Universitat Autònoma de Barcelona

Treball de Fi de Grau

Informe inicial

Disseny i implementació d'un llenguatge de programació amb LLVM

Autor
Josep Maria Domingo Catafal

Tutor
Javier Sánchez Pujadas

8 d'octubre de 2022

Índex

1 Introducció i el problema a resoldre

Històricament, hi ha hagut sempre un dilema entre la rapidesa d'execució, i la rapidesa de desenvolupament.

Alguns llenguatges són fàcils de programar: permeten al programador despreocupar-se de conceptes de baix nivell com ara la gestió de la memòria, i creen abstraccions que agilitzen el desenvolupament. El problema és que aquestes abstraccions, limiten l'eficiència del llenguatge, i creen programes més lents. Un altre motiu que permet agilitzar el desenvolupament és el tipat dinàmic, ja que permet despreocupar-se de quin tipus s'ha de fer servir. Però això també té els seus inconvenients, donat que és molt probable que et trobis amb errors en temps d'execució. Aquests llenguatges acostumen a ser interpretats per tal d'estalviar l'espera al programador, però té un impacte en el rendiment d'execució del programa si ho comparem amb llenguatges compilats. Python, per exemple, seria un dels majors representants d'aquest grup de llenguatges.

En l'altra cara de la moneda i tenim llenguatges com C, els quals no tenen gairebé cap abstracció i el programador ha de ser conscient del que fa en cada línia de codi que escriu. Són llenguatges que aporten un molt bon rendiment, però alenteixen molt el desenvolupament, ja que el programador ha de tenir en compte molts conceptes de baix nivell. Un problema addicional és que al gestionar la memòria de forma manual, s'obre la porta a una gran quantitat d'errors en temps d'execució en la forma de memory leaks. Actualment, hi ha llenguatges com Rust que soluciones aquestes qüestions de gestió de memòria sense perdre rendiment, però el desenvolupament continua sent lent i el temps de compilació llarg. Aquests llenguatges acostumen a ser fortament tipats, cosa que redueix els errors en temps d'execució, però alenteix el desenvolupament.

2 Objectiu

Podem veure que disposem de dos paradigmes ben oposats: un busca el rendiment a canvi d'un desenvolupament lent, mentre l'altre prefereix agilitzar el desenvolupament a canvi d'alentir l'execució. L'objectiu d'aquest treball, és crear un llenguatge de programació que se situï al mig d'aquests dos paradigmes, fent sacrificis per ambdues bandes.

Sacrificis en l'agilitat de desenvolupament:

- **Compilat**: Serà un llenguatge compilat, per tal d'obtenir un bon rendiment i evitar al màxim els erros en temps d'execució.
- **Fortament tipat**: Per evitar al màxim l'aparició d'errors en temps d'execució, i també perquè ajuda a documentar el codi, facilitant la seva lectura.
- **Immutabilitat**: S'intentarà forçar la immutabilitat sempre que sigui possible, per així evitar els efectes secundaris, i facilitar la depuració del codi

Sacrificis en el rendiment d'execució

- **Gestió de memòria automàtica**: L'usuari no s'haurà de preocupar d'eliminar la memòria de forma manual. Hi haurà un Garbage Collector, cosa que penalitza en el rendiment, però redueix memory leaks i agilitza el desenvolupament.

3 Definició del llenguatge

Per importar llibreries externes s'utilitzen imports

```
1 import math
```

El punt d'inici del programa és a la funció main.

```
fn main() {
    // Les instruccions acaben sempre amb un punt i coma
    println("Hello World!");
}
```

Les variables, per defecte, són immutables. Amb això es busca limitar sempre que es pugui els efectes secundaris. Si es vol mutar una variable, s'ha d'afegir la paraula *mut* al declarar-la:

```
let a = 0;
a += 1; // invalid

let mut a = 0;
a += 1; // valid
```

Podem definir constants i variables globals, però les variables no poden ser mai mutables, és a dir un cop inicialitzades, el seu valor serà constant. Amb això es busca evitar comportaments estranys del programa i bugs difícils de trobar, ja que les variables globals poden ser modificada des de qualsevol lloc i costa trobar qui és que l'està canviant. Un altre problema de les variables globals és el multithreading, ja que s'ha de controlar l'accés a aquestes, per tal d'evitar *race conditions*.

```
const MAX_SPEED = 120;
let MIN_SPEED = get_config("MIN_SPEED");
```

En l'exemple anterior, la funció get_config s'executa a l'iniciar el programa, retorna un valor, i s'assigna a MIN_SPEED. Després d'això, el valor ja no es pot modificar.

Si segueixes necesitant variables globals que siguin mutables, hauràs de canviar el disseny del teu codi perquè passi les variables a les funcions per paràmetre. Si tens molts paràmetres, els pots englobar en un struct, que veurem més endavant. Aquesta filosofia és similar a la que segueix Haskell [10].

Per declarar una funció es fa amb la paraula reservada fn, seguit del nom de la funció, els paràmetres entre parèntesis, i finalment el tipus del valor de retorn. Quan l'última expressió no acaba en punt i coma, el resultat serà el que retornarà la funció. És l'únic cas en què es permet no posar punt i coma.

```
fn radius(circumference: i32) i32 {
    circumference / (2 * math.PI)
}
```

3.1 Tipus

Els tipus definits pel llenguatge són:

- **i32**: Enters de 32 bits
- **i64**: Enters de 64 bits
- **f32**: Nombres en coma flotant de 32 bits
- **f64**: Nombres en coma flotant de 64 bits

- **bool**: true o false
- **string**: Cadenes de caràcters
- **arrays**: Conjunt d'elements del mateix tipus

Per declarar arrays, es fa de la forma següent:

```
let array = [1, 2, 3, 4];

// o

let mut array = i32[4]; // len == 4
array[0] = 1;
array[1] = 2;
// ...
```

3.2 Definició de tipus

Si volem definir tipus genèrics, podem definir interfícies:

```
interface Person {
    fn change_name(name: string) string;
}
```

Això ens permet tenir diferents implementacions, amb comportaments diferents.

Per crear tipus propis, es fan servir structs. Permeten definir camps de diferents tipus i definir mètodes que modifiquen l'estat de l'objecte.

Els camps i els mètodes, per defecte són privats, però es poden fer públics. En el cas de les funcions amb la paraula reservada "pub". En el cas dels camps, afegim entre claus ({}), les paraules get o set, depenent de si volem que sigui pública per escriptura, lectura o per les dues coses.

Per implementar una interfície, afegim dos punts i el nom de la interfície just després del nom del struct.

```
struct User : Person {
    age: i32 { get, set };
    name: string { get };
    id: i32;

    fn change_name(name: string) {
        self.name = name;
    }
}
```

Amb l'struct anterior, podríem per exemple fer el següent:

```
let user = User { age: 24, name: "John Doe" };
user.age = 25;
println(user.name);
```

Però no podríem modificar el camp *name*.

No és obligatori definir constructor, però si es vol, l'estàndard és crear una funció estàtica que es digui *new*.

```
// constructor
pub static fn new(age: i32, name: string) self {
    self {
        age,
        name,
    }
}
```

Per cridar les funcions estàtiques es fa de la següent manera:

```
// static: struct_name::static_fn()
let user = User::new(24, "John Doe");
// non-static: object.fn()
user.change_name("Jane Doe");
```

Podem definir mètodes fora de la definició del struct. Un cas on seria útil, per exemple, seria quan treballem amb una entitat amb moltes funcions i volem dividir-ho en múltiples fitxers.

```
struct User {
    name: string;
}

pub fn (User) get_name() string {
    self.name
}
```

Quan un mètode muta l'estat d'un objecte, aquesta funció ho ha d'indicar de forma explícita. Amb això es busca facilitar la vida al programador a l'hora de programar, llegir el codi o depurar, ja que només llegint la crida o la definició, sap que la funció té efectes secundaris.

Per tal d'indicar aquesta mutació, el programador, ha d'afegir la paraula "mut", a la definició de la funció, i cada cop que la crida, s'ha d'afegir un signe d'exclamació al final de l'identificador (ex: `function!(a, b)`).

El mateix passa amb les variables. Per defecte són immutables. Si es vol mutar una variable, s'ha d'afegir la paraula "mut".

```
struct User {
    name: string;
    pub mut fn set_name(name: string) string {
        self.name = name;
    }
}

fn modify_struct() {
    let mut user = User::new();
    user.set_name!("new name");
}
```

3.3 Bucles

Per tal d'executar repetidament instruccions es fan servir bucles. N'hi ha de diferents tipus. El més bàsic és el while:

```
let mut a = 0;
let mut i = 0;
while i < 10 {
    a += i;
    i += 1;
}
```

Un altra opció, és el bucle for:

```
let mut a = 0;
// No need for "mut" when declaring variables on for loop declarations, it's
    implied
for let i = 0; i < 10; i += 1 {
    a += i;
}
```

Per iterar arrays, tenim el bucle for in, que recorre l'array element per element.

```
for number in numbers {
    print(number);
}
```

Podem fer break d'un bucle en concret posant una etiqueta:

```
// Break outer loop from inner loop
outer: for number in numbers {
    for number in numbers {
        if number % 2 == 0 {
            break outer;
        }
    }
}
```

O fer un break normal:

```
for number in numbers {
    if number % 2 == 0 {
        break;
    }
}
```

3.4 Programació Funcional

Per tal de simplificar algunes operacions que es realitzen amb freqüència, hi haurà algunes funcions com ara map, reduce, fold, etc. que ens permetran escriure aquestes operacions amb moltes menys línies. Per tal de poder fer servir aquestes funcions, es farà ús de çlosures"o clausures.

```
fn functional_style(numbers: [i32]) {
    // Functional style with closures
    let numbers_plus_one = numbers.map(|x| -> x + 1);

    let sum = numbers.reduce(|prev, current| -> {
        if current % 2 == 0 {
            prev + current
        } else {
            prev
        }
    });

    if number.any(|x| -> x > 5) {
        print("There's at least one number greater than 5");
    }

    if number.all(|x| -> x > 5) {
        print("All numbers are greater than 5");
    }
}
```

Les clausures ens permeten tenir funcions de primer ordre, el que vol dir que les podem tractar com a qualsevol expressió, assignant-les a una variable, per exemple, o sent retornades per una altra funció.

```
fn two_plus_two() {
    let sum = |a: int, b: int| int -> {
        a + b
    };
    print(sum(2, 2)); // Output: 4
}
```

Un altre concepte inspirat en llenguatges funcionals, és l'operador pipe (|>). Ens permet millorar la lectura de casos tipus el següent:

```
validate_age(get_age(parse_data(person)))
```

Que passaria a ser així:

```
let is_valid =
    person
    |> parse_data
    |> get_age
    |> validate_age;
```

El que fa el pipe, és passar l'expressió de l'esquerra, com a paràmetre de la funció que hi ha a la dreta.

3.5 Gestió d'errors

Les funcions poden retornar el que se'n diu un Result. Un Result, és un enum que té dos valors possibles: `Ok(T) i Err(E)`, on T és l'element que retorna la funció, i E és l'error que retorna la funció.

En indicar el tipus de retorn com a Result, hem d'indicar el tipus del valor que retorna la funció, i el tipus de l'error: `Result<TipusOk, TipusErr>`

```
fn divide(a: i32, b: i32) Result<i32, string> {
    if b == 0 {
        return Err("Divide by zero");
    }
    Ok(a / b)
}
```

Quan cridem a la funció, hem de comprovar que no ens hagi retornat cap error:

```
fn call_divide() -> Result<i32, string> {
    let result = match divide(5, 0) {
        Ok(num) => num,
        Err(err) => return Err(err),
    }

    let result = try divide(5, 0) -> |err| {
        log("Error: ", err);
        return Err(err);
    };

    let result = try divide(5, 0);

    print("The result is: ", result);
    Ok(result)
}
```

Com que pot ser una mica empipador haver de comprovar l'error en cada crida, hi ha l'operador "?"que ens permet retornar immediatament en cas que el resultat sigui un error:

```
fn call_divide() -> Result<i32, string> {
    let result = divide(5, 0)?;
    print("The result is: ", result);
    Ok(result)
}
```

D'aquesta manera, com que la funció divide ens retorna un error, call_divide, retorna immediatament l'error i ja no s'executa el print. En cas que divide hagués retornat un Ok, l'execució hagués continuat. Cal destacar que la funció que fa servir l'operador "?", també ha de retornar un Result, ja que implica que pot retornar un error.

3.6 Referències

Si a un tipus l'hi afegim un `&` a davant, passarà a ser una referència. Les referències en permeten apuntar directament a una posició de memòria, i passar aquest apuntador, sense la necessitat de copiar l'objecte. Per defecte les referències, igual que les variables, són immutables, però es poden mutar, si afegim la paraula mut. En cas que la referència sigui mutable, es modificarà la posició de memòria a on apunta, per tant, totes les referències que llegeixin d'aquesta posició de memòria veuran el canvi. És a dir, venen a ser apuntadors ensucrats, similar a C++.

```
// References are immutable by default but we can make them mutable
fn references(numbers: &[i32], other_numbers: &mut [i32]) {
    for number in numbers {
        print(number);
    }

    for number in other_numbers {
        *number += 1; // This will modify the original vector
    }
}
```

4 Passos a seguir

Abans de començar a fer qualsevol cosa, hem d'escollir la tecnologia amb la qual volem implementar el compilador. Per aquest projecte, s'ha escollit a Rust com llenguatge de programació i a LLVM [1] com a toolchain per a la generació de codi del back-end. LLVM ens permetrà generar assemblador per pràcticament qualsevol arquitectura que existeixi avui en dia, sense haver de fer cap esforç addicional. Només ens haurem d'encarregar de generar la representació intermèdia (IR) de LLVM, i aquest s'encarregarà de generar l'assemblador a partir de l'IR. L'API de LLVM està només per C/C++, per tant, també farem servir una llibreria que es diu Inkwell [2], que ens ofereix una API per a Rust, interactuant amb l'API de C oficial.

Un cop escollides les tecnologies el següent pas és crear un entorn i una estructura de projecte base, on hi podem anar afegint totes les funcionalitats addicionals.

Per tal d'implementar les funcionalitats definides, seguirem un procés similar amb totes. Haurem de fer tres coses:

1. **Lexer**: El lexer, format per l'scanner i el tokenizer, s'encarrega de llegir el nostre codi, i convertir-lo a un seguit de tokens. Un token és un conjunt de caràcters en el codi font que té algun significat en el llenguatge, com ara una paraula clau del llenguatge (if, for, ...), o l'identificador d'una variable. Per cada funcionalitat, haurem d'adaptar el Lexer perquè ens reconegui aquests nous tokens que s'introdueixin al llenguatge.
2. **Parser**: El parser recorre la seqüència de tokens que ens retorna el lexer i l'hi busca el significat. El resultat del parser és un arbre sintàctic, que ens indicarà com s'han d'executar les instruccions. Per cada nova funcionalitat, haurem de modificar el parser perquè pugui interpretar els nous constructes.
3. **Generació de codi**: Per cada nova funcionalitat s'haurà de generar la IR de LLVM que després generarà l'assemblador que s'executarà.

5 Metodologia i planificació

Per tal d'organitzar el desenvolupament del projecte s'ha optat per una metodologia basada en sprints, on cada sprint té una duració d'una setmana. Donat que és un projecte gran, s'ha dividit en un conjunt d'èpiques, que representen un conjunt de funcionalitats. Cada una d'aquestes èpiques, conte un seguit de subtasques, i completar cada una d'aquestes èpiques requereix d'uns quants sprints. Les èpiques que s'han definit són les següents:

1. **Implementació mínima de l'especificació**
   Implementar tota la funcionalitat bàsica per poder fer programes senzills. Inclou operacions aritmètiques, funcions, control de flux bàsic (if statements i bucles while) i declaració de variables.
2. **Declaració de tipus**
   Implementar la funcionalitat necessària per definir tipus de dades propis (structs, interficies i type aliases).
3. **Collections**
   Implementació d'arrays estàtics, arrays dinàmics i tuples.
4. **Mecanisme de gestió d'errors**
   Implementar un mecanisme per tal de gestionar errors en temps d'execució.
5. **Syntactic sugar**

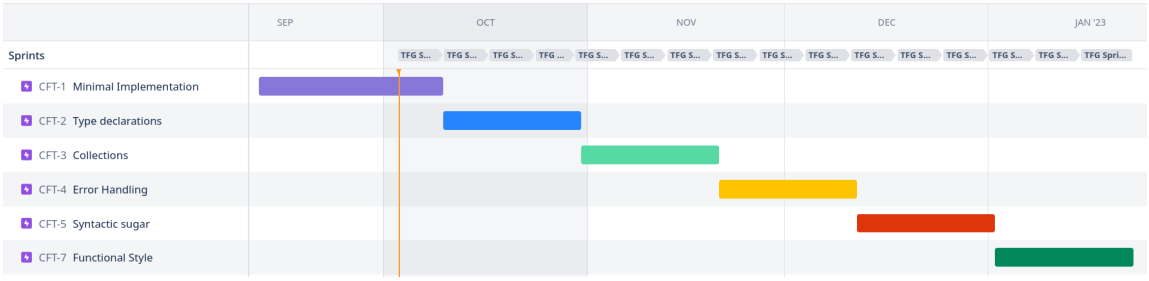

Figura 1: Roadmap on es mostren les diferents èpiques i l'espai de temps on s'han de desenvolupar (fer zoom per veure-ho amb més resolució)

Implementació de "Syntactic sugar", per tal de simplificar operacions que s'utilitzen freqüentment. Inclou bucles for in, match (similar a un switch), list comprehension, etc.

6. **Suport per programació d'estil funcional**
   Afegir suport per programació d'estil funcional, integrant funcions com ara map, reduce, fold, etc.

Per tal de gestionar totes aquestes èpiques, les seves subtasques i els sprints, es farà servir **Jira**, un programa de gestió de projectes que ho facilita molt.

6 Fonts d'informació

Donat que crear un llenguatge de programació és un projecte de certa magnitud, les fonts d'informació no són poques. Principalment, podem dividir les fonts en dos grups: fonts per dissenyar el llenguatge i fonts per implementar el llenguatge.

6.1 Dissenyar del llenguatge

La gran majoria de funcionalitats descrites en aquest document han sigut inspirades d'altres llenguatges ja existents. S'han agafat les següents idees de diferents llenguatges i paradigmes:

1. **List comprehension**: Python
2. **Iteradors (map, reduce, ...)**: Llenguatges funcionals, Javascript, Rust, C#...
3. **Result i Option**: Rust, Scala, Haskell, Swift
4. **Immutabilitat**: Llenguatges funcionals, Scala, Rust, Swift
5. **Structs**: C, Go, Rust
6. **Pipes**: Elixir, F#
7. **Referències**: C++, Rust, Go
8. **Sintaxi get, set**: C#
9. **Expression-oriented**: Scala, Rust, Kotlin

6.2 Implementar el llenguatge

A l'hora d'implementar el compilador, hi ha dues fonts principals d'on treure'm informació detallada i força pràctica de com fer-ho.

El primer de tots és un llibre anomenat *Crafting Interpreters* [6], el qual explica de forma detallada i amb exemples de codi, com escriure des de zero un llenguatge de programació creat per l'autor. El llibre es divideix en dues parts, una en Java, en el qual crear un intèrpret i l'altre amb C, en el qual crea un compilador amb màquina virtual. En el nostre cas, no en farem ni un ni l'altre, ja que serà un llenguatge que compilarà a codi màquina utilitzant LLVM. Però podem reutilitzar molts conceptes dels exposats, com ara com escriure un scanner o un parser, i patrons de disseny que són utils de cara al desenvolupament d'un compilador. Un altre llibre similar és *Writing An Interpreter In Go* [7] i la seva continuació *Writing A Compiler In Go* [8], tot i que ens basarem principalment en *Crafting Interpreters*.

El segon recurs va enfocat a la part de LLVM i generació de codi. Es tracta del tutorial oficial de LLVM, el qual detalla com fer un llenguatge senzill amb C++ i LLVM. Igual que amb ell llibre de *Crafting Interpreters*, també es descriu amb exemples de codi com crear un compilador des de zero.

Donat que el nostre llenguatge serà implementat amb Rust, també ens serà útil el llibre *The Rust Programming Language* [4], conegut coloquialment com *The Rust Book*. És el llibre oficial del llenguatge on s'explica amb detall com funciona el llenguatge i el perquè d'algunes decisions en el seu disseny.

Finalment, també farem ús de la documentació de *Inkwell* [2], la llibreria que ens permet fer servir LLVM des de Rust, així com la mateixa documentació de LLVM. Respecte a LLVM, també resultarà útil una pàgina web que s'anomena *Mapping High Level Constructs to LLVM IR* [9], la qual, com bé diu el seu nom, mostra exemples de com alguns conceptes d'alt nivell (en C/C++) es tradueixen a la representació intermèdia (IR) de LLVM.

7 Gramàtica

```
<lowercase>     ::= "a" | ... | "z"
<uppercase>     ::= "A" | ... | "Z"
<letter>        ::= <lowercase> | <uppercase>
<digit>         ::= "0" | ... | "9"

<program>       ::= <fn>
                    | <method>
                    | <const>
                    | <global_var>
                    | <imports>
                    | <interface>

<fn>            ::= <prototype> <block>
<params>        ::= "(" { <param> { , <param> } } ")"
<param>         ::= <id>: <type>
<prototype>     ::= fn <id> "(" <params> ")" <type>

<method>        ::= [pub] [mut] fn "(" <custom_type> ")" <id> "(" <params> ")" <type>
                    <block>

<block>         ::= "{" { <stmt> } [ <expr> ] "}"

<type>          ::= i32 | i64 | f32 | f64 | bool | string | <custom_type> | <array>
<custom_type>   ::= <uppercase> { ( <lowercase> | <digit> ) }
<array>         ::= "[" <type> "]"
```

```
<interface>     ::= interface <id> "{" { <prototype> ; } "}"

<struct>        ::= struct <id> { <implements> } "{"
                        { <member_var> }
                        { [pub] [static] <prototype> <block> }
                    "}"

<member_var>    ::= <id> : type [ "{" [get [, set]] | [set] "}" ];

<implements>    ::= : <id> { , <id> }

<stmt>          ::= <var> | <fn> | <for> | <while> | <for_in> | <expr_stmt>
<var>           ::= let <id> : <type> = <expr>;
<id>            ::= letter { letter | digit | "_" }

<const>         ::= const <uppercase> { <uppercase> | <digit> } = <number>
                    | <string>
                    | true
                    | false ;
<global_var>    ::= let <uppercase> { <uppercase> | <digit> } = <expr> ;

<imports>       ::= { import <id>; }

<expr>          ::= <eq>
<eq>            ::= <comparison> { ( != | == ) <comparison> }
<comparison>    ::= <term> { ( > | >= | < | <= ) <term> }
<term>          ::= <factor> { ( - | + ) <factor> }
<factor>        ::= <unary> { ( / | * ) <unary> }
<unary>         ::= ( ! | - ) <unary> | <primary>

<primary>       ::= <number>
                    | <string>
                    | true
                    | false
                    | "(" <expr> ")"
                    | <id> ( <assign> |  <fn_call> | <pipe> )
                    | <if_expr>
                    | <clojure>

<assign>        ::= "=" <expr>
<fn_call>       ::= "(" { (<id> | <expr>) { , (<id> | <expr>) } } ")"
<pipe>          ::= |> <id> { |> <id> }

<if_expr>       ::= if <expr> <block> [ else ( <if_expr> | <block> ) ]
<clojure>       ::= | <params> | -> <block>

<for>           ::= [<break_tag>] for let <id> = <expr> ; <comparison> ; <expr>
<while>         ::= while <expr> <block>
<for_in>        ::= [<break_tag>] for <id> in <id> <block>
<expr_stmt>     ::= <expr> ;

<break_tag>     ::= <id>:
```

Referències

[1] The LLVM Compiler Infrastructure, https://llvm.org

[2] Inkwell, https://github.com/TheDan64/inkwell

[3] The Rust Programming Language https://www.rust-lang.org

[4] The Rust Programming Language https://doc.rust-lang.org/book/

[5] LLVM Kaleidoscope Tutorial: Implementing a Language with LLVM, https://llvm.org/docs/tutorial

[6] Crafting Interpreters, Bob Nystrom, http://www.craftinginterpreters.com

[7] Writing An Interpreter In Go, Thorsten Ball, https://interpreterbook.com

[8] Writing A Compiler In Go, Thorsten Ball, https://compilerbook.com

[9] Mapping High Level Constructs to LLVM IR, https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/README.html

[10] Global variables, Haskell Wiki, https://wiki.haskell.org/Global_variables