Universitat Autònoma de Barcelona

Treball de Fi de Grau

Informe de progrés II

Disseny i implementació d'un llenguatge de programació amb LLVM

Autor
Josep Maria Domingo Catafal

Tutor
Javier Sánchez Pujadas

18 de desembre de 2022

Índex

1 Seguiment de la planificació

A l'inici del projecte, vam identificar diversos objectius que volíem assolir i vam definir una sèrie d'èpiques. A continuació repassarem l'estat d'aquestes èpiques: quines s'han completat, quines no, i si hi ha hagut algun canvi respecte a l'informe anterior.

1.1 Implementació mínima de l'especificació

***Completada**: 13/09/2022 - 16/10/2022* (5 esprints)

Aquesta tasca ja va ser completada en l'informe anterior.

1.2 Declaració de tipus

***Completada**: 17/10/2022 - 6/11/2022* (3 esprints)

Aquesta tasca ja va ser completada en l'informe anterior.

1.3 Arrays

***Completada**: 7/11/2022 - 11/12/2022* (3 esprints) Els arrays són un tipus d'estructura de dades que ens permet emmagatzemar múltiples valors del mateix tipus en un sol bloc contigu de memòria.

Per crear un array en LLVM, fem servir la instrucció alloca, que assigna memòria en la pila (és la mateixa que fem servir quan creem variables). La instrucció alloca pren un tipus i un nombre d'elements com a arguments i retorna un apuntador al començament de l'array.

Per exemple aquest seria el codi LLVM per un array de 10 elements de tipus enter de 64 bits:

```
1 %array = alloca [10 x i64], align 8
```

El que diu el codi anterior és simplement que es reservin 10 slots de 64 bits (8 bytes, per això posa align 8). Per generar aquest codi amb la API de LLVM, ho hem de fer en dos passos: primer creem un ArrayType indicant el tipus dels elements i la mida, després generem l'alloca passant el tipus array que em creat. D'aquesta manera estem indicant que ens generi un alloca del tipus array.

Per accedir a un element en l'array, utilitzem la instrucció getelementptr, que permet obtenir l'adreça d'un element en un array o struct. Rep com a entrada l'apuntador al començament de l'array i un índex (al que volem accedir), i retorna el punter a l'element que volem. És similar a la instrucció LEA [7] de l'assemblador x86.

Seguint l'exemple de codi anterior, el codi que hauríem de generar és el següent:

```
1 %element_ptr = getelementptr inbounds [10 x i64], [10 x i64]* %array, i64 0
```

Els paràmetres que especifiquem són el tipus d'array (10 enters de 64 bits), el punter a l'inici de l'array, i finalment l'índex al qual volem accedir (0, és a dir el primer element).

Cal destacar la paraula clau **inbounds** que fem servir després de getelementptr. No és necessari posar-la, també ens retornarà l'adreça que busquem. Però si la posem, estem indicant a LLVM que ja hem comprovat nosaltres que l'índex que estem indicant és vàlid, és a dir, que està inbounds. Això permet a LLVM optimitzar de manera més eficient el codi.

Després fem un load del contingut en el punter que ens ha retornat, i ja tenim el valor:

```
%val = load i64, i64* %element_ptr
```

Per guardar-hi un valor fem un store:

```
store i64 %newval, i64* %element_ptr
```

Així doncs a partir d'ara podem fer servir arrays en el llenguatge de la forma següent:

```
struct SomeStruct {
    arr: [i64; 3]
}

fn main() {
    let a = [1, 2, 3] i64;
    printf("[%d, %d, %d]", a[0], a[1], a[2]);
}
```

1.3.1 Strings

Dins d'aqueta èpica s'han desenvolupat els strings. La seva implementació és relativament senzilla un cop implementats els arrays, ja que són simplement arrays d'enters de 8 bits. S'ha hagut d'ampliar el parser perquè identifiqui les strings i després es tracten com a arrays.

1.3.2 printf

Amb les strings implementades, també vaig aprofitar per implementar el printf, ja que la funció de print que tenia fins ara era bastant rudimentària i només podia printar enters. Ara amb els strings ja es podia fer un print més decent. Per tal d'implementar el printf el que he fet és enllaçar el compilador amb la llibreria estàndard de c (libc), i d'aquesta manera en generar el codi es pot fer una crida a la funció printf de C.

1.4 Mecanisme de gestió d'errors

Cancel·lada

Aquesta èpica ha sigut cancel·lada principalment perquè començaven a haver-hi bastants bugs al programa i vaig decidir que era més important solucionar-los i tenir una base sòlida que seguir afegint funcionalitats. L'he cancel·lat en comptes d'ajornar-la, ja que prefereixo fer l'èpica de Syntactic Sugar.

1.5 Syntactic sugar

***Planificada**: 19/12/2022 - 15/01/2023* (4 esprints)

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.

1.6 Suport per programació d'estil funcional

Cancel·lada

Pel mateix motiu que l'èpica de gestió d'errors, he decidit cancel·lar aquesta èpica. Les tres setmanes en les quals s'havia de fer aquesta èpica seran les tres últimes setmanes i m'estimo més dedicar aquests temps a millorar el que ja tingui fent i estabilitzar-ho tot.

1.7 Deute tècnic

***En progrés**: Termini indefinit* (∞ esprints)

Com hem comentat, hi ha diversos bugs que han anat apareixent i per tant, el temps que estava destinat a l'èpica de gestió d'errors s'ha destinat a aquesta èpica. A continuació veurem alguns dels bugs que s'han solucionat (els més greus o que comporten més feina d'arreglar).

1.7.1 Taula de símbols

La taula de símbols que tenia fins ara no funcionava del tot bé i estava donant problemes. Així que en vaig implementar una de nova, aquest cop fent-ho bé.

Vaig crear una classe amb diferents funcions per tal de poder crear o eliminar un context nou quan fes falta. Consisteix simplement d'un vector de HashMaps, que actua com a pila. Cada HashMap és un context diferent. La classe conté les següents funcions:

- **`new_context`**: Afegeix un nou HashMap a la pila.
- **`pop_context`**: Treu l'últim HashMap de la pila.
- **`insert`**: Afegeix una nova variable a la taula de dalt de tot de la pila.
- **`get`**: Busca una variable en les taules que hi ha a la pila. En cas de no trobar-la, retorna un error (UndeclaredVariableOrOutOfScope)

Amb aquestes funcions en tenim prou per tenir una taula de símbols funcional. Es crea un context nou cada vegada que es crea un bloc (s'obre amb '{') i s'elimina quan s'acaba el bloc (es tanca amb '}').

1.7.2 Els blocs sempre havien de retornar un valor

Una de les característiques clau del llenguatge és que és un llenguatge basat en expressions, el que significa que la majoria de les declaracions del llenguatge retornen un valor. Un exemple són els blocs de codi (conjunt de sentències dins de {}), els quals si l'última sentència del bloc no porta punt i coma, aquesta serà una expressió i per tant el seu valor serà retornat. Per exemple el següent bloc retorna el resultat de 2 + 2:

```
1 let four = { 2 + 2 };
```

El problema és que tal com s'havien implementat els blocs, sempre havien de retornar un valor, encara que no volguessis retornar res. Aleshores, en casos com un bucle while, per exemple, que està format per un bloc, no té massa sentit voler retornar un valor. Per tant era una cosa que havia de ser solucionada.

1.7.3 Error en cridar funcions que no retornaven valor

Com hem comentat abans, el llenguatge està orientat a expressions i per tant pràcticament tot pot retornar un valor. Però a vegades no volem que en retorni cap. Igual que amb els blocs, hi havia un problema al cridar funcions que no retornaven cap valor (void), ja que tal com estava estructurat el compilador, esperava que una expressió **sempre** retornes un valor. I com que una crida a una funció és una expressió, quan cridàvem una funció que no retorna valor fallava. Per solucionar

aquest problema, el que s'ha fet és que les funcions void, també retornin un valor, l'únic que és un valor buit, que no té res, ni mida ni tipus ni valor, com una tupla buida.

1.7.4 Conflicte entre instaciació d'structs i blocs condicionals

Per tal com està implementat el parser hi havia un conflicte entre la instaciació d'structs (`let point = Point { x: 10, y: 20, };`), i els blocs condicionals (if i while). El problema venia perquè els blocs condicionals no porten parentesis, i per tant quan fas per exemple `if a < b { ... }`, el parser quan arriba a la 'b' i comprova que el següent token és un '{', es pensa que és la instaciació d'un struct. Això és perquè el parser no sap distingir entre identificadors (si és un tipus o una variable). Per solucionar aquest problema, he canviat la sintaxi de la instaciació d'structs, i ara s'instancien de la següent forma: `let point = Point!{ x: 10, y: 10, }`. Afegint el signe d'exclamació, el parser ja es capaç de distingir entre les dues casuístiques.

2 Control de qualitat

Els tests automatitzats són una part important del desenvolupament de programari, ja que ajuda a assegurar la qualitat i la fiabilitat del codi. Ens permet detectar errors que poder no hauríem detectat provant-ho a mà. A més a més, a mesura que el projecte es va fent gran, és inviable provar-ho tot de forma manual.

Hi ha diferents tipus de tests automatitzats que es poden implementar. Els tests unitaris són un tipus de test que se centren en unitats individuals de codi, com ara funcions o mètodes. En el meu cas, he implementat tests unitaris pel parser, per així verificar que l'arbre sintàctic que es genera és correcte.

Tot i això, és poc pràctic implementar aquests tests, ja que has de tenir l'arbre correcte, "hardcoded"en el test per comprovar que el resultat sigui aquest, i en arbres més grans és poc viable. Per aquest motiu la majoria de tests que he implementat són tests d'integració.

Els tests d'integració són un altre tipus de test automatitzat que se centren en com diferents unitats de codi treballen juntes. La forma en la qual els he implementat és escrivint programes en el llenguatge desenvolupat, compilant-los , executant-los i comprovant que el resultat és l'esperat. Per exemple pels arrays hi ha un petit programa que crea un array, el passa com a paràmetre a una funció i fa print dels seus elements. Aleshores el test comprovaria que els valors printats siguin els de l'array. I després faríem un altre test que accedeix a un índex de l'array que no existeix i comprovaríem que el programa falla amb un error 'index out of bounds'.

Aquests tests són importants perquè ajuden a assegurar que tots els components del llenguatge funcionen correctament entre ells.

2.1 Integració amb el control de versions del codi font

Com vam comentar a l'informe anterior per al control de versions del codi font s'està utilitzant Git i GitHub (https://github.com/josepmdc/craft). Respecte a l'anterior informe ha canviat una cosa, i és que com que ara disposem de tests automatitzats, podem fer ús d'una funcionalitat de GitHub que s'anomena GitHub Actions. El que ens permeten fer les Actions, és executar en els servidors de GitHub alguna tasca, depenent d'alguns events com ara quan es fa un commit a master. Per exemple, jo he configurat que cada vegada que s'obri una pull request i cada vegada que es faci un commit a master, que s'executin tots els tests del projecte. D'aquesta manera cada vegada que es vulguin incorporar canvis a master, t'assegures que no s'ha trencat res i tot funciona amb normalitat. A part dels tests, he afegit 3 tasques més que s'executen en els dos casos que hem comentat anteriorment. Així que tenim les 4 tasques següents:

- Executa tots els tests, tant d'integració com unitaris.
- Comprova que el projecte compili correctament.
- Comprova que el codi estigui formatat seguint la guia d'estil.
- Executa clippy, un linter que ve incorporat amb el llenguatge de Rust i que et permet millorar el codi a través de suggeriments i t'informa de possibles bugs.

En cas que alguna d'aquestes quatre tasques falli o retorni algun warning, no es podrà fer merge fins que es resolguin els problemes.

Per configurar les Actions és força senzill, ja que es fa a través d'uns arxius de configuració en format YAML, en el qual l'hi especifiques el sistema operatiu que vols fer servir així com totes les tasques que vols que executi. Hi ha algunes Actions que ha desenvolupat la comunitat i que tu simplement les referències a la teva configuració indicant que les executi. Per exemple per la tasca d'executar els tests la configuració seria així:

```
on:
  pull_request:
  push:
    branches: master

name: Continuous integration

jobs:
  test:
    name: Test Suite
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions-rs/toolchain@v1
        with:
          profile: minimal
          toolchain: stable
          override: true
      - uses: actions-rs/cargo@v1
        with:
          command: test
```

El que ens indica la configuració anterior és que quan s'obri una pull request o es faci un push a master, s'executin els jobs que hi ha indicats. El job de test indica que s'executarà en l'última versió d'Ubuntu i que té tres passos (steps). El primer fa servir una action que es diu checkout, que el que fa és descarregar-se el teu repositori en el servidor. El segon pas fa servir una action que es diu toolchain, que instal·la totes els eines necessaries per compilar Rust. I finalment l'ultim fa servir l'action de cargo. Cargo és l'eina de Rust que es fa servir per compilar (`cargo build`), executar els tests (`cargo test`), o descarregar llibreries, entre d'altres. Amb aquesta action l'hi estem indicant que executi `cargo test`.

Per la resta de tasques que em comentat com ara la de clippy, simplement hem d'afegir un altre job a l'arxiu de configuració, de manera similar a com hem fet amb el de test.

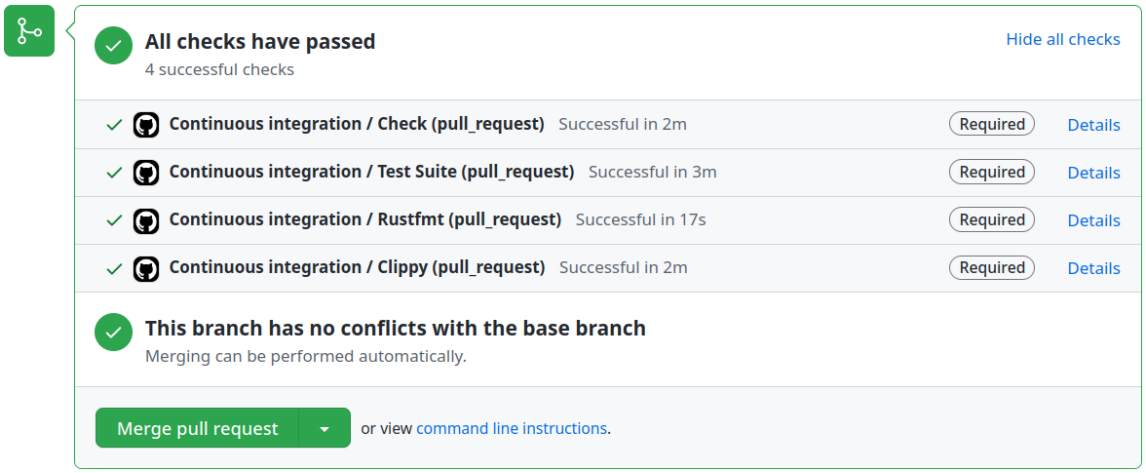


Figura 1: Pull Request on els checks han passat tots i que per tant es pot fer merge

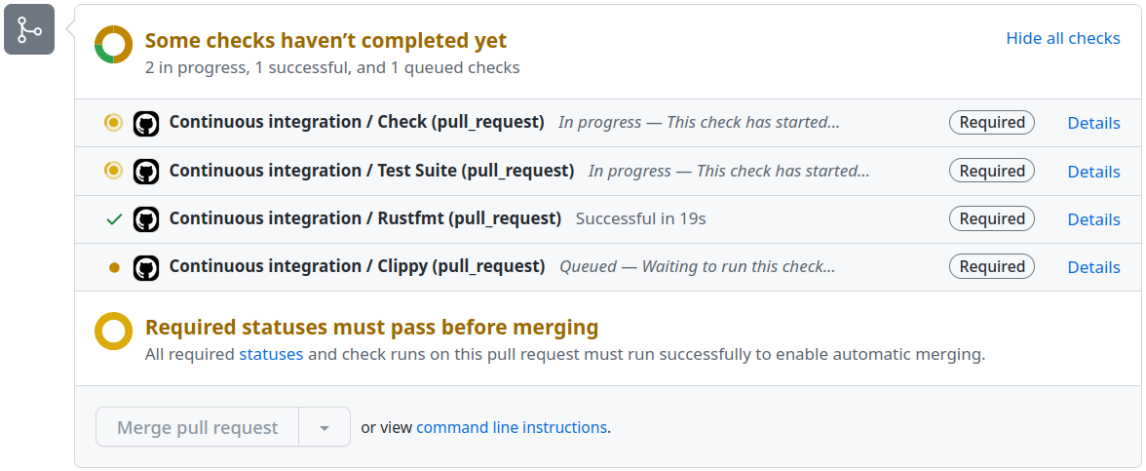


Figura 2: Pull Request on els checks encara no s'han executat i que per tant no es pot fer merge

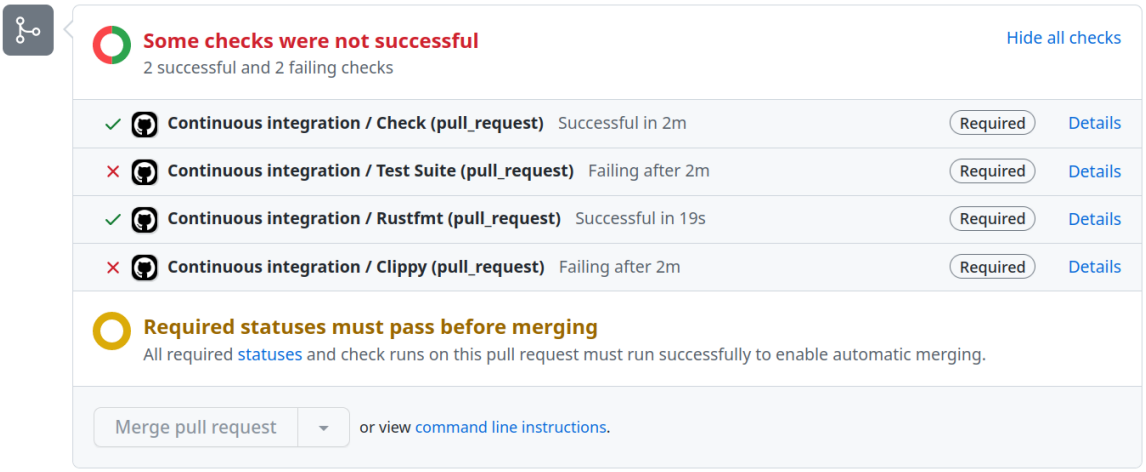


Figura 3: Pull Request on els checks han fallat i que per tant no es pot fer merge

3 Exemples de codi

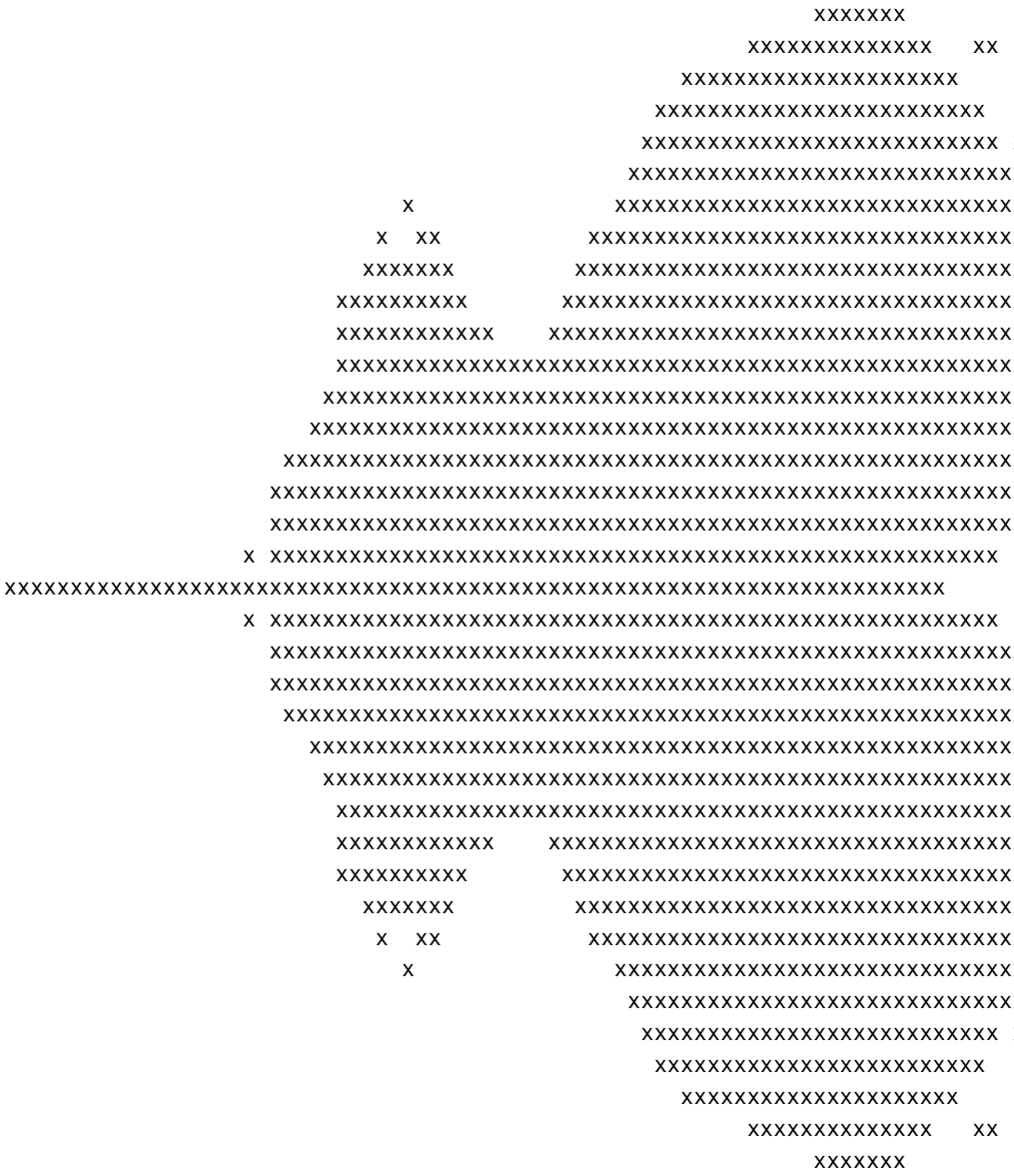
A continuació veurem alguns programes que es poden fer amb el llenguatge tal com està ara.

3.1 Mandelbrot set

Programa que mostra una aproximació del set de Mandelbrot en format ASCII:

```
fn mandelbrot(a: f64, b: f64) f64 {
    let za = 0.0;
    let zb = 0.0;

    let i = 0;

    while i < 50 {
        za = (za*za - zb*zb) + a;
        zb = (za*zb + za*zb) + b;
        i = i+1;
    }

    za*za + zb*zb
}

fn main() {
    let xstart = -2.0;
    let xend = 0.5;
    let ystart = 1.0;
    let yend = -1.0;

    let xstep = 0.0315;
    let ystep = -0.05;

    let x = xstart;
    let y = ystart;

    while y > yend {
        x = xstart;
        while x < xend {
            if mandelbrot(x, y) < 4.0 {
                printf("x");
                0
            } else {
                printf(" ");
                0
            }
            x = x + xstep;
        }
        printf("\n");
        y = y + ystep;
    };
}
```

La sortida és la següent:



3.2 Factorial recursiu

```
fn factorial(n: i64) i64 {
    if n == 1 {
        n
    } else {
        n * factorial(n-1)
    }
}

fn main() {
    printf("%d", factorial(10));
}
```

Sorrtida: 3628800

3.3 Fibonacci recursiu

```
fn fib(n: i64) i64 {
    if n <= 1 {
        n
    } else {
        fib(n - 1) + fib(n - 2)
    }
}

fn main() {
    printf("%d", fib(20));
}
```

Sortida: 6765

3.4 Fibonacci iteratiu

```
fn fib(n: i64) i64 {
    let curr = 1;
    let prev = 0;
    let prev_prev = 0;

    let i = 1;
    while i < n {
        prev_prev = prev;
        prev = curr;
        curr = prev_prev + prev;
        i = i + 1;
    }

    curr
}

```

```
fn main() {
    printf("%d", fib(20));
}
```

Sortida: 6765

3.5 Nested structs

```
struct InnerInnerStruct {
    field: i64
}

struct InnerStruct {
    inner_inner: InnerInnerStruct
}

struct OuterStruct {
    inner: InnerStruct
}

fn extract_field(outer: OuterStruct) i64 {
    outer.inner.inner_inner.field
}

fn main() {
    let outer = OuterStruct!{
        inner: InnerStruct!{
            inner_inner: InnerInnerStruct!{
                field: 42,
            },
        },
    };

    printf("%d", extract_field(outer));
}
```

Resultat: 42

3.6 Calculadora de nombres complexos

```
struct ComplexNumber {
    real:      f64
    imaginary: f64
}

fn add(x: ComplexNumber, y: ComplexNumber) ComplexNumber {
    ComplexNumber!{
        real: x.real + y.real,
        imaginary: x.imaginary + y.imaginary,
    }
```

```
}

fn multiply(x: ComplexNumber, y: ComplexNumber) ComplexNumber {
    ComplexNumber!{
        real: x.real * y.real - x.imaginary * y.imaginary,
        imaginary: x.real * y.imaginary + x.imaginary * y.real,
    }
}

fn main() {
    let x = ComplexNumber!{
        real: 1.0,
        imaginary: 2.0,
    };
    let y = ComplexNumber!{
        real: 3.0,
        imaginary: 4.0,
    };

    let z = add(x, y);
    printf("The sum of x and y is %f + %fi\n", z.real, z.imaginary);

    z = multiply(x, y);
    printf("The product of x and y is %f + %fi\n", z.real, z.imaginary);
}
```

Resultat:

The sum of x and y is 4.000000 + 6.000000i

The product of x and y is -5.000000 + 10.000000i

Bibliografia

[1] *LLVM Kaleidoscope Tutorial: Implementing a Language with LLVM*. Accedit el 25 d'octubre, 2022, des de https://llvm.org/docs/tutorial

[2] Rathi, M. *A complete guide to LLVM for programming language creators*. Mukuls Blogs. Accedit el 2 de Novembre, 2022, des de https://mukulrathi.com/create-your-own-programming-language/llvm-ir-cpp-api-tutorial

[3] Nystrom, R. *Crafting Interpreters*. Publicat Juliol, 2021. ISBN 0990582930.

[4] *Mapping High Level Constructs to LLVM IR*. Accedit el 20 d'octubre, 2022 des de https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/README.html

[5] *Inkwell Documentation*. Accedit el 4 d'octubre, 2022 des de https://thedan64.github.io/inkwell/inkwell/index.html

[6] Ball, T. *Writing A Compiler In Go*. Publicat Agost, 2018. ISBN 398201610X.

[7] LEA, x86 Instruction Set Reference. Accedit el 18 de desembre, 2022 https://c9x.me/x86/html/file_module_x86_id_153.html