Lang compiler

Lang is a functional programming language based on Ocaml and F#. *Langc* is a LLVM compiler for this language (written in Ocaml).

This is a work in progress, although usable version is already available.

Language features

- indent sensitive syntax - produce cleaner and shorter code
- functional first with imperative features (printing, mutable arrays)
- functions are first class citizen
- simple modules (like in *F#*)
- nested lets, recursion
- easy and zero-cost C bindings
- currently only supports `int`, `bool` and `unit` as basic types

Table of Contents

- *Lang* compiler
- Language features
- Table of Contents
- How to build, test & run
  - Install dependencies
  - Test
  - Ast pretty-printer
  - Compile some code!
- Project structure
- Used libraries and code:
- Examples
  - Introduction: external functions and minimal program
  - Let, Let rec and tail-call optimisation
  - Arrays
  - Modules
- Important notes

How to build, test & run

- Get the source code:

```
$ git clone https://github.com/mateuszlewko/lang-compiler.git && cd lang-compiler
```

Install dependencies

- install *ocaml* and *opam*
- switch to *ocaml* version `4.05.0`:

```
$ opam switch 4.05.0
```

- configure *opam* in the current shell:

```
$ eval `opam config env`
```

- install *jbuilder*:

```
$ opam install jbuilder
```

- install rest of dependencies by following output from this commands (except for `menhirLib`):

```
$ jbuilder external-lib-deps --missing @runlangc
$ jbuilder external-lib-deps --missing @runtest
```

You will be asked to install required modules through *opam*, and some external libraries through *depext*. - install LLVM 5 and *gcc* (*gcc* is usually present on linux)

Finally check whether you installed everything correctly:

```
$ llc --version    # Expect something like LLVM version 5.0.1, later versions should also be
                   # NOTE: Version 3.8 will *not* work.
$ gcc --version
$ jbuilder --version
```

Test

Run following command from the root directory of the project. Relative paths to input files used for tests are hardcoded into source code (files in `test/compiler-test-srcs/`).

```
$ make test && ./_build/default/test/test.exe
```

You can add your own tests by creating *lang* source files in `test/compiler-test-srcs/` and specifying expected output in file: `test/compiler_tests.ml` (take a look at this file for examples).

Ast pretty-printer

```
make astprinter && ./_build/default/printer/printer.exe
```

Compile some code!

- First make sure compiler is built:

```
$ make langc -B
```

It's best if you compile files in project root directory as compiler expects file `external.c` to be present at currenty directory. If you are compiling *lang* source from other directory, make sure to copy `external.c` in there.

- Compile `example.la`:

```
$ _build/default/langc/langc.exe example.la
```

If everything went fine, compiler generated binary `a.out`. Check it by running:

`bash $ ./a.out` *Langc* compiler also other options like saving generated *LLVM IR* code in `out.ll`, or changing binary output file with `-o other.out`. For a full list of available options check:

```
$ _build/default/langc/langc.exe --help
```

Project structure

- `compiler/` - compiler library which contains: lexer, parser and codegen
- `compiler/grammar.mly` - grammar in Menhir format
- `compiler/lexer.cppo.sedlex.ml` - lexer
- `compiler/codegen.ml` - main *LLVM IR* code generation
- `compiler/codegenUtils.ml` - some helper functions for code generation
- `compiler/ast.ml` - abstract syntax tree
- `printer/` - pretty-printer for abstract syntax tree
- `langc/` - compiler executable
- `test/` - tests for compiler and parser
- `test/compiler-test-srcs` - input files for compiler tests

Used libraries and code:

- ocaml-parsing - boilerplate code for parsing in OCaml
- Menhir - LR(1) parser generator
- Sedlex - lexer generator
- Jane Street's core, the inofficial standard library for OCaml
- Jane Street's jbuilder, an OCaml build system
- Other libraries specified in jbuild files

Examples

Syntax is very similar to the one in F#. Currently there is no type inference so **arguments and functions without type annotation are assumed to be of type `int`.**

Introduction: external functions and minimal program

First of all we need to declare some external functions for printing. Also let's put them in module called `Prelude`:

```
module Prelude =
  external ll_putint : int -> ()
  external ll_print_line : () -> ()
```

File with implementation of external functions must be called `external.c` and present in current directory during compilation.

Example `external.c`:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

extern void ll_putint(int x) {
  printf("%d", x);
}

extern void ll_print_line() {
  printf("\n");
}
```

Notice that declarations of external functions defined in module need to be indented further then keyword `module`. However you don't need any other tokens like `struct` or `end`.

Program in order to be valid needs a `main` function of type `main : () -> int`. Main will be an entry point for a built binary. Integer returned from `main` will be an exit code.

Note that `()` specifies both unit type and unit literal.

Minimal program in *lang* can look like this:

```
module Prelude =
  external ll_putint : int -> ()
  external ll_print_line : () -> ()

let putint : int -> () = Prelude.ll_putint
```

```
let ln : () -> () = Prelude.ll_print_line

let main () =
  putint 42
  ln ()

  0
```

Make sure to print new line at the end of `main`, so buffer for stdout gets flushed (source).

Let, Let rec and tail-call optimisation

Now let's move to more complex examples. In *lang* we can define recursive `let` bindings with familiar syntax `let rec`:

```
let rec power a n =
  if n = 0
  then 1
  else a * power a (n - 1)

let main () =
  putint (power 2 10)
  ln ()

  0
```

You can also specify values with let:

```
let zero = 0
let zero_gt_than_zero : bool =
  if zero > 0
  then true
  else false
```

Second `let` definition needs type annotation of returned value as it isn't an integer.

`Let` expression can also be nested:

```
let fib n =
  let rec aux n a b =
    if n = 0
    then b
    else aux (n - 1) b (a + b)

  aux n 0 1
```

This fibonacci function is tail recursive. I can't guarantee that tail-call optimisation will be applied here, but in all tests I've done tail calls get optimised. The reason for this is that I'm suggesting *llvm* compiler for every function that it could be tail recursive. It's up to *llvm* static compiler whether this optimisation will actually happen. This is something that could be improved to a level of strong guarantee for tail recursive functions.

Arrays

What would be a language without mutable arrays? Let's add a couple more external functions, needed for creating new array, getting and setting elements of array.

`external.c`:

```
extern int ll_get_ith_elem_of_int_array(int* arr, int i) {
  return *(arr + i);
}

extern void ll_set_ith_elem_of_int_array(int* arr, int i, int val) {
  *(arr + i) = val;
}

extern int* ll_new_int_array(int size) {
  return (int *)malloc(sizeof(int) * size);
}
```

Example usage:

```
module Prelude =
  module Array =
    external ll_get_ith_elem_of_int_array : int array -> int -> int
    external ll_set_ith_elem_of_int_array : int array -> int -> int -> ()
    external ll_new_int_array : int -> int array

    (* Let's give better name to our external functions and
       define other useful functions like printn *)

     let new size : int array =
      ll_new_int_array size

    let set (arr : int array) ix elem : () =
      ll_set_ith_elem_of_int_array  arr ix elem

    let get (arr : int array) ix =
      ll_get_ith_elem_of_int_array arr ix
```

```
    let printn (arr : int array) size : () =
      let rec aux pos left : () =
        if left = 0
        then ln ()
        else printin (get arr pos)
             aux (pos + 1) (left - 1)

      aux 0 size

open Prelude (* module array will be now available *)

let main () : int =
  let arr : int array = Array.new 10

  Array.set arr 0 1
  Array.set arr 0 9

  Array.printn arr 10
  0
```

Array literals:

```
let global_arr : int array =
  [|1; 10;
     1
  |] (* valid array literal may be defined over multiple lines,
        but must have closing bracket at the same column as opening
        bracket*)

  (*
  [|1; 2
    |]  --> invalid array literal
   *)
```

Currently there are only arrays of integers available.

Modules

You've already seen some examples of modules usage. It worth mentioning that when opening some module, definitions inside opened module may shadow previous ones (as expected).

```
(* Previously defined Prelude module here *)
open Prelude

let zero = 0
```

```
module A =
  let testA = 1

  module B =
    let testB  = 2

    (* we can use values defined in outer scope *)
    let testB2 = testA + testB + zero

  let testA = 4 (* overrides previous definition of testA *)

let testA = 999

open A (* opening module A will override top-level definition of testA *)

let main () : int =

  putint testA
  Prelude.putint A.testA (* we can use qualified name although
                            Prelude has been opened *)
  Prelude.putint B.testB (* we opened module A so module B is available *)

  putint A.B.testB2 (* qualified name is still fine *)

  0
```

For more examples check `examples` folder and `test/compiler-test-srcs`.

Important notes

- *Langc* compiler currently doesn't do any type checking, it's only done on *llvm* static compiler level. If you forget about type annotation (for non-integer type) compiler will complain with an error relating to produced *llvm* intermediate representation code.

For this code:

```
  open Prelude

  let main () : int =
    let ar = Array.new 10 (* missing type annotation *)
    0
```

Compiler will produce following error: > llc: .langc_build_temp_1517087594.000000.ll:142:18: error: '@Prelude.Array.set' defined with type 'void ([0 x i32]*, *i32, i32)*' > tail call void @Prelude.Array.set(i32 %load_res, i32 0, i32 9999) > ^ > llc ".langc_build_temp_1517087594.000000.ll" -o ".langc_build_temp_1517087594.000000.ll.s"

-relocation-model=pic -O 3 > Uncaught exception: > > (src/core_sys.ml.Command_failed_with_status 1 > “llc ".langc_build_temp_1517087594.000000.ll" -o ".langc_build_temp_1517087594.000000.ll.s" -relocation-model=pic -O 3”) > > Raised at file “src/core_sys.ml”, line 74, characters 22-73 > Called from file “langc/compilation.ml”, line 51, characters 6-25 > Called from file “langc/compilation.ml”, line 56, characters 4-100 > Called from file “src/batInnerPervasives.mlv”, line 26, characters 6-9 > Re-raised at file “src/batInnerPervasives.mlv”, line 28, characters 22-29 > Called from file “langc/langc.ml”, line 20, characters 7-59

You can check produced llvm code in file `.langc_build_temp_1517087594.000000.ll`, present in current directory.

In rare cases compiler may crash with segmentation fault or other low-level exception when given incorrect code. This happens because of bindings to *C++ llvm* api being used. Make sure to double check type annotations or indentation.

- There is a subtle difference between negative integer literal and minus operator:

```
(* This is subtraction *)
let x = 5 - 5
let sub x = 5 - x

(* This is negative literal *)

let ten = sub -5 (* notice '-' is just before integer without any whitespace *)
```

- There is no garbage collector