Douglas Gastonguay-Goddard
Compiler Design - Ferguson
February 18, 2017

Description

This phase of the project implemented a compiler for the TINY language. I implemented this compiler in OCaml. My target language was LLVM intermediate representation (IR). I chose LLVM IR because of the extensive number of compiler tools available in LLVM and Clang. Later on, I will be able to write compiler passes (eg. optimization passes, obfuscation passes) using the LLVM infrastructure, which will save me a lot of time and effort compared to implementing these from scratch in another language.

OCaml has proven to be an excellent choice for compiler construction. This was my first project using it so learning the language added some difficulty to the project. However, with the resources I found I have been able to implement this phase of the project in less than 400 lines of code (not counting the millions of lines behind LLVM and OCaml).

The project is fairly simple. The lexer is defined in lexer.mll. This defines the regexes to match and the tokens to produce from those regexes. In ast.ml, the abstract syntax tree types and their parameters are defined. The file parser.mly defines how to transform the tokens to the types in the AST. The AST is passed into codegen.ml which transforms the AST types into LLVM IR. All of this is coordinated in main.ml.

As an example I compiled the factorial example included with the loucomp source files. That file and the LLVM IR output are included toward the end of the source files. LLVM IR has no mechanism for file IO, so I implemented the read() and write(int32_t) functions in a C file, io.c. I compiled the LLVM IR and C code using clang.

I am hosting the code, the rendered README with build instructions and resources, at https://github.com/douggard/cminus.

cminus/lexer.mll

```
{
(* Included code *)
open Parser
}

(* Regexes *)

let white = [' ' '\t' '\n']+
let digit = ['0'-'9']
let int = '-'? digit+
let letter = ['a'-'z' 'A'-'Z']
let id = letter+
let comment = '{' [^'}']* '}'

(* Character stream to token *)

rule read = parse
    | white     { read lexbuf }
    | comment   { read lexbuf }
    | "if"      { IF }
    | "then"    { THEN }
    | "else"    { ELSE }
    | "end"     { END }
    | "repeat"  { REPEAT }
    | "until"   { UNTIL }
    | "read"    { READ }
    | "write"   { WRITE }
    | ":="      { ASSIGN }
    | "+"       { PLUS }
    | "-"       { MINUS }
    | "*"       { MUL }
    | "/"       { DIV }
    | "="       { EQUALS }
    | "<"       { LT }
    | "("       { LPAREN }
    | ")"       { RPAREN }
    | ";"       { SEMI }
    | id        { ID (Lexing.lexeme lexbuf) }
    | int       { INT (int_of_string (Lexing.lexeme lexbuf)) }
    | eof       { EOF }
```

cminus/ast.ml

```
(* AST definition *)
type expr =
    | Var of string
    | Int of int
    | Ift of expr*expr list
    | Ife of expr*expr list*expr list
    | Repeat of expr list*expr
    | Read of expr
    | Write of expr
    | Assign of expr*expr
    | Add of expr*expr
    | Sub of expr*expr
    | Mul of expr*expr
    | Div of expr*expr
    | Equiv of expr*expr
    | Less of expr*expr

let ccatl l = String.concat "" l

let rec to_string = function
    | Var s -> ccatl ["( Var "; s; " )"]
    | Int i -> ccatl ["( Int "; (string_of_int i); " )"]
    | Assign (e1, e2) -> ccatl ((to_string e1) :: " := " :: (to_string e2) :: ["\n"])
    | Equiv (e1, e2) -> ccatl ((to_string e1) :: " = " :: (to_string e2) :: [])
    | Less (e1, e2) -> ccatl ((to_string e1) :: " < " :: (to_string e2) :: [])
    | Add (e1, e2) -> ccatl ((to_string e1) :: " + " :: (to_string e2) :: [])
    | Sub (e1, e2) -> ccatl ((to_string e1) :: " - " :: (to_string e2) :: [])
    | Mul (e1, e2) -> ccatl ((to_string e1) :: " * " :: (to_string e2) :: [])
    | Div (e1, e2) -> ccatl ((to_string e1) :: " / " :: (to_string e2) :: [])
    | Ift (e, el) -> ccatl
        ("If ( " :: (to_string e) :: " )\n(\n" :: (List.map to_string el) @ [")\n"])
    | Ife (e, e1, e2) -> ccatl
        (("If ( " :: (to_string e) :: " )\n(\n" :: (List.map to_string e1))
        @ ("\n) else (\n" :: (List.map to_string e2) @ [")\n"]))
    | Repeat (el, e) -> ccatl ("Repeat (\n" ::
        (List.map to_string el) @ [") until ("; (to_string e); " )\n"])
    | Read v -> ccatl ("( Read" :: (to_string v) :: [" )\n"])
    | Write e -> ccatl ("( Write" :: (to_string e) :: [" )\n"])
```

cminus/parser.mly

```
(* Included code *)

%{
open Ast
%}

(* Tokens *)

%token <int> INT
%token <string> ID
%token IF
%token THEN
%token ELSE
%token END
%token REPEAT
%token UNTIL
%token PLUS
%token MINUS
%token MUL
%token DIV
%token LPAREN
%token RPAREN
%token EQUALS
%token LT
%token ASSIGN
%token READ
%token WRITE
%token SEMI
%token EOF

(* Precedence and associativity *)

%right ASSIGN
%left EQUALS
%left LT
%left PLUS
%left MINUS
%left MUL
%left DIV

(* Entry point rule *)
```

```
%start <Ast.expr list> prog

%%

(* Grammar rules *)

prog:
    | s = stmt; EOF { [ s ] }
    | s = stmt; p = prog { s :: p }
    ;

stmt:
    | f = if_stmt { f }
    | i = io_stmt { i }
    | r = repeat_stmt { r }
    | e = expr; SEMI { e }
    ;

stmt_list:
    | s = stmt { [ s ] }
    | s = stmt ; sl = stmt_list { s :: sl }
    ;

if_stmt:
    | IF; e = expr; THEN; sl = stmt_list; END { Ift(e, sl) }
    | IF; e = expr; THEN; slt = stmt_list; ELSE; sle = stmt_list; END { Ife(e, slt, sle) }
    ;

io_stmt:
    | READ; x = ID; SEMI { Read(Var(x)) }
    | WRITE; e = expr; SEMI { Write(e) }
    ;

repeat_stmt:
    | REPEAT; sl = stmt_list; UNTIL; e = expr SEMI { Repeat(sl, e) }
    ;

expr:
    | i = INT { Int i }
    | x = ID { Var x }
    | e1 = expr; EQUALS; e2 = expr { Equiv(e1, e2) }
    | e1 = expr; LT; e2 = expr { Less(e1, e2) }
    | e1 = expr; PLUS; e2 = expr { Add(e1, e2) }
    | e1 = expr; MINUS; e2 = expr { Sub(e1, e2) }
    | e1 = expr; MUL; e2 = expr { Mul(e1, e2) }
    | e1 = expr; DIV; e2 = expr { Div(e1, e2) }
    | LPAREN; e = expr; RPAREN { e }
    | x = ID; ASSIGN ; e = expr { Assign(Var(x), e) }
    ;
```

cminus/codegen.ml

```
open Llvm

exception Error of string

let llvm_ctx = global_context ()
let llvm_mod = create_module llvm_ctx "tiny";;

let llvm_builder = builder llvm_ctx
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let i32_t = i32_type llvm_ctx
let void_t = void_type llvm_ctx

let fn_t = function_type i32_t [| |]
let wrt_t = function_type void_t [| i32_t |]

let read = declare_function "read" fn_t llvm_mod
let write = declare_function "write" wrt_t llvm_mod

let main = define_function "main" fn_t llvm_mod
let entry = entry_block main;;
position_at_end entry llvm_builder


let create_entry_block_alloca func name =
  let block_builder = builder_at llvm_ctx (instr_begin (entry_block func)) in
  let alloca = build_alloca i32_t name block_builder in
  Hashtbl.add named_values name alloca;
  alloca

let rec codegen_expr = function
    | Ast.Var n ->
        let v = try Hashtbl.find named_values n with
```

```
        | Not_found -> raise (Error "unknown variable name")
    in
    build_load v n llvm_builder
| Ast.Int i -> const_int i32_t i
| Ast.Assign (e1, e2) ->
    let name = match e1 with
        | Ast.Var name -> name
        | _ -> raise (Error "lhs of := must be var")
    in
    let _val = codegen_expr e2 in
    let variable = try Hashtbl.find named_values name with
        | Not_found -> create_entry_block_alloca main name;
    in
    ignore(build_store _val variable llvm_builder);
    _val
| Ast.Equiv (e1, e2) ->
    let lhs = codegen_expr e1 in
    let rhs = codegen_expr e2 in
    build_icmp Llvm.Icmp.Eq lhs rhs "eqtmp" llvm_builder
| Ast.Less (e1, e2) ->
    let lhs = codegen_expr e1 in
    let rhs = codegen_expr e2 in
    build_icmp Llvm.Icmp.Slt lhs rhs "slttmp" llvm_builder
| Ast.Add (e1, e2) ->
    let lhs = codegen_expr e1 in
    let rhs = codegen_expr e2 in
    build_add lhs rhs "addtmp" llvm_builder
| Ast.Sub (e1, e2) ->
    let lhs = codegen_expr e1 in
    let rhs = codegen_expr e2 in
    build_sub lhs rhs "subtmp" llvm_builder
| Ast.Mul (e1, e2) ->
    let lhs = codegen_expr e1 in
    let rhs = codegen_expr e2 in
    build_mul lhs rhs "multmp" llvm_builder
| Ast.Div (e1, e2) ->
    let lhs = codegen_expr e1 in
    let rhs = codegen_expr e2 in
    build_sdiv lhs rhs "divtmp" llvm_builder
| Ast.Ift (e, el) ->
    (* Create conditional *)
    let cond = codegen_expr e in
    (* Create if block *)
    let if_blk = append_block llvm_ctx "if.true" main in
    (* Add instructions to if block *)
    ignore(position_at_end if_blk llvm_builder);
    ignore(List.map codegen_expr el);
    (* Create end block *)
    let end_blk = append_block llvm_ctx "if.end" main in
    (* Create explicit branch from if -> end block *)
    let ifendbr = build_br end_blk llvm_builder in
    (* Create conditional branch *)
    let cond_blk = instr_parent cond in
    ignore(position_at_end cond_blk llvm_builder);
    let condbr =
        build_cond_br cond if_blk end_blk llvm_builder in
    (* Move to end *)
    ignore(position_at_end end_blk llvm_builder);
    condbr
| Ast.Ife (e, el1, el2) ->
    (* Create conditional *)
    let cond = codegen_expr e in
    (* Create if block *)
    let if_blk = append_block llvm_ctx "if.true" main in
    (* Add instructions to if block *)
    ignore(position_at_end if_blk llvm_builder);
    ignore(List.map codegen_expr el1);
    (* Create else block *)
    let else_blk = append_block llvm_ctx "if.false" main in
    (* Add instructions to else block *)
    ignore(position_at_end else_blk llvm_builder);
    ignore(List.map codegen_expr el2);
    (* Create end block *)
    let end_blk = append_block llvm_ctx "if.end" main in
    (* Create conditional *)
    let cond_blk = instr_parent cond in
    ignore(position_at_end cond_blk llvm_builder);
    let condbr =
        build_cond_br cond if_blk else_blk llvm_builder in
    (* Create explicit branch from if -> end *)
    ignore(position_at_end if_blk llvm_builder);
    let ifendbr = build_br end_blk llvm_builder in
    (* Create explicit branch from else -> end *)
    ignore(position_at_end else_blk llvm_builder);
    let elseendbr = build_br end_blk llvm_builder in
    (* Move to end *)
    ignore(position_at_end end_blk llvm_builder);
    elseendbr
| Ast.Repeat (el, e) ->
    (* Create repeat block *)
```

```
            let rpt_blk = append_block llvm_ctx "repeat.true" main in
            (* Create explicit branch to repeat *)
            let brrpt = build_br rpt_blk llvm_builder in
            (* Add instructions to repeat block *)
            ignore(position_at_end rpt_blk llvm_builder);
            ignore(List.map codegen_expr el);
            (* Set up conditional *)
            let cond = codegen_expr e in
            (* Add end block *)
            let end_blk = append_block llvm_ctx "repeat.end" main in
            (* Create branch from rpt -> rpt | end *)
            ignore(position_at_end rpt_blk llvm_builder);
            let condbr =
                build_cond_br cond end_blk rpt_blk llvm_builder in
            ignore(position_at_end end_blk llvm_builder);
            condbr
        | Ast.Read v ->
            let read_val =
                build_call read [| |] "readtmp" llvm_builder in
            let name = match v with
                | Ast.Var name -> name
                | _ -> raise (Error "argument to read must be var")
            in
            let variable = try Hashtbl.find named_values name with
                | Not_found -> create_entry_block_alloca main name;
            in
            ignore(build_store read_val variable llvm_builder);
            read_val
        | Ast.Write e ->
            let _val = codegen_expr e in
            ignore(build_call write [| _val |] "" llvm_builder);
            _val

    let codegen expr_list =
        List.map codegen_expr expr_list;
        let ret = const_int i32_t 0 in
        build_ret ret llvm_builder;
```

cminus/main.ml

```
    open Ast

    let parse s =
        let lexbuf = Lexing.from_string s in
        let ast = Parser.prog Lexer.read lexbuf in
        ast

    let file = "sample.tny"
    let contents = Core.Std.In_channel.read_all file
    let parsed = parse contents;;

    (* let str_list = List.map to_string parsed;;
    print_string (String.concat "" (str_list));; *)

    Codegen.codegen parsed;;
    Llvm.dump_module Codegen.llvm_mod;;
```

cminus/samples/io.c

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

    int32_t read() {
        int val = 0;
        printf("READ: ");
        scanf("%d", &val);
        return val;
    }

    void write(int32_t val) {
        printf("WRITE: %d\n", val);
    }
```

cminus/samples/fact/fact.tny

```
    { Sample program
      in TINY language -
```

```
  computes factorial
}
read x;
if 0 < x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1;
  until x = 0;
  write fact;
end
```

cminus/samples/fact/fact.ll

```
; ModuleID = 'tiny'
source_filename = "tiny"

declare i32 @read()

declare void @write(i32)

define i32 @main() {
entry:
  %fact = alloca i32
  %x = alloca i32
  %readtmp = call i32 @read()
  store i32 %readtmp, i32* %x
  %x1 = load i32, i32* %x
  %slttmp = icmp slt i32 0, %x1
  br i1 %slttmp, label %if.true, label %if.end

if.true:                                          ; preds = %entry
  store i32 1, i32* %fact
  br label %repeat.true

repeat.true:                                      ; preds = %repeat.true, %if.true
  %fact2 = load i32, i32* %fact
  %x3 = load i32, i32* %x
  %multmp = mul i32 %fact2, %x3
  store i32 %multmp, i32* %fact
  %x4 = load i32, i32* %x
  %subtmp = sub i32 %x4, 1
  store i32 %subtmp, i32* %x
  %x5 = load i32, i32* %x
  %eqtmp = icmp eq i32 %x5, 0
  br i1 %eqtmp, label %repeat.end, label %repeat.true

repeat.end:                                       ; preds = %repeat.true
  %fact6 = load i32, i32* %fact
  call void @write(i32 %fact6)
  br label %if.end

if.end:                                           ; preds = %entry, %repeat.end
  ret i32 0
}
```

cminus/README.md

```
# cminus

This project is to implement a compiler from C- to LLVM IR. C Minus is a subset of C, defined in the book
[Compiler Construction: Principles and Practice by Kenneth C. Louden](http://www.cs.sjsu.edu/~louden/cmptext/).

This project will initially implement the TINY language, also defined in Compiler Construction. A tag will be
made when a full compiler for TINY is completed.

This is my first project in OCaml, so pardon the mess.

## Resources

I'm using the [Cornell's CS3110 parsing
code](http://www.cs.cornell.edu/courses/cs3110/2015fa/l/12-interp/rec.html) as my basis. This was the cleanest
code I found that defined a language and produced an AST. The comments are very thorough (though I will delete
most of them as time goes on). It uses the [menhir](http://gallium.inria.fr/~fpottier/menhir/) parser
generator.
```

A good resource for code is the [LLVM Kaleidoscope OCaml tutorial](http://llvm.org/docs/tutorial/OCamlLangImpl1.html). It uses camlp4 and the code is more complex than the Cornell tutorial. It includes examples of code generation and other further pieces of a compiler.

The [documentation for the LLVM OCaml bindings](https://llvm.moe/ocaml/).

[Part 3](https://www.wzdftpd.net/blog/ocaml-llvm-03.html) of @chifflier's OCaml LLVM bindings tutorial is very helpful for code generation.

Dependencies

Install ocaml.

Build LLVM from source, it should detect ocaml and install bindings when you `sudo make install`. On OS Sierra you can check for `/usr/local/lib/ocaml/llvm*`.

Alternatively, you can try to install LLVM with your package manager, then install the bindings with `opam install llvm`.

If you want these in utop you have to build a custom one. You can build an `llvmutop` with step 3 [here](https://xysun.github.io/posts/install-llvm-ocaml-bindings-and-toplevel.html), credit @xysun.

```

echo "let () = UTop_main.main ()" > myutop_main.ml

ocamlfind ocamlmktop -o llvmutop -thread -linkpkg -package utop llvm.cma myutop_main.ml -cc g++

```

Building

`ocamlbuild -lib=llvm -use-ocamlfind -pkg core -tag thread main.byte`

Installing OCaml on CentOS 6

TBD (for professor)

cminus/_tags

```
true: use_menhir, debug
```

code2pdf/code2pdf.py

```python
# Copyright Douglas Gastonguay-Goddard 2017

import os
import sys
import json
from datetime import datetime as dt
from reportlab.lib.units import inch
from reportlab.lib.pagesizes import landscape, letter
from reportlab.lib.pygments2xpre import pygments2xpre as p2x
from reportlab.lib.styles import ParagraphStyle, getSampleStyleSheet
from reportlab.platypus import Paragraph, SimpleDocTemplate, XPreformatted

def add_heading(heading, paragraphs):
    for line in heading:
        if line == '[DATE]':
            line = dt.now().strftime('%B %d, %Y')
        paragraphs.append(Paragraph(line, style=styles['heading']))

def add_description(fname, paragraphs):
    paragraphs.append(Paragraph('Description', style=styles['title']))
    with open(fname) as description_file:
        contents = description_file.read()
        for paragraph in contents.split('\n'):
            if paragraph == '':
                continue
            paragraphs.append(Paragraph(paragraph, style=styles['paragraph']))

def init_styles():
    global styles
    styles = {}

    heading_style = ParagraphStyle('heading')
```

```
    heading_style.alignment = 2 # right
    styles['heading'] = heading_style

    paragraph_style = ParagraphStyle('paragraph')
    paragraph_style.spaceBefore = 0.1*inch
    paragraph_style.leftIndent = 0.5*inch
    styles['paragraph'] = paragraph_style

    code_style = getSampleStyleSheet()["Code"]
    styles['code'] = code_style

    title_style = getSampleStyleSheet()["Title"]
    title_style.alignment = 0   # left
    title_style.spaceBefore = 0.5*inch
    styles['title'] = title_style

    markdown_style = ParagraphStyle('markdown')
    markdown_style.spaceBefore = 0.1*inch
    markdown_style.leftIndent = 0.5*inch
    markdown_style.fontName = 'Courier'
    markdown_style.fontSize = 8
    styles['markdown'] = markdown_style

def add_markdown(fname, paragraphs):
    with open(fname) as md_file:
        contents = md_file.read()

        for paragraph in contents.split('\n'):
            if paragraph == '':
                continue
            paragraphs.append(Paragraph(paragraph, style=styles['markdown']))

def add_file(fname, paragraphs):
    extensions = {
        'py': 'python',
        'ml': 'ocaml',
        'mll': 'ocaml',
        'mly': 'ocaml',
        'c': 'c',
        'h': 'c++',
        'cpp': 'c++',
        'cc': 'c++',
        'll': 'llvm',
        'java': 'java',
    }

    paragraphs.append(Paragraph(fname, style=styles['title']))
    with open(fname) as source_file:
        language = 'python'
        if '.' in fname:
            ext = fname.split('.')[-1]
            if ext == 'md':
                add_markdown(fname, paragraphs)
                return
            if ext in extensions:
                language = extensions[ext]

        source = source_file.read()
        pretty = p2x(source, language=language)
        formatted = XPreformatted(pretty, style=styles['code'])
        paragraphs.append(formatted)

def main():
    paragraph_style = ParagraphStyle({})
    paragraphs = []
    init_styles()

    config_file = 'config.json'
    if len(sys.argv) > 1 and os.path.exists(sys.argv[1]):
        config_file = sys.argv[1]

    doc = SimpleDocTemplate(
            "out.pdf",
            # pagesize=landscape(letter),
            rightMargin=0.1*inch,
            leftMargin=0.1*inch,
            topMargin=0.25*inch,
            bottomMargin=0.25*inch)

    with open(config_file) as cfg:
        contents = cfg.read()
        config = json.loads(contents)

    add_heading(config['heading'], paragraphs)
    add_description(config['description_file'], paragraphs)

    for fname in config['files']:
        add_file(fname, paragraphs)

    doc.build(paragraphs)
```

```
if __name__ == '__main__':
    main()
```