

Milner

Alan Hu

2020

# Milner

Over the past three weeks, I've been creating my own programming language, Milner

<https://github.com/milner-lang/milner>

```
external puts : (Cstring) -> ()
```

```
val main : () -> Int32  
fun main() =  
  puts("Hello, world!");  
  0i32
```

# What is a Programming Language?

# What is a Programming Language?

## ► Syntax

$$e ::= \lambda x.e \mid e(e) \mid x, y, z \dots$$

# What is a Programming Language?

## ► Syntax

$$e ::= \lambda x. e \mid e(e) \mid x, y, z \dots$$

## ► Semantics

$$\frac{e_1 \longrightarrow e'_1}{e_1(e_2) \longrightarrow e'_1(e_2)}$$
$$\frac{}{(\lambda x. e_1)(e_2) \longrightarrow e_1[x/e_2]}$$

# What is a Programming Language?

## ► Syntax

$$e ::= \lambda x.e \mid e(e) \mid x, y, z \dots$$

## ► Semantics

$$\frac{e_1 \longrightarrow e'_1}{e_1(e_2) \longrightarrow e'_1(e_2)}$$

$$\overline{(\lambda x.e_1)(e_2) \longrightarrow e_1[x/e_2]}$$

A programming language is an abstract idea

A programming language *implementation* realizes the abstract language definition

A programming language *implementation* realizes the abstract language definition

- ▶ Compiler



A programming language *implementation* realizes the abstract language definition

- ▶ Compiler

- ▶ Translate source language into machine language (executable file)

A programming language *implementation* realizes the abstract language definition

- ▶ Compiler
  - ▶ Translate source language into machine language (executable file)
- ▶ Interpreter

A programming language *implementation* realizes the abstract language definition

- ▶ Compiler
  - ▶ Translate source language into machine language (executable file)
- ▶ Interpreter
  - ▶ Execution is evaluation

A *reference implementation* is a specification for the language

# Compilers

How does a compiler or interpreter work?

# Compilers

How does a compiler or interpreter work?

- ▶ Lexer

# Compilers

How does a compiler or interpreter work?

- ▶ Lexer
- ▶ Parser

# Compilers

How does a compiler or interpreter work?

- ▶ Lexer
- ▶ Parser
- ▶ Analysis

# Compilers

How does a compiler or interpreter work?

- ▶ Lexer
- ▶ Parser
- ▶ Analysis
- ▶ Codegen



# Compilers

How does a compiler or interpreter work?

- ▶ Lexer
- ▶ Parser
- ▶ Analysis
- ▶ Codegen

In practice, interpreters compile to a low-level *bytecode* which is then executed





- ▶ OCaml - Implementation language



- ▶ OCaml - Implementation language
- ▶ Developed at Inria - Coq proof assistant as motivation



- ▶ OCaml - Implementation language
- ▶ Developed at Inria - Coq proof assistant as motivation
- ▶ Descendant of Robin Milner's proof assistant meta language, ML



- ▶ OCaml - Implementation language
- ▶ Developed at Inria - Coq proof assistant as motivation
- ▶ Descendant of Robin Milner's proof assistant meta language, ML
- ▶ Excels at compiler development, symbolic manipulation

# Reading

Compilers aren't magical

# Reading

Compilers aren't magical

- ▶ A programming language is a file format



# Reading

Compilers aren't magical

- ▶ A programming language is a file format
- ▶ Read file, process data, output file

# Reading

Compilers aren't magical

- ▶ A programming language is a file format
- ▶ Read file, process data, output file
- ▶ Compilers are glorified file converters

# Reading

Compilers aren't magical

- ▶ A programming language is a file format
- ▶ Read file, process data, output file
- ▶ Compilers are glorified file converters
- ▶ Files are sequences of bytes on your hard disk

# Reading

Compilers aren't magical

- ▶ A programming language is a file format
- ▶ Read file, process data, output file
- ▶ Compilers are glorified file converters
- ▶ Files are sequences of bytes on your hard disk
- ▶ Parse text into an abstract syntax tree

# Reading

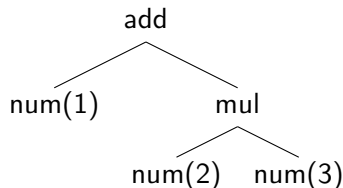
Compilers aren't magical

- ▶ A programming language is a file format
- ▶ Read file, process data, output file
- ▶ Compilers are glorified file converters
- ▶ Files are sequences of bytes on your hard disk
- ▶ Parse text into an abstract syntax tree
- ▶  $1 + 2 \times 3$

# Reading

Compilers aren't magical

- ▶ A programming language is a file format
- ▶ Read file, process data, output file
- ▶ Compilers are glorified file converters
- ▶ Files are sequences of bytes on your hard disk
- ▶ Parse text into an abstract syntax tree
- ▶  $1 + 2 \times 3$
- ▶



```

type 'a annot = {
  annot_item : 'a;
  annot_begin : Lexing.position;
  annot_end   : Lexing.position;
}

```

Figure: Source location annotation

```

type expr =
| Apply_expr of expr annot * expr annot list
| Lit_expr   of literal
| Seq_expr   of expr annot * expr annot
| Var_expr   of string

```

Figure: Expression tree definition

# Typechecking



# Typechecking

- ▶ Not all programs have a meaning

# Typechecking

- ▶ Not all programs have a meaning
  - ▶ “Hello” + 1

# Typechecking

- ▶ Not all programs have a meaning
  - ▶ “Hello” + 1
  - ▶ True(3)
- ▶ Ascribe a “type” to each term

# Typechecking

- ▶ Not all programs have a meaning
  - ▶ “Hello” + 1
  - ▶ True(3)
- ▶ Ascribe a “type” to each term
- ▶ Static semantics

# Typechecking

- ▶ Not all programs have a meaning
  - ▶ “Hello” + 1
  - ▶ True(3)
- ▶ Ascribe a “type” to each term
- ▶ Static semantics
  - ▶

$$\frac{e_1 : Int \quad e_2 : Int}{e_1 + e_2 : Int}$$

# Typechecking

- ▶ Not all programs have a meaning
  - ▶ “Hello” + 1
  - ▶ True(3)
- ▶ Ascribe a “type” to each term
- ▶ Static semantics



$$\frac{e_1 : Int \quad e_2 : Int}{e_1 + e_2 : Int}$$



$$\frac{e_1 : A \rightarrow B \quad e_2 : A}{e_1(e_2) : B}$$

- ▶ Hindley-Milner type system

# Code Generation

# Code Generation

- ▶  $M$  languages,  $N$  target architectures



# Code Generation

- ▶  $M$  languages,  $N$  target architectures
- ▶  $M \times N$  compiler combinations

# Code Generation

- ▶  $M$  languages,  $N$  target architectures
- ▶  $M \times N$  compiler combinations
- ▶ Separate compiler into *frontend* and *backend*

# Code Generation

- ▶  $M$  languages,  $N$  target architectures
- ▶  $M \times N$  compiler combinations
- ▶ Separate compiler into *frontend* and *backend*
- ▶ *frontend* :  $Source \rightarrow IR$

# Code Generation

- ▶  $M$  languages,  $N$  target architectures
- ▶  $M \times N$  compiler combinations
- ▶ Separate compiler into *frontend* and *backend*
- ▶ *frontend* :  $Source \rightarrow IR$
- ▶ *backend* :  $IR \rightarrow Target$

# Code Generation

- ▶  $M$  languages,  $N$  target architectures
- ▶  $M \times N$  compiler combinations
- ▶ Separate compiler into *frontend* and *backend*
- ▶ *frontend* :  $Source \rightarrow IR$
- ▶ *backend* :  $IR \rightarrow Target$
- ▶ *compiler* :  $Source \rightarrow Target$

# Code Generation

- ▶  $M$  languages,  $N$  target architectures
- ▶  $M \times N$  compiler combinations
- ▶ Separate compiler into *frontend* and *backend*
- ▶  $frontend : Source \rightarrow IR$
- ▶  $backend : IR \rightarrow Target$
- ▶  $compiler : Source \rightarrow Target$
- ▶  $compiler = backend \circ frontend$

# LLVM



# LLVM



- ▶ LLVM - Low-Level Virtual Machine



# LLVM



- ▶ LLVM - Low-Level Virtual Machine
- ▶ SSA - Static Single Assignment

# LLVM



- ▶ LLVM - Low-Level Virtual Machine
- ▶ SSA - Static Single Assignment
- ▶ Official OCaml API - <https://llvm.moe/>

Questions?