variables. There are two ways to bind a variable to a value.

1. The input statement reads an integer value from the console and binds it to the variable.
2. The val statement evaluates an arithmetic expression and binds the result to the variable.

We refer to the above operations as *expression statements*, and represent them using the type `estmt`.

Because variable bindings are immutable, they can only be bound once. To refer to variables, the declaration for expressions now includes a variable lookup constructor, `Var of varname`. An expression program is a list of statements (that is, either input or val bindings) followed by one return expression. The type `eprog` captures this in its definition as a tuple consisting of two items

1. A list of expression statements `estmt`, and

2. An expression

For representing variables, we use the built-in OCaml string type.

2.2.1 Concrete syntax

For concrete syntax, we adopt the following notation

1. Input statements are written using the syntax `input x`
2. Value binding statements are written using the syntax `val x = e`, where `e` is the binding expression.
3. Return expression is written as `return e`.

For example, the following concrete syntax

```
input x
val y = x + 1
return x + y
```

represents the program given by this AST

```
let eprog_01: eprog = (
      [ Input "x" ; Val ("y", BinOp (Add, Var "x", Int 1)) ],
      BinOp (Add, Var "x", Var "y"))
```

Task 1: Pretty printer for expression programs

Write a function `string_of_eprog` that has type `eprog -> string` for pretty printing expression programs using the above notation. You probably want to reuse and extend the pretty printer for expressions you have written in the first assignment.

2.3 Semantic analysis

Semantic analysis is a compilation phase that reports type and other semantic errors in the program. In our case, we have two kinds of errors to report

1. Undeclared variables. An important consideration in language design is what to do with undeclared variables. For example, the undeclared variable `x` in the program

   ```
   let y = x + 1
   return y
   ```

   In this assignment, we want to prevent such programs and therefore report undeclared variables as errors.

2. Duplicate bindings. Because our language does not have any notion of nested scopes and mutation, there is little value in duplicate bindings, e.g., the following program will be rejected because of the duplicate binding of `y`

   ```
   input y
   let y = x
   return y + 2
   ```

We use the following Ocaml declarations for error reporting

```
(* -- Use this in your solution without modifications --  *)
type semant_error
  = Undeclared of varname
  | Duplicate of varname


type semant_result
  = Ok
  | Error of semant_error list
```

Note

If your semantic analysis and expression program declarations live in different modules (as we suggested earlier) remember to open the expression program module.

Task 2: Semantic analysis

Write a function `semant` that has type `eprog -> semant_result` that returns whether the program has any type errors, in which case the errors are collected in a list of type `semant_error`.

Question 1
Does `semant` need to be recursively inspect its argument? Why or why not? What auxiliary functions do you define? Are any of them recursive or not and why?

Question 2
Describe how you keep track of variable declarations in your implementation. What data structure(s) do you use?

Question 3 (glory *)
Implement the warning of unused variables. Extend the definition of `semant_result` to incorporate a warning possibility and extend the implementation of `semant` appropriately.

2.4 Evaluator

Our evaluator takes programs that are accepted by the semantic analysis and runs them.

Task 2: Evaluation of expression programs

Write a function `eval` that has the type `eprog -> int` for evaluating expression programs. For evaluating the input statements, you can use something like the following OCaml code

```
let eprog_input()
   = Printf.printf "Please enter an integer: " ; read_line () |> int_of_string
```

Question 4
In the above code, `;` is the OCaml sequencing and `|>` is the OCaml pipeline operator. How would you write the implementation of `eprog_input` using just the `let` expressions without these operators?

As you further work on this task, answer the following questions.

Question 5 (glory)
The function `int_of_string` that we use above raises an exception if its argument is not an integer. How would you handle this situation?

Question 6
What data structure(s) do you use in the implementation of the interpreter?

Question 7
What runtime errors are possible during the execution of your interpreter? Are they preventable, and how?

2.5 Compiling to x86 assembly

Task 3: Compiling expression programl to x86

Write a function `eprog_to_x86` that has the type `eprog -> X86.prog` that compiles expression programs to x86. For reading input from the console, use the following C function

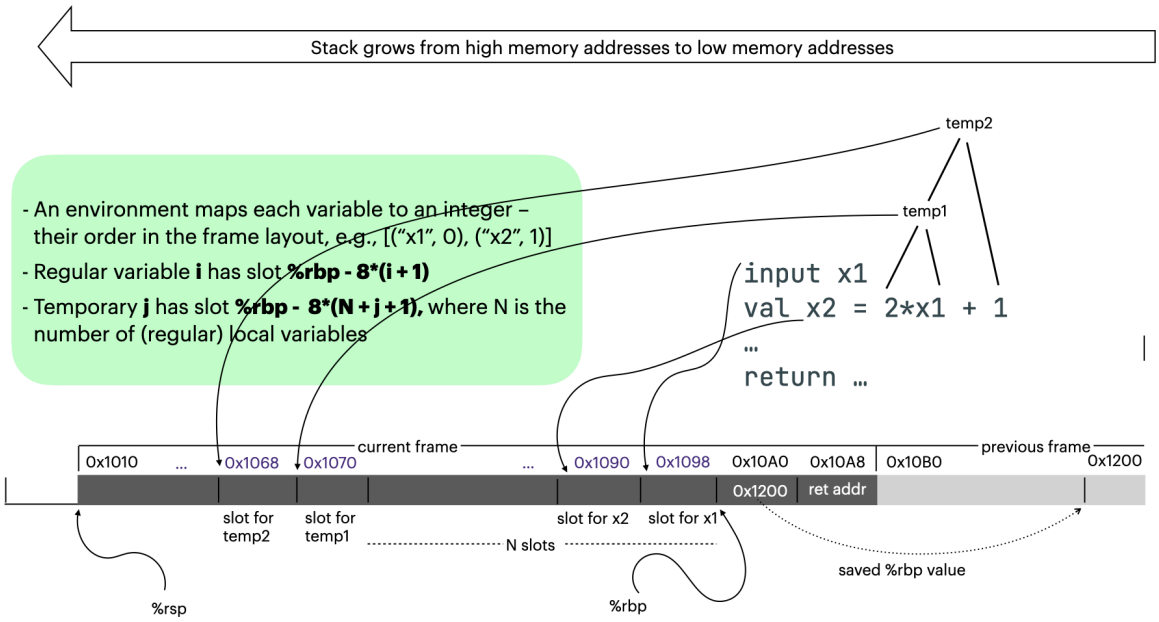
```c
#include <stdio.h>          /* make sure these two includes are    */
#include <inttypes.h>       /* present in the start of your C file */

int64_t read_integer () {
    int64_t value;
    printf("Please enter an integer: ");
    scanf("%" PRId64 "" , &value);
    return value;
}
```

For reporting the result of the program, use the function `print_int`, as in *Assignment 1*.

The following two sections describe one potential approach to this task. Note that this is only one of the ways of implementing this task. Other approaches are possible, and you are welcome to pursue them in your assignment; in that case it is still a good idea to understand the approach outlined below.

2.5.1 Stack organization via spilling

A simple (admittedly quite inefficient) way of x86 code generation is via *spilling*, where each local variable has a reserved slot on the stack. Additionally, the results of the intermediate binary operations are likewise stored on the stack in a reserved location. The following image illustrates such a stack organization.



2.5.2 Implementation via spilling

To implement code generation via spilling, we need a number of building blocks.

- Introduce a `layout` type that is an *associative map* for mapping variables to their respective slot positions on the stack.
- Introduce a counter for keeping track of the temporaries for the intermediate results of the binary operations.
- Break down your implementation into auxiliary functions, such as
  - `cg_stmt` of type `layout -> estmt -> X86.ins list` that takes the layout, a single statement, and returns a list of instructions corresponding to the assembly code for that statement. The layout is needed to store the result of the statement in the right stack slot.
  - `cg_expr` of type `layout -> expr -> X86.ins list * X86.operand` that takes the layout, a single expression, and returns a pair of the instructions corresponding to the assembly code for that expression and an *X86 operand* that has the result of that expression. The layout is needed to read variables from the stack.

    This programming pattern of returning a list of instructions (or something akin to that) and an operand, indicating where to find the result after those instructions are executed, is commonplace for "flattening" phases, as is the case here because we translate from a tree data structure (expressions) to a list data structure (assembly instructions).

    For example, for integer expression (`Int n`), function `cg_expr` can return a pair `[], (~$ n)`. For binops, the instructions list will include moving data into the right registers, doing the binop, and storing the result on the stack in a temporary slot. The operand will be the offset from the base pointer to the temporary. For example, if the temporary has index `j` the operand is the displacement `X86.Ind3 (Lit(-8*(N+j+1)), Rbp)`, where `N` is the number of the regular locals. Remember to increase the temp counter, when needed.

One more concern is what information we need to generate the correct function prologue and epilogues. For example, the function prologue must include an instruction for extending the stack – such as subtracting from the `%rsp` register.

But how much to subtract? The answer depends on two things.

1. The number of the variables in the source program. This is easy to compute by, e.g., taking the length of the list of expression statements.
2. The number of temporaries needed for the code generation of all of the expressions. This number is not as readily available, because it depends on the expressions in the rest of the program. This means that the code generation of the prologue might need to be postponed until the code generation of the rest of the program is complete. This is an interesting, but not unusual in compilation, phenomenon: the prologue – an early section of the x86 program – turns out to be the one generated at the very end of the translation process!

As you further work on this assignment, answer the following questions.

Question 8 (glory *)
The C function `scanf` that we use here provides no means of error reporting or error detection when the input is malformed. How would you implement a robust input of integers from a console in C?

Question 9
What data structure(s) do you use in the implementation of the compiler?

Question 10
Describe inefficiencies that are present in your solution, if any.

Question 11 (glory **)
Pick one or several optimizations to implement in your compilation, and generate optimized code. Explain which optimizations you are implementing and how they affect the produced code. What data structures and intermediate representations do you use in the implementation of these optimizations?

Extra-glory: can you benchmark the quantitative improvements from your optimizations? Explain your benchmarking approach.

Task 4: Putting it all together, examples, and cross-testing

In this task, we put together all the ingredients developed in the earlier tasks.

1. Write a function `interpret` of the type `eprog -> unit` that takes an expression program as its input, runs the semantic analysis, and depending on the result of the semantic analysis either prints out the errors or proceeds to evaluate the program using the `eval` function you have implemented.
2. Write a function `compile` of the type `eprog -> unit` that takes an expression program as its input, runs the semantic analysis, and depending on the result of the semantic analysis either prints out the errors or proceeds to compile the program into assembly.
3. Write 5 example programs that cover all the features of the language.
4. Cross-test that both evaluation and compilation agree on the results.

Question 12.
Why are your example programs good programs for testing your implementations? Did you discover any bugs during the cross-testing?

CHAPTER

THREE

LLVM-- PROGRAMMING

3.1 Assignment overview

This assignment covers writing and debugging `LLVM--` programs with a focus on loops and conditionals. There are 3 tasks and 6 questions in this assignment. Questions marked for *glory* are optional. Do them only if you feel ambitious and have the time.

3.1.1 What you need to get started

Make sure you understand everything from the lectures on `LLVM--` (the lectures on Friday, Sep 15th, and Wednesday, Sep 20th). That is, everything in *`LLVM--`* chapter except for aggregate and named types, GEP, and phi nodes.

Important

Revisit *Arithmetic Expressions in OCaml and x86 Assembly* and be sure you recall the specialties of your setup with respect to architecture and/or operating system.

3.1.2 What you need to hand in

Please hand in a `.zip` file containing the following

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each task and question, briefly (1 – 4 sentences) describe your implementation or answer. Write concisely.
2. All the source files needed to reproduce your solution. This also includes the `C` code provided. Please explain in your report how the solution could be reproduced, e.g., calling `make` (if you have made a `Makefile`), the command line to call `clang`, etc. You do not need to provide any code for Task 3. Just write the answer in the report.

Note

If you are answering any of the glory questions, please submit them as separate solutions, i.e., in a separate, clearly marked subfolder in the `.zip` file of your submission.

Important

Make sure to understand all the code you hand in, including what is copied from here.

3.2 Write LLVM-- by hand

Consider the following Dolphin program.

```
void number_perfect(n : int){
  var i = 1;
  while(i <= n){
    var temp = 0;
    for (var j = 1; j < i; j = j + 1){
      let d = i % j;
      temp = temp + (d == 0 ? j : 0);
    }
    if(temp == i)
      print_perfect(i);
    i = i + 1;
  }
}
```

The function `print_perfect` is defined as follows in a `C` file that is to be linked with the LLVM file you produce:

```
#include <stdio.h>         /* make sure these two includes are    */
#include <inttypes.h>      /* present in the start of your C file */

void print_perfect(int64_t i) {
  printf("%" PRId64 ": perfect\n", i);
}

int main() {
  number_perfect(10000);
  return 0;
}
```

The following is the expected output:

```
6: perfect
28: perfect
496: perfect
8128: perfect
```

Task 1: Rewrite in your favorite language

Pick your favorite programming language (second-favourite if your favourite is already Dolphin) and write the program above, including the part written in C above. In your report, explain briefly, in a short paragraph, what the program does. Write comments explaining the most important lines of code in the code you hand in (the code in your favorite language). Try to keep your program as close as you can to the code given above. A note about perfect numbers: a number is said to be perfect if it is equal to the sum of all its proper divisors, i.e., not the number itself.

Task 2: Translate to LLVM--

Translate the Dolphin code above, the function `number_perfect`, into `LLVM--`.

Remember to test your `LLVM--` program by linking and running with different inputs by changing the number with which the function `number_perfect` is called (on the `C` side). Recall that this can be done using the following commands:

```
clang perfect.ll main.c
./a.out
```

where `perfect.ll` is your `LLVM--` code and `main.c` is the `C` code above.

Question 1
How many basic block does your program have? Explain how you count basic blocks.

Question 2
How do you handle division by zero? Do you produce code for checking that the divisor is non-zero? Why?

Question 3 (glory **)
Use `Phi` nodes to translate the ternary operator (`?:`) above.

Question 4 (glory *)**
This is an extension of question 3 above. Write your `LLVM--` code without the use of the `alloca` instruction.

Task 3: Debug `LLVM--`

Consider the following Dolphin program.

```
int rec_fun(acc : int, n : int){
  if(n > 0)
    return rec_fun(acc + 2, n - 1);
  return acc;
}
```

First, convince yourself that when called `rec_fun(0, -10)` will return 0.

A buggy compiler could produce the following code for this program:

```
define i64 @rec_fun (i64 %acc, i64 %n) {
 %n6 = alloca i64
 %acc5 = alloca i64
 store i64 %acc, i64* %acc5
 store i64 %n, i64* %n6
 %load_local_var7 = load i64, i64* %n6
 %arith_comp_op8 = icmp slt i64 %load_local_var7, 0
 br i1 %arith_comp_op8, label %ifthenelse_true_branch9, label %ifthenelse_false_
 ↪branch10
ifthenelse_true_branch9:
 %load_local_var12 = load i64, i64* %acc5
 %arith_bin_op13 = add i64 %load_local_var12, 2
 %load_local_var14 = load i64, i64* %n6
 %arith_bin_op15 = sub i64 %load_local_var14, 1
 %call16 = call i64 @rec_fun (i64 %arith_bin_op13, i64 %arith_bin_op15)
 ret i64 %call16
after_return17:
 br label %ifthenelse_merge11
ifthenelse_false_branch10:
 br label %ifthenelse_merge11
ifthenelse_merge11:
 %load_local_var18 = load i64, i64* %acc5
 ret i64 %load_local_var18
after_return19:
 unreachable
}
```

In the `LLVM--` code above, when the function `rec_fun(0, -10)` is called it crashes with a segmentation fault.

Identify the issue in the code above that causes the segmentation fault. Explain why the segmentation fault occurs.

Question 5

Make an educated guess as to what the compiler has done wrong to produce such a buggy code (this is about the bug in the compiler that leads to the bug in the code of Task 3).

Question 6

Find at least one other way a subtle mistake in the code generation phase of the compiler can cause a segmentation fault in a program like the one above.

CHAPTER

FOUR

DOLPHIN – PHASE 1

Attention

This is a group assignment. You should have formed groups by now. If not, please do so as soon as possible. There will an announcement on BrightSpace explaining how you should register your groups on BrightSpace.

4.1 Assignment overview

This assignment covers translating a fragment of Dolphin language, in the AST form, into `LLVM--`. There are 3 tasks and 3 questions in this assignment. Questions marked for *glory* are optional. Do them only if you feel ambitious and have the time.

4.1.1 What you need to get started

- For `LLVM--`: make sure you understand everything from the lectures on `LLVM--` (the lectures on Friday, Sep 15th, and Wednesday, Sep 20th). That is, everything in the *`LLVM--`* chapter except for aggregate and named types, GEP, and phi nodes.
- For semantic analysis: this will be covered in lectures in week 39.
- For programmatic `LLVM--` code generation: this is partly covered in TA classes in week 39. This will be complemented in lectures in week 39. (This part, in spirit, resembles programmatic generation of x86 code we hand in *assignment 2*.)
- For OCaml, the assignment uses OCaml records which are covered in Section 3.4 of the OCaml book.

Important

You will need files `symbol.ml`, `symbol.mli`, `cfgBuilder.ml`, `cfgBuilder.mli`, and `ll.ml` as they have been released in the TA classes.

4.1.2 What you need to hand in

Please hand in a `.zip` file containing the following

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each task and question, briefly (1 – 4 sentences) describe your implementation or answer. Write concisely.
2. All the source files needed to reproduce your solution. This also includes the `C` code provided. Please explain in your report how the solution could be reproduced, e.g., calling `make` (if you have made a `Makefile`), the command line to call `clang`, etc.

> **Note**
>
> If you are answering any of the glory questions, please submit them as separate solutions, i.e., in a separate, clearly marked subfolder in the `.zip` file of your submission.

> **Important**
>
> Make sure to understand all the code you hand in, including what is copied from here. (The code for pretty printing (typed) ASTs is an exception here; see the *appendix* below.)

4.2 The Abstract Syntax Tree (AST) of Dolphin (phase 1)

In this phase, a Dolphin program is simply a sequence of statements. Intuitively, this is to be understood as the body of the main function.

The OCaml types for the AST describing programs is given below:

```
(* -- Use this in your solution without modifications *)
type ident = Ident of {name : string;}

type typ = | Int | Bool

type binop =
| Plus | Minus | Mul | Div | Rem | Lt | Le | Gt | Ge | Lor | Land | Eq | NEq

type unop = | Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident

type statement =
| VarDeclStm of {name : ident; tp : typ option; body : expr}
| ExprStm of {expr : expr option}
```

(continues on next page)

(continued from previous page)

```
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option}
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type program = statement list
```

Action item

Put the AST declarations above in a module called `Ast`. That is, make a file called `ast.ml` and copy the code above into it. Do **not** change the code above.

The types `ident` (program identifiers), `typ` (types), `binop` (binary operators), and `unop` (unary operations) are self-explanatory.

Expressions consist of the following (in the order they appear in type `expr` above):

- Integer literals
- Booleans literals
- Binary operations consisting of the a left operand, a binary operation, and a right operand
- Unary operations consisting of a unary operation and an operand
- L-values, i.e., values that can appear on the left-hand-side of an assignment expression (see below), in our fragment the only possible l-values are variables
- Assignments consisting of an l-value, and right-hand-side expression assigned to the l-value in question
- Function call consisting of a function name (an identifier), and a list of arguments. **Note**: even though the phase-1 fragment of Dolphin has no function definitions, programs do have access to a very limited standard library, customised for this phase.

Statements consist of the following:

- Variable declarations consisting of a name (identifier), *optionally* a type, and a body (an initialization expression). The optionality of the type annotation for variable declarations can be seen in the difference following two declarations of variable `x` in `var x = 10;` and `var x : int = 10;` as written in Dolphin's syntax.
- Expression statements which allow us to use expressions as statements (this is restricted in *semantic analysis*). **Note** that the expression is optional. A `None` expression statement corresponds to the empty statement, i.e., `;` on its own in languages like C and Java.
- If statements where `cond` is the condition (an expression), `thbr` is the statement for the then branch, and `elbro` is the *optional* else branch.
- Compound statements: these correspond to statement blocks. Think of code blocks in C and Java enclosed in curly braces. **Note**: compound statements serve two purposes. They delimit the scope of declarations, and allow us to use a statement blocks inside other statements, e.g., a statement block as the body of the then, or else branch of an if statement.
- Return statement where the result of an expression is returned. (Recall that in this phase, the program being the body of the main function should return an integer.)

4.3 The Typed Abstract Syntax Tree (AST) of Dolphin (phase 1)

The typed AST differs from the AST in that it includes all the necessary type information for (LLVM) code generation. The differences between the two notions of AST can be subtle. Please pay attention to the details.

The OCaml types for the typed AST describing programs is given below:

```
(* -- Use this in your solution without modifications *)
module Sym = Symbol

type ident = Ident of {sym : Sym.symbol}

type typ = Void | Int | Bool | ErrorType

type binop = Plus | Minus | Mul | Div | Rem | Lt | Le | Gt | Ge | Lor | Land | Eq |␣
↪NEq

type unop = Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr; tp : typ}
| UnOp of {op : unop; operand : expr; tp : typ}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr; tp : typ}
| Call of {fname : ident; args : expr list; tp : typ}
and lval =
| Var of {ident : ident; tp : typ}

type statement =
| VarDeclStm of {name : ident; tp : typ; body : expr}
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option}
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type param = Param of {paramname : ident; typ : typ}

type funtype = FunTyp of {ret : typ; params : param list}

type program = statement list
```

Action item

Put the AST declarations above in a module called `TypedAst`. That is, make a file called `typedAst.ml` and copy the code above into it. Do **not** change the code above.

Below, we enumerate and explain the main differences between (untyped) ASTs we saw earlier and typed ASTs:

- Types now include the `Void` type, and the `ErrorType` type. While the source (ASTs) do not have void in them, library functions, which *are* featured in this phase 1 Dolphin fragment, can involve the `Void` type. It is needed for *error* recovery where the compiler does not merely stop when it first encounters a type error in the program, but continues and reports all errors that it detects.
- Identifiers, instead using strings for names, use symbols. Symbols are much more efficient to use than strings.

- Many expressions, e.g., unary and binary operations, now also include a type. This extra type, is the type of the entire expression, that is, a binary operation for comparison, with both left and right expressions bing integers, will have its as `Bool`. Note also how variable declarations, which *optionally* included types in ASTs, always include a type in typed ASTs (it is not an option anymore). Note that assignments are also accompanied by a type. The reason for this is that an assignment, in addition to storing the right-hand-side value into the relevant memory address, also produces this right-hand-side value as a result. That is, the expression `x = y = 1 + 4` is a valid expression, which should be understood in two steps, in the first step assigns 5 to `y` and results in 5, in the second step, the result of the first step, i.e., 5, is assigned to `x`. The following is also a valid expression: `x = (y = 4 - 2) + 7`. It assigns 2 to `y` and 9 to `x`.

Inclusion of the typing information allows one to directly determine the type of an expression with a shallow inspection of the expression when necessary, e.g., when we need to decide what LLVM type to use when generating `LLVM--` code. This can be done using the following OCaml functions:

```
let rec type_of_expr = function
  | TAst.Integer _ -> TAst.Int
  | TAst.Boolean _ -> TAst.Bool
  | TAst.BinOp {tp; _} -> tp
  | TAst.UnOp {tp; _} -> tp
  | TAst.Lval lvl -> type_of_lval lvl
  | TAst.Assignment {tp; _} -> tp
  | TAst.Call {tp; _} -> tp
and type_of_lval = function
  | TAst.Var {tp; _} -> tp
```

4.4 Semantic analysis of Dolphin programs (phase 1)

The first task asks you to implement semantic analysis. The semantic analysis must check the program semantically. That is, it must make sure that the program satisfies to the criteria for being a valid program that we can produce executable code for. At a high level view this process takes an AST (something of type `Ast.program` above) and produces a typed AST (something of type `TypedAst.program` above). This requires to determine all the necessary types to produce a typed AST, e.g., the types of all variable declaration, if omitted, must be *inferred*. This, however, **is not** the only requirement. There are a few extra requirements, some of which are specific to this phase. In particular, not all OCaml values of type `TypedAst.program` are semantically valid programs. The following are extra conditions, that should be **guaranteed** by semantic analysis, and can therefore be **assumed** when producing `LLVM--` code.

1. No expression, except for a call to a function with `Void` return type can have type `Void`. This also includes the type (inferred for) declared variables. That is, a program `var x = f();` where `f` is a function with `Void` return type is not semantically valid. Hence, one needs to be careful when inferring types.
2. Arithmetic operations `Plus`, `Minus`, `Mul`, `Div`, and `Rem` can only be applied to integer expressions and produce integer results.
3. Comparison operators `Lt`, `Le`, `Gt`, and `Ge` (in this phase) can only be applied to integer operands and result in a boolean.
4. Comparison operators `Eq` and `NEq` require the two operands to have the same (non-void) type and result in a boolean.
5. Boolean operations `Lor` and `Land` can only be applied to boolean operands and result in a boolean. These operators are *short-circuiting* (explained later).
6. Variables should be properly resolved and type checked. Note that this is a *non-local* requirement. In other words, semantic analysis must use *environment* to remember the type of all variables that are in scope.
7. A well-typed assignment must have a right-hand-side that has the same (non-void) type as the l-value it is being assigned to.

8. A valid call must correspond to a call to a known function, in this phase just library functions. The types and number of arguments must match the parameters of the function. Furthermore, the type of the overall expression must the same as the return type of the function.
9. The body given for variable declaration should match its type, if provided. If not provided, the type of the variable is inferred to be the type of the body expression which must not be void.
10. A valid expression statement can either be an assignment, or a function call. Any other expression is not considered valid as an expression statement.
11. The condition of an if statement must be an expression of type boolean. The then branch and the else branch (if present) must valid statements.
12. A compound statement is valid if all its statements are valid. Any variable declared inside a compound statement are only valid within that block (and obviously only after that declaration itself within the block).
13. Variables can shadow one another. That is, when used, a variable refers to its closest declaration, e.g., `var x = 10; var x = 12; print_integer(x); print_integer(x);` is valid and should print 12 twice, `var x = 10; {var x = 12; print_integer(x);} print_integer(x);` should print 12 followed by 10. However, `var x = 10; {var x = 12;} print_integer(x); print_integer(x);` should print 10 twice, etc.
14. Note that function names and local variables live in the same scope. This means that local variables can mask functions. That is, `var f = 10; f(12);` should always produce an error even if function `f` exists and has the appropriate type.
15. Return statements are only valid (in this phase) if the expression returned is an integer.
16. All programs (in this phase) must always end in a return statement (which must be an integer).

Note

In case of questions regarding ambiguity in the above semantic rules ask questions on the forum. We will try our best to answer promptly. In case necessary, e.g., if you are in doubt and there is no enough time (please do not leave assignments for the last minute!), use your best judgment (you must be able to reasonably defend it) and explain your reasoning in your report. We will do our best to be understanding.

Important

The details of type checking and type inference are discussed in class.

Task 1: Implement semantic analysis

Implement semantic analysis for phase 1 of Dolphin as described above.

The following template OCaml code is a good starting point to embark in this assignment. Here, we assume that a module `Errors` exists which declares an exception `TypeError` and a number of errors, e.g., the `TypeMismatch` error used below. This is explained below.

```
exception Unimplemented (* your code should eventually compile without this exception␣
↪*)

let typecheck_typ = function
| Ast.Int -> TAst.Int
| Ast.Bool -> TAst.Bool
```

(continues on next page)

(continued from previous page)

```
(* should return a pair of a typed expression and its inferred type. you can/should␣
 ↪use typecheck_expr inside infertype_expr. *)
let rec infertype_expr env expr =
  match expr with
  | _ -> raise Unimplemented
and infertype_lval env lvl =
  match lvl with
  | _ -> raise Unimplemented
(* checks that an expression has the required type tp by inferring the type and␣
 ↪comparing it to tp. *)
and typecheck_expr env expr tp =
  let texpr, texprtp = infertype_expr env expr in
  if texprtp <> tp then raise Unimplemented;
  texpr

(* should check the validity of a statement and produce the corresponding typed␣
 ↪statement. Should use typecheck_expr and/or infertype_expr as necessary. *)
let rec typecheck_statement env stm =
  match stm with
  | _ -> raise Unimplemented
(* should use typecheck_statement to check the block of statements. *)
and typecheck_statement_seq env stms = raise Unimplemented

(* the initial environment should include all the library functions, no local␣
 ↪variables, and no errors. *)
let initial_environment = raise Unimplemented

(* should check that the program (sequence of statements) ends in a return statement␣
 ↪and make sure that all statements are valid as described in the assignment. Should␣
 ↪use typecheck_statement_seq. *)
let typecheck_prog prg = raise Unimplemented
```

We recommend that you create three separate modules for this task. One module is for the semantic analysis following the example above; let us call this the `Semant` module. The other two should be the `Errors` module, and the `Env` module, roughly following the structures below. (These modules will be discussed further in class.)

```
(* Errors module *)
module Sym = Symbol
module TAst = TypedAst
module TPretty = TypedPretty

type error =
| TypeMismatch of {expected : TAst.typ; actual : TAst.typ}
(* other errors to be added as needed. *)

(* Useful for printing errors *)
let error_to_string err =
  match err with
  | TypeMismatch {expected; actual; _} -> Printf.sprintf "Type mismatch: expected %s␣
 ↪but found %s." (TPretty.typ_to_string expected) (TPretty.typ_to_string actual)
```

```
(* Env module *)

exception Unimplemented (* your code should eventually compile without this exception␣
 ↪*)
```

(continues on next page)

(continued from previous page)

```
type environment = | (* to be defined *)

(* create an initial environment with the given functions defined *)
let make_env function_types = raise Unimplemented

(* insert a local declaration into the environment *)
let insert_local_decl env sym typ = raise Unimplemented

(* lookup variables and functions. Note: it must first look for a local variable and␣
 ↪if not found then look for a function. *)
let lookup_var_fun env sym = raise Unimplemented
```

Question 1
Reflect on the usage of type inference as opposed to type checking in your implementation. Briefly describe what principle guides you in writing your code, that is when you decide to use one vs the other.

Question 2
Reflect on the error recovery strategy in your implementation. How do you implement it?

4.5 Code generation

The compiler must generate valid LLVM code for all semantically valid programs. For this purpose we use the `cfgBuilder` module.

Task 2: Implement code generation

Implement code generation for phase 1 of Dolphin as described above.

There are a few key points to pay attention to when producing code for Dolphin programs:

1. Dolphin has a guaranteed order of evaluation: left to right. For instance, in binary operations, the left operand is executed first. Once the result of the first operand is obtained, then the program continues with executing the right operand, followed by the operation itself. Similarly, function arguments are computed from left to right before the function call.
2. Pay attention to the *short-circuiting* nature of boolean and (`&&`) and or (`||`) operations. These operations first compute the left operand. If the left operand determines the result of the operation (if it is `false` in case of `&&` or `true` in case of `||`), the right operand is **not** computed. This is the standard semantics for such operations. Here, the LLVM program generated must perform the analysis of branch appropriately so as to not compute the right operand, if not necessary. To maintain SSA form, we must either allocate temporary stack space for the result (as we have seen earlier), or use `phi` nodes (this is one of the glory points below).
3. As we have done before when translating programs by hand to LLVM, local variables are allocated on the stack (using `LLVM--`'s `alloca` instruction). We need an environment to remember the assigned `LLVM--` register for each variable and its `LLVM--` type. This is in many ways similar to the environment we use for semantic analysis. However, we no longer need to have functions and their types in the environment. All the necessary information, i.e., the return type, and the type of arguments, is already stored in the typed AST.

Question 3 (glory **)
Use `phi` nodes to re-implement short-circuiting behavior of boolean `and` and `or` operators.

Task 3: Testing and consolidation

In this task, we consolidate the two parts from the previous tasks, and test our project in an end-to-end fashion.

1. Write a top-level function `compile_prog` that given an AST, runs the semantic analysis on it. If there are any errors, they should be printed on standard error output, and the program exits with exit code 1. If there are no errors, it proceeds to generate the LLVM translation. The result of the translation should be output on standard output, and the program exits with exit code 0.
2. Write a test corpus consisting of 10 or more example programs that cover all the features of the language. It is important that the tests you write are relevant. Think about negative and positive tests. For negative tests, the critical aspect is the semantic analysis. In particular, if your semantic analysis reports a particular type of an error, your test corpus should include a program that exhibits that kind of error. For positive tests, your tests should cover all features of the language, i.e., all possible binary operations, calls to the standard library, etc.

> **Attention**
>
> Do not take this part of the assignment lightly. Use this task as a vehicle for discovering bugs and logical errors in your project.

4.6 Appendix

You will need libraries `printbox` and `printbox-text` to use pretty printers below. These can be installed using opam using the following command `opam install printbox printbox-text`

These pretty printers produce a so-called `box` which is the terminology that `printbox` uses to refer to formatted, structured texts. A `box` ca be printed as follows:

```
PrintBox_text.output stdout (Pretty.program_to_tree prog)
```

This will print the AST of the program as a tree.

4.6.1 pretty printer for ASTs (module `Pretty`)

```
module PBox = PrintBox
open Ast

(* producing trees for pretty printing *)
let typ_style = PBox.Style.fg_color PBox.Style.Green
let ident_style = PBox.Style.fg_color PBox.Style.Yellow
let fieldname_style = ident_style
let keyword_style = PBox.Style.fg_color PBox.Style.Blue

let info_node_style = PBox.Style.fg_color PBox.Style.Cyan

let make_typ_line name = PBox.line_with_style typ_style name
let make_fieldname_line name = PBox.line_with_style fieldname_style name
let make_ident_line name = PBox.line_with_style ident_style name
let make_keyword_line name = PBox.line_with_style keyword_style name

let make_info_node_line info = PBox.line_with_style info_node_style info

let ident_to_tree (Ident {name}) = make_ident_line name

let typ_to_tree tp =
  match tp with
```

(continues on next page)

(continued from previous page)

```
  | Bool -> make_typ_line "Bool"
  | Int -> make_typ_line "Int"

let binop_to_tree op =
  match op with
  | Plus -> make_keyword_line "Plus"
  | Minus -> make_keyword_line "Minus"
  | Mul -> make_keyword_line "Mul"
  | Div -> make_keyword_line "Div"
  | Rem -> make_keyword_line "Rem"
  | Lt -> make_keyword_line "Lt"
  | Le -> make_keyword_line "Le"
  | Gt -> make_keyword_line "Gt"
  | Ge -> make_keyword_line "Ge"
  | Lor -> make_keyword_line "Lor"
  | Land -> make_keyword_line "Land"
  | Eq -> make_keyword_line "Eq"
  | NEq -> make_keyword_line "NEq"

let unop_to_tree op =
  match op with
  | Neg -> make_keyword_line "Neg"
  | Lnot -> make_keyword_line "Lnot"

let rec expr_to_tree e =
  match e with
  | Integer {int; _} -> PBox.hlist ~bars:false [make_info_node_line "IntLit("; PBox.
↪line (Int64.to_string int); make_info_node_line ")"]
  | Boolean {bool; _} -> PBox.hlist ~bars:false [make_info_node_line "BooleanLit(";␣
↪make_keyword_line (if bool then "true" else "false"); make_info_node_line ")"]
  | BinOp {left; op; right; _} -> PBox.tree (make_info_node_line "BinOp") [expr_to_
↪tree left; binop_to_tree op; expr_to_tree right]
  | UnOp {op; operand; _} -> PBox.tree (make_info_node_line "UnOp") [unop_to_tree op;␣
↪expr_to_tree operand]
  | Lval l -> PBox.tree (make_info_node_line "Lval") [lval_to_tree l]
  | Assignment {lvl; rhs; _} -> PBox.tree (make_info_node_line "Assignment") [lval_to_
↪tree lvl; expr_to_tree rhs]
  | Call {fname; args; _} ->
    PBox.tree (make_info_node_line "Call")
      [PBox.hlist ~bars:false [make_info_node_line "FunName: "; ident_to_tree fname];
       PBox.tree (make_info_node_line "Args") (List.map (fun e -> expr_to_tree e)␣
↪args)]
and lval_to_tree l =
  match l with
  | Var ident -> PBox.hlist ~bars:false [make_info_node_line "Var("; ident_to_tree␣
↪ident; make_info_node_line ")"]

let rec statement_to_tree c =
  match c with
  | VarDeclStm {name; tp; body} -> PBox.tree (make_keyword_line "VarDeclStm")
    [PBox.hlist ~bars:false [make_info_node_line "Ident: "; ident_to_tree name];
    PBox.hlist ~bars:false [make_info_node_line "Type: "; Option.fold ~none:PBox.
↪empty ~some:typ_to_tree tp];
    PBox.hlist ~bars:false [make_info_node_line "Body: "; expr_to_tree body]]
  | ExprStm {expr; _} -> PBox.hlist ~bars:false [make_info_node_line "ExprStm: ";␣
↪Option.fold ~none:PBox.empty ~some:expr_to_tree expr]
  | IfThenElseStm {cond; thbr; elbro; _} ->
```

(continues on next page)

(continued from previous page)

```
    PBox.tree (make_keyword_line "IfStm")
      ([PBox.hlist ~bars:false [make_info_node_line "Cond: "; expr_to_tree cond];␣
 ↪PBox.hlist ~bars:false [make_info_node_line "Then-Branch: "; statement_to_tree␣
 ↪thbr]] @
       match elbro with None -> [] | Some elbr -> [PBox.hlist ~bars:false [make_info_
 ↪node_line "Else-Branch: "; statement_to_tree elbr]])
  | CompoundStm {stms; _} -> PBox.tree (make_info_node_line "CompoundStm") (statement_
 ↪seq_to_forest stms)
  | ReturnStm {ret; _} -> PBox.hlist ~bars:false [make_keyword_line "ReturnValStm: ";␣
 ↪expr_to_tree ret]
and statement_seq_to_forest stms = List.map statement_to_tree stms

let program_to_tree prog =
  PBox.tree (make_info_node_line "Program") (statement_seq_to_forest prog)
```

4.6.2 pretty printer for ASTs (module `TypedPretty`)

```
module Sym = Symbol
module PBox = PrintBox
open TypedAst

let typ_to_string = function
| Void -> "void"
| Int -> "int"
| Bool -> "bool"
| ErrorType -> "'type error'"

(* producing trees for pretty printing *)
let ident_to_tree (Ident {sym}) = Pretty.make_ident_line (Sym.name sym)

let typ_to_tree tp =
  match tp with
  | Void -> Pretty.make_typ_line "Void"
  | Int -> Pretty.make_typ_line "Int"
  | Bool -> Pretty.make_typ_line "Bool"
  | ErrorType -> PBox.line_with_style (PBox.Style.set_bg_color PBox.Style.Red PBox.
 ↪Style.default) "ErrorType"

let binop_to_tree op =
  match op with
  | Plus -> Pretty.make_keyword_line "PLUS"
  | Minus -> Pretty.make_keyword_line "Minus"
  | Mul -> Pretty.make_keyword_line "Mul"
  | Div -> Pretty.make_keyword_line "Div"
  | Rem -> Pretty.make_keyword_line "Rem"
  | Lt -> Pretty.make_keyword_line "Lt"
  | Le -> Pretty.make_keyword_line "Le"
  | Gt -> Pretty.make_keyword_line "Gt"
  | Ge -> Pretty.make_keyword_line "Ge"
  | Lor -> Pretty.make_keyword_line "Lor"
  | Land -> Pretty.make_keyword_line "Land"
  | Eq -> Pretty.make_keyword_line "Eq"
  | NEq -> Pretty.make_keyword_line "NEq"

let unop_to_tree op =
```

(continues on next page)

(continued from previous page)

```
  match op with
  | Neg -> Pretty.make_keyword_line "Neg"
  | Lnot -> Pretty.make_keyword_line "Lor"

let rec expr_to_tree e =
  match e with
  | Integer {int; _} -> PBox.hlist ~bars:false [Pretty.make_info_node_line "IntLit(";␣
↪PBox.line (Int64.to_string int); Pretty.make_info_node_line ")"]
  | Boolean {bool; _} -> PBox.hlist ~bars:false [Pretty.make_info_node_line
↪"BooleanLit("; Pretty.make_keyword_line (if bool then "true" else "false"); Pretty.
↪make_info_node_line ")"]
  | BinOp {left; op; right; tp; _} -> PBox.tree (Pretty.make_info_node_line "BinOp")␣
↪[typ_to_tree tp; expr_to_tree left; binop_to_tree op; expr_to_tree right]
  | UnOp {op; operand; tp; _} -> PBox.tree (Pretty.make_info_node_line "UnOp") [typ_
↪to_tree tp; unop_to_tree op; expr_to_tree operand]
  | Lval l -> PBox.tree (Pretty.make_info_node_line "Lval") [lval_to_tree l]
  | Assignment {lvl; rhs; tp; _} -> PBox.tree (Pretty.make_info_node_line "Assignment
↪") [typ_to_tree tp; lval_to_tree lvl; expr_to_tree rhs]
  | Call {fname; args; tp; _} ->
    PBox.tree (Pretty.make_info_node_line "Call")
      [typ_to_tree tp;
      PBox.hlist ~bars:false [Pretty.make_info_node_line "FunName: "; ident_to_tree␣
↪fname];
       PBox.tree (Pretty.make_info_node_line "Args") (List.map (fun e -> expr_to_
↪tree e) args)]
and lval_to_tree l =
  match l with
  | Var {ident; tp} -> PBox.hlist ~bars:false [Pretty.make_info_node_line "Var(";␣
↪ident_to_tree ident; Pretty.make_info_node_line ")"; PBox.line " : "; typ_to_tree␣
↪tp;]

let rec statement_to_tree c =
  match c with
  | VarDeclStm {name; tp; body; _} ->
    PBox.tree (Pretty.make_keyword_line "VarDeclStm")
      [PBox.hlist ~bars:false [Pretty.make_info_node_line "Ident: "; ident_to_tree␣
↪name];
      PBox.hlist ~bars:false [Pretty.make_info_node_line "Type: "; typ_to_tree tp];
      PBox.hlist ~bars:false [Pretty.make_info_node_line "Body: "; expr_to_tree body]]
  | ExprStm {expr; _} -> PBox.hlist ~bars:false [Pretty.make_info_node_line "ExprStm:
↪"; Option.fold ~none:PBox.empty ~some:expr_to_tree expr]
  | IfThenElseStm {cond; thbr; elbro; _} ->
    PBox.tree (Pretty.make_keyword_line "IfStm")
      ([PBox.hlist ~bars:false [Pretty.make_info_node_line "Cond: "; expr_to_tree␣
↪cond]; PBox.hlist ~bars:false [Pretty.make_info_node_line "Then-Branch: ";␣
↪statement_to_tree thbr]] @
       match elbro with None -> [] | Some elbr -> [PBox.hlist ~bars:false [Pretty.
↪make_info_node_line "Else-Branch: "; statement_to_tree elbr]])
  | CompoundStm {stms; _} -> PBox.tree (Pretty.make_info_node_line "CompoundStm")␣
↪(statement_seq_to_forest stms)
  | ReturnStm {ret; _} -> PBox.hlist ~bars:false [Pretty.make_keyword_line
↪"ReturnValStm: "; expr_to_tree ret]
and statement_seq_to_forest stms = List.map statement_to_tree stms

let program_to_tree prg =
  PBox.tree (Pretty.make_info_node_line "Program") (statement_seq_to_forest prg)
```

4.6.3 C runtime

```c
#include <stdio.h>          /* make sure these two includes are    */
#include <inttypes.h>       /* present in the start of your C file */

// your LLVM program must produce a function called dolphin_main of the following type.
extern int64_t dolphin_main();

int64_t read_integer () {
    int64_t value;
    printf("Please enter an integer: ");
    scanf("%" PRId64 "" , &value);
    return value;
}

void print_integer (int64_t value) {
    printf("%" PRId64 "\n" , value);
}

int main(){
    return dolphin_main();
}
```

The following OCaml code defines the type of the two library functions. Use this code, placed in a separate module, to bootstrap your semantic analysis and code generation.

```
module TAst = TypedAst
module Sym = Symbol

let make_ident name = TAst.Ident {sym = Sym.symbol name}

let library_functions =
  [
    (Symbol.symbol "read_integer", TAst.FunTyp {ret = TAst.Int; params = []});
    (Symbol.symbol "print_integer", TAst.FunTyp {ret = TAst.Void; params = [Param
 ↪{paramname = make_ident "value"; typ = TAst.Int}]})
  ]
```

CHAPTER

FIVE

DOLPHIN – PHASE 2

Attention

This is a group assignment. The workload is calibrated for a group of 3.

In case of questions regarding ambiguity of what you should do, ask questions on the forum. If you are in doubt and there is no enough time, use your best judgment and explain your reasoning in your report.

5.1 Assignment overview

This assignment is builds on *the previous assignment* and covers translating a fragment of Dolphin language, in the AST form, into `LLVM--`. This phase extends the language in two ways.

1. the language is extended with loops.
2. the language is extended to support multiple variable declarations in one declaration statement.

There are 2 tasks and *no* questions in this assignment. There are no *glory* questions in this assignment.

5.1.1 What you need to get started

This assignment is continuation of the previous assignment. To get started you need to edit the code from the previous assignment.

5.1.2 What you need to hand in

Please hand in a `.zip` file containing the following

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each task and question, briefly (1 – 4 sentences) describe your implementation or answer. Write concisely.
2. All the source files needed to reproduce your solution. This also includes the `C` code provided. Please explain in your report how the solution could be reproduced, e.g., calling `make` (if you have made a `Makefile`), the command line to call `clang`, etc.

Important

Make sure to understand all the code you hand in, including what is copied from here. (The code for pretty printing (typed) ASTs is an exception here; see the *appendix* below.)

5.2 The Abstract Syntax Tree (AST) of Dolphin (phase 2)

In this phase, a Dolphin program is *still* simply a sequence of statements. Recall that, intuitively, this is to be understood as the body of the main function.

The OCaml types for the AST describing programs is given below:

```
(* -- Use this in your solution without modifications *)
type ident = Ident of {name : string;}

type typ = | Int | Bool

type binop = | Plus | Minus | Mul | Div | Rem | Lt
  | Le | Gt | Ge | Lor | Land | Eq | NEq

type unop = | Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr}
| UnOp of {op : unop; operand : expr}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr;}
| Call of {fname : ident; args : expr list}
and lval =
| Var of ident

type single_declaration =
  Declaration of {name : ident; tp : typ option; body : expr}

type declaration_block = DeclBlock of single_declaration list

type for_init =
| FIExpr of expr
| FIDecl of declaration_block

type statement =
| VarDeclStm of declaration_block
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option}
| WhileStm of {cond : expr; body : statement}
| ForStm of { init : for_init option
            ; cond : expr option
            ; update : expr option
            ; body : statement }
| BreakStm
| ContinueStm
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type program = statement list
```

Action item

The AST declarations above should replace the contents of the module called `Ast`. Do **not** change the code above.

Note that in the AST above the types `typ`, `binop`, `unop`, `expr`, `lval`, and `program` have not changed at all. Note that the `VarDeclStm` is changed. It no longer consists of an inlined record. Instead, it consists of a `declaration_block`, which in turn is a list of `single_declaration`s. This change corresponds to allowing declaration statements to declare multiple variables in one statement, e.g.,

```
var x : int = 2, z = 5, w = "abc", t = x + z;
```

The meaning of multiple declarations is to be understood from left to right, and variables declared before can be used in the initialization expressions of later variables. The declaration statement above is identical *in meaning* to the following sequence of declarations.

```
var x : int = 2;
var z = 5;
var w = "abc";
var t = x + z;
```

Hint

There should be no major changes necessary, in either semantic analysis or code generation, to support multiple declarations in one statement, *if variable declarations are handled properly in the previous phase*. A simple fold over the list should suffice. Also note that there is no need to assume/enforce that the length of the list is non-empty in this phase.

The type statement, in the AST declaration above, has four new cases: `WhileStm` (while loops), `ForStm` (for loops), `BreakStm` (the break statement), and `ContinueStm` (the continue statement).

- The while loop construction carries a condition expression and a body which is a statement.
- The for loop on the other hand is more involved. It carries an optional initialization, an optional condition expression, an optional update expression, and a body statement.

  The initialization, if present, is either a declaration block, or an expression. If the initialization of a for loop is a declaration block, the scope of those declarations is the entire for loop, i.e., condition, update, and the body of the loop. If either of the initialization or the update parts of the for loop are absent, they should be understood as *do nothing*, similarly to empty expression statements (no expression present).

  In case of the update expression, or the initialization (when it is an expression), there is no requirement about the type. They could be expressions of any type, *as long as they are well-typed*. The condition, on the other hand, when present should be a boolean expression. When absent, the condition is to be understood as `true`. As a simple example, the for loop `for(;;){}` with no initialization, condition, or update present is an infinite loop that does nothing, just like `while(true){}`. Do note that the condition of the while loop must always be present. This is reflected in type of the AST.
- The break and continue statements, respectively break out of, and go to the beginning of the current loop. Here, current loop means the innermost enclosing loop (either a while or a for). Note that break and continue statements should never appear outside a loop. Semantic analysis should check this. For the code generation of the continue statement it is important to note that continue behaves differently based on whether it innermost enclosing loop is a for or a while loop. In case of the while loop, the continue statement causes the loop to start over starting with evaluating the condition of the loop. In case of the for loop, however, the continue statement causes the update clause to be evaluated before evaluating the loop condition.

Hint

In order to handle break and continue properly one needs to extend the environments. For semantic analysis, the environment should track whether the part of the program being analyzed is inside a loop or not. For code generation,

the environment should track the label to jump to in case of break and continue.

5.3 The Typed Abstract Syntax Tree (AST) of Dolphin (phase 2)

Recall the typed AST differs from the AST in that it includes all the necessary type information for (LLVM) code generation. The differences between the two notions of AST can be subtle. Please pay attention to the details.

The OCaml types for the typed AST describing programs is given below:

```
(* -- Use this in your solution without modifications *)
module Sym = Symbol

type ident = Ident of {sym : Sym.symbol}

type typ = | Void | Int | Bool | ErrorType

type binop = | Plus | Minus | Mul | Div | Rem | Lt
  | Le | Gt | Ge | Lor | Land | Eq | NEq

type unop = | Neg | Lnot

type expr =
| Integer of {int : int64}
| Boolean of {bool : bool}
| BinOp of {left : expr; op : binop; right : expr; tp : typ}
| UnOp of {op : unop; operand : expr; tp : typ}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr; tp : typ}
| Call of {fname : ident; args : expr list; tp : typ}
and lval =
| Var of {ident : ident; tp : typ}

type single_declaration = Declaration of {name : ident; tp : typ; body : expr}

type declaration_block = DeclBlock of single_declaration list

type for_init =
| FIDecl of declaration_block
| FIExpr of expr

type statement =
| VarDeclStm of declaration_block
| ExprStm of {expr : expr option}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option}
| WhileStm of {cond : expr; body : statement}
| ForStm of { init : for_init option
            ; cond : expr option
            ; update : expr option
            ; body : statement }
| BreakStm
| ContinueStm
| CompoundStm of {stms : statement list}
| ReturnStm of {ret : expr}

type param = Param of {paramname : ident; typ : typ}
```

(continues on next page)

(continued from previous page)

```
type funtype = FunTyp of {ret : typ; params : param list}

type program = statement list
```

Action item

The AST declarations above should replace the contents of the module called `TypedAst`. Do **not** change the code above.

For an explanation of the main differences between (untyped) ASTs we saw earlier and typed ASTs consult *the previous assignment*.

5.4 Semantic analysis of Dolphin programs (phase 2)

The first task asks you to update semantic analysis of the previous assignment to support the new extensions.

Task 1: Implement semantic analysis

Implement semantic analysis for phase 2 of Dolphin as described above. Extend the errors in the `Errors` module to accommodate new ways a program can be semantically malformed.

5.5 Code generation

Recall that the compiler must generate valid LLVM code for all semantically valid programs. For this purpose we use the `cfgBuilder` module.

Task 2: Implement code generation

Implement code generation for phase 2 of Dolphin as described above.

Task 3: Testing and consolidation

In this task, we consolidate the two parts from the previous tasks, and test our project in an end-to-end fashion.

1. You should already have a top-level function `compile_prog`. If implemented properly before, this function should not require any changes for this assignment. In other words, ensure the following behavior for `compile_prog`.
    - Given an AST, `compile_prog` runs the semantic analysis on it. If there are any errors, they should be printed on standard error output, and the program should exit with exit code 1. If there are no errors, it proceeds to generate the LLVM translation. The result of the translation should be output on standard output, and the program exits with exit code 0.
2. Write a test corpus consisting of 10 or more example programs that cover all the *new* features of the language. It is important that the tests you write are relevant. Think about negative and positive tests. For negative tests, the critical aspect is the semantic analysis. In particular, if your semantic analysis reports a particular type of an error, your test corpus should include a program that exhibits that kind of error. For positive tests, your tests should cover all *new* features of the language, i.e., all possible cases of using (and nesting) for and while loops, as well the

break and continue statements in them, etc. Do remember to produce positive and negative tests for declaration statements declaring multiple variables.

> **Attention**
>
> Do not take this part of the assignment lightly. Use this task as a vehicle for discovering bugs and logical errors in your project.

5.6 Appendix

Recall that you will need libraries `printbox` and `printbox-text` to use pretty printers below. These can be installed using opam using the following command `opam install printbox printbox-text`

These pretty printers produce a so-called `box` which is the terminology that `printbox` uses to refer to formatted, structured texts. A `box` ca be printed as follows:

```
PrintBox_text.output stdout (Pretty.program_to_tree prog)
```

This will print the AST of the program as a tree.

5.6.1 pretty printer for ASTs (module `Pretty`)

```
module PBox = PrintBox
open Ast

(* producing trees for pretty printing *)
let typ_style = PBox.Style.fg_color PBox.Style.Green
let ident_style = PBox.Style.fg_color PBox.Style.Yellow
let fieldname_style = ident_style
let keyword_style = PBox.Style.fg_color PBox.Style.Blue

let info_node_style = PBox.Style.fg_color PBox.Style.Cyan

let make_typ_line name = PBox.line_with_style typ_style name
let make_fieldname_line name = PBox.line_with_style fieldname_style name
let make_ident_line name = PBox.line_with_style ident_style name
let make_keyword_line name = PBox.line_with_style keyword_style name

let make_info_node_line info = PBox.line_with_style info_node_style info

let ident_to_tree (Ident {name}) = make_ident_line name

let typ_to_tree tp =
  match tp with
  | Bool -> make_typ_line "Bool"
  | Int -> make_typ_line "Int"

let binop_to_tree op =
  match op with
  | Plus -> make_keyword_line "PLUS"
  | Minus -> make_keyword_line "Minus"
  | Mul -> make_keyword_line "Mul"
```

(continues on next page)

(continued from previous page)

```
  | Div -> make_keyword_line "Div"
  | Rem -> make_keyword_line "Rem"
  | Lt -> make_keyword_line "Lt"
  | Le -> make_keyword_line "Le"
  | Gt -> make_keyword_line "Gt"
  | Ge -> make_keyword_line "Ge"
  | Lor -> make_keyword_line "Lor"
  | Land -> make_keyword_line "Land"
  | Eq -> make_keyword_line "Eq"
  | NEq -> make_keyword_line "NEq"

let unop_to_tree op =
  match op with
  | Neg -> make_keyword_line "Neg"
  | Lnot -> make_keyword_line "Lor"

let rec expr_to_tree e =
  match e with
  | Integer {int; _} -> PBox.hlist ~bars:false [make_info_node_line "IntLit("; PBox.
 ↪line (Int64.to_string int); make_info_node_line ")"]
  | Boolean {bool; _} -> PBox.hlist ~bars:false [make_info_node_line "BooleanLit(";␣
 ↪make_keyword_line (if bool then "true" else "false"); make_info_node_line ")"]
  | BinOp {left; op; right; _} -> PBox.tree (make_info_node_line "BinOp") [expr_to_
 ↪tree left; binop_to_tree op; expr_to_tree right]
  | UnOp {op; operand; _} -> PBox.tree (make_info_node_line "UnOp") [unop_to_tree op;␣
 ↪expr_to_tree operand]
  | Lval l -> PBox.tree (make_info_node_line "Lval") [lval_to_tree l]
  | Assignment {lvl; rhs; _} -> PBox.tree (make_info_node_line "Assignment") [lval_to_
 ↪tree lvl; expr_to_tree rhs]
  | Call {fname; args; _} ->
    PBox.tree (make_info_node_line "Call")
      [PBox.hlist ~bars:false [make_info_node_line "FunName: "; ident_to_tree fname];
       PBox.tree (make_info_node_line "Args") (List.map (fun e -> expr_to_tree e)␣
 ↪args)]
and lval_to_tree l =
  match l with
  | Var ident -> PBox.hlist ~bars:false [make_info_node_line "Var("; ident_to_tree␣
 ↪ident; make_info_node_line ")"]

let single_declaration_to_tree (Declaration {name; tp; body; _}) =
  PBox.tree (make_keyword_line "Declaration")
    [PBox.hlist ~bars:false [make_info_node_line "Ident: "; ident_to_tree name];
    PBox.hlist ~bars:false [make_info_node_line "Type: "; Option.fold ~none:PBox.
 ↪empty ~some:typ_to_tree tp];
    PBox.hlist ~bars:false [make_info_node_line "Body: "; expr_to_tree body]]

let declaration_block_to_tree (DeclBlock declarations) =
PBox.tree (make_keyword_line "VarDecl")  (List.map single_declaration_to_tree␣
 ↪declarations)

let for_init_to_tree = function
| FIDecl db -> PBox.hlist ~bars:false [PBox.line "ForInitDecl: "; declaration_block_
 ↪to_tree db]
| FIExpr e -> PBox.hlist ~bars:false [PBox.line "ForInitExpr: "; expr_to_tree e]

let rec statement_to_tree c =
  match c with
```

(continues on next page)

(continued from previous page)

```
  | VarDeclStm db -> PBox.hlist ~bars:false [PBox.line "DeclStm: "; declaration_block_
↪to_tree db]
  | ExprStm {expr} -> PBox.hlist ~bars:false [make_info_node_line "ExprStm: "; Option.
↪fold ~none:PBox.empty ~some:expr_to_tree expr]
  | IfThenElseStm {cond; thbr; elbro} ->
    PBox.tree (make_keyword_line "IfStm")
      ([PBox.hlist ~bars:false [make_info_node_line "Cond: "; expr_to_tree cond];␣
↪PBox.hlist ~bars:false [make_info_node_line "Then-Branch: "; statement_to_tree␣
↪thbr]] @
       match elbro with None -> [] | Some elbr -> [PBox.hlist ~bars:false [make_info_
↪node_line "Else-Branch: "; statement_to_tree elbr]])
  | WhileStm {cond; body} ->
    PBox.tree (make_keyword_line "WhileStm")
      [PBox.hlist ~bars:false [make_info_node_line "Cond: "; expr_to_tree cond];
       PBox.hlist ~bars:false [make_info_node_line "Body: "; statement_to_tree body]]
  | ForStm {init; cond; update; body} ->
    PBox.tree (make_keyword_line "ForStm")
      [PBox.hlist ~bars:false [make_info_node_line "Init: "; Option.fold ~none:PBox.
↪empty ~some:for_init_to_tree init];
       PBox.hlist ~bars:false [make_info_node_line "Cond: "; Option.fold ~none:PBox.
↪empty ~some:expr_to_tree cond];
       PBox.hlist ~bars:false [make_info_node_line "Update: "; Option.fold ~
↪none:PBox.empty ~some:expr_to_tree update];
       PBox.hlist ~bars:false [make_info_node_line "Body: "; statement_to_tree body]]
  | BreakStm -> make_keyword_line "BreakStm"
  | ContinueStm -> make_keyword_line "ContinueStm"
  | CompoundStm {stms} -> PBox.tree (make_info_node_line "CompoundStm") (statement_
↪seq_to_forest stms)
  | ReturnStm {ret} -> PBox.hlist ~bars:false [make_keyword_line "ReturnValStm: ";␣
↪expr_to_tree ret]
and statement_seq_to_forest stms = List.map statement_to_tree stms

let program_to_tree prog =
  PBox.tree (make_info_node_line "Program") (statement_seq_to_forest prog)
```

5.6.2 pretty printer for ASTs (module `TypedPretty`)

```
module Sym = Symbol
module PBox = PrintBox
open TypedAst

let typ_to_string = function
| Void -> "void"
| Int -> "int"
| Bool -> "bool"
| ErrorType -> "'type error'"

(* producing trees for pretty printing *)
let ident_to_tree (Ident {sym}) = Pretty.make_ident_line (Sym.name sym)

let typ_to_tree tp =
  match tp with
  | Void -> Pretty.make_typ_line "Void"
  | Int -> Pretty.make_typ_line "Int"
  | Bool -> Pretty.make_typ_line "Bool"
```

(continues on next page)

(continued from previous page)

```
  | ErrorType -> PBox.line_with_style (PBox.Style.set_bg_color PBox.Style.Red PBox.
↪Style.default) "ErrorType"

let binop_to_tree op =
  match op with
  | Plus -> Pretty.make_keyword_line "PLUS"
  | Minus -> Pretty.make_keyword_line "Minus"
  | Mul -> Pretty.make_keyword_line "Mul"
  | Div -> Pretty.make_keyword_line "Div"
  | Rem -> Pretty.make_keyword_line "Rem"
  | Lt -> Pretty.make_keyword_line "Lt"
  | Le -> Pretty.make_keyword_line "Le"
  | Gt -> Pretty.make_keyword_line "Gt"
  | Ge -> Pretty.make_keyword_line "Ge"
  | Lor -> Pretty.make_keyword_line "Lor"
  | Land -> Pretty.make_keyword_line "Land"
  | Eq -> Pretty.make_keyword_line "Eq"
  | NEq -> Pretty.make_keyword_line "NEq"

let unop_to_tree op =
  match op with
  | Neg -> Pretty.make_keyword_line "Neg"
  | Lnot -> Pretty.make_keyword_line "Lor"

let rec expr_to_tree e =
  match e with
  | Integer {int; _} -> PBox.hlist ~bars:false [Pretty.make_info_node_line "IntLit(";␣
↪PBox.line (Int64.to_string int); Pretty.make_info_node_line ")"]
  | Boolean {bool; _} -> PBox.hlist ~bars:false [Pretty.make_info_node_line
↪"BooleanLit("; Pretty.make_keyword_line (if bool then "true" else "false"); Pretty.
↪make_info_node_line ")"]
  | BinOp {left; op; right; tp; _} -> PBox.tree (Pretty.make_info_node_line "BinOp")␣
↪[typ_to_tree tp; expr_to_tree left; binop_to_tree op; expr_to_tree right]
  | UnOp {op; operand; tp; _} -> PBox.tree (Pretty.make_info_node_line "UnOp") [typ_
↪to_tree tp; unop_to_tree op; expr_to_tree operand]
  | Lval l -> PBox.tree (Pretty.make_info_node_line "Lval") [lval_to_tree l]
  | Assignment {lvl; rhs; tp; _} -> PBox.tree (Pretty.make_info_node_line "Assignment
↪") [typ_to_tree tp; lval_to_tree lvl; expr_to_tree rhs]
  | Call {fname; args; tp; _} ->
    PBox.tree (Pretty.make_info_node_line "Call")
      [typ_to_tree tp;
      PBox.hlist ~bars:false [Pretty.make_info_node_line "FunName: "; ident_to_tree␣
↪fname];
       PBox.tree (Pretty.make_info_node_line "Args") (List.map (fun e -> expr_to_
↪tree e) args)]
and lval_to_tree l =
  match l with
  | Var {ident; tp} -> PBox.hlist ~bars:false [Pretty.make_info_node_line "Var(";␣
↪ident_to_tree ident; Pretty.make_info_node_line ")"; PBox.line " : "; typ_to_tree␣
↪tp;]

let single_declaration_to_tree (Declaration {name; tp; body; _}) =
  PBox.tree (Pretty.make_keyword_line "Declaration")
    [PBox.hlist ~bars:false [Pretty.make_info_node_line "Ident: "; ident_to_tree␣
↪name];
    PBox.hlist ~bars:false [Pretty.make_info_node_line "Type: "; typ_to_tree tp];
    PBox.hlist ~bars:false [Pretty.make_info_node_line "Body: "; expr_to_tree body]]
```

(continues on next page)

(continued from previous page)

```
let declaration_block_to_tree (DeclBlock declarations) =
  PBox.tree (Pretty.make_keyword_line "VarDecl") (List.map single_declaration_to_tree␣
 ↪declarations)

let for_init_to_tree = function
| FIDecl db -> PBox.hlist ~bars:false [PBox.line "ForInitDecl: "; declaration_block_
 ↪to_tree db]
| FIExpr e -> PBox.hlist ~bars:false [PBox.line "ForInitExpr: "; expr_to_tree e]

let rec statement_to_tree c =
  match c with
  | VarDeclStm db -> PBox.hlist ~bars:false [PBox.line "DeclStm: "; declaration_block_
 ↪to_tree db]
  | ExprStm {expr; _} -> PBox.hlist ~bars:false [Pretty.make_info_node_line "ExprStm:
 ↪"; Option.fold ~none:PBox.empty ~some:expr_to_tree expr]
  | IfThenElseStm {cond; thbr; elbro; _} ->
    PBox.tree (Pretty.make_keyword_line "IfStm")
      ([PBox.hlist ~bars:false [Pretty.make_info_node_line "Cond: "; expr_to_tree␣
 ↪cond]; PBox.hlist ~bars:false [Pretty.make_info_node_line "Then-Branch: ";␣
 ↪statement_to_tree thbr]] @
       match elbro with None -> [] | Some elbr -> [PBox.hlist ~bars:false [Pretty.
 ↪make_info_node_line "Else-Branch: "; statement_to_tree elbr]])
  | WhileStm {cond; body; _} ->
    PBox.tree (Pretty.make_keyword_line "WhileStm")
      [PBox.hlist ~bars:false [Pretty.make_info_node_line "Cond: "; expr_to_tree␣
 ↪cond];
       PBox.hlist ~bars:false [Pretty.make_info_node_line "Body: "; statement_to_
 ↪tree body]]
  | ForStm {init; cond; update; body; _} ->
    PBox.tree (Pretty.make_keyword_line "ForStm")
      [PBox.hlist ~bars:false [Pretty.make_info_node_line "Init: "; Option.fold ~
 ↪none:PBox.empty ~some:for_init_to_tree init];
       PBox.hlist ~bars:false [Pretty.make_info_node_line "Cond: "; Option.fold ~
 ↪none:PBox.empty ~some:expr_to_tree cond];
       PBox.hlist ~bars:false [Pretty.make_info_node_line "Update: "; Option.fold ~
 ↪none:PBox.empty ~some:expr_to_tree update];
       PBox.hlist ~bars:false [Pretty.make_info_node_line "Body: "; statement_to_
 ↪tree body]]
  | BreakStm -> Pretty.make_keyword_line "BreakStm"
  | ContinueStm -> Pretty.make_keyword_line "ContinueStm"
  | CompoundStm {stms; _} -> PBox.tree (Pretty.make_info_node_line "CompoundStm")␣
 ↪(statement_seq_to_forest stms)
  | ReturnStm {ret; _} -> PBox.hlist ~bars:false [Pretty.make_keyword_line
 ↪"ReturnValStm: "; expr_to_tree ret]
and statement_seq_to_forest stms = List.map statement_to_tree stms

let program_to_tree prg =
  PBox.tree (Pretty.make_info_node_line "Program") (statement_seq_to_forest prg)
```

5.6.3 C runtime

The C runtime we use has not changed compared to the previous assignment. See the corresponding description in the *previous assignment*

CHAPTER

SIX

DOLPHIN – PHASE 3

Attention

This is a group assignment. The workload is calibrated for a group of 3.

In case of questions regarding ambiguity of what you should do, ask questions on the forum. If you are in doubt and there is no enough time, use your best judgment and explain your reasoning in your report.

6.1 Assignment overview

This assignment builds on the previous two assignments: *Phase 1* and *Phase 2*. The main focus of this assignment is building a *frontend* for the language of Phase 2. We extend the Dolphin implementation with the following:

1. A lexer that translates the source code into a stream of *tokens*.
2. A parser that translates a stream of tokens into an AST.

There are 4 tasks and *no* questions in this assignment. There are no *glory* questions in this assignment.

6.1.1 What you need to get started

- This assignment is continuation of the previous assignment. To get started you need to edit the code from the previous assignment.
- You need to understand how OCamllex lexer generator works. See OCamllex documentation.
- You need to understand how Menhir parser generator works. See Menhir documentation.

6.1.2 What you need to hand in

Please hand in a `.zip` file containing the following

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each task and question, briefly (1 – 4 sentences) describe your implementation or answer. Write concisely.
2. All the source files needed to reproduce your solution. This also includes the `C` code provided. Please explain in your report how the solution could be reproduced, e.g., calling `make` (if you have made a `Makefile`), the command line to call `clang`, etc.
3. All the tests that you create (see Task 4) should be placed into a directory `assignment-06-tests` as individual `.dlp` files.

Important

Make sure to understand all the code you hand in, including what is copied from here. (The code for pretty printing (typed) ASTs is an exception here; see the *appendix* below.)

6.2 The Abstract Syntax Tree (AST) of Dolphin (phase 2)

The AST that we use in this assignment is conceptually the same as before, except for one techical change – most of the nodes of the AST now carry location information. Location information should be used in error reporting.

The OCaml types for the AST describing programs is given below:

```
(* -- Use this in your solution without modifications *)
module Loc = Location

type ident = Ident of {name : string; loc : Loc.location}

type typ =
| Int of {loc : Loc.location}
| Bool of {loc : Loc.location}

type binop =
| Plus of {loc : Loc.location}
| Minus of {loc : Loc.location}
| Mul of {loc : Loc.location}
| Div of {loc : Loc.location}
| Rem of {loc : Loc.location}
| Lt of {loc : Loc.location}
| Le of {loc : Loc.location}
| Gt of {loc : Loc.location}
| Ge of {loc : Loc.location}
| Lor of {loc : Loc.location}
| Land of {loc : Loc.location}
| Eq of {loc : Loc.location}
| NEq of {loc : Loc.location}

type unop =
| Neg of {loc : Loc.location}
| Lnot of {loc : Loc.location}

type expr =
| Integer of {int : int64; loc : Loc.location}
| Boolean of {bool : bool; loc : Loc.location}
| BinOp of {left : expr; op : binop; right : expr; loc : Loc.location}
| UnOp of {op : unop; operand : expr; loc : Loc.location}
| Lval of lval
| Assignment of {lvl : lval; rhs : expr; loc : Loc.location}
| Call of {fname : ident; args : expr list; loc : Loc.location}
and lval =
| Var of ident

type single_declaration = Declaration of {name : ident; tp : typ option; body : expr;␣
↪loc : Loc.location}

type declaration_block = DeclBlock of {declarations : single_declaration list; loc :␣
```

(continues on next page)

(continued from previous page)

```
 ↪Loc.location}

type for_init =
| FIExpr of expr
| FIDecl of declaration_block

type statement =
| VarDeclStm of declaration_block
| ExprStm of {expr : expr option; loc : Loc.location}
| IfThenElseStm of {cond : expr; thbr : statement; elbro : statement option; loc :␣
 ↪Loc.location}
| WhileStm of {cond : expr; body : statement; loc : Loc.location}
| ForStm of {init : for_init option; cond : expr option; update : expr option; body :␣
 ↪statement; loc : Loc.location}
| BreakStm of {loc : Loc.location}
| ContinueStm of {loc : Loc.location}
| CompoundStm of {stms : statement list; loc : Loc.location}
| ReturnStm of {ret : expr; loc : Loc.location}

type program = statement list
```

Action item

The AST declarations above should replace the contents of the module called `Ast`. Do **not** change the code above.

6.2.1 Location module

The AST module above uses the module Location below:

```
(* -- Use this in your solution without modifications *)
module PBox = PrintBox

type location = {start_pos : Lexing.position; end_pos : Lexing.position}

let make_location (startp, endp) = {start_pos = startp; end_pos = endp}

let dummy_loc = {start_pos = Lexing.dummy_pos; end_pos = Lexing.dummy_pos}

let location_style = PBox.Style.fg_color PBox.Style.Cyan

let location_to_tree ?(includefile = true) {start_pos; end_pos} =
  if includefile then
    if start_pos.pos_lnum = end_pos.pos_lnum then
      PBox.sprintf_with_style location_style "\"%s\" @ %d:%d-%d"
        start_pos.pos_fname start_pos.pos_lnum
        (start_pos.pos_cnum - start_pos.pos_bol)
        (end_pos.pos_cnum - end_pos.pos_bol)
    else
      PBox.sprintf_with_style location_style "\"%s\" @ %d:%d-%d:%d"
        start_pos.pos_fname start_pos.pos_lnum
        (start_pos.pos_cnum - start_pos.pos_bol)
        end_pos.pos_lnum
        (end_pos.pos_cnum - end_pos.pos_bol)
```

(continues on next page)

(continued from previous page)

```
  else
    if start_pos.pos_lnum = end_pos.pos_lnum then
      PBox.sprintf_with_style location_style "@ %d:%d-%d"
        start_pos.pos_lnum
        (start_pos.pos_cnum - start_pos.pos_bol)
        (end_pos.pos_cnum - end_pos.pos_bol)
    else
      PBox.sprintf_with_style location_style "@ %d:%d-%d:%d"
        start_pos.pos_lnum
        (start_pos.pos_cnum - start_pos.pos_bol)
        end_pos.pos_lnum
        (end_pos.pos_cnum - end_pos.pos_bol)
```

Action item

The definition of locations above should be placed in a module called `Location`. That is, the contents above should be placed in a file called `location`. Do **not** change the code above.

Note

The typed AST module remains the same as in the previous assignment.

6.3 The syntax of Dolphin

You should by now have an intuitive understanding of the syntax and semantics of Dolphin. See *Exercises for Week 7* for an overview. Below, we focus on the details of the syntax that are relevant for the implementaiton of the Dolphin frontend.

6.3.1 Whitespace

Dolphin's syntax recognizes, and ignores, the following (sequences of) characters as whitespace: `\n` (linefeed character, i.e., Unix new line indicator), `\r\n` (carriage return, followed by linefeed, i.e., Windows new line indicator), tab, and space. All whitespace characters are completely ignored in Dolphin. New line indicators also indicate the end of a single-line comment.

Hint

When using `ocamllex`, the semntic action of recognizing new line indicators should invoke `Lexing.new_line` function to increment line number in the lexer state to maintain consistent source location.

6.3.2 Comments

In Dolphin, single-line comments start with `//`, and extend up to the end of the line. Multi-line (block) comments are delimited by `/*` and `*/`. **Note**: Dolphin supports nested multi-line comments. That is, both `/* abc */` and `/*/* abc */*/` are valid comments.

6.3.3 Keywords

The keywords of Dolphin are as follows: `true`, `false`, `nil`, `var`, `let`, `if`, `else`, `while`, `for`, `break`, `continue`, `return`, `int`, `byte`, `bool`, `string`, `void`, `record`, `new`, `length_of`

Some of these keywords are not yet used in our implementation Dolphin. However, they should be recognized by the lexer so as that identifiers are handled properly; see below.

6.3.4 Identifiers

An identifier is any string consisting of underscore, _, lowercse or capitl letters of English alphabet, and digits, `0`, …, `9`, that does not start with a digit, and is not a Dolphin keyword.

6.3.5 Operators

The operators of the Dolphin language are as follows:

Operator	Description
`+`, `-`, `*`, `/`, `%`	Arithmetic operators: addition, subtraction, unary negation, multiplication, division, remainder.
`<`, `<=`, `>`, `>=`	Relative comparison operators; these operators *only* apply to integers and strings.
`==`, `!=`	Equality comparison operator; these operators apply to *any type*, as long as the two sides are the same type, and that type is not `void`.
`||`, `&&`, `!`	Logical operators on booleans: or, and, and not.
`.`	Looking up fields in a record.
`,`	See comma expressions explained below. (*Note that `,` is also used as punctuation.*)

Precedences and associativity

- Arithmetic operator precedences are as usual: + and − have the same precedence which is lower than that of `*`, `/`, and `%`, which also have the same precedence. All these operators left-associative.
- The operator `&&` has a higher priority than `||` and they are both left associative.
- Unary operations, − and `!`, have a higher precedence than binary operations.
- The comparison operators are non-associative.

6.3.6 Punctuation

Punctuation mark	Description
`,`	Used for separating function arguments in a call, and separating variable declarations in a variable declaration block.
`{` and `}`	Delimits compound statements. Also, they will later be used to delimit function body.
`(` and `)`	Used for grouping, e.g., overriding precedences. Also used for function calls (see below).

6.3.7 Literals

Dolphin supports integer and string literals.

- An integer literal is any number within the range -2^{63} to $2^{63} - 1$. Integer literals out of this range are invalid.
- Boolean literals: keywords `true` and `false`.

6.3.8 Types

- Primitive types: `int`, `bool`.
- Void type: `void`. This type does not yet syntactically appear in our programs because it may **only** be used the return type of a function in a function declaration, which we do not yet support in Dolphin.

6.3.9 L-values

An l-value, also written lvalue, is any value in the programming language that roughly-speaking corresponds to a memory location. For the current subset of Dolphin, l-values can only be local variable names.

6.3.10 Expressions

Dolphin expressions are as follows:

- Integer and string literals.
- Boolean literals `true` and `false`.
- Binary operators applied to two expressions: `e1 o e2` where `e1` and `e2` are two expressions and `o` is any binary operator; see above for details of precedences.
- Unary operators applied to expressions: `o e` where `o` is a unary operator and `e` is an expression; see above for details of precedences.
- L-values are also expressions; this corresponds to reading the location.
- Assignment: `l = e` where `l` is an l-value and `e` is an expression; this corresponds to writing the location.
- Function calls are written by writing the name of the function followed by its arguments passed in parentheses, separated by a comma, e.g., `func(a, b, 2 + 3)`. Function arguments are expressions.
- Parentheses can be used to group expressions: `(e)` is a valid expression whenever `e` is.

6.3.11 Statements

- Expression statements: an expression statement is an assignment, a function call, or nothing, followed by a semicolon. That is, both `func(2, 3);` and `;` are valid expression statements. However, `2 + 5;` is not a valid expression statement.
- Declaration blocks: A declaration block consists of a `var` keyword, followed by a sequence of declarations, separated with a comma, terminated by a semicolon. A declaration consists of an identifier and a type, separated by a colon, and an initialization (= followed by an expression, which cannot itself be a comma operator; recall parentheses for grouping). Finally, the type ascription is optional. That is, `var x : int = 2;`, `var y = false, w : int = 5;`, `var x = 3, z = true;` are all valid declaration blocks.
- Conditional statement of the form `if(cond) true_body else false_body` where `cond` is an expression and `true_body` and `false_body` are both statements (see compound statements below). The `else false_body` part of the statement is optional and may be omitted. **Note**: the parentheses around the condition are part of the syntax of if statements.
- While loops of the form `while(cond) body` where `cond` is an expression and `body` is a statement (see compound statements below). **Note**: the parentheses around the condition are part of the syntax of while loops.
- For loops of the form `for(init; cond; update) body`. The initialization part, `init` may be a declaration block (without a semicolon, as `init` and `cond` are already separated by a `;` as part of the syntax of the for loop). Both `cond` and `update` are expressions and `body` is a statement (see compound statements below). **Note**: the parentheses around `init; cond; update` are part of the syntax of while loops. Each of the parts `init`, `cond`, and `update` may be omitted. That is, `for(;;);`, `for(; i < 10;);`, `for(var i : int = 12; i > 10;);`, etc. are all valid for loops.
- Break statement: `break;` (Note: the `;` ist part of the syntax of the statement.)
- Continue statement: `continue;` (Note: the `;` ist part of the syntax of the statement.)
- Return statement of the form `return res;` where `res` is an expression.
- Compound statement (block of statements): a compound statement is a sequence of statements, one followed after the other, surrounded in curly braces, e.g., `{if (n > 0) {r = 10; return 1; } else continue; {} }`. A compound statement may appear at any point where a statement is expected, e.g., as the body of a loop, or conditional, as a statement inside another compound statement, etc.

6.4 Lexer

6.4.1 Lexemes

The Dolphin language has the following lexeme kinds. (This is written in the language of Menhir.)

```
// end of file
%token EOF
// string literals
%token <string> STRING_LIT   (* Strings quoted with "" *)
// integer literals
%token <int64> INT_LIT
// booleans
%token TRUE FALSE
// length operation; for arrays and strings
%token LENGTHOF
// arithmetic oprations
%token PLUS MINUS MUL DIV REM
```

(continues on next page)

(continued from previous page)

```
// comparison operators
%token LT LE GT GE
// logical operations
%token LOR LAND LNOT
// equality
%token EQ NEQ
// assignment
%token ASSIGN
// punctuation
%token QUESTIONMARK COLON COMMA SEMICOLON
// accessors
%token DOT LBRACKET RBRACKET
// braces
%token LBRACE RBRACE
// parentheses
%token LPAREN RPAREN
// identifiers
%token <string> IDENT
// keywords
%token NIL VAR LET IF ELSE WHILE FOR BREAK CONTINUE RETURN NEW
// types
%token INT BOOL STRING BYTE VOID RECORD
```

6.4.2 Lexer

Task 1: Implement lexical analysis using OCamllex

Implement lexical analysis for the present subset of Dolphin as described above, where the input is read from a file, using `ocamllex``

> **Hint**
>
> Use a dummy parser for token generation.

6.5 Parser

Task 2: Implement the parser using Menhir parser generator

Implement the parser for the present subset of Dolphin as described above, using Menhir parser generator.

6.6 Updated semantic analysis

Task 3: Update semantic analysis

Update your semantic analysis so that error reporting now includes the location information.

6.7 Consolidation

Task 4: Testing and consolidation

In this task, we put the previous tasks together and test our project in an end-to-end fashion.

1. Extend your function `compile_prog` from the previous phase to include the frontend. In other words, ensure the following behavior for `compile_prog`.
   - Given a path to a file with Dolphin source code, it runs the lexer. If there are no errors in lexing, `compile_prog` runs the parser. If there are no errors in the parser, `compile_prog` runs the semantic analysis (from here on, it is as just as in the previous assignment). In case of errors, they should be printed on standard error output, and the program should exit with exit code 1. If there are no errors, `compile_prog` proceeds to generate the LLVM translation. The result of the translation should be output on standard output, and the program exits with exit code 0.
2. Port your earlier tests to source code. Add 10 new tests that negatively test your lexer and the parser. All the tests should be included into your submission as individual `.dlp` files.

6.8 Appendix

Recall that you will need libraries `printbox` and `printbox-text` to use pretty printers below. These can be installed using opam using the following command `opam install printbox printbox-text`

These pretty printers produce a so-called `box` which is the terminology that `printbox` uses to refer to formatted, structured texts. A `box` ca be printed as follows:

```
PrintBox_text.output stdout (Pretty.program_to_tree prog)
```

This will print the AST of the program as a tree.

6.8.1 pretty printer for ASTs (module `Pretty`)

```
module PBox = PrintBox
open Ast

(* producing trees for pretty printing *)
let typ_style = PBox.Style.fg_color PBox.Style.Green
let ident_style = PBox.Style.fg_color PBox.Style.Yellow
let fieldname_style = ident_style
let keyword_style = PBox.Style.fg_color PBox.Style.Blue

let info_node_style = PBox.Style.fg_color PBox.Style.Cyan
```

(continues on next page)

(continued from previous page)

```
let make_typ_line name = PBox.line_with_style typ_style name
let make_fieldname_line name = PBox.line_with_style fieldname_style name
let make_ident_line name = PBox.line_with_style ident_style name
let make_keyword_line name = PBox.line_with_style keyword_style name

let make_info_node_line info = PBox.line_with_style info_node_style info

let ident_to_tree (Ident {name; _}) = make_ident_line name

let typ_to_tree tp =
  match tp with
  | Bool _ -> make_typ_line "Bool"
  | Int _ -> make_typ_line "Int"

let binop_to_tree op =
    match op with
    | Plus _ -> make_keyword_line "PLUS"
    | Minus _ -> make_keyword_line "Minus"
    | Mul _ -> make_keyword_line "Mul"
    | Div _ -> make_keyword_line "Div"
    | Rem _ -> make_keyword_line "Rem"
    | Lt _ -> make_keyword_line "Lt"
    | Le _ -> make_keyword_line "Le"
    | Gt _ -> make_keyword_line "Gt"
    | Ge _ -> make_keyword_line "Ge"
    | Lor _ -> make_keyword_line "Lor"
    | Land _ -> make_keyword_line "Land"
    | Eq _ -> make_keyword_line "Eq"
    | NEq _ -> make_keyword_line "NEq"

let unop_to_tree op =
  match op with
  | Neg _ -> make_keyword_line "Neg"
  | Lnot _ -> make_keyword_line "Lnot"

  let rec expr_to_tree e =
    match e with
    | Integer {int; _} -> PBox.hlist ~bars:false [make_info_node_line "IntLit("; PBox.
↪line (Int64.to_string int); make_info_node_line ")"]
    | Boolean {bool; _} -> PBox.hlist ~bars:false [make_info_node_line "BooleanLit(";␣
↪make_keyword_line (if bool then "true" else "false"); make_info_node_line ")"]
    | BinOp {left; op; right; _} -> PBox.tree (make_info_node_line "BinOp") [expr_to_
↪tree left; binop_to_tree op; expr_to_tree right]
    | UnOp {op; operand; _} -> PBox.tree (make_info_node_line "UnOp") [unop_to_tree␣
↪op; expr_to_tree operand]
    | Lval l -> PBox.tree (make_info_node_line "Lval") [lval_to_tree l]
    | Assignment {lvl; rhs; _} -> PBox.tree (make_info_node_line "Assignment") [lval_
↪to_tree lvl; expr_to_tree rhs]
    | Call {fname; args; _} ->
      PBox.tree (make_info_node_line "Call")
        [PBox.hlist ~bars:false [make_info_node_line "FunName: "; ident_to_tree␣
↪fname];
         PBox.tree (make_info_node_line "Args") (List.map (fun e -> expr_to_tree e)␣
↪args)]
  and lval_to_tree l =
    match l with
    | Var ident -> PBox.hlist ~bars:false [make_info_node_line "Var("; ident_to_tree␣
```

(continues on next page)

(continued from previous page)

```
 ↪ident; make_info_node_line ")"]

let single_declaration_to_tree (Declaration {name; tp; body; _}) =
  PBox.tree (make_keyword_line "Declaration")
    [PBox.hlist ~bars:false [make_info_node_line "Ident: "; ident_to_tree name];
    PBox.hlist ~bars:false [make_info_node_line "Type: "; Option.fold ~none:PBox.
 ↪empty ~some:typ_to_tree tp];
    PBox.hlist ~bars:false [make_info_node_line "Body: "; expr_to_tree body]]

let declaration_block_to_tree (DeclBlock {declarations; _}) =
PBox.tree (make_keyword_line "VarDecl")  (List.map single_declaration_to_tree␣
 ↪declarations)

let for_init_to_tree = function
| FIDecl db -> PBox.hlist ~bars:false [PBox.line "ForInitDecl: "; declaration_block_
 ↪to_tree db]
| FIExpr e -> PBox.hlist ~bars:false [PBox.line "ForInitExpr: "; expr_to_tree e]

let rec statement_to_tree c =
  match c with
  | VarDeclStm db -> PBox.hlist ~bars:false [PBox.line "DeclStm: "; declaration_block_
 ↪to_tree db]
  | ExprStm {expr; _} -> PBox.hlist ~bars:false [make_info_node_line "ExprStm: ";␣
 ↪Option.fold ~none:PBox.empty ~some:expr_to_tree expr]
  | IfThenElseStm {cond; thbr; elbro; _} ->
    PBox.tree (make_keyword_line "IfStm")
      ([PBox.hlist ~bars:false [make_info_node_line "Cond: "; expr_to_tree cond];␣
 ↪PBox.hlist ~bars:false [make_info_node_line "Then-Branch: "; statement_to_tree␣
 ↪thbr]] @
       match elbro with None -> [] | Some elbr -> [PBox.hlist ~bars:false [make_info_
 ↪node_line "Else-Branch: "; statement_to_tree elbr]])
  | WhileStm {cond; body; _} ->
    PBox.tree (make_keyword_line "WhileStm")
      [PBox.hlist ~bars:false [make_info_node_line "Cond: "; expr_to_tree cond];
       PBox.hlist ~bars:false [make_info_node_line "Body: "; statement_to_tree body]]
  | ForStm {init; cond; update; body; _} ->
    PBox.tree (make_keyword_line "ForStm")
      [PBox.hlist ~bars:false [make_info_node_line "Init: "; Option.fold ~none:PBox.
 ↪empty ~some:for_init_to_tree init];
       PBox.hlist ~bars:false [make_info_node_line "Cond: "; Option.fold ~none:PBox.
 ↪empty ~some:expr_to_tree cond];
       PBox.hlist ~bars:false [make_info_node_line "Update: "; Option.fold ~
 ↪none:PBox.empty ~some:expr_to_tree update];
       PBox.hlist ~bars:false [make_info_node_line "Body: "; statement_to_tree body]]
  | BreakStm _ -> make_keyword_line "BreakStm"
  | ContinueStm _ -> make_keyword_line "ContinueStm"
  | CompoundStm {stms; _} -> PBox.tree (make_info_node_line "CompoundStm") (statement_
 ↪seq_to_forest stms)
  | ReturnStm {ret; _} -> PBox.hlist ~bars:false [make_keyword_line "ReturnValStm: ";␣
 ↪expr_to_tree ret]
and statement_seq_to_forest stms = List.map statement_to_tree stms

let program_to_tree prog =
  PBox.tree (make_info_node_line "Program") (statement_seq_to_forest prog)
```

6.8.2 pretty printer for ASTs (module `TypedPretty`)

The typed AST has not changed compared to the previous assignment. Hence, there is no need to update the `TypedPretty` module.

6.8.3 C runtime

The C runtime we use has not changed compared to the previous assignment. See the corresponding description in the *previous assignment*

CHAPTER

SEVEN

DOLPHIN – PHASE 4

Attention

This is a group assignment. The workload is calibrated for a group of 3. Please also see the *recommended workflow* section in the Appendix below.

In case of questions regarding ambiguity of what you should do, ask questions on the forum. If you are in doubt and there is no enough time, use your best judgment and explain your reasoning in your report.

7.1 Assignment overview

This assignment builds on the previous three assignments: *Phase 1*, *Phase 2*, and *Phase 3*.

Phase 4 of Dolphin extends the language with two new features

1. Top-level functions, and
2. Comma expressions that are used to sequentialize expressions.

There are 5 tasks and no questions in this assignment. There is one *glory* task. Do it only if you have the time and feel ambitious.

7.1.1 What you need to get started

- This assignment is a continuation of the previous assignment. You will need to edit the code from the previous assignment.
- You will need to understand Menhir parser generator. See Menhir documentation.

7.1.2 What you need to hand in

Please hand in a `.zip` file containing the following.

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each task and question, briefly (1 – 4 sentences) describe your implementation and answer. Write concisely.
2. All the source files needed to reproduce your solution. This also includes the `C` code provided. Please explain in your report how the solution could be reproduced, e.g., calling `make` (if you have created a `Makefile`), the command line to call `clang`, etc.
3. All the tests that you create (see Task 5) should be placed into a directory `assignment-07-tests` as individual `.dlp` files.

> **Important**
>
> Make sure to understand all the code you hand in.

7.2 Functions in Dolphin

Phase 4 Dolphin program consists of a list of one or more function declarations. The following applies to function declarations.

1. The syntax for function declarations is `<return-type> <function-name> ( <arguments> ) { <function-body> }`, where `<arguments>` is a list of zero or more comma-separated pairs of the form `<argument-name>: <argument-type>`. Note that both the return type and the argument types must be explicitly given. The following is an example of a declaration of function named `f` with return type `int` that takes two arguments `x` and `y`, both of type `int`.

```
/* valid function declaration */
int f (x: int, y : int) {
 return x+y;
}
```

2. The information about the function name, return type, and the argument number and types, is called *function signature*.
3. Exactly one function in the program must have name `main` with the return type `int`, and no arguments.
4. Within a program, all function names must be unique.
5. Within a function, all argument names must be unique. For example, the following declaration is invalid

```
/* invalid function declaration
   - duplicate argument x
*/
int f (x: int, x : int) {
 return x;
}
```

6. Within a function, lexical scoping rules apply.
7. Arguments can be modified in the function as if they were local variables.

```
/* valid program */
int f (x:int) {
    x = x * 10;
    return x;
}

int main () {
    var x:int = 1;
    var y = f (x);
    return x + y ; /* returns 11 */
}
```

8. Within a program, all functions are mutually recursive. For example, in the following program, function `odd` is accessible from `even` despite their declaration order.

```
/* even-odd example
   - purpose: showcases mutual recursion
   - this program is valid.
   - returns 1
*/
int even (x:int) {
  if (x == 0) {
    return 0;
  } else {
    return odd (x - 1);   /* odd is in scope */
  }
}

int odd (x: int) {
  if (x == 0) {
    return 1;
  } else {
    return even (x - 1);  /* even is in scope */
  }
}

int main () {
  return even (5);
}
```

9. If the return type of the function is not `void`, all paths in the program must return. See, for example, function `even` above. The type of the return statements must agree with the return-type of the function. In void functions, the return statement may be omitted.
10. Variables and function names share their namespace.

7.3 Comma expressions

Comma expressions are a form of expressions that are separated with a comma. The semantics is that of *sequencing the evaluation of the expressions*. The expression to the left of the comma is evaluated before the expression to the right of the comma. The result of the comma expression is that of the right subexpression. In other words, the result of the left-subexpression is ignored. Comma expressions are useful for their side-effects, e.g., assignments or side-effects through function calls. A common use case for comma expressions is in the update expression in `for`-loops.

```
int main () {
    var _o = get_stdout ();
    for ( var i: int = 0, j: int = 9
         ; i < 10
         ; i = i+1, j=j-1 /* obs: comma expression here */ )
   {
        output_string (int_to_string (i), _o);
        output_string (" ", _o);
        output_string (int_to_string (j), _o);
        output_string ("\n", _o);
   }
    return 0;
}
```

The following applies to comma experissions.

1. In particular, comma expressions are not valid expression statements. That is, as in the earlier phases, a valid expression statement can either be an assignment, or a function call. In particular, the following program is invalid.

```
int main () {
    var x = 0;
    x = 1,x = 2; /* invalid expression statement                        */
                 /* the programmer should use semi-colon instead ;)    */
    return x;
}
```

The following program, however, is valid, because the comma expression appears as the right-hand side of the assignment expression statement.

```
/* valid program */
int main () {
    var x = 0;
    x = (1,2); /* valid expression statement */
    return x;  /* returns 2 */
}
```

Note

C allows comma expression statements.

2. Comma expressions cannot appear as the top-level expression in the arguments to function calls.

7.4 Abstract syntax trees

Task 1: Design and implement an AST and Typed AST for Phase 4

The structure of the new AST should follow the idea that a program is as list of functions as described *earlier in the document*.

7.5 Parser

Task 2: Extend your parser to support new language features

Note that lexer does not need to change.

7.6 Semantic analysis

In order to tackle mutual recursion, the type checking can be done with two passes over the AST.

1. In the first pass, check the validity of all function signatures only. If signatures of all functions are valid, gather them into a function environment (similar to one for the build-in functions) that will be used in pass two.
2. In the second pass, check the validity of each function body using the environment created in the previous pass.

Task 3: Extend your semantic analysis to support new language features

Extend the error module as needed.

7.7 Code generation

For code generation, we map each Dolphin function to an LLVM function. Each generated LLVM function has the same number of arguments as the source Dolphin function that it corresponds to. Because arguments may be used as local variables, the generated function needs to copy them from the LLVM to stack `alloca`-ted memory. For example, the source level function `int f (x:int) { x = x * 10; return x; }` when translated to LLVM may look as follows

```
define i64 @dolphin_fun_f (i64 %x_arg) {
 ; Allocate memory for copying the argument %x_arg.
 %x_local_arg_copy = alloca i64
 ; Copy the argument
 store i64 %x_arg, i64* %x_local_arg_copy
 ; Identifier %x_arg is not used from this point on.
 ; The rest of the function uses %x_local_arg_copy.
 ...
}
```

Task 4: Design and implement code generation for Phase 4

7.8 Testing and consolidation

Because of the changes in the syntax the tests from the previous phases will not immediately work. To port them, wrap them as `int main() { ... }` functions. Additionally, create at least 10 new test programs that test the functionality of the new phase.

All the tests should be included into your submission as `.dlp` files.

Task 5: Testing and consolidation

Port your old tests and create new ones as specified by the description above.

Task 6 (glory): Add support for tail call optimization

Full LLVM supports tail call annotations. See `tail` or `musttail` annotations on call instructions in LLVM documentation. Add support for tail-call optimization.

7.9 Appendix

7.9.1 C runtime

The C runtime we use has not changed compared to the previous assignment. See the corresponding description in the *previous assignment*

7.9.2 Recommended workflow

1. Start by designing the AST and the Typed AST declarations for both functions and comma expressions.
2. Tackle function and comma expressions separately.
3. Start with functions, and proceed with modifying and debugging each phase that needs to be changed in this assignment. a. if you decide to split work among group members, each of the parser, semantic analysis, and code generator, can be worked on independently, for as long as you adhere to the AST interfaces, and know how to put together the individual pieces. b. if you decide to work on the assignment in the order of the phases, work from the parser to semantic analysis to code generation.
4. Move onto comma expressions, and implement support for them in the parser, semantic analysis, and code generation, as in the previous step.

CHAPTER

EIGHT

DOLPHIN – PHASE 5

Attention

This is a group assignment. The workload is calibrated for a group of 3. Please also see the *recommended workflow* section in the Appendix below.

In case of questions regarding ambiguity of what you should do, ask questions on the forum. If you are in doubt and there is no enough time, use your best judgment and explain your reasoning in your report.

8.1 Assignment overview

This assignment extends the language with three language features: records, arrays, and strings. For examples of programs that use these features, we refer to the *Exercises for week 7*, and the example programs provided with this assignment.

There are 6 tasks and no questions in this assignment. There is one *glory* task. Do it only if you have the time and feel ambitious.

8.1.1 What you need to get started

- This assignment is a continuation of the previous assignment. You will need to edit the code from the previous assignment.
- The following features of LLVM that we have not previously used are relevant for this assignment
  - *LLVM named type definitions* and *agggregate types*.
  - *LLVM global identifiers* and string literals.
  - LLVM cast operations such as `bitcast` and `ptrtoint`.
  - LLVM's `getelemntptr` instruction. See *our reference material on GEP* and *Exercises for week 11*.
- You need to understand OCamllex lexer generator. Refer to OCamllex documentation for details.
- You need to understand Menhir parser generator. Refer to Menhir documentation for details.

8.1.2 What you need to hand in

Please hand in a `.zip` file containing the following.

1. A brief report documenting your solution. Acceptable report formats are `.pdf`, `.rtf`, and `.md`. For each task and question, briefly (1 – 4 sentences) describe your implementation and answer. Write concisely.
2. All the source files needed to reproduce your solution. This also includes the `C` code provided. Please explain in your report how the solution could be reproduced, e.g., calling `make` (if you have created a `Makefile`), the command line to call `clang`, etc.
3. All the tests that you create (see Task 6) should be placed into a directory `assignment-08-tests` as individual `.dlp` files.

Important

Make sure to understand all the code you hand in.

8.2 Records in Dolphin

Records in Dolphin are similar to structs in C (or Java classes without methods). They are the only form of user-defined data types, allow the grouping of data of different types under a single entity. Record types are declared using `record` keyword, which has the following syntax:

```
record <record-type-name> {
  <field-name-1>:<type-1>;
  <field-name-2>:<type-2>;
  ...
  <field-name-n>:<type-n>;
}
```

For example,

`record Tuple { x: int; y : int; }`. A record type is referred to in the program using its name, for example via `var x:MyRec`.

New records are declared using the syntax

```
new <record-type-name> {
  <field-name-1> = <field-init-expression-1>;
  <field-name-2> = <field-init-expression-2>;
  ...
  <field-name-n> = <field-init-expression-n>;
}
```

where each `<field-init-expressions>` must have the type corresponding to the field it initializes.

Given an expression `e` of record type, its field `f` is accessed using the dot notation: `e.f`. The keyword `nil` denotes an invalid record of any record type, like `null` in Java; attempting to access one of its fields should be reported as an error at runtime. Below is an example of a simple program with records:

```
record Tuple { x: int; y : int ; }

int main () {
    var a:Tuple = new Tuple { x = 0; y = 1; };
```

(continues on next page)

(continued from previous page)

```
    var b:Tuple = nil;
    return a.x;  /* dot notation to refer to the field `x` of variable `a` */
}
```

The following applies to records and their use in Dolphin programs.

1. All records are declared at the top-level. A Dolphin program is a collection of function and record declarations.
2. All records are mutually recursive.
3. Within a program, all record names must be unique.
4. The following record names are *reserved* for the standard library. They may not be used in the declarations of the user-defined records: `stream`, `socket`, `socket_address`, `ip_version`, `accepted_connection`, `udp_recvfrom_result`, and `connection_type`.
5. The only way to obtain non-nil values for the reserved records is through standard library. In particular, reserved records cannot be created in the program using the `new` keyword.
6. All fields within the record must be unique.
7. The number of fields may be zero.
8. At record creation, the fields may appear in any order. For example, `var a:Tuple = new Tuple { y = 1; x = 0; };` is a valid record creation.
9. All fields must be initialized. If a field initialization is missing, an error must be reported.
10. The only binary operations allowed on the records are equalty and inequality. Records are compared by reference. The following example program illustrates record equality.

```
/* valid program; returns 1 */
record Tuple { x: int; y : int ; }

int main () {
    var a:Tuple = new Tuple { x = 0; y = 1;};
    var b:Tuple = new Tuple { x = 0; y = 1;};
    var c = a;
    if (b == c) {
        return 0;
    }
    if (a == c) {
        return 1;
    }
    return 2;
}
```

11. If one of the operands of equality (or inequality) is a record, the other operand must be either (a) another record of the same type, or (b) `nil`.

> **Note**
>
> This particular design of record comparison is compatible with mainstream languages such as Java and C. This also means that other ways of definging equality, i.e., structurally, have to be implemented in code, e.g., by writing a function `bool tuple_eq (Tuple t1, Tuple t2)`.

8.3 Arrays in Dolphin

Arrays in Dolphin are similar to arrays in languages such as Java or C. They hold a fixed number of values of a single type. The type of the array is denoted using the syntax `[<element-type>]`, e.g., `[int]` is the type of array of integers. Arrays are initialized using the syntax `new <element-type> [<length-expression>]`, where `<length-expression>` must evaluate to an integer. For example, `var x = new int[1+3]` initializes `x` to be an array of 4 elements. The length of the arrays is, in general, not known at compile time. Arrays are indexed using the bracket notation `<variable-name> [ <index-expression> ]`.

The following applies to arrays:

1. All array accesses – reading and writing – are bounds-checked.
2. Similar to records, the only allowed binary operations on arrays are equality and inequality. Similar to records, the equality checks are done by reference.
3. The number of array elements must be non-negative.
4. After array creation, the length of the array cannot change at runtime.
5. Array length is accessed using `length_of (<expression>)`, where `<expression>` must evaluate to an array, and `length_of` is a keyword.

8.4 Strings in Dolphin

In Dolphin, strings are a built-in type. They can be created in one of the following ways:

- using string literals in the program source, e.g,. `"Hello"`
- using Dolphin standard library functions, e.g., `string_concat ("Hello", "World")` concatenates two strings.

The following applies to strings:

1. The binary operations on strings are equality, inequality, and string comparison. Unlike arrays and records, strings are compared by value.
2. String literals may include escape characters: for example '\n' stands for a newline. Dolphin follows OCaml's lexical convention for representing escape sequences.
3. String literals may span over several lines.
4. The length of a string can be obtained using `length_of` keyword.

The following example program illustrates some string operations:

```
int main() {
    var x = "hello
world";
    var y = "hello\nworld";
    if (x == y) {
        return 0;
    }
    return length_of(x);
}
```

8.5 Abstract Syntax Tree

Task 1: Design and implement an AST and Typed AST for Phase 5.

As you work on this task, take the following aspects into account.

8.5.1 Type representation

We suggest the following OCaml data structures for representing strings, arrays, and records.

```
(* suggested code snippet to incorporate into your AST *)
type recordname = RecordName of {name : string; loc : Loc.location}
type fieldname = FieldName of {name : string; loc : Loc.location}
type ident = Ident of {name : string; loc : Loc.location}

type typ =
| ...  (* placeholder for previous constructors *)
| Str of {loc : Loc.location}
| Array of {typ : typ; loc : Loc.location}
| Record of {recordname : recordname}
```

8.5.2 Program structure

The structure of the new AST should follow the idea that a program is a list of functions or record declarations. You should create a way of representing record declarations in the AST.

8.5.3 Expressions

To accommodate the new features, we suggest to extend your `expr` type by adding constructors for the following:

- record creation
- array creation
- `nil` expression
- string values
- the `length_of` keyword

8.5.4 Extending Lvals

In programming languages, *lvals*, correspond to the program entities that may appear to the *left* of the assignment statement, hence the use of the letter *l* in the name. In previous phases, lvals were only identifiers. With the addition of records and arrays, the space of lvals is now richer. It includes indexing into array or a record field, as well as their combination, e.g., `f().x[1+g()].y[2].z`.

With regards to extending the AST to support lvals, we suggest the following.

```
type expr =
...
and lval =
| Var of ident
| Idx of { arr   : expr
         ; index : expr
         ; loc   : Loc.location
         }
| Fld of { rcrd  : expr
         ; field : fieldname (* see declaration of fieldname earlier *)
         ; loc   : Loc.location
         }
```

8.6 Lexer

Task 2: Extend the lexer to support the new language features

As you work on this task, recognize what new tokens need to be added to the language. For full support of strings, you may need to add a new lexer rule, in order to properly treat escape characters and strings spanning multiple lines. Note that because Dolphin follows OCaml's specification of escape characters, we can use the OCaml standard library function `Scanf.unescape` to recognize escape characters.

8.7 Parser

Task 3: Extend the parser to support the new language features

To score full points on this task, your parser must not have any shift/reduce or reduce/reduce conflicts. As you work on this task, pay attention to proper parsing of lvals in your parser. Consult the examples and the specification earlier in this section regarding the correct syntax, i.e., the use of semicolons when delimiting fields in records - and write your own tests based on the specification.

8.8 Semantic analysis

Task 4: Extend the semantic analysis to support the new language features

8.8.1 Record declarations

In the implementation of semantic analysis, pay special attention to the mutual recursion of records. Similarly to the mutual recursion of functions, use two passes over the record declarations.

1. Identify all record names, and collect them into a data-structure (a list or a set).
2. Check each record individually, ensuring that the user-defined fields correspond to valid types.

8.8.2 Checking functions

Once the record declarations are checked, the information about all the records can be collected into an environment where their names map to their types. This environment should also include the reserved records. Use this resulting environment when type checking function signatures and bodies.

8.9 Code generation

Task 5: Extend the code generation to support the new language features

We start off by going over the runtime and standard library integration.

8.9.1 Runtime and standard library integration

Before you proceed with code generation, it is necessary to port your project to the new runtime. The runtime and the standard library are split across several modules, because there is a qualitative distinction between the runtime and the standard library (unlike in the earlier phases), which we explain below. We have the following files.

1. `runtime.c` contains the extended core runtime functionality, for operations such as record and array allocation, string equality, etc. What makes these functions part of the runtime (as opposed to the standand library) is that none of these C functions are exposed to the programmer. It is the compiler's code generator that embeds calls to them, based on the (typed) AST.
2. `stdlib.c` contains Dolphin standard library that includes functions that are user-visible. These include a number of "everyday" functionality, such as functions for string concatenation, printing a string, etc. This is a relatively large module.
3. `runtime.h` is a header file that is included from `stdlib.c`.

These files are to be downloaded from Brightpsace.

Representation of reserved records and strings

We map reserved records, e.g., `stream` to empty LLVM records. This is sufficient because When passing these arguments back and forth to the runtime, we will only use pointers to these records. This also means that we could have represented the reserved records as generic pointers, i.e., `i8*`; but the empty record approach has an added benefit of tagging the intended use of the signature functions (though the typing guarantees are weak because they can be overcome via casts).

Because strings are represented as LLVM structs, we also need to include a definition of the string type.

Declaring external functions and types

Because your generated LLVM file uses the external functionality provided by the runtime and standard library, it needs to include `@declare` instructions to let LLVM know of the the function signatures and that they are implemented elsewhere.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; The following declarations should be included in the generated LLVM file ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

(continues on next page)

(continued from previous page)

```
; LLVM struct corresponding to the reserved stream type
%dolphin_record_stream = type {  }

; LLVM struct correpsonding to the string representation
%string_type = type {i64, i8* }

@dolphin_rc_empty_string = external global %string_type

; signatures for the runtime functions
; -- not user visible --
declare i64 @compare_strings(%string_type*, %string_type*)
declare i8* @allocate_record(i32)
declare i8* @raw_allocate_on_heap(i32)
declare i8* @allocate_array(i32, i64, i8*)
declare void @report_error_array_index_out_of_bounds()
declare void @report_error_nil_access()
declare void @report_error_division_by_zero()
declare i64 @string_length(%string_type*)

; signatures for the core standard library
; -- these are user visible --
declare %string_type* @bytes_array_to_string(i8*)
declare i8* @string_to_bytes_array(%string_type*)
declare i64 @byte_to_int_unsigned(i8)
declare i64 @byte_to_int_signed(i8)
declare i8 @int_to_byte_unsigned(i64)
declare i8 @int_to_byte_signed(i64)
declare i64 @ascii_ord(%string_type*)
declare %string_type* @ascii_chr(i64)
declare %string_type* @string_concat(%string_type*, %string_type*)
declare %string_type* @substring(%string_type*, i64, i64)
declare %string_type* @int_to_string(i64)
declare i64 @string_to_int(%string_type*)
declare i64 @input_byte(%dolphin_record_stream*)
declare i1 @output_byte(i8, %dolphin_record_stream*)
declare i8* @input_bytes_array(i64, %dolphin_record_stream*)
declare void @output_bytes_array(i8*, %dolphin_record_stream*)
declare void @output_string(%string_type*, %dolphin_record_stream*)
declare i1 @seek_in_file(i64, i1, %dolphin_record_stream*)
declare i64 @pos_in_file(%dolphin_record_stream*)
declare i1 @close_file(%dolphin_record_stream*)
declare i1 @flush_file(%dolphin_record_stream*)
declare i1 @error_in_file(%dolphin_record_stream*)
declare i1 @end_of_file(%dolphin_record_stream*)
declare i64 @get_eof()
declare %dolphin_record_stream* @open_file(%string_type*, %string_type*)
declare %dolphin_record_stream* @get_stdin()
declare %dolphin_record_stream* @get_stderr()
declare %dolphin_record_stream* @get_stdout()
declare %string_type** @get_cmd_args()
declare void @exit(i64)


; array length utility functions
define i32 @dolphin_rc_compute_array_length_size () {
 %size_ptr = getelementptr i64, i64* null, i64 1
 %size = ptrtoint i64* %size_ptr to i32
```

(continues on next page)

(continued from previous page)

```
 ret i32 %size
}

define void @dolphin_rc_set_array_length (i64* %array, i64 %size) {
 %len_ptr = getelementptr i64, i64* %array, i64 -1
 store i64 %size, i64* %len_ptr
 ret void
}

define i64 @dolphin_rc_get_array_length (i64* %array) {
 %len_ptr = getelementptr i64, i64* %array, i64 -1
 %size= load i64, i64* %len_ptr
 ret i64 %size
}
```

8.9.2 Records

Record representation

We represent records using LLVM structs. The translation is quite straightforward and is one-to-one. A Dolphin record with `n` fields is translated to an LLVM struct with `n` fields. The types of the LLVM fields should correspond to the Dolphin types. For example, the Dolphin declaration of two types.

```
record T1 { x: int; t2 : T2;}
record T2 { y: int; t1 : T1;}
```

is translated to LLVM as follows

```
%dlp_rec_T2 = type { i64, %dlp_rec_T1* }
%dlp_rec_T1 = type { i64, %dlp_rec_T2* }
```

Note that the choice of the naming `%dlp_rec_T2` here is compiler-chosen; you may chose a different naming convention. What is important is that the code generation phase of the compiler is aware of the mapping between the source and the target types, and that of course the generated LLVM code is valid.

Nil representation

Nil record values can be represented as `null` in LLVM.

Record initialization

Allocation of a record is done through the runtime function `allocate_record`. This function takes an argument corresponding to the size of the record. Because the size depends on LLVM, we need to implement an architecture-independent way of obtaining the size information. This is accomplished using the GEP size hack[Lat05] as illustrated below. To actually save the payload of the record, we further need to use a combination of GEP and store instructions.

For example, consider the following record creation

```
var t2 = new T2{y = 10; t1 = t1; }  /* for some previously computed value `t1` */
```

The LLVM code for it can look as follows (we abbreviate `getelementptr` as `<GEP>` for brevity in the listing).

In this example, we need to use `bitcast` to cast the generic pointer `i8*` returned by the runtime allocation function into the LLVM type. Note that casts typically are erased in the LLVM to x86 translation; in other words, they have no performance overhead.

```
; Suppose %var_t1 and %var_t2 are the identifiers
; that we alloca-ed for the source level t1, t2

; GEP size hack
%size_ptr = getelementptr %dlp_rec_T2, %dlp_rec_T2* null, i32 1
; Cast ptr to integer
%size = ptrtoint %dlp_rec_T2* %size_ptr to i32
; Call into the runtime for allocation
%ptr_rt = call i8* @allocate_record (i32 %size)

; Turn generic pointer into a more specific one, so we can use GEP
%t2_ptr = bitcast i8* %ptr_rt to %dlp_record_T2*

; Access field y of the struct
%ptr_field_y_of_var_t2 = getelementptr %dlp_rec_T2, %dlp_rec_T2* %t2_ptr, i32 0, i32 0
; Save 10 in the field y
store i64 10, i64* %ptr_field_y_of_var_t2

; Read from %var_t1
%ptr_t1 = load %dlp_rec_T1*, %dlp_rec_T1** %var_t1
; Access field t1 of the struct
%ptr_field_t1_of_var_t2 = getelementptr %dlp_rec_T2, %dlp_rec_T2* %t2_ptr, i32 0, i32␣
↪1
; Save in the field t1
store %dlp_rec_T1* %ptr_t1, %dlp_rec_T1** %ptr_field_t1_of_var_t2

; Save in %var_t2
store %dlp_rec_T2* %t2_ptr %dlp_rec_T2** %var_t2
```

Record access

To access the record, we need to use the GEP instruction similarly to how it is used in the initialization above.

8.9.3 Array translation

Array representation

An array is represented as a contiguous block of memory, with the metadata information about the length stored in memory before the content.

```
┌──────────┬──────────────────────────────────────────┐
│ Length   │                Array content             │
└──────────┴──────────────────────────────────────────┘

▲          ▲
|          |
|          The pointer to the start of the array
|
|
Length is accessed by casting the pointer to the start of the
array to i64* and using GEP with index -1
```

Array initialization

Array initialization takes place via function `allocate_array` in the runtime. Note that the last argument to that function needs to include the pointer to the initialization value.

To correctly access the array length, our runtime depends on three functions to be generated by our compiler. These functions are declared in the header file `dolphin_rc.h`.

```
/* dolphin_rc.h - include this file in your project */
#include <stdint.h>

extern int32_t dolphin_rc_compute_array_length_size();
extern void dolphin_rc_set_array_length(void *, int64_t);
extern int64_t dolphin_rc_get_array_length(void *);
```

Array access

To access the `i`-th element of the array (counting from zero), we can use GEP instruction with `i` as the first index.

8.9.4 String translation

Runtime representation

Strings are represented at runtime as C structs of the type.

```
struct string { int64_t len; char * contents; };
```

String literals in LLVM code.

String literals are represented in the source as global identifiers.

A string literal is represented as an LLVM global of LLVM array type. Recall that LLVM arrays have static length (and are only used here to represent string literals). Consider the following source program and the associated translation.

```
int main () {
    var x = "Hello World\n";
    output_string (x, get_stdout());
    return 0;
}
```

This corresponding simplified LLVM code can look as follows

```
;; external functions and reserved type declarations omitted

@str_lit_struct = global { i64, [12 x i8]* } {i64 12, [12 x i8]* @str_lit}
@str_lit = global [12 x i8] c"Hello World\0A"

define i64 @dolphin_fun_main () {
 %var_x = alloca %string_type*
 %str_cast = bitcast { i64, [12 x i8]* }* @str_lit_struct to %string_type*
 store %string_type* %str_cast, %string_type** %var_x
 %tmp_1 = load %string_type*, %string_type** %var_x
 %tmp_2 = call %dolphin_record_stream* @get_stdout ()
```

(continues on next page)

(continued from previous page)

```
  call void @output_string (%string_type* %tmp_1, %dolphin_record_stream* %tmp_2)
  ret i64 0
}
```

8.9.5 Comparing strings

String comparison is to be translated to calls to the runtime function `compare_strings`. This function compares two strings lexicographically. It returns `0` if the strings are identical, `-1` if the first strings is less than the second one, and `1` if the first argument is greater than the second one.

8.10 Consolidation and testing

Task 6: Put all the phases together and test your functionality

- Add at least 10 new tests that check for the negative behavior in the semantic analysis and the frontend that you have implemented.
- Add at least 10 new tests that check for the positive behavior of the frontend.
- Run your compiler on the provided example programs and check their behavior.

Describe why your tests are useful.

8.10.1 Glory task

Task 7 (glory): Add support for the full standard library

Full standard library includes networking API. This will allow you to run the HTTP server example. See *full standard library signature* in the Appendix.

8.11 Appendix

8.11.1 Example programs

We provide a handful of example programs together with their expected output. Download them from brightspace.

8.11.2 Recommended workflow

As a general rule of thumb, work your way very slowly through the assignment, especially because as you add code generation, the size of the generated LLVM programs will grow substantially, and it is crucial that you understand it. Read and test the LLVM code you generate as you incrementally add new aspects of code generation. For example, test record creation immediately after you implement it, even before you complete other aspects of records, i.e., record lookups.

At a high level, we suggest the following order for the assignment:

1. Implement the most rudimentary support for strings in the frontend. In particular, for lexing, ignore escape characters and newlines.
2. Add runtime integration and code generation for strings. At this point, you should be able to compile and run something as simple as

```
int main () {
  output_string ("Hello World!", get_stdout());
  return 0;
}
```

3. Add support for records through all the compiler phases.
4. Add support for arrays through all the compiler phases. Get the core functionality of arrays (creation, lookup, update) working first, and add bounds-checking afterwards.
5. Add support for lexing of complex strings, including escape characters and newlines.
6. Consolidate.

8.11.3 Full standard library signature

```
%dolphin_record_udp_recvfrom_result = type { i8*, %dolphin_record_socket_address* }
%dolphin_record_accepted_connection = type { %dolphin_record_socket*, %dolphin_record_
 ↪socket_address* }
%dolphin_record_socket = type {  }
%dolphin_record_socket_address = type {  }
%dolphin_record_ip_address = type {  }
%dolphin_record_ip_version = type {  }
%dolphin_record_connection_type = type {  }
%dolphin_record_stream = type {  }
%string_type = type {i64, i8* }

@dolphin_rc_empty_string = external global %string_type

declare i64 @compare_strings(%string_type*, %string_type*)
declare i8* @allocate_record(i32)
declare i8* @raw_allocate_on_heap(i32)
declare i8* @allocate_array(i32, i64, i8*)
declare void @report_error_array_index_out_of_bounds()
declare void @report_error_nil_access()
declare void @report_error_division_by_zero()
declare %dolphin_record_udp_recvfrom_result* @socket_recvfrom_udp(%dolphin_record_
 ↪socket*)
declare i64 @socket_sendto_udp(%dolphin_record_socket*, %dolphin_record_socket_
 ↪address*, i8*)
declare i1 @socket_close(%dolphin_record_socket*)
declare i1 @socket_activate_udp(%dolphin_record_socket*)
declare i1 @socket_connect(%dolphin_record_socket*, %dolphin_record_socket_address*)
declare %dolphin_record_accepted_connection* @socket_accept(%dolphin_record_socket*)
declare i1 @socket_listen(%dolphin_record_socket*, i64)
declare i1 @socket_bind(%dolphin_record_socket*, %dolphin_record_socket_address*)
declare i64 @get_port_of_socket_address(%dolphin_record_socket_address*)
declare %dolphin_record_ip_address* @get_ip_address_of_socket_address(%dolphin_record_
 ↪socket_address*)
declare %dolphin_record_socket_address* @create_socket_address(%dolphin_record_ip_
 ↪address*, i64)
```

(continues on next page)

(continued from previous page)

```
declare %string_type* @ip_address_to_string(%dolphin_record_ip_address*)
declare %dolphin_record_ip_address* @string_to_ip_address(%dolphin_record_ip_version*,
 ↪ %string_type*)
declare %dolphin_record_stream* @socket_get_output_stream(%dolphin_record_socket*)
declare %dolphin_record_stream* @socket_get_input_stream(%dolphin_record_socket*)
declare %dolphin_record_socket* @create_socket(%dolphin_record_ip_version*, %dolphin_
 ↪record_connection_type*)
declare %dolphin_record_ip_address* @get_ipv6_address_any()
declare %dolphin_record_ip_address* @get_ipv4_address_any()
declare %dolphin_record_ip_version* @get_ipv6()
declare %dolphin_record_ip_version* @get_ipv4()
declare %dolphin_record_connection_type* @get_tcp_connection_type()
declare %dolphin_record_connection_type* @get_udp_connection_type()
declare i64 @string_length(%string_type*)
declare %string_type* @bytes_array_to_string(i8*)
declare i8* @string_to_bytes_array(%string_type*)
declare i64 @byte_to_int_unsigned(i8)
declare i64 @byte_to_int_signed(i8)
declare i8 @int_to_byte_unsigned(i64)
declare i8 @int_to_byte_signed(i64)
declare i64 @ascii_ord(%string_type*)
declare %string_type* @ascii_chr(i64)
declare %string_type* @string_concat(%string_type*, %string_type*)
declare %string_type* @substring(%string_type*, i64, i64)
declare %string_type* @int_to_string(i64)
declare i64 @string_to_int(%string_type*)
declare i64 @input_byte(%dolphin_record_stream*)
declare i1 @output_byte(i8, %dolphin_record_stream*)
declare i8* @input_bytes_array(i64, %dolphin_record_stream*)
declare void @output_bytes_array(i8*, %dolphin_record_stream*)
declare void @output_string(%string_type*, %dolphin_record_stream*)
declare i1 @seek_in_file(i64, i1, %dolphin_record_stream*)
declare i64 @pos_in_file(%dolphin_record_stream*)
declare i1 @close_file(%dolphin_record_stream*)
declare i1 @flush_file(%dolphin_record_stream*)
declare i1 @error_in_file(%dolphin_record_stream*)
declare i1 @end_of_file(%dolphin_record_stream*)
declare i64 @get_eof()
declare %dolphin_record_stream* @open_file(%string_type*, %string_type*)
declare %dolphin_record_stream* @get_stdin()
declare %dolphin_record_stream* @get_stderr()
declare %dolphin_record_stream* @get_stdout()
declare %string_type** @get_cmd_args()
declare void @exit(i64)

define i32 @dolphin_rc_compute_array_length_size () {
 %size_of_length_addr$0 = getelementptr i64, i64* null, i64 1
 %size_of_length$1 = ptrtoint i64* %size_of_length_addr$0 to i32
 ret i32 %size_of_length$1
}

define void @dolphin_rc_set_array_length (i64* %array, i64 %size) {
 %array_length$2 = getelementptr i64, i64* %array, i64 -1
 store i64 %size, i64* %array_length$2
 ret void
}
```

(continues on next page)

(continued from previous page)

```
define i64 @dolphin_rc_get_array_length (i64* %array) {
 %array_length$3 = getelementptr i64, i64* %array, i64 -1
 %size$4 = load i64, i64* %array_length$3
 ret i64 %size$4
}
```

Part II

Exercises (TA sessions)

CHAPTER

NINE

EXERCISES FOR WEEK 2

In compiler implementation, it is often desired to inspect the data structers we work with. A common scenario includes working with an AST that we either process or synthesize, where we want to understand its shape. The necessary device in this case is *visualization* of our data structures. Often times, visualization techniques we use are very rudimentary, yet it helps to be familiar with them, which is the purpose of this exercise.

We will explore two visualization approaches. In both cases, we will use the AST for arithmetic expressions from Assignment 01.

```
(* Defining the type for binary operations *)
type binop = Add | Sub | Mul | Div

(* Defining the type for arithmetic expressions *)
type expr
  = Int of int                        (* Integer constant *)
  | BinOp of binop * expr * expr      (* Binary operation *)
```

9.1 Formatting output to console

Our first approach is to print the tree on console using indentation. Write function `print_tree` that when called would produce the output similar to Unix `tree` utility. For example, `print_tree (BinOp (Add, Int 1, BinOp (Mul, Int 2, Int 3)))` would generate the output

```
BinOp Add
├── Int 1
└── BinOp Mul
    ├── Int 2
    └── Int 3
```

Hint: if you are struggling with getting the ASCII connectives right, start with a simpler version that just uses the indentation

```
BinOp Add
    Int 1
    BinOp Mul
        Int 2
        Int 3
```

9.2 Formatting to graphviz

Another popular way of visualizing intermediate representations is to synthesize a `.dot` file to be used with Graphviz toolkit. Write a function `print_aexp_to_dot` that generates a .dot formatted output corresponding to the input expression. For example, for the expression above it can generate something like this

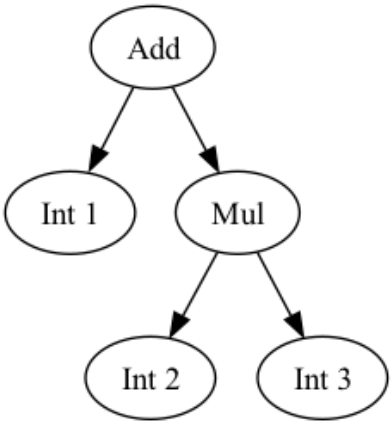
```
digraph AST {
  0 [label="Add"];
  1 [label="Int 1"];
  2 [label="Mul"];
  3 [label="Int 2"];
  4 [label="Int 3"];

  0 -> 1;
  0 -> 2;
  2 -> 3;
  2 -> 4;
}
```

Note that while the full spec of .dot format is quite extensive, it is sufficient to just the use the basic features, such as declaration of nodes and connectives, as in the example above.

If this output is saved to a file, e.g., `aexp.dot` we can invoke the `dot` command (which is available as part of graphviz package and should be already installed in the dev container) to get an image file, e.g.,

```
dot -Tpng aexp.dot -o aexp.png
```

will generate an image as follows



CHAPTER

TEN

EXERCISES FOR WEEK 3

10.1 Exercise overview

In compiler implementation, an essential data structure is that of *associative maps*, also often referred to as *environments*. For example, type checking uses environments for associating variables with types, code generation uses environments for mapping variables to target registers, and optimization phases use environments extensively, e.g., for tracking uses and definitions of variables. Note that while it is usual that keys in the environments are identifiers, (that is, variable names), that is not always the case. For example, in common subexpression elimination optimization that optimizes `x = a + b; y = a + b` to `x = a + b; y = x`, the keys are expressions such as `a + b`.

The exercises below explore four different approaches for implementing environments. For simplicity, we consider enviromnents where keys are strings and values are integers. The examples make use of OCaml's module system. This fulfills the secondary goal of these exercises to familiarize oneself with OCaml modules. To read up on OCaml module system, see Chapter 5 of the OCaml book.

The four approaches for implementing environments we consider are

1. List-based environments.
2. Functional environments.
3. Map-based environments.
4. Map-based environments with integer hashtable.

10.2 Relevant reading

For the implementation of approach 4, the relevant reading is Section 5.1 of Andrew Appel's book Modern Compiler Implementation in ML.

10.3 Environment signature

We start off by defining the type of our keys, the `NotFound` exception, and a module signature that will be shared by all four implementations

```
type value = int
exception NotFound (* we throw this exception when the key is not found *)

module type EnvSig = sig
  type env                    (* abstract environment type *)
```

(continues on next page)

(continued from previous page)

```
  type symbol                (* abstract symbol type      *)
  val empty_env : env        (* create an empty environment *)
  val insert : env -> symbol -> value -> env  (* insert; returns updated environment␣
↪*)
  val lookup : env -> symbol -> value         (* lookup; returns the value or raises␣
↪NotFound *)
  val symbol : string -> symbol    (* create a symbol value from string *)
  val approach_name: string (* the name of our approach, for benchmarking later *)
end
```

Most of the signature is standard, except perhaps the `symbol` type and the `symbol` function. They introduce a layer of indirection. Instead of inserting a key-value pair directly into the environment, we first convert the key into a so-called symbol value (that has type `symbol`) which we pass over as the argument to `insert`. This layer of indirection is needed by the last approach, which promises a bit of extra performance. We added it to the signature in order to parametrize the benchmarking; but everywhere except the last implementation, the `symbol` function is implemented as identity function `fun x -> x`, and the `symbol` type is a synonym to `string`.

For example, if E is a module implementing our environments, we can write a function such as

```
let example () =
  let open E in
  let e0 = empty_env in
  let x_sym = symbol "x" in
  let e1 = insert e0 x_sym 5 in
  let result = lookup e1 x_sym in
  Printf.printf "%d\n" result          (* prints 5 *)
```

10.4 List-based environments

Our first implementation uses associative lists as environments. See the following start of an implementation. Two functions are left to you as an exercise – they are the ones raising the `TODO` exception.

```
(* replace the code that raises TODO with your implementation *)
exception TODO

(* Approach 1 - list based environments  *)
module ListEnv : EnvSig = struct
  let approach_name = "List-based environments"
  type env = (string * value) list
  type symbol = string
  let symbol = fun x -> x
  let empty_env = []
  let insert e k v = raise TODO
  let lookup e k = raise TODO
end
```

10.4.1 Testing your module

When you complete the module you can start testing it. For example, you can repurpose the `example` question from above where you replace `E` with `ListEnv`. Do create your own examples to see that this module works as expected.

This approach to testing applies to all of the remaining implementations.

10.5 Functional environments

The following approach uses functions as environments. It is closer in spirit to the mathematical notion of an associative map being a function from keys to values. We have now included the implementation of `insert`, and we ask you to only implement the `lookup` function here.

```
(* Approach 2 - functional environments *)

module FunEnv : EnvSig = struct
  let approach_name = "Functional environments"
  type env = string -> value
  type symbol = string
  let symbol = fun x -> x
  let empty_env = fun _ -> raise NotFound
  let insert e k v = fun x -> if x = k then v else e x
  let lookup e k = raise TODO
end
```

10.6 Map-based environments

Next implementation is a thin wrapper around the OCaml's `Map` standard library. This way of wrapping allows us to see how this implementation relates to the other approaches.

```
(* Approach 3 - Map-based environments  *)
module MapEnv : EnvSig = struct
  let approach_name = "Map-based environments"
  module StringMap = Map.Make (String)    (* using Map functor; do read up on this !*)
  type symbol = string
  let symbol = fun x-> x

  type env = value StringMap.t
  let empty_env = StringMap.empty
  let lookup e k = raise TODO
  let insert e k v = raise TODO
end
```

10.7 Map-based environments with int hashtables

The following implementation is based on the symbol tables as described in Modern Compiler Implementation in ML, Chapter 5.1.

```
(* Approach 4 - Map-based environments with int hashtable;
   based on MCIML, Chapter 5.1 *)

module MapHashEnv : EnvSig = struct
  let approach_name = "Map-based envs with int H/T"
  let nextsym = ref 0
  type symbol = string * int
  module H = Hashtbl
  let hashtbl: (string, int) H.t = H.create 2048 (* some init size *)
  module SymbolMap =
      Map.Make (
          struct
            type t = symbol
            let compare (_,n1) (_,n2) = compare n1 n2
          end)

  let symbol name =
    match Hashtbl.find_opt hashtbl name with
    | Some i -> (name, i)
    | None ->
      let i = !nextsym in
      nextsym := i + 1;
      H.add hashtbl name i ;
      (name, i)

  type env = value SymbolMap.t
  let empty_env = SymbolMap.empty
  let insert e k v = raise TODO
  let lookup e k = raise TODO
end;;
```

Make sure you understand the code and the purpose of the symbol table-based indirection.

10.8 Benchmarking

So, four ways to do the same thing!? But what is the right way? A back-of-an-envelop asymptotic analysis will tell us that both the list-based environments and functional environments are inferior to the map-based ones (can you see why?). It may also be instructive to do an empirical analysis to get a sense of how all four of the approaches relate to each other.

How should one benchmark these? Ideally, we should have a finished implementation of a compiler where the environment module can be replaced and we perform an end-to-end evaluation of how different implementations affect compilper performance. Alas, we do not have a finished (or even started) compiler. Instead, we will try to use a micro-benchmarking approach, using an OCaml library called `core_bench`. You can read about the idea of micro-benchmarking and the library we use in this Jane Street blog post.

```
(* Benchmarking using core_bench *)
(* This probably requires extra libraries to install using opam *)
let _ = Random.self_init();;

let random_string n =
```

(continues on next page)

(continued from previous page)

```
  String.init n (fun _ -> Char.chr(97 + (Random.int 26)))

let max_n = 2000
let name_strings = List.init max_n (fun _ -> (random_string 10))
let values = List.init max_n (fun x -> x )

module type BenchSig = sig
  val approach_name : string
  val bench : unit -> int
end


(* Question to students: is this a good way of doing benchmarking? *)
(* Are there any issues to discuss about this? *)
module EnvBench (E:EnvSig):BenchSig = struct
  include E
  let names = List.map symbol name_strings
  let names_and_values = List.combine names values
  let bench () =
    let e1 = List.fold_left (fun e (k,v) -> insert e k v ) empty_env names_and_values␣
 ↪in
    let s = List.fold_left (fun s n -> s + lookup e1 n ) 0 names  in
    s
end

(* main function *)
let () =
  let open Core in
  let open Core_bench in
  let f m = let module M = (val m : BenchSig) in
            Bench.Test.create ~name: M.approach_name M.bench
  in (List.map [(module EnvBench (ListEnv)     : BenchSig);
                (module EnvBench (FunEnv)      : BenchSig);
                (module EnvBench (MapEnv)      : BenchSig);
                (module EnvBench (MapHashEnv) : BenchSig);
                ]  ~f: f)
  |> Bench.make_command    (* we're plugging into the bench main functionality *)
  |> Command_unix.run
```

10.8.1 Configuring and running the benchmark

We use the following dune configuration; see the list of libraries that you may need to install using `opam`.

```
(executable
   (name envbench)
   (libraries core core_bench core_unix.command_unix))

(install
    (section bin)
    (files (envbench.exe as envbench)))

(env
  (dev
    (flags (:standard -warn-error -A))))
```

With the above configuration, `dune build` will produce a binary `envbench` that we can call

```
_build/install/default/bin/envbench
```

You will get an output that looks like follows. We have redacted the actual numbers with asterisks – run the benchmark to see the numbers for yourself!

```
Estimated testing time 40s (4 benchmarks x 10s). Change using '-quota'.
┌──────────────────┬──────────┬──────────┬──────────┬──────────┬────────────┐
│ Name             │ Time/Run │ mWd/Run  │ mjWd/Run │ Prom/Run │ Percentage │
├──────────────────┼──────────┼──────────┼──────────┼──────────┼────────────┤
│ List-based env   │ ********** │ ******** │ ********* │ ********* │ ********** │
│ Functional env   │ ********** │ ******** │ ********* │ ********* │ ********** │
│ Map-based env    │ ********** │ ******** │ ********* │ ********* │ ********** │
│ Map + int H/T    │ ********** │ ******** │ ********* │ ********* │ ********** │
└──────────────────┴──────────┴──────────┴──────────┴──────────┴────────────┘
```

The core_bench library provides a number of flags that we can study. Check them out by running

```
_build/install/default/bin/envbench --help
```

CHAPTER

ELEVEN

EXERCISES FOR WEEK 4

11.1 Exercise overview

This week's exercises are about writing LLVM by hand, which will help us get some understanding of the basic structure of LLVM programs.

11.2 Division function

11.2.1 Basic division

Write an LLVM function `f_mod (a, b)` that takes two 64-bit arguments `a` and `b` and returns as its result the value. `a mod b`.

To start, all you need is an empty file, in which you declare your function (single-line comments in LLVM start with a semicolon)

```
; save this as prog.ll
define i64 @f_mod(i64 %a, i64 %b) {
    ; Write your code here
    ; ...
}
```

To test your program, proceed as follows. Suppose your program is saved in a file `prog.ll`. Then we can use the following C wrapper `main.c`.

```
/* save this as main.c */
#include <stdio.h>
#include <stdlib.h>

extern int f_mod(int a, int b);

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Please call this program with two arguments: %s <a> <b>\n", argv[0]);
        return 1;
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);

    int result = f_mod(a, b);
```

(continues on next page)

(continued from previous page)

```
    printf("%d\n", result);

    return 0;
}
```

Compile and link everything with `clang`

```
clang main.c prog.ll
```

Run the resulting program, on several arguments, e.g., `./a.out 1 1` and ensure that you get the right behavior. What happens when the second argument is 0?

11.2.2 Adding division by zero protection

Let us ensure that we check against division by zero. We will need two things

1. The error handling code that will "gently" stop program execution.
2. The comparison in the LLVM code.

For the error handling code, let us add a function

```
int error_div_by_zero () {
    printf ("Error: division by zero\n");
    exit (1);
}
```

Our LLVM program will need a line saying that `error_div_by_zero` is an external function.

```
declare void @error_div_by_zero()
```

> **Note**
>
> Observe the difference between `declare` and `define`.

Compile, and test your implementmation, and validate that the error handling that you have implemented indeed works.

11.3 Euclidean algorithm for computing GCD in LLVM

In this task, we implement GCD calculation using Euclidean algorithm.

11.3.1 Reference implementation in a programming language of choice.

Start off by recalling the definition of GCD and Eucli's algorithm by implementing it in a programming language of your choice. Write two implementation: one using a loop, and one using recursion. Check that they compute the same things.

11.3.2 LLVM implementations

- Write an LLVM function `gcd_1` that implements the Euclid's algorithm. Decide whether you want that to be a loop-based one or a recursive one. Modify the C wrapper so that it calls `gcd_1` instead of the `f_mod` from before.
- Write an LLVM function `gcd_2` that uses the other approach for the Euclid's algorithm. Modify the C wrapper, one more time.

Which of the two is shorter? Which one is easier to understand?

- How many basic blocks do you have in your program. Can you rearrange the basic blocks and run your program again. Does it change program semantics and how?

11.3.3 Hardening against divisions by zero

An important aspect of code generation in compilers is that most of the code generation for simple operations, such as division, is context-independent. That means that if your general compilation strategy is such that you generate division by zero checks, then it is normal for these checks to be present against all divisions. The idea is that usually it is the job of the later phases, e.g., compilation, to perform the cleanup and remove unnecessary checks.

- Write the code for `gcd_1` and `gcd_2` as if it was generated by a compiler that generates division by zero checks but that is unaware of the context surrounding the division operation.

CHAPTER

TWELVE

EXERCISES FOR WEEK 5

12.1 Exercise overview

This week's exercises are about programmatic generation of LLVM. There are two workhorse modules that we give you:

- `ll.ml` that is the module for representing LLVM programs.
- `cfgBuilder` is the module for on-the-fly generation of LLVM programs.

12.2 Using `Ll` module

This module includes the auxiliary data structures for representing LLVM– programs. The main top-level declaration is that of `prog`. This module includes a pretty printer for LLVM programs in the function `string_of_prog`.

As an exercise, create a simple LLVM program for returning an integer value.

12.3 Using `CfgBuilder`

For the actual code generation, we have an auxiliary module for building CFGs. Check out the interface stored in the `.mli` file.

As an exercise, generate the code for a factorial function in LLVM using this API.

CHAPTER

THIRTEEN

EXERCISES FOR WEEK 6

13.1 Exercise overview

In this exercise we programmatically generate LLVM code for the following function (in pseudo-code)

```
var x = read_integer ()
if x >= 0 then
   x = x + 1
else
   x = x - 1
print_integer x
```

We start off by studying example final LLVM code that we may want to generate. We use `alloca` for allocating space on the stack for variable `x`, and note that we do not use `phi` instructions here.

```
declare i64 @read_integer()
declare void @print_integer(i64)
define void @dolphin_main () {
 %x0 = alloca i64
 %t1 = call i64 @read_integer ()
 store i64 %t1, i64* %x0
 %t2 = icmp sge i64 %t1, 0
 br i1 %t2, label %then3, label %else4
then3:
 %t6 = load i64, i64* %x0
 %t7 = add i64 %t6, 1
 store i64 %t7, i64* %x0
 br label %merge5
else4:
 %t8 = load i64, i64* %x0
 %t9 = sub i64 %t8, 1
 store i64 %t9, i64* %x0
 br label %merge5
merge5:
 %t10 = load i64, i64* %x0
 call void @print_integer (i64 %t10)
 ret void
}
```

Save this in `prog.ll` and compile together with the runtime library provided for Assignment 4.

```
clang prog.ll runtime.c
```

Run and ensure that the program behaves as expected.

13.2 Skeleton

One particular aspect of the CFG Builder library is that the CFG construction is *non-destructive*. All the helper functions there return a new version of the CFG Builder instead of modifying it in-place. Moreover, much of the code generation does not depend on the state of the CFG builder but is generally about extending it. For this reason, we use the notion of CFG transformers, that we dub *buildlets*, and that have type `cfgbuilder -> cfgbuilder` so that they can be conveniently combined.

We exemplify the usage of buildlets in the rest of the exercise. We start off with the following skeleton

```
module Sym = Symbol
open Ll
let symbol = Sym.symbol

let fresh_symbol =
  let c = ref 0 in
  fun initial ->
    let n = !c in c := n + 1; symbol (initial ^ (string_of_int n))


exception NotImplemented
(* string -> insn -> CfgBuilder.buildlet * operand *)
let add_instruction_with_fresh s i =
  raise NotImplemented


(* bop -> operand -> CfgBuilder.buildlet *)
let change_by_one_buildlet op loc =
  let b_load, load_op = add_instruction_with_fresh "t" (Load (I64, loc)) in
  let b_binop, op = add_instruction_with_fresh "t" (Binop (op, I64, load_op, IConst64␣
↪1L)) in
  let b_save = CfgBuilder.add_insn (None, Store (I64, op, loc)) in
  CfgBuilder.seq_buildlets [ b_load; b_binop; b_save]

(* CfgBuilder.buildlet *)
let exercise_buildlet =


(* let b_update_add = change_by_one_buildlet Add alloca_op in  *)
(* ... *)
(* let b_update_sub = change_by_one_buildlet Sub alloca_op in  *)


(* CfgBuilder.seq_buildlets [ b_update_add; ... ; b_update_sub ] *)
  CfgBuilder.term_block (Ret (Void, None)) (* replace this with your code *)


let p : prog =
  let b = exercise_buildlet CfgBuilder.empty_cfg_builder in
  let cfg = CfgBuilder.get_cfg b in
  let f = { fty = ([], Void); param = []; cfg = cfg} in
  {
  tdecls = [];
  extgdecls = [];
  gdecls = [];
  extfuns = [ (symbol "print_integer",  ([I64], Void))
            ; (symbol "read_integer", ([], I64))];
```

(continues on next page)

(continued from previous page)

```
  fdecls = [ (symbol "dolphin_main", f )]
}

let _ = Printf.printf "%s" (Ll.string_of_prog p)
```

13.3 Building parts

A common task in code generation is to generate a fresh name for some instruction. To help us with this, we start off by writing an auxiliary function `add_instruction_with_fresh` that takes a string and an instruction, and returns a pair of two things:

- a buildlet function that corresponds to the instruction
- a freshly generated new name that is used by the instruction and can be used by the caller of this function

We exemplify the usage of this function in `change_by_one_buildlet` that takes a binary operation, a memory location (presumably a result of alloca or some other memory allocation), and does the following:

- it loads the value from memory
- performs the binop
- saves the result in-place

It also returns a builder that is a sequence of the underlying buildlet combinators. Do check out the definition of `CfgBulider.seq_buildlets` and make sure you understand it.

13.4 Putting it all together

Write the main body of the example. Implement the function `exercise_buildlet`. For the bodies of the 'then' and 'else' branches, do make use of the helper function `change_by_one_buildlet`. Once done, use the generated LLVM code and test your solution.

CHAPTER

FOURTEEN

EXERCISES FOR WEEK 7

14.1 Exercise overview

In this exercise we get to know the language Dolphin a bit better. For this purpose, we will use the online Dolphin interpreter: https://cs.au.dk/~timany/dolphin_web/.

14.2 Syntax of Dolphin

Throughout the course we have seen many Dolphin code snippets. Therefore, we will not repeat things in great detail. Remember, Dolphin's syntax is heavily inspired by that of `C`, `Java`, and `C#`.

A Dolphin program consists of a number of function declarations and a number of record declarations. The order of these declarations does not matter. All Dolphin programs must have a main function that takes no arguments and returns an integer (the exit code of the program). Records are declared as follows using the `record` keyword:

```
record person {name : string; height : int; }
```

Records are referred to by their names. Given the record above, we can write the following program which finds the tallest person. Here, `[person]` is the type of array of `person`.

```
record person {name : string; height : int; }

person tallest(prs : [person]){
  var tlp : person = nil;

  if(length_of(prs) > 0){
    tlp = prs[0];
  }

  for(var i = 1; i < length_of(prs); i = i + 1){
    if(prs[i].height > tlp.height)
      tlp = prs[i];
  }
  return tlp;
}

int main(){
  let stdout = get_stdout();
  let prs = new person[5];
  prs[0] = new person {name = "Chris"; height = 163; };
  prs[1] = new person {name = "Pernille"; height = 152; };
  prs[2] = new person {name = "Gudmund"; height = 180; };
```

(continues on next page)

(continued from previous page)

```
  prs[3] = new person {name = "Mathias"; height = 162; };
  prs[4] = new person {name = "Nina"; height = 171; };
  let tlp = tallest(prs);
  output_string("The tallest person is ", stdout);
  output_string(tlp.name, stdout);
  output_string(" who is ", stdout);
  output_string(int_to_string(tlp.height), stdout);
  output_string(" cm tall.\n", stdout);
  return 0;
}
```

Note how the `new` keyword is used to create both arrays and records. Note that arrays are not explicitly initialized by the program. Dolphin automatically initializes array entries with default values, 0 for integers, `false` for booleans, `""` for strings, and `nil` for arrays and records. The `let` keyword creates an immutable variable, i.e., a variable that can never be updated after it's initialization.

Run the program above in the interpreter.

14.3 Dolphin's standard library in the interpreter

The following standard library functions are available in the interpreter. (Things like file I/O and networking are not supported in the interpreter.)

```
void exit(code : int);
stream get_stdout();
stream get_stderr();
stream get_stdin();
bool flush_file(srm : stream); // returns false when errors encountered
void output_string(str : string, srm : stream);
void output_bytes_array(bta : [byte], srm : stream);
[byte] input_bytes_array(len : int; srm : stream);
bool output_byte(b : byte, srm : stream); // returns false when errors encountered
int input_byte(stream); // returns a byte as a signed integer
int string_to_int(str : string);
string int_to_string(i : int);
string substring(str : string, start : int, len : int);
string string_concat(str1 : string, str2 : string);
string ascii_chr(c : int);
int ascii_ord(string);
byte int_to_byte_signed(i : int);
byte int_to_byte_unsigned(i : int);
int byte_to_int_signed(b : byte);
int byte_to_int_unsigned(b : byte);
[byte] string_to_bytes_array(str : string);
string bytes_array_to_string(bts : [byte]);
```

The type `stream` is a type in Dolphin defined by the standard library. It is technically treated by Dolphin as a record type with no fields. However, programmers cannot create instances of this record. (Try it!) The `stream` type is used for file and console I/O.

The type `byte` is a primitive type in Dolphin of exactly 1 byte size. Programmers cannot write `byte` literal values. The standard library uses bytes for I/O and allows programmers to convert to and from strings and integers. Dolphin does not enforce a *signedness* for bytes, hence, the conversion functions to and from integers have a signed and unsigned variant.

14.4 Exercise

Write a program for a phone directory. It should store entries as records, with two string fields, one for the name of the person, and another for the phone number. The directory data structure should be a collection of entries, e.g., an array, a linked list, etc., we leave the details to you. The data structure should support adding entries, removing entries, searching for entries (by name), and printing all entries. The latter should print entries in alphabetical order according to their names. These operations can be destructive (update in place), or non-destructive (return a new collection), again, we leave the details to you.

Write a main function that enters 20 entries, searches for a few names (test both existing and non-existing names), removes 2 entries, and prints the final directory.

This exercise gives a lot of freedom in how you choose to write your program, e.g., in the choice of presenting menu, taking user's input, etc. The purpose is to focus on getting familiar with Dolphin, not a particular way of programming.

14.4.1 Bonus

Implement a basic command-line interface (CLI), e.g., the main function should present the user with a menu like below:

```
Please choose one of the following actions:
1. Add an entry
2. List entries
3. Edit an entry
4. Delete an entry
5. Export as HTML
6. Exit
```

CHAPTER

FIFTEEN

EXERCISES FOR WEEK 9

Extend the hashtags example (available on Brightspace) with identification of emails.

CHAPTER

SIXTEEN

EXERCISES FOR WEEK 10

This week's TA exercise is getting to know two items.

1. Creating dummy parsers for testing the implementation of lexers.
2. End-to-end testing of compilers (time permitting).

16.1 Dummy Parsers

A dummy parser is a parser that, rather than building an AST from the lexed tokens, simply emits each token as it is read from the input. This is especially useful for verifying that your parser works as expected.

Your first task is to write a parser that does this.

16.2 End-to-end testing

Unlike other domains, compiler testing generally does not involve unit testing. Instead, most positive tests are end-to-end, while negative tests are end-to-wherever-the-error-occurs. This makes the test suite more robust to modifications of the AST and the addition of features.

Your second task is to develop an end-to-end testing framework for your compiler. You may choose to use a unit testing framework to write your end-to-end tests, such as OUnit or Alcotest, or you may choose to hand-write your test execution.

In any case, for each test, your framework should:

- read Dolphin source code
- try to compile the source code to a binary
  - if compilation fails, compare the failure to an expected error
  - if compilation succeeds, execute the compiled code and compare the output to an expected output

Ideally, the entire suite should be runnable without any human interaction, and make it easy to identify which tests failed.

CHAPTER

SEVENTEEN

EXERCISES FOR WEEK 11

This week's exercises are about getting to know how to use LLVM's `getelementptr` (GEP) instruction.

17.1 Tuples

1. Create an LLVM struct for tuples, called `Tuple`, consisting of two `i64` fields.
2. Create the following LLVM functions:
    1. `create_tuple` that takes two `i64` arguments, and returns a tuple initialized with these arguments. Use the C `malloc` function to allocate the tuple on the heap. Discuss in class what size argument to pass to `malloc`.
    2. `get_first` that takes a tuple and returns its first element.
    3. `get_second` that takes a tuple and returns its second element.
    4. `set_first` that takes a tuple, and an `i64` value and performs an in-place update of the first element of the tuple.
    5. `set_second` that takes a tuple, and an `i64` value and performs an in-place update of the second element of the tuple.
    6. `swap_elements` that in-place swaps the first and the second elements of the tuple.
3. Write an LLVM program that tests the functionality of the above functions.
4. Modify your `create_tuple` function so that the allocation takes place on stack instead of the heap. Does the program created in step (3) above still work? If yes, can you come up with a program that works in the heap-allocated version of `create_tuple` but does not work in the stack-allocated version?
5. Write an LLVM function `swap_tuple_array` that takes three arguments:
    1. a pointer to an array of tuples
    2. an `i64` integer `i`
    3. an `i64` integer `j`

    This function should swap the elements `i` and `j` in the provided array.
6. Write an LLVM program that tests the functionality of the above functions.
7. Consider two alternative implementations of the Tuple
    1. as an LLVM array of two fields.
    2. as an LLVM pointer.

    How would your implementation of the above functions change in each case?

17.2 Nested GEPs

Consider the following C program

```c
#include <stdlib.h>

struct B {
  int p;
  int *f;
  int q;
};

struct A {
  int g;
};

void update_two (struct A** a, struct B b) {
  a[b.f[2]]->g = b.p + b.q;
}

int main() {
  struct A **a = malloc(10 * sizeof(struct A*));

  for (int i = 0; i < 10; ++i) {
    a[i] = malloc(sizeof(struct A));
    a[i]->g = 0;
  }

  struct B b;
  update_two (a,b);

  int array_for_f[3] = {0, 1, 4};
  b.f = array_for_f;


  return 0;
}
```

Compile the above function to LLVM, using `-emit-llvm -S` flag passed to `clang`. Modify the LLVM code for function `update_two` to use as few GEPs as possible.

> **Note**
>
> Counting the number of GEPs is generally not a meaningful metric for the quality of code generation. We use it here only as an exercise device to learn GEP.

Part III

Reference

CHAPTER

EIGHTEEN

LLVM--

18.1 Introduction to LLVM--

LLVM-- is a subset of the LLVM intermediate representation. Our subset is heavily based on LLVMlite developed at the University of Pennsylvania, but is further customized to the specifics of the Compilation course at Aarhus University. This document outlines the structure of programs in the LLVM-- intermediate representation.

18.1.1 A minimal LLVM-- program

The following is an example of a small LLVM-- program that returns number 42 as its result.

```
define i64 @main () {
 ret i64 42
}
```

This program can be compiled, executed, and inspected for the return value using the following sequence of shell commands.

```
$ clang program.ll
$ ./a.out
$ echo $?
42
```

Here, `clang` invokes the LLVM compiler, assuming that the source file is `program.ll`. Because we do not specify any options to `clang`, the compiler uses the default name `a.out` for the generated executable. After invoking the executable, the OS stores its return value in the `$?` shell variable, and we print out the exit code using the `echo` shell command.[1]

18.1.2 Identifiers

LLVM-- differentiates between two kinds of identifiers: global and local. Global identifiers start with the `@` symbol and are used for the definitions of global data definitions and functions. Identifiers that start with the `%` symbol are used for the names of *registers*, named type definitions (see below), and for referring to the labels in branch instructions. LLVM registers, e.g., `%x`, `%y` are local to each function. There is no limit on the number of registers in a function.

[1] On POSIX systems the exit-code is a value between 0 and 255.

On implicit registers in full LLVM

Note that full LLVM supports implicit register naming scheme (`%0`, `%1`, ...) for function arguments and labels. Our subset requires all arguments and labels to be explicitly named.

18.1.3 Types

LLVM is a typed intermediate language. In our subset, we distinguish between *single value types*, such as integers and pointers, and *aggregate types*, such as structures and arrays. Additionally, we can define named types. Finally, we distinguish a special `void` type used in function declarations.

Single value types

Single value types include integers and pointers.

- **Integer types** These are `i1`, `i8`, `i32`, `i64`. The number after the `i` symbol corresponds to the number of bits. For example, `i1` is used for booleans, `i8` is used for characters, and `i64` is used for 64-bit integers. Note that `LLVM--` does not use 32-bit integers (with the exception of indexing into structs via *GEP*).
- **Pointer types** The syntax for declaring a pointer to a base type t is t`*`. For example, `i64*` is a type of a pointer to a 64-bit integer.

Aggregate types

Aggregate types include structures and statically-sized arrays.

- **Structure types** A structure type resembles record types in languages such as C. The syntax for the structure types is `{ t1, ..., tk }` where each of `t1` ... `tk` is a type. For example, the following is a structure consisting of three types `{i1, i64 *, i8}`. Structure types may appear nested within structure types, for example `{i1, {i64, i64*}}`, `{i64, {i8, i8*, i64}, i1}`.
- **Statically-sized array types** The syntax for array types is `[n x t]`, where `n` is an integer indicating the static size of the array, and `t` is the array base type. For example, the type of an array with base type `i64` of size 10 is defined as [10 x i64]`. Array base types can include other types, including structure and array types.

Named types

Named types allow creating an abbreviation to other types. The identifiers used in the type definitions need to start with `%` symbol; the named type definition moreover needs to use a designated `type` keyword. The following example declares `tuple` to be a structure of two 64-bit integers.

```
%tuple = type {i64, i64}
```

Other named types may appear in named type definitions:

```
%tuplearray = type [10 x %tuple]
```

Note that our subset `LLVM--` does not allow cycles in the declarations of named types.

Void type

In addition to the types above, there is a special `void` type that is used in the declaration of the methods (cf. *function header*). Its purpose is similar to the void type in languages such as C or Java.

On LLVM types that are not part of our subset

Note that full LLVM contains other primitive types such as floats, vectors, labels, and x86-mmx; we do not consider these in our subset, `LLVM--`.

18.2 Structure of `LLVM--` Programs

An `LLVM--` program consists of three parts: global data definitions, named type definitions, and function definitions.

18.2.1 Global data definitions

Global data definitions assign *global identifiers* to global constants. Each global identifier starts with an `@` character. The syntax uses the `global` keyword. An example of a global data definition is

```
@x = global i64 42
```

Here, `@x` is the name of the global identifier, `i64` is the type of the definition and constant `42` is the value assigned to `@x`.

- **Global initializers** Global initializers in LLVM are used to declare program constants. Similarly to types, global initializers may be simple and aggregate: *Simple initializers* include integers, e.g., 0, 1, 10, 42, or a *null* pointer constant. Aggregate initializers include structure initializers, e.g., `{i64 0, {i64 1, i64 2}}`, and static array initializers, e.g., `{i64 0, i64 1, i64 2}`.
- **String literals** A common representation of string literals in LLVM is as arrays of type `i8`. There is a shorthand syntax for declaring string literals, using `c` character and the string in quotes:

  ```
  @s = global [5 x i8] c"Hello"
  ```

  Note, however, that in our Dolphin compiler, there is some more work involved in the compilation of string literals and we use a slightly different representation.

Finally, note that a global data definitions for type `t` returns a pointer to type `t`. Thus, the type of `@x` above is `i64*` and the type of `@s` is `i8*`.

18.2.2 Named type definitions

The definitions of the named types uses the syntax explained in *Named types*. Several definitions of named types follow each other in a sequence.

```
%t1 = type ...
%t2 = type ...
%t3 = type ...
```

18.3 Declaring external functions

External functions, e.g., functions implemented in the C runtime, can be called from LLVM if they are declared as external functions. We use the `declare` keyword for declaring external functions as follows:

```
declare i64 @libfun (i64 %x)
```

In the above, the external function `@libfun` takes one arguments, `x` of type `i64`, and returns an `i64`.

18.4 Function declarations

Function declarations consist of a *function header* followed by a sequence of *basic blocks* in braces.

18.4.1 Function header

Function header consists of the keyword `define` followed by the return type of the function, the function name, and the list of the typed argument identifiers in parentheses:

```
define i64 @foo (i64 %x, i8* %y, %sometype %z)
```

In the above, the function `@foo` takes three arguments: `x` of type `i64`, `y` of type `i8*`, and `z` of the named type `%sometype`. Note the use of `define` keyword as opposed to `declare` keyword above used to *declare external functions*.

18.4.2 Basic blocks

Basic blocks are the primary unit of control flow in `LLVM--`. They consist of a sequence of *LLVM instructions* followed by a *terminator*. The instructions in a basic block are always executed sequentially until the execution reaches the terminator. The terminators include returning from a function, conditional and unconditional branching (jumps), and a special terminator indicating unreachable code (see below). Branching instructions take the *label* of the target block as an argument. This organization ensures that the execution cannot jump into a middle of a basic block via the branching instructions.

18.4.3 Instructions and terminators

`LLVM--` subset supports the following instructions:

- **Non-terminating instructions** The following table lists the non-terminating instructions. Here, *binop* ranges over one of the `add`, `sub`, `mul`, `sdiv`, `srem`, `shl`, `lshr`, `ashr`, `and`, `or`, `xor`, and the *comparator* argument to the `icmp` instruction is one of `eq`, `ne`, `slt`, `sle`, `sgt`, `sge`.

Name	Description	Example	Returns a result
binop	binary operation on two operands	`%x = add i64 %y, 42`	Yes
`alloca`	allocate memory on the current stack frame	`%x = alloca {i64, i64}`	Yes
`load`	load a value from memory	`%x = load i64, i64* %loc`	Yes
`store`	store a value in memory	`store i64 %x, i64* %loc`	No
`icmp` *comparator*	compare values	`%x = icmp ne i64 %y, 0`	Yes
`call`	call a function	`%x = call i64 @f(i64 %y)`	If non-void
`bitcast .. . to`	cast a value between types w/o changing any bits	`%x = bitcast i8* %y to i64*`	Yes
`ptrtoint . .. to`	convert a pointer to an integer	`%x = ptrtoint i64* %y to i64`	Yes
`getelementptr`	compute address of a subelement in an aggregate	see *GEP* below	Yes
`phi`	compute result based on control-flow (see *Phi nodes*)	`phi i64 [0, %true_branch], [1, %false_branch]`	yes

- **Terminators** A terminator is one of the following four instructions:

Description	Example
Conditional branch	`br i1 %x, label %L1, label %L2`
Unconditional branch	`br label %L`
Return with a return value	`ret i64 42`
Return w/o a return value	`ret void`
End of basic block is unreachable	`unreachable`

The semantics of the conditional branch is that it jumps to the first label if the `i1` value is `1` and the second label otherwise.

18.4.4 Example CFG: factorial function

We consider a simple C function

```
int factorial(int X) {
  if (X == 0) return 1;
  return X*factorial(X-1);
}
```

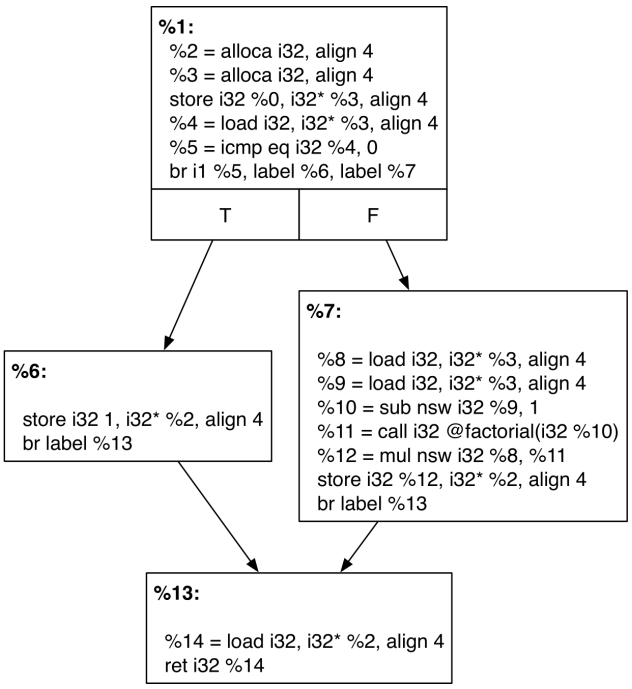
Below is the LLVM code for this function (generated by `clang`) and the corresponding Control Flow Graph (CFG).

```
define i32 @factorial(i32) {
  %2 = alloca i32
  %3 = alloca i32
  store i32 %0, i32* %3
  %4 = load i32, i32* %3
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:
  store i32 1, i32* %2
```

(continues on next page)

(continued from previous page)

```
  br label %13

7:
  %8 = load i32, i32* %3
  %9 = load i32, i32* %3
  %10 = sub nsw i32 %9, 1
  %11 = call i32 @factorial(i32 %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2
  br label %13

13:
  %14 = load i32, i32* %2
  ret i32 %14
}
```



There are four basic blocks here. Because this code is generated by `clang` it makes use of implicit identifiers (`%0`,`%1`, `%2`,…). For example, `%0` refers to the identifier that is the argument to the `@factorial` function, and `%1` is the label of the entry block.

Note that our subset, `LLVM--`. requires that the entry basic block is unlabeled, enforcing that the control flow cannot accidentally jump there – this looses no generality because we can always have an empty instruction sequence in the first basic block.

18.5 Phi nodes

LLVM programs are in so-called SSA form (short for static single-assignment form). That is, each register, when declared, is initialized with a value and can never be changed subsequently. Note that the `load` and `store` instructions perform loading from and storing to *memory locations* (the address may be in a register) and not registers. This raises an issue: what if we want to assign a value to a register based on which branch we take in the program? An example of this can be seen in *the factorial example above* where this problem is solved by allocating space on the stack, `%2` which is written to in both branches `%6` and `%7`, and which is read and returned at the end of the function. A better solution to this problem is to use so-called `phi` nodes which do not use stack space. A phi node is of the form `phi ty [val_1, %block_1], [val_2, %block_2], ..., [val_n, %block_n]` where `ty` is the type of the value produced by the `phi` node. The labels `%block_1`, ..., `%block_n` are the labels of *all* the basic blocks that can branch to the basic block where the `phi` node is in. The result of the `phi` node is `val_i` if the basic block where the `phi` is in is reached by branching from `%block_i`. All values `val_1`, ..., `val_n` must be of type `ty`.

Warning

A valid, well-formed `phi` node must always specify a value for any basic block that can reach the block where it appears in. Note that in some cases a malformed `phi` node with the wrong labels referenced can trigger a bug in some versions of `clang` that causes it to crash.

The following is the code for the factorial function adapted to use a `phi` node instead of stack space.

```
define i32 @factorial(i32) {
  %2 = alloca i32
  store i32 %0, i32* %2
  %3 = load i32, i32* %2
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %5, label %6

5:
  br label %12

6:
  %7 = load i32, i32* %2
  %8 = load i32, i32* %2
  %9 = sub nsw i32 %8, 1
  %10 = call i32 @factorial(i32 %9)
  %11 = mul nsw i32 %7, %10
  br label %12

12:
  %13 = phi i32 [1, %5], [%11, %6]
  ret i32 %13
}
```

Note

In a valid LLVM program, implicit registers in a function *must* be named in-order, and without skipping any numbers. That is, if the last declared implicit register is `%10`, the next implicit register declared must be `%11`, otherwise, LLVM would reject the program. This is the reason why in the code above we have renamed registers compared to *the original factorial example above*. In particular, in the code above `%2` is what `%3` was in *the original factorial example above*.

In the code above the returned value, register `%12`, is computed using a `phi` node. The `phi` node assigns value of register

`%10` to register `%12` if the program reaches label `%11` by branching from the basic block labeled `%6`. Otherwise, if the label `%11` was reached by branching from the basic block labeled `%5`, register `%12` is assigned 1.

18.6 The `getelementptr` (GEP) instruction

LLVM uses the `getelementptr` instruction (briefly, GEP) to implement an architecture-independent way of computing addresses of subelements in aggregate data structures, such as structures or arrays. One way to make sense of GEP is to study how C array and structs can be compiled to LLVM because C influenced the design of LLVM (and GEP in particular). Consider the C program below:

```
struct Tuple {
  int x;
  int y;
};

int foo(struct Tuple* tuple) {
  return tuple[2].y;
  // Try:   return tuple[0].y
  //        return tuple->y
}

int main (int argc, char ** argv) {
  struct Tuple tuples [] = { {11, 22},
                             {33, 44},
                             {55, 66} };
  return foo (tuples); // returns 66
}
```

This program declares a tuple structure, initializes an array of tuples and uses the function `foo` to access the `y`-component of the second (counting from zero) element of the array. Assuming the file `example.c` contains this program, we can compile, execute, and inspect the result of this program using the following shell commands:

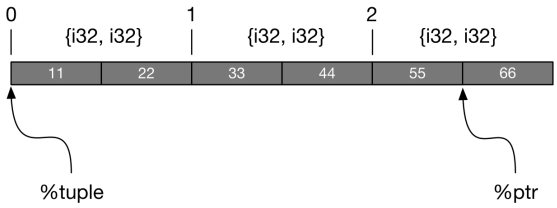
```
$ clang example.c
$ ./a.out
$ echo $?
66
```

The following listing shows how `clang` translates the definition of the tuple structure to LLVM and a *possible* translation for function `foo`. Note that this listing differs from the actual compiler output in two ways: first, we use the name `Tuple` instead of `struct.Tuple`; second, we simplified the code for function `foo` in order to focus on the explanation of GEP. We revisit the actual output at the end of this section.

```
%Tuple = type { i32, i32 }

define i32 @foo(%Tuple* %tuple)  {
  %ptr = getelementptr inbounds %Tuple, %Tuple* %tuple, i64 2, i32 1
  %x = load i32, i32* %ptr
  ret i32 %x
}
```

Observe the two last arguments to GEP: `i64 2, i32 1`. They encode the *path* to the address of `tuple[2].y` in the memory. First, we ask for the pointer to the memory where the second element of the array lives – that is `i64 2`. Next, we need the pointer to the `y`-component of the tuple. Counting from 0, it is the 1-st element of the `Tuple` structure. Note that LLVM uses `i32` for indexing into structures because it is highly unlikely for structures to contain more than 2^{32} elements.

The following picture visualizes the layout of the memory for this example, illustrating where the `%tuple` and `%ptr` registers point to in this layout.



If we want to index into `tuple[1].x`, we should use the GEP instruction with the path `i64 1, i32 0`. For `tuple[0].y`, it should be `i64 0, i32 1`, and so on.

There is a subtlety to GEP that is induced by the close relationship of arrays and pointers in C. A reference to an array in C is just a pointer to the memory where the array starts, that is, the pointer to the zeroth element of the array. This relationship is already apparent in the C declaration of the function `foo` where the `tuple` variable has type `struct Tuple*`. Therefore, GEP treats every pointer as a potential array. This treatment implies that if we are given a pointer to an LLVM structure, and we want to access the `n`-th element of that structure, we need to treat that pointer as if it was a reference to a *one-element array* and we were accessing that one element at position 0 in that array. This means that GEP paths for such accesses must have the form `i64 0, i32 n`.

18.6.1 Examining the actual output of the compiler

Armed with the above observation, let us check out the actual code generated for `@foo` by `clang` (we ignore the specifics of `inbounds` and `align` annotations here):

```
%struct.Tuple = type { i32, i32 }

define i32 @foo(%struct.Tuple*) #0 {
 %2 = alloca %struct.Tuple*, align 8
 store %struct.Tuple* %0, %struct.Tuple** %2, align 8
 %3 = load %struct.Tuple*, %struct.Tuple** %2, align 8
 %4 = getelementptr inbounds
                     %struct.Tuple, %struct.Tuple* %3, i64 2
 %5 = getelementptr inbounds
                     %struct.Tuple, %struct.Tuple* %4, i32 0, i32 1
 %6 = load i32, i32* %5, align 4
 ret i32 %6
}
```

The relevant parts here are the two GEPs. They do the same computation as in the version we studied earlier, except that the task of computing the address to `tuple[2].y` is split into two GEP operations. The first one finds the offset to the start of the second element of the array. That results in a pointer to a `%struct.Tuple` structure. The second GEP finds the offset to the field at position 1 in that structure (once again, treating that pointer as a reference to a one-element array and hence prefixing the path with 0).

To experiment further with this example, one can compile the C code with `clang -S -emit-llvm example.c`, and modify the generated LLVM code for `foo` in an editor. The modified `.ll` file can be compiled using the command `clang example.ll`.

18.6.2 A more involved GEP example

We now consider a slightly more elaborate C program:

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

The generated `.ll` file has the following code for `foo` and structure declarations.

```
%struct.ST = type { i32, double, %struct.RT }
%struct.RT = type { i8, [10 x [20 x i32]], i8 }

define i32* @foo(%struct.ST*) #0 {
  %2 = alloca %struct.ST*, align 8
  store %struct.ST* %0, %struct.ST** %2, align 8
  %3 = load %struct.ST*, %struct.ST** %2, align 8
  %4 = getelementptr inbounds %struct.ST, %struct.ST* %3, i64 1
  %5 = getelementptr inbounds %struct.ST, %struct.ST* %4, i32 0, i32 2
  %6 = getelementptr inbounds %struct.RT, %struct.RT* %5, i32 0, i32 1
  %7 = getelementptr inbounds
                [10 x [20 x i32]], [10 x [20 x i32]]* %6, i64 0, i64 5
  %8 = getelementptr inbounds [20 x i32], [20 x i32]* %7, i64 0, i64 13
  ret i32* %8
}
```

We invite the reader to study the GEPs generated by the compiler and see whether they can match them to the source. One can ask an exercise question: can we rewrite this sequence of GEPs as one? The answer is yes. In fact, the code for this function could just as well be:

```
define i32* @foo(%struct.ST* %arg) #0 {
  %x = getelementptr inbounds
      %struct.ST, %struct.ST* %arg, i64 1, i32 2, i32 1, i64 5, i64 13
  ret i32* %x
}
```

Here, the GEP path is the sequence of operands `i64 1, i32 2, i32 1, i64 5, i64 13`. We can check if the two versions compute the same offsets with the help of the `clang` itself. First, we expand our example as follows:

```
#include <stdio.h>

struct RT {
  char A;
  int B[10][20];
  char C;
};
```

(continues on next page)

(continued from previous page)

```
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}

int *bar(struct ST *s) {
  return &s[1].Z.B[5][13];
}

int main (int argc, char** argv) {
  int* a = foo (NULL);
  int* b = bar (NULL);
  printf ("foo: %p\nbar: %p\n", a, b);
  return 0;
}
```

Observe that we have duplicated the code for `foo` in the function `bar`. The `main` function calls each of these with the argument `NULL`. The results `a` and `b` are the offsets from the `NULL` pointers. This trick of using the `NULL` pointer argument effectively gives us the offsets calculated by each of the functions from the base argument provided to them. We print the offsets in `main` (expecting the two to be the same). Note that the program does not actually dereference anything – dereferencing an offset from a NULL pointer would likely cause a segmentation fault at runtime – again, we are only computing the offsets.

If we compile and run this program, we get the following output:

```
$ clang example-orig.c
$ ./a.out
foo: 0x510
bar: 0x510
```

Here, `0x510` is the offset in hexadecimal (1296 in decimal). Next, we can ask the compiler to generate the `.ll` file

```
$ clang -S -emit-llvm example.c
```

Now, we can go ahead and `edit` the code for `@foo` in `example.ll`. Save, compile, and run it.

```
$ clang example.ll -o example-edited
$ ./example-edited
foo: 0x510
bar: 0x510
```

At this point, one can further play with the LLVM code for `@foo`. For example, changing the last argument in the GEP path from 13 to 12 will yield a different offset.

18.7 Further reading

For more on LLVM instructions and GEP, please consult the relevant pages on the LLVM website:

- LLVM Reference Manual: https://www.llvm.org/docs/LangRef.html
- The Often Misunderstood GEP Instruction: https://www.llvm.org/docs/GetElementPtr.html

CHAPTER

NINETEEN

LLVM-- CHEAT SHEET

It is a good idea to have this cheat sheet with you, e.g., by printing it, when writing LLVM-- code.

Description	Example(s)
Integer types	`i1`, `i8`, `i32`, `i64`
Structure types	`{i1, i64 *, i8}`
Fixed-size array type	`[20 x i64]`
Named types	`%tuple = type {i64, i64}`
Global variables	`@x = global i64 10`
Global string variables	`@s = global [5 x i8] c"Hello"`
External function declarations	`declare i64 @libfun (i64 %x)`
Function declarations	`define i64 @foo (i64 %x){%y = add i64 %x, 1 ret i64 %y}`
Comparison operators	`eq`, `ne`, `sle`, `slt`, `sge`, `sgt`
Arith/logical operators	`add`, `sub`, `mul`, `sdiv`, `srem`, `shl`, `lshr`, `ashr`, `and`, `or`, `xor`
Arith/logical computation	`%x = add i64 %y, 42`
Allocating on the stack	`%ptr = alloca i32`
Loading from a pointer	`%val = load i32, i32* %ptr`
Storing to a pointer	`store i32 42, i32* %ptr`
Integer comparison	`%cmp = icmp eq i32 %val, 10`
Calling void functions	`call void @myFun(i32 %arg1, i32 %arg2)`
Calling non-void functions	`%z = call i8 @myFun(i32 %arg1, i32 %arg2)`
Casting	`%ptr2 = bitcast i32* %ptr to i8*`
Address of pointer	`%int_ptr = ptrtoint i32* %ptr to i64`
Pointer arithmetic	`%element_ptr = getelementptr i32, i32* %array, i32 3`
Phi node	`%res = phi i32 [ %val1, %block1 ], [ %val2, %block2 ]`
Conditional branching	`br i1 %x, label %L1, label %L2`
Unconditional branching	`br label %L`
Returning a value	`ret i64 42`
Returning void	`ret void`
End of block unreachable	`unreachable`

CHAPTER

TWENTY

X86-64

> **Note**
>
> We use AT&T assembly syntax. This is what `clang` uses.

> **Tip**
>
> You can use the x86-64 emulator x64-emu to experiment and practice x86-64 assembly in your browser — it supports all (and only) the instructions and features described here.

20.1 Registers that we will use

Register	Description	Preserved Across Calls
rip	Instruction pointer; cannot be manipulated directly	*Irrelevant* but no!
rax	General purpose; stores return value	No
rbx	General purpose; sometimes also used as the base pointer	Yes
rcx	General purpose; used for 4th argument	No
rdx	General purpose; used for 3rd argument	No
rsp	Stack pointer	Yes (*automatically*)
rbp	Can be used as base pointer	Yes
rsi	General purpose; used for 2nd argument	No
rdi	General purpose; used for 1st argument	No
r8	General purpose; used for 5th argument	No
r9	General purpose; used for 6th argument	No
r10	General purpose	No
r11	General purpose	No
r12	General purpose	Yes
r13	General purpose	Yes
r14	General purpose	Yes
r15	General purpose	Yes

20.2 The stack

In x86 the stack starts in a high address and grows towards lower addresses.

- Pushing on the stack: first *decrements* the stack point (rsp) by the size of the object being pushed and then writes the value in memory pointed by the stack pointer.
- Popping from the stack: reads the value from memory where the stack pointer points and then *increments* the stack pointer by the size of the object being popped.
- To allocate 16 bytes space on the stack, e.g., for storing function's local variables, we _decrement the stack by 16 bytes:

```
subq $16, %rsp
```

20.3 Endianness

x86 is a little-endian architecture. This means that low-significance bytes are written at lower address. For example, the four byte number 0xFFA02B1C, when written to memory at address n, is stored as follows:

Address:	$\cdots$	n	$n+1$	$n+2$	$n+3$	$\cdots$
Contents:	$\cdots$	1C	2B	A0	FF	$\cdots$

20.4 CPU flags

Flag	Meaning
SF	Set if the last arithmetic/logic instruction resulted in a negative number
ZF	Set if the last arithmetic/logic instruction resulted in zero
OF	Set if the last arithmetic/logic instruction resulted in an overflow
CF	Set if the last arithmetic/logic instruction produced a carry
PF	Set to 1 if number of 1's resulting from the last arithmetic/logic instruction is even

20.5 Instructions that we will use

Below are categorized lists of instructions that we will use. Note that when an instruction, like `movq`, takes both a source and a destination operand, both the source and the destination operands cannot be memory at the same time.

20.5.1 Data movement instructions

Instruction	Description	Affected Flags (OF, SF, ZF, CF, PF)	Source Operand(s)	Destination Operand
`movq`	Move 64-bit value from source to destination	-	Register, Memory	Register, Memory
`leaq`	Load effective address into register	-	Memory	Register

20.5.2 Arithmetic/logic instructions

Instruction	Description	Affected Flags (OF, SF, ZF, CF, PF)	Source Operand(s)	Destination Operand
`incq`	Increment value by 1	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`decq`	Decrement value by 1	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`negq`	Negate value (two's complement)	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`addq`	Add source to destination	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`subq`	Subtract source from destination	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`imulq`	Signed multiply destination by source	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`cqto`	Convert quadword (`rax`) to octaword (`rdx:rax`)	-	-	`rdx:rax`
`idivq`	Signed divide `rdx:rax` by divisor	OF, SF, ZF, CF, PF	Register, Memory	`rax` (quotient), `rdx` (remainder)
`cmpq`	Compare source and destination (sets flags)	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`notq`	Bitwise NOT (complement) operation	-	Register, Memory	Register, Memory
`xorq`	Bitwise XOR destination with source	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`orq`	Bitwise OR destination with source	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`andq`	Bitwise AND destination with source	OF, SF, ZF, CF, PF	Register, Memory	Register, Memory
`shlq`	Shift left destination by count bits	OF, SF, ZF, CF, PF	Immediate, `cl`	Register, Memory
`sarq`	Arithmetic shift right destination by count bits	OF, SF, ZF, CF, PF	Immediate, `cl`	Register, Memory
`shrq`	Logical shift right destination by count bits	OF, SF, ZF, CF, PF	Immediate, `cl`	Register, Memory

Note

The instruction `idivq` does not set flags; in fact there is no guarantee what values the flags will have after `idivq`. If there is an overflow (the quotient does not fit in `rax`), or if the divisor is zero, `idivq` will result in a trap (CPU level exception caught by the operating system and usually passed to language's runtime). Proper use of `cqto` instruction should prevent overflows. It is a good idea (this is what we do in this course) to (produce code to) check that the divisor is not zero before every division operation.

Note

The `imulq` instruction multiplies two 64-bit values (source and destination). The result is computed as a 128-bit number. If the result is too big to fit in the 64-bit destination, both the OF and CF flags are set.

Note

The register `cl` which is used to indicate the number of shifts in shifting operations is simply the lowest byte of the `rcx` register. We don't have direct access to it in fragment we use in this course (explained in this document) but it can be set indirectly by setting `rcx` accordingly.

20.5.3 Stack management

Instruction	Description	Affected Flags (OF, SF, ZF, CF, PF)	Source Operand(s)	Destination Operand
`pushq`	Push value onto the stack	-	Register, Memory	Stack
`popq`	Pop value from the stack	-	Stack	Register, Memory

20.5.4 Call and return

Instruction	Description	Affected Flags (OF, SF, ZF, CF, PF)	Source Operand(s)	Destination Operand
`retq`	Return from function	-	-	-
`callq`	Call a function at destination	-	-	Memory address

20.5.5 Jumps

Instruction	Description	Affected Flags (OF, SF, ZF, CF, PF)	Source Operand(s)	Destination Operand
`jmp`	Unconditionally jump to destination	-	-	Memory address
`je`	Jump if equal (ZF=1)	-	-	Memory address
`jne`	Jump if not equal (ZF=0)	-	-	Memory address
`jg`	Jump if greater (ZF=0, SF=OF)	-	-	Memory address
`jge`	Jump if greater or equal (SF=OF)	-	-	Memory address
`jl`	Jump if less (SF!=OF)	-	-	Memory address
`jle`	Jump if less or equal (ZF=1 or SF!=OF)	-	-	Memory address

20.5.6 Setting memory conditionally

Instruction	Description	Affected Flags (OF, SF, ZF, CF, PF)	Source Operand(s)	Destination Operand
`sete`	Set byte to 1 if equal (ZF=1)	-	-	Memory
`setne`	Set byte to 1 if not equal (ZF=0)	-	-	Memory
`setg`	Set byte to 1 if greater (SF=OF)	-	-	Memory
`setge`	Set byte to 1 if greater or equal (SF=OF)	-	-	Memory
`setl`	Set byte to 1 if less (SF!=OF)	-	-	Memory
`setle`	Set byte to 1 if less or equal (ZF=1 or SF!=OF)	-	-	Memory

20.5.7 Instruction operands

- We write register operands with a preceding `%`,e.g., `xorq %rax, %rax`.
- We write immediate integer operands with a preceding `$`, e.g., `$10`.
- Labels (to stand for the memory addresses they refer to) can be directly used as memory operand, however, rip-relative addressing (see below) is the preferred mode of referring to labels.
- We write indirect addresses using parentheses, e.g., `(%rax)` for the memory location whose address is in the register `rax` — as an operand for jumping such addresses must also be preceded with an asterisk, i.e., `jmp *(%rax)`..
- We write relative indirect addresses with an offset preceding parentheses, e.g., `10(%rax)` for the memory location whose address is 10 bytes after the address stored in the register `rax` — as an operand for jumping such addresses must also be preceded with an asterisk, i.e., `jmp *10(%rax)`.

Conditional jumps do not support indirect addressing. That is, while `jmp *10(%rax)` is a valid instruction, `je *10(%rax)` is not. Such (relative) indirect conditional jumps must be encoded manually. That is, one should write the following code instead of `je *10(%rax)`

```
; code before jump
jne after_jump
jmp *10(%rax)
after_jump:
; code after jump
```

> **Tip**
>
> X86-64 uses so-called rip-relative addressing for making code relocatable. That is, in order to move data from a label `abc`, one would write `movq abc(%rip), %rax` instead of `movq abc, %rax`. This effectively does the same thing but is compiled in a different way so that the code is relocatable, i.e., functions correctly regardless of where in the memory it is loaded. Rip-relative addressing can only be used both for accessing data and not jumps. Jumps to labels are automatically made relative.

20.6 Sections in assembly code

Assembly programs are divided into so-called sections: data section(s) and code section(s). These are indicated by `.data` and `.text` indicators in the assembly program.

20.7 labels

Points in the code or data section can be marked with a label by including a line `label:` before the declaration or code being labeled.

We write `.global label` (on a separate line) to export a label (code or data) so it can be referred to from other modules (other source files) that are linked with the code, e.g., to export the function `func`, we would write:

```
.text
.global main
main:
  movq $0, %rax
  retq
```

20.7.1 Storing constants

- Constant 64-bit integers can be stored in the data section by writing `.quad n` where n is a the constant integer stored.
- Constant strings can be stored in the data section by writing `.asciiz "str"` to store the string "str"

These constants can be accessed by their preceding labels:

```
string1:
.asciiz "abcd"

integer1:
.quad 123456
```

20.8 Calling convention (System V ABI)

We use the System V ABI calling convention. This is to be compatible with Linux and Mac OS, and the C programming language on these operating systems. See https://www.intel.com/content/dam/develop/external/us/en/documents/mpx-linux64-abi.pdf for details.

20.8.1 Passing arguments

When calling a function, the first 6 arguments arguments are passed in registers: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`. All the remaining arguments, if any, are passed on the stack, from right to left. That is, the last argument of the function is pushed on the stack, and then, the one before last, until finally the 7th argument is pushed on the stack, before calling the target function.

20.8.2 Callee versus caller saved registers

Each function must ensure that it preserves all callee saved registers (registers marked to be preserved across calls in the table of registers above, e.g., rbp). These must be stored on the stack as part of the function prologue, i.e., the code that runs immediately the beginning of the function. The only exception is the rsp register itself which is automatically preserved by the return instruction. (Note that the function must ensure that the rsp register is exactly as it was at the beginning of the call before it can safely invoke the return instruction because the return address is read from the stack.)

20.8.3 Function prologue and epilogue

Typical simple function prologue and epilogue can be as follows. Here we assume that rbp is the only callee-saved register the function will touch.

function prologue

```
pushq %rbp          ; save the caller's base pointer on the stack
movq %rsp, %rbp     ; set our base pointer to the current stack pointer
                    ; this is useful to be able to restore it and
                    ; as an anchor for referring to variables on the stack
subq 112, %rsp      ; reserve memory on stack for function's local
                    ; variables (112 = 14 * 8)
```

function epilogue

```
movq %rbp, %rsp   ; restore the rsp to where it was right after pushing %rbp
                  ; of the caller
popq %rbp         ; restore the rbp to caller's value
                  ; the stack pointer is now exactly where it was before entering
                  ; the function, i.e., right at the return address
retq              ; return to the caller
```

20.8.4 Stack alignment

System V ABI calling convention mandates that at any function call the stack pointer (`rsp`) must be 16-byte aligned. That is, the numeric value of `rsp` must be divisible by 16. The best way to ensure it is to pre-allocate all the space that function needs on the stack in the prologue and make sure it is 16-byte aligned. The function prologue above does ensure that the stack is in 16-byte alignment. It is a good exercise to try to convince yourself that this is indeed the case.

CHAPTER

TWENTYONE

SCALA OCAML CHEATSHEET

This page presents a quick reference for various OCaml features as they contrast with Scala. Please see OCaml resources for more detail.

21.1 Operators

> **Caution**
>
> For equality checking use = for equal and <> for not-equals.
> **Do not** use == or != in OCaml unless you have good reason to do so!

Scala

OCaml

```
==
```

```
=
```

```
!=
```

```
<>
```

21.2 Variables

Scala

OCaml

```
val x = e
```

```
let x = e
```

```
val x = e
x
```

```
let x = e in
x
```

```
val x: Int = 1
```

```
let x: int = 1
```

21.3 Functions

Scala

OCaml

```
def f(x: Int): Int = x
```

```
let f (x: int): int = x
```

```
def add(x: Int, y: Int): Int = x + y
```

```
let add (x: int) (y: int): int = x + y
```

```
add(2, 3)
```

```
add 2 3
```

```
(x: Int) => x
```

```
fun (x: int) -> x
```

21.3.1 Recursive functions

OCaml requires recursive functions be marked with the `rec` keyword.

Scala

OCaml

```
def fac(n: Int): Int =
    if (n < 2)
        1
    else
        n * fac(n - 1)
```

```
let rec fac (n: int): int =
    if n < 2
    then 1
    else n * fac (n - 1)
```

For mutually recursive functions we use the `and` keyword to make a later declared function visible.

Scala

OCaml

```
def isEven(n: Int): Boolean =
  if (n == 0)
    true
  else
    isOdd(n - 1)

def isOdd(n: Int): Boolean =
  if (n == 0)
    false
  else
    isEven(n - 1)
```

```
let rec is_even (n: int): bool =
  if n = 0
  then true
  else is_odd (n - 1)
and is_odd (n: int): bool =
  if n = 0
  then false
  else is_even (n - 1)
```

21.4 Control structures

Scala

Ocaml

```
if (test) x else y
```

```
if test then x else y
```

```
iOpt match {
  case Some(i) => i
  case None    => 0
}
```

```
match i_opt with
| Some(i) -> i
| None    -> 0
```

```
while (test) {
  body
}
```

```
while test do
  body
done
```

21.5 Data structures

21.5.1 Tuples

Scala

Ocaml

```
(1, 2, 3)
```

```
(1, 2, 3)
```

```
x._1
```

```
fst x
```

```
x._2
```

```
snd x
```

21.5.2 List

Scala

Ocaml

```
x::Nil
```

```
x::[]
```

```
List(1, 2, 3)
```

```
[1; 2; 3]
```

```
def sum(xs: List[Int]): Int =
  xs.foldLeft(0)((x, acc) => x + acc)
```

```
let sum (xs: int list): int =
  List.fold_left (fun x acc -> x + acc) 0 xs
```

21.5.3 Option

Scala

Ocaml

```
Some(42)
```

```
Some(42)
```

```
None
```

```
None
```

21.6 Types

Scala

OCaml

```
(Int, Boolean)
```

```
int * bool
```

```
Int => Boolean
```

```
int -> bool
```

```
Option[T]
```

```
'a option
```

21.7 Operator pitfalls

21.7.1 Equality operators

OCaml has operators for both *structural* equality and *physical* equality. = is structual equals and <> is structural not-equals. == is physical equals and != is physical not-equals. To illustrate the difference, consider the following example:

```
# let x = ref 0;;
# let y = ref 0;;
# x = y;;
- : bool = true
# x == y;;
- : bool = false
```

In this example, `x = y` evaluates to `true` because both are references to `0`, while `x == y` evaluates to `false` since they are not the same references.

21.7.2 Comparison

Be mindful when you compare elements, as OCaml will infer an ordering on elements of any datatype you declare. This will be the order of declaration. As an example, consider the following:

```
# type ex = C | B | A;;
type ex = C | B | A
# C < A;;
- : bool = true
# C < B;;
- : bool = true
# A < B;;
- : bool = false
```

21.8 Mutability

In Scala we can declare mutable variables with the `var` keyword. Instead of mutable variables OCaml has mutable references. These represent a reference to a location in memory similar to a pointer in C. To create a reference cell in OCaml we can use `ref`:

```
# let x = ref 0;;
val x : int ref = {contents = 0}
```

Note the type `int ref`, indicating a reference to a cell containing an integer. To read from and write to the memory location, we can use `!` (dereference) and `:=` (assignment).

```
# !x;;
- : int = 0
# x := 1;;
- : unit = ()
# !x;;
- : int = 1
```

If we have two references two the same memory cell, changes will be reflected:

```
# let y = x;;
val y : int ref = {contents = 1}
# !y;;
- : int = 1
# x := 0;;
- : unit = ()
# !y;;
- : int = 0
```

Finally, consider the following examples of an imperative factorial function:

Scala

OCaml

```
def fac(n: Int): Int = {
  var acc = 1
  var res = 1
  while (acc <= n) {
```

(continues on next page)

(continued from previous page)

```
    res = res * acc
    acc = acc + 1
  }
  res
}
```

```
let fac (n: int): int =
  let acc = ref 1 in
  let res = ref 1 in
  while !acc <= n do
    res := !res * !acc;
    acc := !acc + 1
  done;
  !res
```

CHAPTER

TWENTYTWO

DEVELOPMENT CONTAINER

We have a Docker container with most of the relevant software for the course. This container can be plugged into the VSCode's development-in-the-container extension as follows.

1. Make sure you have the Visual Studio Code Dev Containers extension.
2. Make sure you have a recent version of Docker (usually a good idea to update from the previous semester).

In the main folder of your project, create a folder `.devcontainer` and in that folder create a file `devcontainer.json` that includes the following configuration:

```
{
        "name": "dOvs container",
        "image": "ghcr.io/au-compilers/dovs_container:main",
        "customizations": {
                "vscode": {
                  "extensions": ["ocamllabs.ocaml-platform"]
                },
                "settings": {
                        "terminal.integrated.profiles.linux": {
                                "bash": {
                                        "path": "bash",
                                        "icon": "terminal-bash"
                                }
                        },
                        "terminal.integrated.defaultProfile.linux": "bash"
                }
        }
}
```

Ask VSCode to reopen your project in a container: `F1 -> Dev Container: Reopen in Container`. First time, it will download the container and build it locally. On subsequent runs, this should not be necessary, unless there are changes either in the configuration or the container itself. If you experience slow build times, it may make sense to slightly tweak mounting configuration of your _build directories. There may be some caveats with that, so please ask on the forum.

CHAPTER

TWENTYTHREE

BIBLIOGRAPHY

Note

This page has a known rendering bug due to upstream dependencies.

BIBLIOGRAPHY

[Lat05] Chris Lattner. Implementing portable sizeof, offsetof and variable sized structures in llvm. https://nondot.org/sabre/LLVMNotes/SizeOf-OffsetOf-VariableSizedStructs.txt, 2005. Accessed: 2023-11-19.

[Lev00] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, USA, 2000. ISBN 1-55860-496-0.