Language implementations: Arithmetic expressions

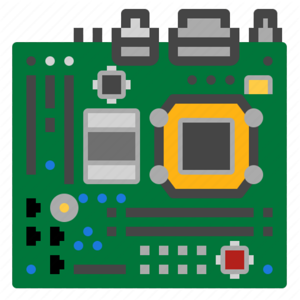
FPLI 2023, week 3

Morten Rhiger

Roskilde University

Version 2023-03-19 17:48:21

March 20, 2023

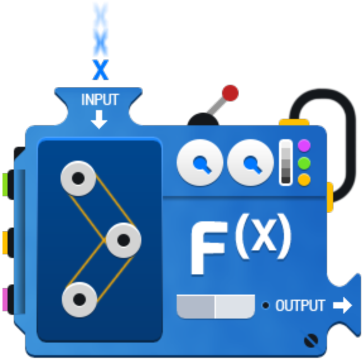


- Programs processors
- Parsing
- Arithmetic expressions
  - — Interpreter
  - — “Assembly code”
  - — Emulating Assembly code
  - — Compiler

Program processors

One program takes another program as input or produces another program as output.

- Interpreting
- Compiling
- Type checking
- Analyzing
- Optimizing
- Refactoring



Interpreting a program

Input

$\downarrow$

(1) Source program $\longrightarrow$ **Interpreter** $\longrightarrow$ Output

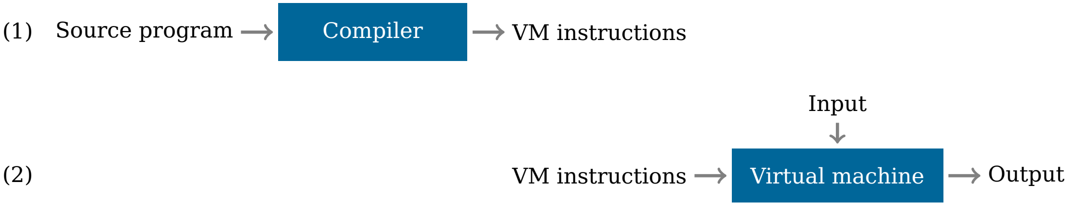
Compile to executable

(1) Source program → Compiler → Executable

Input ↓

(2) Executable → Output

Compile to virtual machine



Example interpreter (python)

```
> cat pgm.py
print("hello world\n")
> python pgm.py
hello world
>
```

Example interpreter (python)

Source program

Interpreter

```
> cat pgm.py
print("hello world\n")
> python pgm.py
hello world
>
```

Output

Example compiler (gcc)

```
> cat pgm.c
#include <stdio.h>
void main() { printf("hello world\n"); }
> gcc pgm.c -o pgm
> ./pgm
hello world
>
```

Example compiler (gcc)

Source program

Compiler

```
> cat pgm.c
#include <stdio.h>
void main() { printf("hello world\n"); }
> gcc pgm.c -o pgm
> ./pgm
hello world
>
```

Executable

Output

Example virtual machine (java)

```
> cat Pgm.java
public class Pgm {
  public static void main(String[] args) {
    System.out.println("hello world");
  }
}
> javac Pgm.java
> java Pgm
hello world
>
```

Example virtual machine (java)

Source program

```
> cat Pgm.java
public class Pgm {
  public static void main(String[] args) {
    System.out.println("hello world");
  }
}
```

Compiler

```
> javac Pgm.java
```

Virtual machine

```
> java Pgm
```

VM instructions

```
hello world
>
```

Output

Interpreters

An interpreter is a program

that runs programs

and returns the results of these programs.

Interpreters

An interpreter is a program

(implemented in the *metalanguage* or *interpreting* language, F#)

that runs programs

(implemented in the *source* or *interpreted* or *object* language)

and returns the results of these programs.

Compilers

A compiler is a program

that translates programs

into programs

such that the results

are the same.

Compilers

A compiler is a program

(implemented in the *metalanguage*, F#)

that translates programs

(implemented in the *source* language)

into programs

(implemented in the *target* language),

such that the results of running a source program and

its target program

are the same.

In the rest of the course

For a number of small languages,

1. Implement an interpreter
2. Define low-level instructions
3. Implement a compiler
4. Implement virtual machine

In the rest of the course

For a number of small languages,

1. Implement an interpreter
2. Define low-level instructions
3. Implement a compiler
4. Implement virtual machine

Why?

- Teaches us the fundamental principles of the language we implement
- Different languages $\Rightarrow$ different fundamental principles

How?

- We start simple, and extend with more and more features

Defining a language

We have introduced F# in a

- ad hoc (rather that systematic),
- informal,
- “top down,”
- incomplete, and
- example-based

manner.

But programmers and compiler-writers need *formal, complete,* and *unambiguous* definitions.

Defining a language, how?

- Syntax
- Type system
- Semantics

Syntax of arithmetic expressions

expression:

integer

expression + *expression*

(*expression*)

integer: one of

`...  -2  -1  0  1  2  ...`

Syntax of arithmetic expressions

expression:

    integer

    expression + *expression*

    (*expression*)

integer: one of

    ... -2 -1 0 1 2 ...

Example expressions: `1` `((1))` `1+((2+3))+4+(5)` `1 + -2`

Syntax of arithmetic expressions

expression:

  integer

  expression + *expression*

  `(` *expression* `)`

integer: one of

  `...  -2  -1  0  1  2  ...`

Example expressions: `1` `((1))` `1+((2+3))+4+(5)` `1 + -2`

Not expressions: `()` `pi` `1-2` `1++2` `1 2`

Representing programs

Concrete syntax?

```
"14 + (2 + 19)"
```

- No program structure
- Difficult to manipulate
- Useless for practical purposes

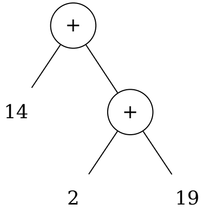
Representing programs

Concrete syntax?

```
"14 + (2 + 19)"
```

- No program structure
- Difficult to manipulate
- Useless for practical purposes

Abstract syntax tree!



- Program structure is preserved
- Easily to manipulate by recursive functions

AST of arithmetic expressions

```
type exp = | INT of int
           | ADD of exp * exp
```

AST of arithmetic expressions

```
type exp = | INT of int
           | ADD of exp * exp
```

Concrete syntax	Abstract syntax tree
3	`INT 3`
1 + 2	`ADD (INT 1, INT 2)`
3 + (4 + 5)	`ADD (INT 3, ADD (INT 4, INT 5))`
(3 + 4) + 5	`ADD (ADD (INT 3, INT 4), INT 5)`

Interpreting arithmetic expressions

In practice, an interpreter

- traverses the AST of the source program,
- decodes the current AST node, and
- performs operations in the metalanguage that mimics the source language constructs.

```
let rec eval = function
    | INT i        -> i
    | ADD (e1, e2) -> eval e1 + eval e2
```

Interpreting arithmetic expressions

In practice, an interpreter

- traverses the AST of the source program,
- decodes the current AST node, and
- performs operations in the metalanguage that mimics the source language constructs.

```
let rec eval = function
    | INT i          -> i
    | ADD (e1, e2) -> eval e1 + eval e2

// or better

let rec eval = function
    | INT i          -> i
    | ADD (e1, e2) -> let v1 = eval e1
                       let v2 = eval e2
                       v1 + v2
```

Implementing a compiler

In practice, a compiler

- traverses the AST of the source program,
- decodes the current AST node, and
- generates code that performs operations in the target language that mimics the source language constructs.

Stack-based target languages

Stack of things:

- Add a thing on the top
- Remove a thing from the top

In computers,

- a stack store intermediate results.

Instruction set

- Instructions:

```
type inst = | IPUSH of int
            | IADD
```

Instruction set

- Instructions:

```
type inst = | IPUSH of int
            | IADD
```

- Executing instructions on a stack:

Instruction	Stack before		Stack after
IPUSH i	$v_0, \ldots, v_{n-1}$	$\longrightarrow$	$i, v_0, \ldots, v_{n-1}$
IADD	$y, x, v_0, \ldots, v_{n-1}$	$\longrightarrow$	$(x + y), v_0, \ldots, v_{n-1}$

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

```
IPUSH 1
IPUSH 3
IADD
IPUSH 5
IPUSH 2
IADD
IADD
```

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

	Stack after
`IPUSH 1`	1
`IPUSH 3`	
`IADD`	
`IPUSH 5`	
`IPUSH 2`	
`IADD`	
`IADD`	

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

	Stack after
IPUSH 1	1
IPUSH 3	3, 1
IADD	
IPUSH 5	
IPUSH 2	
IADD	
IADD	

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

	Stack after
`IPUSH 1`	`1`
`IPUSH 3`	`3, 1`
`IADD`	`4`
`IPUSH 5`	
`IPUSH 2`	
`IADD`	
`IADD`	

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

	Stack after
`IPUSH 1`	1
`IPUSH 3`	3, 1
`IADD`	4
`IPUSH 5`	5, 4
`IPUSH 2`	
`IADD`	
`IADD`	

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

	Stack after
`IPUSH 1`	1
`IPUSH 3`	3, 1
`IADD`	4
`IPUSH 5`	5, 4
`IPUSH 2`	2, 5, 4
`IADD`	
`IADD`	

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

	Stack after
`IPUSH 1`	1
`IPUSH 3`	3, 1
`IADD`	4
`IPUSH 5`	5, 4
`IPUSH 2`	2, 5, 4
`IADD`	7, 4
`IADD`	

Execution on a stack machine

- Source program

```
(1 + 3) + (5 + 2)
```

- Abstract syntax tree

```
ADD (ADD (INT 1, INT 3),
     ADD (INT 5, INT 2))
```

- Compiled program

	Stack after
`IPUSH 1`	1
`IPUSH 3`	3, 1
`IADD`	4
`IPUSH 5`	5, 4
`IPUSH 2`	2, 5, 4
`IADD`	7, 4
`IADD`	11

Compiling to a stack machine

If expression e compiles to instructions `ins`, then executing `ins` with a stack

$$v_0, \ldots, v_{n-1}$$

must result in a stack

$$i, v_0, \ldots, v_{n-1}$$

where i is the value of e.

Compiling arithmetic expressions

```
type exp =
  | INT of int
  | ADD of exp * exp

type inst =
  | IPUSH of int
  | IADD
```

```
let rec comp = function
  | INT i          -> [IPUSH i]
  | ADD (e1, e2) -> comp e1 @
                    comp e2 @
                    [IADD]
```

Emulating the virtual machine

```
let rec exec ins st =
  match (ins, st) with
    | ([],              v :: _)       -> v
    | (IPUSH i :: ins, st)            -> exec ins (i     :: st)
    | (IADD    :: ins, y :: x :: st) -> exec ins (x + y :: st)
```

Phases of a compiler

Stream of characters

↓

Lexer

↓

Stream of tokens

↓

Parser

↓

Abstract syntax tree

↓

Code generator

↓

Instructions (mnemonic)

↓

Assembler

↓

Instructions (bytes)

Digression: Arithmetic expressions in Java

Abstract syntax trees and an interpreter

```java
abstract class Exp {
  abstract int eval();
}

class INT extends Exp {
  int i;
  INT(int i) { this.i = i; }
  int eval() { return i; }
}
```

```java
class ADD extends Exp {
  Exp e1, e2;
  ADD(Exp e1, Exp e2) {
    this.e1 = e1;
    this.e2 = e2;
  }
  int eval() {
    return e1.eval() + e2.eval();
  }
}
```