Universitat Autònoma de Barcelona

Treball de Fi de Grau

Informe de progrés I

Disseny i implementació d'un llenguatge de programació amb LLVM

Autor
Josep Maria Domingo Catafal

Tutor
Javier Sánchez Pujadas

13 de novembre de 2022

Índex

1 Metodologia

Per tal d'organitzar el projecte es va optar per una metodologia Agile/Scrum però adaptada per a una sola persona. Principalment consisteix a crear esprints d'una setmana. Un esprint és simplement un bloc de temps (una setmana en el nostre cas) en el que s'han de completar un seguit de tasques. Les tasques han de ser petites per tal d'oferir la màxima flexibilitat i s'agrupen en èpiques. Les èpiques ens indiquen una funcionalitat que ha de tenir el llenguatge, i totes les tasques que la conformen són les tasques necessàries per poder desenvolupar aquesta funcionalitat.

1.1 Eines

Tot i ser un projecte d'una envergadura no molt gran, i format només per una persona, és difícil organitzar-se sense fer ús de cap eina. És per això que s'han utilitzat principalment dues eines: una per gestionar les tasques i els terminis, i un altre per gestionar el control de versions del codi font.

1.1.1 Gestió de les tasques

Per crear i gestionar les tasques s'ha estat fent servir Jira, com vam comentar en el primer informe. Es va triar aquesta eina, ja que l'he fet servir prèviament en projectes professionals i ja la tenia per mà. Permet crear tasques i assignar-les a esprints, a més de crear un roadmap amb les èpiques, mostrant de manera gràfica quan inicien i quan han d'acabar. També té moltes funcions de mètriques, tot i que no es faran servir en aquest projecte.

En el temps que ha transcorregut des de l'inici, està funcionant realment bé, i ajuda bastant en l'organització.

1.1.2 Control de versions del codi font

L'altra eina important és el control de versions. És imprescindible, des del meu punt de vista, si has de col·laborar amb més gent, però també, en projectes com aquest que són individuals. Et permet estar al cas de tots els canvis que has fet en el projecte, i en casos que trobis un bug, és molt més fàcil tirar enrere en el temps per trobar on va aparèixer per primer cop.

De totes les eines de control de versions, s'està fent servir Git, ja que és l'estàndard en el desenvolupament del software i és amb la que estic més familiaritzat. També s'està fent servir GitHub per allotjar el repositori, ja que ofereix moltes funcionalitats i a més és on es troben la gran majoria de projectes de codi obert i disposa d'una gran comunitat. El repositori es pot trobar al següent enllaç: https://github.com/josepmdc/craft

1.2 Control de qualitat

Actualment el projecte disposa de tests automatitzats per al parser. Però ara que el projecte està ja una mica més avançat la intenció és crear tests que compilin un programa sencer i comprovin que el resultat obtingut sigui el correcte. Això facilitarà molt més la detecció d'errors de programació i ajudarà a fer refactor sense causar nous errors.

2 Seguiment de la planificació

Si recordem, a l'inici del treball vam definir quines eren les èpiques que conformarien el projecte i en quin moment es desenvoluparien. A hores d'ara aquestes èpiques segueixen més o menys com s'havien plantejat al seu moment, però amb algun canvi pel que fa al moment en el qual s'han desenvolupat (alguna ha durat més de l'esperat). També s'ha creat una nova èpica per albergar

tots els bugs i tasques de manteniment i millora del projecte que van apareixent a mesura que va avançant.

A continuació mirarem aquestes èpiques, una per una, per veure quin és el seu estat, i, per les que ja s'han completat, donar una mica més de detall de què han aportat al projecte i com s'han desenvolupat a un nivell més tècnic.

2.1 Implementació mínima de l'especificació

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

Aquesta èpica és la primera que es va completar i de la qual neixen la resta. Consistia en la creació d'una base sòlida del llenguatge que permetés crear programes simples amb operacions numèriques i control de flux.

Aquesta èpica és la més llarga de totes, ja que requereix molta feina inicial. Va acabar una setmana més tard del previst, però tot i això no ha tingut gaire afectació al conjunt del projecte.

El primer que es va fer va ser crear el projecte i un conjunt de funcions d'utilitat. El projecte es divideix principalment en 4 mòduls:

2.1.1 Mòdul principal

S'encarrega de gestionar l'entrada de l'usuari (per exemple llegir l'arxiu a compilar) i crida a la resta de mòduls, passant la sortida d'un a l'entrada del següent. Bàsicament és el conductor de la resta de mòduls.

2.1.2 Lexer

Aquest mòdul s'encarrega de llegir la seqüència de caràcters del codi a compilar i transformar-lo en un conjunt de tokens. Principalment ens permet identificar els símbols del llenguatge i alertar a l'usuari en cas que n'estigui fent servir algun que no és vàlid. La manera com funciona és la següent:

Mentre que no hàgim arribat al final de la seqüència de caràcters, processem el següent caràcter en la seqüència. Si és un parèntesi, per exemple, com que per si sol forma una seqüència vàlida, i no pot generar-ne cap altre, creem un nou token de tipus parèntesis i l'afegim al conjunt de sortida. Si per contra ens trobéssim un caràcter tipus '>', no podem crear un token encara, ja que depenent del següent caràcter, podria ser un '>=' o bé un '>'. Per tant, escanegem també el següent caràcter i en funció d'això creem el token.

Un cas particular que poder cal destacar, és quan ens trobem amb un caràcter alfanumèric, ja que hem de tenir en compte més coses:

- Si és un dígit, vol dir que estem processant un nombre, i per tant hem d'anar avançant mentre ens anem trobant dígits. Mentre processem els dígits pot ser que ens trobem una punt, el que ens indicarà que és un nombre real i per tant generarem un token de float. Si, no té punt en canvi, generem un token d'int.
- Si en canvi és un caràcter alfabètic, haurem d'anar avançant mentre que ens trobem caràcters alfanumèrics (no es permeten dígits a l'inici d'un identificador però si a la meitat) o guions baixos. Un cop troben un altre tipus de caràcter, vol dir que ja hem acabat i podem crear el token. Ara bé, aquest token pot ser un identificador o una paraula reservada del llenguatge, per tant, hem de comprovar si és una paraula reservada i crear un token o l'altre en funció d'això.

En general el lexer, un cop fet no s'haurà de modificar gaire, només caldrà modificar-lo si hem d'afegir alguna paraula reservada nova el llenguatge o algun símbol nou i que per tant no reconeix encara.

2.1.3 Parser

El lexer ens ha permès identificar els símbols del programa, però no ens permet identificar si l'ordre és correcte o si segueix les normes del llenguatge (és a dir si és gramaticalment correcte). Aquesta és la funció del parser, el qual a partir de la seqüència de tokens, n'extraurà el significat i generarà un arbre sintàctic, que ens indica com haurem d'executar les instruccions.

La funció del parser és fer complir la gramàtica del llenguatge. S'ha implementat en la forma d'un *recursive descent parser* el qual és pràcticament una traducció literal de la gramàtica en forma de codi.

El que fa el parser és començar pel primer token de la seqüència que ens ha generat el lexer. A partir d'allà va descendint de forma recursiva per totes les produccions de la gramàtica. Per exemple, si el primer token és 'fn', el parser començarà a explorar la producció que genera una funció. Mirarà que seguidament de 'fn' hi hagi un identificar. Si no hi és, es retorna un error a l'usuari. Si hi és, aleshores segueix descendint, mira que s'especifiquin correctament els paràmetres, si hi són, el tipus del valor de retorn i finalment el cos de la funció. Diem que és recursiu perquè pot ser que mentre estem generant, per exemple una expressió, pot ser que aquesta estigui formada per altres expressions, i per tant, tornarà a cridar de forma recursiva la producció d'expressió.

Posem un exemple més il·lustratiu, amb una petita gramàtica, i com aquesta s'implementaria en el parser. Aquesta seria la gramàtica per generar el prototipus d'una funció (excloem el tipus de retorn de la funció, que aniria després dels paràmetres, per simplificar):

```
<prototype>  ::= fn <id> "(" <params> ")"
<id>         ::= letter { letter | digit | "_" }
<params>     ::= "(" { <param> { , <param> } } ")"
<param>      ::= <id>: <type>
```

La implementació d'aquesta gramàtica seria de la següent manera (en un pseudocodi aproximat a Rust):

```
1  fn parse_prototype() -> (Prototype, Err) {
2      // we expect to find the fn keyword, else it's an error
3      match current_token().kind {
4          // The advance function moves to the next token
5          TokenKind::Fn => advance(),
6          _ => return Err("Expected fn keyword"),
7      };
8
9      // we expect to find the function name, else it's an error
10     let name = match current_token().kind {
11         TokenKind::Identifier => current_token().lexeme,
12         _ => return Err("Expected an identifier"),
13     };
14
15     advance();
16
17     // we call the params rule
18     let params = parse_params();
19
```

```
    // we are done, we return a struct with the info of the prototype
    return Prototype {
        name,
        params,
    };
}
```

Nota: Els comentaris en els fragments de codi estan en anglès, ja que LaTeXdonava problemes amb els accents.

És així de simple, iterem els tokens un per un i comprovem que el token que estem revisant sigui el token que esperem trobar, en funció del que ens diu la gramàtica. I quan el que esperen és una altra producció, simplement cridem a la funció que implementa aquesta producció. En el nostre exemple la funció `parse_params`:

```
fn parse_params() -> (Vec<Variable>, Err) {
    // params start with an opening paren, else it's an error
    match current_token().kind {
        TokenKind::LeftParen => advance(),
        _ => return Err("Expected left paren"),
    };

    // if we find a closing paren, then we are done (no params)
    if current_token().kind == TokenKind::RightParen {
        advance();
        return [];
    }

    let params = [];

    loop {
        // we expect an identifier for the param name
        let name = match current_token().kind {
            TokenKind::Identifier => current_token().lexeme,
            _ => return Err("Expected an identifier"),
        };

        advance();

        // a colon separates the identifier from the type
        match current_token().kind {
            TokenKind::Colon => advance(),
            _ => return Err("Expected ':' after identifier"),
        };

        // finally the type
        let type = match current_token().kind {
            TokenKind::Identifier => current_token().lexeme,
            _ => return Err("Expected an identifier"),
        };

        // add it to the list of params
```

```
38          params.push(Param { name, type });
39
40          advance();
41
42          match current_token().kind {
43              // if it's a ')' we are done and we break from the loop
44              TokenKind::RightParen => {
45                  advance();
46                  break;
47              }
48              // if it's a ',' then there's another param
49              TokenKind::Comma => advance(),
50              // else it's an error
51              _ => return Err("Expected right paren or comma"),
52          }
53      }
54  }
```

2.1.4 Generació de codi

En la definició del projecte vam comentar que fariem servir LLVM. Ara entrarem en una mica més de detall per veure que és LLVM, com funciona i com ha estat incorporat al projecte per tal de generar el codi.

LLVM va ser creat el 2003 per Chris Lattner (també creador del llenguatge de programació Swift) i disposa del suport d'empreses com Apple (LLVM és una part integral de XCode i de Swift per al desenvolupament d'aplicacions iOS), Google, IBM o Intel. Actualment hi ha diversos dels principals llenguatges de programació que en fan ús com ara C/C++ (a través del compilador Clang, una alternativa a GCC), Rust o Swift.

LLVM és un toolchain per crear compiladors, és a dir un conjunt d'eines diferents que ens ajuden en la implementació de compiladors, però principalment, i el que ens interessa a nosaltres, és un back end (de fet n'és molts a la vegada com ja veurem). Un back end és la part del compilador que genera assemblador per alguna arquitectura en concret a partir d'una representació intermèdia. Ara bé, en el cas de LLVM, no genera assemblador per una arquitectura en concret, sinó que en pot generar per pràcticament totes les arquitectures disponibles actualment. D'aquesta manera, tu, com a creador de llenguatges de programació, només t'has de preocupar de generar la representació intermèdia de LLVM. A partir d'aquí hi aplicarà optimitzacions i generarà l'assemblador de l'arquitectura que l'hi indiquis. Fins i tot pot generar Web Assembly, cosa que ens permet executar el llenguatge en navegadors web moderns.

Donat que LLVM és independent de l'arquitectura, quan generem el codi, no ens hem de preocupar del nombre de registres, ja que disposem d'un nombre il·limitat de registres virtuals, els quals LLVM mapejarà posteriorment als registres de l'arquitectura corresponent.

Com hem comentat, LLVM també aplica optimitzacions al codi generat, com pot ser eliminar codi que no es fa servir o avaluar expressions que es poden saber en temps de compilació. Ara bé, perquè LLVM pugui fer aquestes optimitzacions, nosaltres haurem de generar el codi en SSA (Static single-assignment), el que vol dir que només podem assignar valor a una variable una vegada. Si necessitem reassignar un valor, hem de crear una nova variable que la substitueixi. Això es fa simplement perquè facilita molt a l'hora d'aplicar optimitzacions.

LLVM té el concepte de mòduls. Un mòdul conté tota la informació associada a un arxiu de codi. Si tenim múltiples arxius, simplement hem de crear diferents mòduls i enllaçar-los.

Els mòduls contenen funcions i les funcions estan formades per instruccions, similars a les instruccions en assemblador.

Anem a veure un exemple d'un petit fragment de codi i com seria en la representació intermèdia de LLVM. Tenim la següent funció que rep dos enters i en retorna el màxim:

```
1 fn max(int a, int b) int {
2     if a > b { a } else { b }
3 }
```

Si ho traduim a LLVM tenim el següent codi:

```
1  define i32 @max(i32 %a, i32 %b) {
2  entry:
3    %0 = icmp sgt i32 %a, %b
4    br i1 %0, label %btrue, label %bfalse
5
6  btrue:
7    br label %end
8
9  bfalse:
10   br label %end
11
12 end:
13   %retval = phi i32 [%a, %btrue], [%b, %bfalse]
14   ret i32 %retval
15 }
```

La primera linia del codi anterior defineix una funció, la qual rep dos enters de 32 bits. A continuació defineix una etiqueta entry. Aquestes etiquetes són com les etiquetes dels assembladors, i podem anar-hi fent-hi salts.

En l'etiqueta 'entry', el primer que fa és comparar els dos enters, és a dir la condició del 'if'. La paraula clau 'sgt' vol dir 'signed greater than', o, en altres paraules, fa una comparació amb signe de més gran que. El resultat de la comparació és un enter d'un bit. Si és true farà un salt a la label `btrue`, sinó anirà a `bfalse`.

En aquest cas les dues branques fan el mateix: un salt a l'etiqueta `end`. Allà ens hi trobem amb un concepte que són els nodes phi. Els nodes phi venen a ser una espècia d'if invertit. En funció d'on hàgim fet el salt assignarem un valor o un altre a la variable `retval`. Si venim de `btrue` s'assigna a, i si venim de `bfalse` s'assigna b.

Ara bé, per què les dues branques fan al salt a l'etiqueta `end` i després es torna a fer un condicional en el bloc `end`? No podíem assignar el valor de `retval` directament dins de la branca `btrue` o `bfalse` i estalviar-nos aquest tercer condicional? Doncs la resposta és que no, perquè aleshores estaríem generant codi que no està en forma SSA. I com hem comentat anteriorment, LLVM ens obliga a generar codi en forma SSA. I per això existeixen els nodes phi, per poder solucionar aquest tipus de problemes.

2.1.5 API de LLVM

Està bé entendre el codi de LLVM, però generar tot aquest codi a mà pot ser una mica molest i propens a errors. És per això que LLVM ens ofereix una API per poder generar i compilar el

codi LLVM. Aquesta API és per a C++, per això hem de fer servir una altra llibreria anomenada Inkwell, que ens defineix una API de LLVM per a Rust, fent crides a través de FFI a l'API de C++.

L'API disposa d'una sèrie de classes amb les quals interactuarem. La més fonamental és la classe `BasicBlock`, la qual representa els blocs que ens trobem dins d'una funció, és a dir el conjunt d'instruccions que es troben dins d'una etiqueta. El conjunt d'aquests blocs, són les funcions, les quals estan representades per la classe `Function`. I seguint aquesta lògica, tenim la classe `Modue`, que representa els mòduls, que són un conjunt de funcions.

Totes aquestes classes neixen d'una classe base que s'anomena `Value`. Aquesta classe representa qualsevol valor que pugui generar el programa, ja sigui una funció, un bloc, etc. Aquesta classe és útil de cara a implementar les funcions de generació de codi del nostre compilador, ja que totes poden retornar un `Value`, i d'aquesta manera podem crear abstraccions que ens faciliten el desenvolupament.

Una altra classe important és `Context`. Necessitarem una sola instància d'aquesta classe, la qual conte un conjunt d'estructures de dades de LLVM, i que contindrà l'estat de la compilació.

Finalment tenim la classe `Builder`, que és la que s'encarrega de generar el codi com a tal, i al que, per tant, anirem cridant cada com que vulguem generar una instrucció. Per exemle si volem construir la comparació del if de la funció `max` anterior, ho fariem així:

```
Builder.CreateICmpSGT(a, b, "nom")
```

El tercer paràmetre indica el nom de la variable generada, tot i això, només serveix perquè, a l'hora de debugar, ens sigui més fàcil trobar-ho.

La crida anterior ens generarà el següent codi (que és la línia 3 de l'exemple complet que hem vist en l'apartat anterior):

```
%nom = icmp sgt i32 %a, %b
```

2.1.6 Anàlisi semàntica

Un pas addicional que hem de fer és l'anàlisi semàntica, per així assegurar que el codi que estem compilant és correcte, comprovant coses com que en una suma, els dos valors siguin de tipus compatibles i que es puguin sumar. Aquesta anàlisi es fa en el pas de la generació de codi, ja que estan molt lligats. Per exemple, en el cas de la suma que hem comentat, en generar el codi, ens adonarem que els dos tipus que estem sumant no són compatibles, ja que no podrem generar un codi vàlid.

2.1.7 Què ens ha aportat aquesta èpica?

Un cop finalitzada aquesta èpica ja tenim un llenguatge mínim que pot fer coses senzilles amb expressions numèriques. També tenim operacions de control de flux en forma de if i while. Per exemple, el següent programa ja funcionaria:

```
1 fn fib(n) {
2     if n <= 1 {
3         n
4     } else {
5         fib(n - 1) + fib(n - 2)
6     }
7 }
8
9 fn print_i(n) {
```

```
10      let i = 0;
11      while i < n {
12          print(i);
13          i = i + 1;
14      }
15  }
```

Podem veure que el llenguatge encara no l'hi indiquem el tipus dels paràmetres i de retorn, ja que és algo que apareixerà més endavant. En aquesta èpica de moment treballem sempre en doubles.

2.2 Declaració de tipus

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

L'objectiu d'aquesta èpica era permetre a l'usuari definir tipus propis a part dels que venen per defecte en el llenguatge. Aquest objectiu s'ha traduït en la implementació de structs dins del llenguatge, el qual permet definir conjunts de dades de forma estructurada. Funcionen igual que els structs de C, per exemple. La intenció es afegir mètodes als structs, per així poder dotar-los de funcionalitat, similar a les classes d'altres llenguatges, però això es farà més endavant si dona temps, ja que no és una funcionalitat imprescindible, donat que es pot obtenir un funcionament similar al dels mètodes, passant els structs com a paràmetre a les funcions.

Aquesta èpica també inclou les anotacions de tipus als paràmetres de les funcions i el tipus de retorn de les funcions. De moment els tipus disponibles són i64, f64 i els que l'usuari defineixi a partir dels structs.

Implementar els structs, com la gran majoria de noves funcionalitats afegides al llenguatge implica modificar el lexer, el parser i la generació de codi.

En quant al lexer no implica gaires complicacions. Simplement afegim una nova paraula clau ("struct") i també perque detecti els punts com a caràcter vàlid, ja que per accedir al camp d'un struct fem `struct.camp`.

Ampliar el parser, requereix una mica més de feina, però no difereix gaire del que hem fet fins ara. Hem d'implementar en el parser les produccions per a la definició de structs, per la instanciació i per l'accés als camps.

La part més complexa és la de generació de codi, ja que hem de tenir en compte la distribució de les dades en la memòria. Donat que podem tenir tres tipus diferents d'interaccions amb els structs els analitzarem per separat.

- **Definició**: Els structs en LLVM són simplement un conjunt de dades de diferents tipus una rere l'altre. Per exemple, el struct:

  ```
  1  struct SomeStruct {
  2      first: i64
  3      second: f64
  4  }
  ```

  es representa de la següent manera:

  ```
  1  struct { i64, f64 }
  ```

Només indiquem els tipus dels camps i en quin ordre van, però no hi ha cap identificador. Per aquest motiu, es responsabilitat nostra, comprovar coses com ara que un camp no estigui duplicat, o saber en quina posició del struct es troba la dada que ens demana l'usuari a través de l'identificador. Per tant, durant el procés de compilació hem de guardar totes les definicions dels structs, els seus camps, i l'ordre en el qual es troben, per així poder saber on buscar la dada que ens demana l'usuari.

- **Instanciació**: Per instanciar un struct, simplement hem de recórrer els camps que ens ha retornat el parser, un per un, generant les expressions de cada camp, i el resultat l'assignarem a la posició de memòria que li toqui segons l'índex del camp que se li ha assignat en la definició del struct. Aquesta assignació de memòria es fa igual que si assignéssim una variable, l'únic que canvia és com determinem la posició de memòria on guardar-hi el valor. LLVM disposa d'una instrucció que es diu GetElementPtr (GEP), que ens calcula la posició de memòria del camp a partir de l'apuntador a l'inici del struct i l'índex del camp.
- **Accés**: L'accés és similar a la instanciació, però més simple. Simplement hem de tenir l'índex del camp al qual volem accedir dins del struct (per exemple si és el primer camp, l'índex 0), i l'apuntador a l'inici del struct. A partir d'aquesta informació amb la instrucció GEP obtenim la posició de memòria del camp.

2.2.1 Què ens ha aportat aquesta èpica?

Després d'aquesta èpica ja comencen a tenir un llenguatge una mica més complet, ja que podem definir tipus propis a partir de structs i també especificar el tipus dels paràmetres i del valor de retorn de les funcions.

```
1 struct SomeStruct {
2     first: i64
3     second: i64
4 }
5
6 fn sum(a: SomeStruct, b: i64) i64 {
7     a.first + a.second + b
8 }
```

2.3 Collections

***En progrés**: 31/10/2022 - 20/11/2022** (3 esprints)

Aquesta èpica té l'objectiu d'afegir col·leccions d'elements al llenguatge. La idea és implementar arrays (de mida fixa, conegut en temps de compilació), i també slices (igual que els arrays, però la mida es pot conèixer en temps d'execució).

Si donés temps també seria interessant implementar arrays dinàmics.

Aquesta èpica està actualment en progrés. S'estan implementant els arrays, i un cop acabats s'implementaran els slices. LLVM té suport per arrays, i per tant, la seva implementació no és gaire complexa (similar a implementar structs). Pel que fa als slices, és una mica més complicat. La idea és fer servir un struct, on el primer camp sigui un punter al primer element i el segon camp, la longitud del slice.

Donat que els strings també són arrays de caràcters, l'objectiu és poder implementar-los.

Pel que fa a arrays dinàmics, una opció seria fer ús de la funció malloc de C, i enllaçar el programa amb libc.

2.3.1 Què ens aportarà aquesta èpica?

Un cop finalitzada aquesta èpica podrem declarar arrays i slices de la forma seguent:

```
fn test(n: i64) {
    let array = [1, 2, 3, 4]; // array d'enters de mida fixa
    let slice = [i64; n]; // slice d'enters de mida n
    slice[0] = 1;
    print(array[0] + slice[0]);
    let str = "un string"; // array de u8;
}
```

2.4 Mecanisme de gestió d'errors

***Planificada**: 21/11/2022 - 11/12/2022* (3 esprints)

Actualment ell llenguatge no té manera de gestionar errors, a part de retornar un enter que indiqui depenent del valor quin ha estat el resultat (com a C). Però això no és l'ideal, per això aquesta èpica serà per definir un sistema de gestió d'errors una mica més elegant i no tan propens a errors.

2.5 Syntactic sugar

***Planificada**: 12/12/2022 - 02/01/2023* (3 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.

2.6 Suport per programació d'estil funcional

***Planificada**: 03/01/2023 - 23/01/2023* (3 esprints)

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

2.7 Deute tècnic

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

Aquesta és una nova èpica que s'ha afegit per albergar totes les tasques de manteniment i millora del projecte, com ara resolució de bugs. La duració d'aquesta tasca és indefinida, ja que constantment van apareixent bugs i coses a millorar.

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.