

Llenguatges de programació

Curs 2021-2022

MEMÒRIA

Alumne (Grup 1)	Login	Nom
	v.valles	Víctor Vallés Medina
	carles.torrubiano	Carles Torrubiano Rubies
	oscar.julian	Òscar Julián Ponte
	bernat.segura	Bernat Segura Carmona
	rafael.morera	Rafael Morera Jeeninga

Data	29 de Maig del 2022
------	---------------------

Índex

1.	INTRODUCCIÓ	4
2.	MÒDULS DEL COMPILADOR	5
	Preprocessador.....	6
	Anàlisi lèxic.....	6
	Anàlisi Sintàctic.....	6
	Anàlisi Semàntic.....	7
	Optimitzacions.....	7
	MIPS	8
3.	INFORMACIÓ DEL LLENGUATGE.....	9
	Motivació.....	9
	Dades bàsiques	9
	Tipus de llenguatge.....	9
4.	GRAMÀTICA.....	11
	Punt d'entrada	11
	Definició	12
5.	ESPECIFICACIÓ DEL LLENGUATGE	16
	Diccionari	16
	Paraules reservades	16
	Símbols reservats.....	17
	Tipatge.....	18
	Comentaris	19
	Condicionals	19
	Buckles.....	20
	For.....	20
	While	20
	Funcions	21
	Func bàsica.....	21

Func amb múltiples return.....	22
Func recursiva.....	22
Operacions aritmètiques.....	23
Exemples sintaxi.....	24
Expressions regulars.....	26
6. METODOLOGIA DE TREBALL.....	27
7. ESTRUCTURES PRÒPIES.....	28
String of Tokens	28
ParseTree node.....	29
SymbolTable.....	30
Quadrupletes	31
8. CONCLUSIONS	33
9. DIAGRAMA DE CLASSES.....	34
10. BIBLIOGRAFIA	35

1. INTRODUCCIÓ

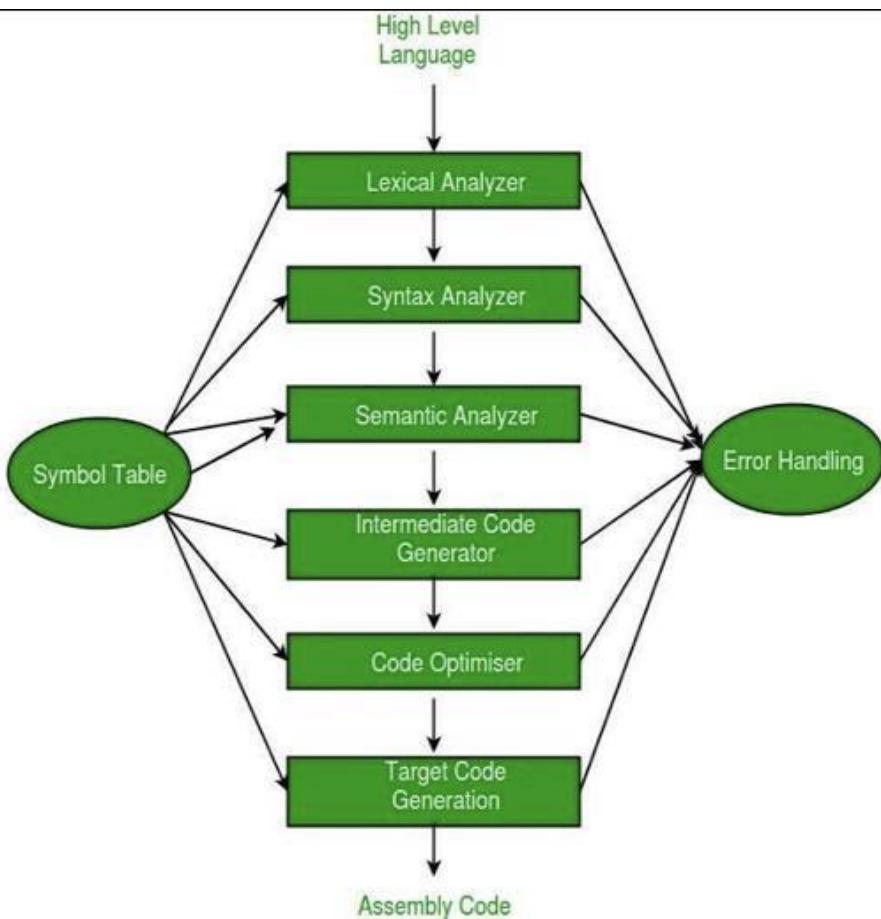
En aquesta pràctica es crea un compilador a partir dels conceptes estudiats a classe. Aquest compilador s'encarrega de passar d'un codi dissenyat per nosaltres a codi màquina. Per tant, l'objectiu final que es pretén aconseguir amb la realització d'aquesta pràctica és dissenyar un llenguatge de programació propi i crear un compilador capaç de transformar-lo perquè un ordinador sigui capaç d'entendre'l en llenguatge màquina.

La primera part consistirà a definir i dissenyar com serà el llenguatge amb tots els membres de l'equip. Un cop definides les dades bàsiques i especificacions que volem que tingui el llenguatge, caldrà fer la gramàtica del nostre llenguatge. Un cop tenim el disseny del llenguatge plasmat en un document formal, ja podem començar a desenvolupar el compilador per al nostre llenguatge. El desenvolupament d'aquest compilador es farà a través del llenguatge Java. La idea és anar implementant progressivament els mòduls dels quals es compon un compilador. Els mòduls d'un compilador es divideixen en dos grans grups, el front-end i el back-end. Nosaltres ens centrarem en la implementació del front-end fins a arribar a la generació del codi intermedi on l'adaptarem per al back-end utilitzant l'arquitectura MIPS.

La metodologia de treball ha sigut una metodologia Agile on amb l'ús d'eines com Jira i Bitbucket. Aquestes eines ens serviran per poder crear tasques i planificar sprints de treball, cosa que ens permetrà tenir una organització de cara a treballar en equip.

2. MÒDULS DEL COMPILADOR

El nostre compilador es compon primerament del mòdul de preprocessor, seguidament agafem el resultat d'aquest mòdul amb l'extensió .preprocessor i l'interpretem al següent mòdul. L'analitzador lèxic s'encarrega de llegir el .preprocessor i s'encarrega de fer la seva tasca i treu el seu resultat com .StringOfTokens. El següent mòdul és l'analitzador sintàctic on llegeix l'arxiu de .StringOfTokens i s'encarrega d'analitzar la gramàtica i construir l'arbre, també conegut com a parse tree. L'arbre l'emmagatzemem en un .json per poder veure les dades més gràficament. El següent mòdul és l'analitzador semàntic on s'encarrega de recórrer l'arbre i fer les seves validacions. El següent mòdul és el intermediate code generator on s'encarrega de fer el tac, és a dir el 3@code. Això genera un arxiu .tac on interactua el següent mòdul que és l'optimitzador, s'encarrega de fer les optimitzacions i finalment acaba l'últim mòdul on llegeix l'arxiu .tac i genera el codi màquina MIPS, aquest últim mòdul finalment acaba generant l'arxiu .asm.



Preprocessador

El preprocessador es la fase prèvia per la que passa el codi abans de ser compilat, per lo que no forma part del compilador en sí, sinó que s'encarrega d'eliminar algunes parts del codi que faciliten la vida al procés de compilació.

Per molt que no formi part del compilador en sí, hem trobat interessant implementar un ja que d'aquesta manera el nostre compilador podia començar el procés de compilació amb un codi més net i amb més facilitat. El nostre preprocessador en concret consta de dos funcionalitats principals les quals són l'eliminació de comentaris / blocs de codi i l'expansió dels defines (abans de començar a compilar el codi tots els diferents llocs on es troba un define es substitueix pel valor que li correspon).

Anàlisi lèxic

Aquesta es la primera fase del compilador i juntament amb les dos següents, són les que formen el conjunt de frontend del compilador. Aquest mòdul és l'encarregat de generar el string of tokens. Aquest el procés el realitza fent una lectura del fitxer generat pel preprocessor on es llegeix token a token on primerament es comprova si aquest token pertany al diccionari que tenim del nostre llenguatge, i en cas de que no sigui així, mirem si compleix els regex de tipus de variable o nom de variable. En cas de que no es compleixi cap d'aquests casos, aquest mòdul ja sap que s'ha trobat amb un token incorrecte. En aquest mòdul es van afegint tots els errors trobats en una llista i al final d'aquest procés llista tots els errors trobats i acaba l'execució. En cas de que tots els tokens siguin vàlids, es genera el string of tokens i també el fitxer pertinent que conte tota la informació interessant d'aquest mòdul.

Anàlisi Sintàctic

L'analitzador sintàctic s'encarrega de construir l'arbre recursivament segons la gramàtica definida. (veure l'índex). Per la implementació de l'arbre utilitzem una sèrie de nodes on els anem emmagatzemant recursivament (per veure l'estructura d'aquest node anar a l'apartat estructures pròpies a l'índex). Quan es detecta una gramàtica on no fa match amb el token corresponent salta l'error handler indicant que el token no és l'adequat i que és un error de syntax. Per un'altra banda, cada cop que l'analitzador sintàctic és troba una variable l'emmagatzema a la taula de símbols amb el seu scope corresponent.

Anàlisi Semàntic

L'analitzador semàntic s'encarrega de validar que les declaracions siguin semànticament correctes. És a dir, s'encarrega de la verificació dels tipus, per exemple si una funció retorna un 'int' el valor que guarda el return d'aquesta funció també sigui un 'int' en el cas que no sigui així es crida a l'error handler i es mostra un error de semàntica on indica la predicció de quin tipus hauria de ser. També l'analitzador semàntic busca a la taula de símbols si la variable s'ha instanciat al scope que toca i té un control de les variables de si es poden utilitzar. Per fer totes aquestes comprovacions el que fa es recórrer l'arbre generat al mòdul del analitzador sintàctic.

Generador de codi intermedi

El generador de codi intermedi s'encarrega de crear el Three Adress Code. Aquest està implementat com una llista de quadrupletes. Per tant recorrent l'arbre es va aturant a cada punt on té la informació que necessita per a crear la pròxima acció i en genera una nova quadrupleta que s'afegeix a la llista. Un cop acaba s'han creat totes les quadrupletes que genera el codi d'entrada. Aquesta es la última fase del frontend del compilador.

Optimitzacions

Un cop arribem al Backend ja disposem d'informació del Hardware on s'executarà el nostre codi intermedi. I per tant el podem adaptar i optimitzar per aquest en concret. En la nostre implementació això s'ha fet amb tenint en ment una arquitectura de MIPS32. S'han implementat 3 optimitzacions. La primera i més necessària es transformar totes les variables temporals que tenim a les quadrupletes en registres, i en cas de que ens quedem sense registres passar-ho a memòria i després llegir-ho de memòria quan es necessitin les dades. La segona optimització va ser inicialitzar el valor de les variables. Quan en el nostre codi es declarava una variable igualada a un valor, directament no es traduïa com a declarar la variable en MIPS32 amb aquell valor inicial. Amb aquesta optimització disminuïm el nombre de línies de codi i per tant d'accions a realitzar mentre que mantenim totes les propietats del codi. La tercera i última optimització que s'ha realitzat són les operacions de nombres constants. En el cas de que ens trobem amb una operació o una concatenació de operacions, les quals no contenen variables. Es realitzarà el càlcul d'aquesta operació en temps de compilació. Un cop hem fet això expandim el valor resultant per poder expandir les constants i poder evitar fer us de registres temporals o accessos a memòria innecessaris. Amb aquesta optimització reduïm significativament l'ús de variables temporals així com el nombre de línies necessàries per realitzar la acció concreta. Tot això incrementa el rendiment de computacional i temporal de l'execució dels programa que es compili.

MIPS

En el bloc de mips s'ha implementat la conversió de Quadrupletes a codi MIPS32. Per a cada una de les operacions que pot contenir una quadrupleta es genera el codi corresponent. En la conversió hi ha operacions que requereixen l'ús de temporals i registres que no es s'han tingut en compte en la creació de les quadrupletes donat que en aquell punt es desconeixen les regions de memòria disponibles. Tot el codi resultant es volca en un fitxer per a posteriorment executar-lo amb l'aplicació “MARS”.

3. INFORMACIÓ DEL LLENGUATGE

Motivació

Després d'haver tingut la possibilitat de programar en diferents llenguatges al llarg de la nostre carrera (o inclús fora), i tenint la possibilitat de crear el nostre propi llenguatge de programació en aquesta assignatura, ens hem decantat per crear-ne un, el qual ens permeti fer totes les coses que altres llenguatges ens limiten. Així que hem decidit basar-nos en un llenguatge com ‘C’, en el qual volem aconseguir, com hem comentat, permetre algunes funcionalitats o implementacions que aquest llenguatge no ens permet fer de per si.

Un dels objectius que volem assolir serà que aquest llenguatge sigui un fàcil d'entendre a primera vista i també que si algú mai l'ha vist, no sigui gens complicat d'aprendre a programar amb ell. Serà un llenguatge molt adient per a començar a programar.

Dades bàsiques

Nom: Ceasy

El nom és un joc de paraules entre ‘C’, el llenguatge de programació, i la paraula ‘easy’ que, en anglès, significa fàcil”. D'aquesta manera fem referència a l'orientació que li hem donat a aquest llenguatge.

Extensió: .cy

L'extensió dels nostres fitxers és directament una abreviatura del nom del llenguatge.

Tipus de llenguatge

Compilat / Interpretat → Compilat

Des del nostre punt de vista, i tenint en compte també la nostre experiència personal, considerem que és de més ajut que el llenguatge sigui compilat, ja que d'aquesta manera el control d'errors es durà a terme durant el temps de compilació, això comporta que, d'alguna manera, l'usuari podrà arribar a conèixer que no ha comés certs errors alhora d'executar el programa, lo que va relacionat amb que, en certa manera, l'usuari pugui programar d'una forma més “senzilla”, la qual cosa és el principal objectiu del nostre llenguatge de programació.

El tipus de compilació serà multi-passada. Cada fase es completa abans de començar la següent. Entre fase i fase es genera un fitxer de codi fins a l'etapa que es troba. Això permet que sigui més

fàcil debuggar i mantenir cada fase, però pel contrari, és més lent i més costos , degut a que es genera un fitxer per cada fase.

Paradigma → Imperatiu.

El paradigma del llenguatge és imperatiu ja que es basa en C. Aquest tipus de paradigma consisteix en una successió d'instruccions. El programador dona ordres concretes escrivint pas per pas el comportament que seguirà el programa.

Tipatge → Tipat feble

Hem decidit tenir un tipat feble, aquest és un dels punts principals la motivació del nostre llenguatge. Com hem comentat, volem tenir un llenguatge senzill, que s'assimili a d'altres però que no doni masses problemes i ens doni facilitat alhora de programar el codi. Per a poder complir aquesta motivació, el primer pas serà definir un tipat feble.

Domini → Genèric

El nostre llenguatge, al no centrar-se en cap àmbit concret, tindrà un domini genèric. Com hem comentat, serà molt adient per a començar a aprendre a programar, així que el domini serà molt genèric per a que es pugui utilitzar des de qualsevol enfoc. Aquest llenguatge no té el propòsit de resoldre un problema en d'un àmbit concret.

4. GRAMÀTICA

Punt d'entrada

El codi pot tenir com a punt d'entrada d'execució la funció Main. Ara bé, anterior a aquest funció es poden trobar, o no, diferents paràmetres com poden ser Defines, variables globals o funcions. L'ordre del fitxer sempre haurà de ser defines, variables globals, funcions i el main.

A continuació s'exposen alguns exemples de possibles casos de punts d'entrada del nostre codi:

Exemple 1:

```
#define DOS 2
```

```
int a = 1  
int b = DOS  
func sum() -> int {  
    return a + b  
}  
func main() -> void {  
    sum()  
}
```

Exemple 2:

```
func main() -> void {  
    int a = 1  
    int b = 2  
    int c = a + b  
}
```

Definició

Entry Point

```
P -> <primary_statement> <main>

<primary_statement> ::= <primary_statement-concat> <primary_statement> | ε

<primary_statement_concat> ::= <define> | <var_statement> | <func_statement> |
    <operacio> | ε
```

Block

```
<block_statement> ::= <block_statement_concat> <block_statement> | ε

<break> ::= 'break' | ε

<block_statement_concat> ::= <if_expr> | <while_expr> | <for_expr> | <func_callback>
    | <return_inside_function> | <assignment> | <var_statement> | <break> | ε
```

If

```
<conditional_block> ::= `(` <binary_cond_expression> `)` '{' <block_statement> `}`

<if_statement> ::= <conditional_block> <if_concat>

<if_expr> ::= 'if' <if_statement>

<if_concat> ::= <elif_expr> | <else_expr> | ε

<elif_exp> ::= 'elif' <if_statement>

<else_expr> ::= 'else' '{' <block_statement> `}`
```

Buckles

```
<while_expr> ::= 'while' <conditional_block>

<for_expr> ::= 'for' `(` <var_init_statement> `;` <binary_cond_expression> `;` <operacio> `)` `{` <block_statement> `}`
```

Function main

```
<main> ::= 'func' 'main' `(` `)` `->` 'void' '{' <block_statement> `}`
```

Function call

```
<func_callback> ::= <name> '(' <func_callback_params> ')'
<func_callback_params> ::= <id> <func_callback_params_concat> | ε
<func_callback_params_concat> ::= ',' <func_callback_params> | ε
```

Function declaration

```
<func_statement> ::= 'func' <name> '(' <func_declaration_params> ')' '->
<return_declaracio_func> '{<block_statement>}'
```

Function params

```
<> ::= <type_var_name> <func_declaration_params_concat> | ε
<func_declaration_params_concat> ::= ',' <func_declaration_params> | ε
```

Function arrow return

```
<return_declaracio_func> ::= <type> <return_declaracio_func_concat> | 'void'
<return_declaracio_func_concat> ::= ',' <return_declaracio_func> | ε
```

Function inside return

```
<return_inside_function> ::= 'return' <return_var>
<return_var> ::= <id> <return_var_concat> | ε
<return_var_concat> ::= ',' <return_var> | ε
```

Variables

```
<type_var_name> ::= <type> <name>
<define> ::= '#define' <name> <id>
<name> ::= a-zA-Z0-9_ //Format per lletres, números i/o '_'
<type> ::= 'int' | 'float' | 'boolean' | 'string' | 'int64' | 'float64'
```

Variables initialization

```
<var_init_statement> ::= '=' <assignment_value>
```

Variables assignation

`<assignment> ::= <name> <assignment_var_arr> '=' <assignment_value>`

`<assignment_var_arr> ::= '[' <id> ']' | ε`

`<assignment_value> ::= <func_callback> | <operacio>`

Variables declaration

`<var_statement> ::= <type_var_name> <var_statement_concat> | ε`

`<var_statement_concat> ::= <var_init_statement> | <assignment_var_arr>`

Condicionals

`<binary_cond_expression> ::= <expression> <binary_expression_prime>`

`<binary_expression_prime> ::= <op_expr> <binary_cond_expression> | ε`

`<expression> ::= <id> <op> <id>`

`<id> ::= Integer (0..9) || Float (0..9 + '.' + 0..9) || String ("a-zA-Z0-9_") || bool (true/false) ||`

`Operació (Ex. 2+3) || Funció (no void) || <name> || <func_callback>`

`<op> ::= '<' | '<=' | '>' | '>=' | '==' | '!='`

`<op_expr> ::= == '&&' | '||'`

Operacions

`<operacio> ::= <operacio_basica> | <operacions_simplificades> | <operacions_increments>`

`//Admet + - * / % <id> i ()`

`<operacio_basica> ::= <sum_res> <sum_res_prima>`

`<sum_res_prima> ::= <sym_op_low> <sum_res> <sum_res_prima> | ε`

`<sym_op_low> ::= '+' | '-'`

`<sum_res> ::= <mul_div> <mul_div_prima>`

`<mul_div_prima> ::= <sym_op_high> <mul_div> <mul_div_prima> | ε`

`<sym_op_high> ::= '/' | '*' | '^'`

`<mul_div> ::= <modul> <modul_prima>`

<modul_prima> ::= '%' <modul> <modul_prima> | ε

<modul> ::= '(' <operacio_basica> ')' | <id>

Operacions simplificades

<operacions_simplificades> ::= <name> <operacio_simp_op> <operacio>

<operacio_simp_op> ::= '+=' | '-=' | '/=' | '*='

Operacions increments

<operacions_increments> ::= <name>'++' | <name>'--'

5. ESPECIFICACIÓ DEL LLENGUATGE

Diccionari

Paraules reservades

Paraula	Descripció
if	Avalua una expressió condicional.
elif	Avalua una expressió quan no es compleix la condició del if.
else	Avalua expressions condicional quan no es compleixen les anteriors.
for	Bucle per repetir un conjunt d'instruccions un número finit de vegades.
while	Bucle infinit fins que no es compleix l'expressió condicional o s'obliga a la seva sortida.
break	Atura la execució d'un bucle i surt del mateix. En cas de no estar en un bucle, surt de l'estruatura de control.
return	Acaba l'execució d'una funció i també pot retornar un valor a la funció cridada.
func	Defineix la creació d'una funció.
nil	Representa un valor nul o buit.
#define	Permet definir una constant que es substitueix en tot el programa.
int	Tipus de variable int per nombres enters.
float	Tipus de variable float per nombres decimals.
int64	Tipus de variable per definir nombres enters de fins a 8bytes.
float64	Tipus de variable per definir nombres decimals de fins a 8bytes.
void	Tipus de variable nula.
string	Tipus de variable string per cadenes de text.
bool	Tipus de variable booleana.
main	Estableix la funció d'inici del programa.
true	Nom que indica el valor del booleans.
false	Nom que indica el valor de booleans.

Símbols reservats

Paraula	Descripció
//	Comentari de línia.
/* */	Comentari línia o en bloc.
+	Operació de suma o concatenació de dos strings.
-	Operació de resta.
*	Operació de multiplicació.
/	Operació de divisió.
=	Iguala una expressió.
++	Increment d'un.
--	Decrement d'un.
+=	Acumula el valor sumat a una variable.
-=	Acumula el valor restat a una variable.
()	Donen prioritat a operacions aritmètiques, s'utilitzen en la crida de funcions i engloben una expressió condicional.
[]	Instanciar o accedir a una posició d'un array.
{}	Defineix el contingut d'una funció o d'expressions condicionals.
""	Defineix el contingut de un variable tipus string.
''	Defineix el contingut de un variable tipus string.
->	Indica el tipus de variable que retorna una funció.
.	Separar la part decimal i entera d'un float.
,	Separar paràmetres d'una funció o entre variables.
%	Operació de mòdul d'una divisió.
;	Separa les parts d'un if i else if (variable, condició i increment).
<	Més petit.
>	Més gran.
>=	Més gran o igual.
<=	Més petit o igual.
==	Igual a.
&	Símbol reservat no vàlid.
&&	Operador lòtic and.
	Símbol reservat no vàlid.
	Operador lòtic or.
!=	Diferent de.

Tipatge

El tipatge del nostre llenguatge, com hem comentat, serà de tipat feble. Volem que sigui estrictament necessari declarar un tipus per a cada variable que es tingui, però volem permetre les variacions o ‘mutacions’ entre elles. Alguns exemples de variacions comentades s’especifiquen a continuació:

- Dividir ints i guardar-los en un float. Ex: $4/3 = 1.333333$
- Realitzar operacions entre ints i float. Retorna float.
- Qualsevol variable es pot sumar amb un string amb l’operant de suma. Retorna les dues variables concatenades.

Tipus generals	Ús	Justificació
int	Definir nombres enters (fins a 4 Bytes)	En el nostre llenguatge necessitarem definir tipus per a nombres enters. El nom ve donat per l’abreviació de ‘nombre enter’ en anglès (integer) i hem aprofitat aquest nom que es comú entre molts llenguatges de programació.
float	Definir nombres decimals (fins a 4 Bytes)	Implementem el tipus float per a poder treballar amb nombres decimals. El nom també ve donar en d’altres idiomes de programació, prové de l’abreviació de ‘coma flotant’ referint-se a la coma dels nombres decimals.
int64	Definir nombres enters grans (fins a 8 Bytes)	Implementem el tipus int64 per a poder treballar amb nombres enters grans.
float64	Definir nombres decimals grans (fins a 8 Bytes)	Implementem el tipus per a poder treballar amb nombres decimals grans.
string	Definir cadenes de caràcters. Ocuparà 1byte per cada caràcter que estigui emmagatzemat al string	Implementem el tipus string per a treballar amb cadenes sense necessitat d’utilitzar chars.
bool	Definir tipus boolean per tindre ‘true’ o ‘false’.	Implementem el tipus el qual ens facilitarà la programació per als casos en el que tinguem dues opcions que seran ‘o blanc o negre’.
Arrays estàtics	Definir estructures que ens permetin emmagatzemar un conjunt de dades.	Implementem arrays estàtics ja que volem fer un llenguatge fàcil d’entendre orientat a l’aprenentatge.

	Ocuparà en bytes: mida del tipus * nombre de casella
--	--

Comentaris

Per a poder escriure comentaris en el codi, ho hem implementat amb aquests símbols: “*// comentari*” per a comentaris d’una línia. També per a comentaris multi-línia i “*/* comentari */*”. Aquests símbols hauran de col·locar-se sempre al principi de la línia per a passar la totalitat de la línia a comentari.

En el nostre llenguatge, per a interpretar el final de línia no usarem cap tipus de símbol, directament serà el salt de línia (*\n*) el que ens indicarà el final d’aquella sentència, i posteriorment l’inici d’una nova.

La decisió de l’ús d’aquests símbols és perquè hi ha múltiples llenguatges que fan ús d’aquesta nomenclatura. Ja que, per a aprendre a programar usant aquest llenguatge, si posteriorment es continua en un altre, el programador es trobarà aquestes similituds.

Condicionals

Necessitem una estructura de control en el nostre llenguatge de programació. Quan el programador necessiti poder determinar quina acció ha de prendre el programa donada o no una certa condició, ho podrà fer mitjançant aquests condicionals. Així aconseguint una separació en l’execució del propi programa, executant certes accions a partir de l’avaluació d’una condició donada. Hem usat aquesta terminologia ja que és molt típica en diferents llenguatges, i de la mateixa manera que en altres punts del nostre llenguatge, serà de utilitat utilitzar la mateixa o de semblant a d’altres idiomes per acostumar-se a programar d’aquesta manera i si en un futur el programador necessita aprendre un idioma diferent ja tindrà una base per a començar si sap programar en Ceasy.

La sintaxi que seguirà en el nostre llenguatge és el següent:

```
if (condicio) {  
    //Code  
} elif (condicio2) {  
    //Code  
} else {  
    //Code  
}
```

Bucle

D'igual manera serà necessari tenir estructures de control en el nostre llenguatge de programació. Aquestes estructures permetran al programador tenir un control del flux del programa, el qual permetrà al programa realitzar repetidament les mateixes instruccions definides en una regió. Realitzar repetidament les mateixes instruccions pot ser de gran utilitat ja que podem anar canviant les dades que s'usen a l'interior d'aquest bucle i poder transformar la seva informació, de manera en que per a les següents instruccions s'apliquin en les noves dades. Per a aquestes estructures de control aplicarem dos tipus diferents de bucles, el for i el while.

For

El bucle for serà útil ja que en aquest podem marcar el nombre de iteracions que volem que el nostre codi repeteixi, s'usa en casi tots els llenguatges de programació.

Seguirem la següent sintaxi:

```
for (int i = 0; i < 10; i++) {  
    //Code  
}
```

Com a especificació cal declarar una variable pel statement del for. Aquesta variable s'utilitzarà únicament en aquesta estructura de control.

En l'exemple anterior, es pot observar com es declara la variable *int i = 0*.

While

El bucle while també el trobem a la majoria de llenguatges de programació. S'encarrega d'executar un tros de codi mentre que es compleixi una condició inicial.

La sintaxi que hem decidit implementar és la següent:

```
while (var > 10) {  
    //Code  
}
```

Funcions

Func bàsica

Les funcions del nostre llenguatge de programació començaran amb la paraula reservada ‘func’, seguit del nom de la funció. Hem considerat que fer ús de la paraula reservada ‘func’ és interessant perquè d’aquesta manera el codi queda molt més senzill a l’hora de llegir-lo (es localitzen les diferents funcions ràpidament) i és alguna cosa que pot ser d’ajuda per a la gent que pugui estar començant a programar ja que podran interpretar el codi d’una manera més senzilla.

Després del nom de la funció, ens trobem amb els diferents paràmetres que se li passen a la funció entre parèntesis (les funcions poden rebre N número de paràmetres), on a cadascun d’aquests s’indica de quin tipus es. Lo següent que hi ha es una fletxa, la qual indica que a continuació es troben els tipus de variables que retornarà aquesta funció. Novament, hem considerat que utilitzar el símbol de la fletxa, és una manera molt visual també de poder saber ràpidament quins són els tipus de variable que retorna la funció.

Un cop dins de la funció, per poder retornar el valor de la/les variables, s’ha de fer ús de la paraula ‘return’ seguida dels noms de les variables les quals es vol retornar el valor. L’execució del tros de codi de la funció s’acabarà quan en algun punt es trobi amb la paraula reservada ‘return’.

```
func fibonacci(int num) -> int {  
    // Code  
    return x  
}
```

També tenim funcions les quals no retornen cap tipus de valor, les quals tenen exactament la mateixa sintaxis amb la única diferència de que darrera la fletxa, en comptes d’indicar el tipus de variable, s’utilitza la paraula ‘void’ indicant que no s’efectuarà cap return dins de la funció.

El funcionament intern serà exactament el mateix que el de les funcions de dalt però en aquest cas, es podrà fer l’ús del return sense valors per acabar d’executar la funció o també, hi haurà la possibilitat de que s’acabi d’executar el tros de codi de la funció quan s’arriba al punt del claudador que la tenca.

```
func doSomething() -> void{  
    // Code  
}
```

Func amb múltiples return

Anteriorment hem vist les funcions com en “C”. Funcions que permeten passar paràmetres i retornar un valor. Però donat que el nostre llenguatge no contempla punters, els valors que passes per paràmetre a les funcions són sempre per valor i mai per referència. Aquest fet limita a l'usuari ja que només pot modificar una variable a cada funció. Amb l'objectiu de solucionar aquesta mancança el nostre llenguatge permetrà retornar més d'un valor de cada funció.

En fer la crida d'una funció amb més d'un valor de “return” s'ha de fer com l'exemple següent. Tenint les variables on rebem el valor retornat ja declarades prèviament, separades per una coma. Finalment s'iguala a la funció.

En declarar la funció s'indicaran els tipus de les variables que es retorna separats per una coma. Un cop dins el cos de la funció el “return” seguirà la mateixa estructura amb els valors a retornar separats per comes. Tenir en compte que sempre es mantindrà l'ordre en els tres casos. Quan es crida la funció, quan es defineix la funció i quan es retornen els valors.

```
var1, var2 = divide(num1, num2)

func divide(int num1, int num2) -> int, int {
    //Code
    return x1, x2
}
```

Func recursiva

El nostre llenguatge permetrà fer crides a funcions de forma recursiva. Donat que és un concepte molt important a entendre quan s'està aprenent a programar.

```
func recursiu() -> void{
    recursiu()
}
```

Operacions aritmètiques

Les operacions que pot realitzar Ceasy són:

Operacions	Exemples
Suma	$x = a + b$
Resta	$x = a - b$
Multiplicació	$x = a * b$
Divisió	$x = a / b$
Mod	$x = a \% b$

Les operacions aritmètiques tenen prioritat per parèntesis i tipus d'operació. L'ordre de prioritat en les operacions és major per les divisions i multiplicacions. Les operacions de suma i resta tenen menor preferència. Per últim, l'operació per obtenir el mòdul d'una divisió és la que més preferència té respecte les anteriors. En cas que es realitzin operacions de la mateixa preferència la seva prioritat s'estableix d'esquerra a dreta. Anem a veure un exemple:

$$x = a + b - c$$

La primera operació que es fa és $a + b$, un cop obtingut el resultat, es resta el valor de la variable c .

$$x = a * b / c$$

La primera operació que es fa és $a * b$, un cop obtingut el resultat, es divideix el valor de la variable c .

$$x = a * b + c$$

La primera operació que es fa és $a * b$, un cop obtingut el resultat, es suma el valor de la variable c degut a la preferència de les operacions.

$$x = a * (b + c)$$

La primera operació que es fa és $b + c$, degut a que els parèntesis atorguen més prioritat. Un cop obtingut el resultat de l'operació entre parèntesis, es multiplica el resultat per a .

Un altre aspecte a tenir en compte són els acumuladors. Es poden usar els operands “ $+ =$ ” i “ $- =$ ” per sumar i restar respectivament un valor directament a la variable. Amb aquests operants es simplifica el haver de declarar tota l'expressió “ $x = x + i$ ” o “ $x = x - i$ ”, tot i que també es pot realitzar d'aquesta manera.

També accepta abreviacions a l'hora de realitzar increments i decrements d'un en variables int i float. L'increment s'expressa com “x++”, que equivaldria a l'operació “ $x = x + 1$ ”, i el decrement “x--“, que equivaldria a “ $x = x - 1$ ”.

Exemples sintaxi

El nom de les variables pot ser tant caràcters com nombres.

Definició de int

```
int numA = 2
```

```
int numB
```

```
numB = 3
```

Definició de float

```
float numA = 2 (Internament serà 2.0)
```

```
float numB = 2.3
```

```
float numC
```

```
numC = 2.4
```

Definició de int64

```
int64 numA = 2
```

```
int64 numB
```

```
numB = 3
```

Definició de float64

```
float64 numA = 2 (Internament serà 2.0)
```

```
float64 numB = 2.3
```

```
float64 numC
```

```
numC = 2.4
```

Definició de string

```
string textA = "hola"
```

```
string textB
```

```
textB = "hola"
```

Definició de bool

```
bool isCorrect = true
```

```
bool isWrong
```

```
IsWrong = false
```

Definició d'arrays

```
type array [ 10 ] //espais entre nomVar i [ i entre el corxetes i el nom
```

```
type pot ser int, float, int64, float64, string o bool.
```

```
array [ i ] = 2
```

Expressions regulars

PR	TOKEN	DESCRIPCIÓ	REGEX	EXEMPLE
1	String content	Valor que es pot assignar a una variable del tipus String	(^"[a-zA-Z0-9\s]+")\$	"hello world"
2	Float/Float64 content	Valor que es pot assignar a una variable del tipus Float	(^[0-9]+[.][0-9]+)\$	43.5633
3	Int/Int64 content	Valor que es pot assignar a una variable del tipus Int	(^[0-9]+)\$	69
4	name	Nom que pot prendre una variable o funció	(^([a-zA-Z][a-zA-Z0-9_]*))\$	nom_VAR_101
5	Comentaris una línia	Identifica el principi d'un comentari	(^//.*)\$	// comentari
6	Comentaris en bloc (principi)	Identifica el principi d'un bloc de comentaris	(^/*[^.]*.)\$	/*
7	Comentaris en bloc (final)	Identifica el final d'un bloc de comentaris	(^[*]+/\$)	*/
8	Defines	Valor que es pot assignar a una variable define	(^#\#define\s[a-zA-Z][a-zA-Z0-9_]*\s.*\$)	#define nom_VAR "hola"

En la taula d'expressions regulars no estan definides les referents al diccionari de paraules ni de símbols ja que, el regex és el mateix (equals) a la paraula que volem identificar. Com per exemple:

(^if)\$ (^for)\$ (^while)\$

6. METODOLOGIA DE TREBALL

La metodologia de treball per aquesta pràctica ha sigut una metodologia Agile. Hem treballat amb sprints gràcies a l'eina Jira. Pel que fa a la codificació hem utilitzat Bitbucket per a tindre un control de versions del codi que hem anat desenvolupant.

Al llarg de tot el semestre hem anat dividint els blocs de treball en sprints. Aquests tenien durada d'una setmana, excepte un de 2 setmanes ja que coincidia amb setmana santa. En aquests assignàvem diverses storys amb les corresponents tasques a cadascun dels components del grup. Per així poder anar desenvolupant el projecte de manera grupal, però amb diferents tasques assignades individualment a cada membre del grup.

Alhora de treballar amb git hem tingut una branca de master, que representa una versió estable i definitiva. Per sobre de master hem creat la branca dev per a poder desenvolupar, aquesta branca conte una versió estable però inacabada del projecte. Finalment a sobre de dev creàvem una branca per a cada història que anàvem creant amb el sistema Jira.

Per a poder anar tot el grup coordinat amb aquesta metodologia també vam coordinar una reunió setmanal. Es a dir, la totalitat del grup ens reuníem els dimarts tarda i els dijous a la hora de classe per posar-nos al dia, prendre les decisions necessàries repartir feines i assignar grups per a poder seguir els dies venidors. A més per realitzar aquelles tasques més complexes feiem pair programming amb un altre membre del equip.

Amb questa pràctica hem vist la gran necessitat de les reunions en la metodologia Agile. El fet de que tot el grup estiguï d'acord i tinguem un objectiu i plantejament del treball alineat. Ja que en diverses reunions hem allargat mes de 2 hores i hem fet us d'un grup de WhatsApp, chat de Discord i trucades de veu per a seguir amb la presa de decisions. Ja que no ens havia donat temps a tractar tots els punts necessaris. Per contra, tots els components del grup sabem com s'ha dissenyat i implementat totes les parts de la practica.

Si bé no es la primera pràctica en la que fem servir una metodologia per sprints i branques, ja que en DPOO ja es va treballar amb una metodologia similar. Si que es la primera en la que aquestes han estat unes eines útils, i realment hem pogut veure el potencial de gestionar el desenvolupament del projecte amb aquestes eines.

7. ESTRUCTURES PRÒPIES

Per a completar el desenvolupament del compilador hem fet us de diverses estructures de dades en múltiples fases.

String of Tokens

La primera estructura que hem necessitat es el “String of Tokens”. Aquesta l’hem implementat com una llista de “Symbol”. Aquesta conte tots els tokens del programa amb el context corresponent. En la següent imatge es veu una part d’una “String of Tokens”.

```
{"token": "func", "regex": "FUNC", "lineNumber": 4}
{"token": "fb", "regex": "NAME", "lineNumber": 4}
{"token": "(", "regex": "PARENTHESES_OPEN", "lineNumber": 4}
 {"token": "int", "regex": "INT", "lineNumber": 4}
 {"token": "num", "regex": "NAME", "lineNumber": 4}
 {"token": ")", "regex": "PARENTHESES_CLOSE", "lineNumber": 4}
 {"token": "-\u003e", "regex": "ARROW", "lineNumber": 4}
 {"token": "int", "regex": "INT", "lineNumber": 4}
 {"token": "{", "regex": "CURLY_BRACKET_OPEN", "lineNumber": 4}
 {"token": "int", "regex": "INT", "lineNumber": 5}
 {"token": "prev", "regex": "NAME", "lineNumber": 5}
 {"token": "\u003d", "regex": "EQUAL", "lineNumber": 5}
 {"token": "1", "regex": "INT_VALUE", "lineNumber": 5}
 {"token": "int", "regex": "INT", "lineNumber": 6}
 {"token": "prevPrev", "regex": "NAME", "lineNumber": 6}
 {"token": "int", "regex": "INT", "lineNumber": 7}
 {"token": "res", "regex": "NAME", "lineNumber": 7}
 {"token": "int", "regex": "INT", "lineNumber": 8}
 {"token": "count", "regex": "NAME", "lineNumber": 8}
 {"token": "\u003d", "regex": "EQUAL", "lineNumber": 8}
 {"token": "1", "regex": "INT_VALUE", "lineNumber": 8}
 {"token": "while", "regex": "WHILE", "lineNumber": 10}
 {"token": "(", "regex": "PARENTHESES_OPEN", "lineNumber": 10}
```

L’estructura “Symbol” conté els camps que es veuen a continuació. On “token” es la paraula del codi a la que fa referència. El camp de “Key”, es una clau que hem assignat a cadascun dels possibles tokens. Aquesta clau es s’obté de 2 llocs. Primerament les claus estàtiques s’obtenen del diccionari. I en segon lloc les dinàmiques que s’obtenen dels regex.

```
SYMBOL {  
    token  
    Key  
    Line number  
}
```

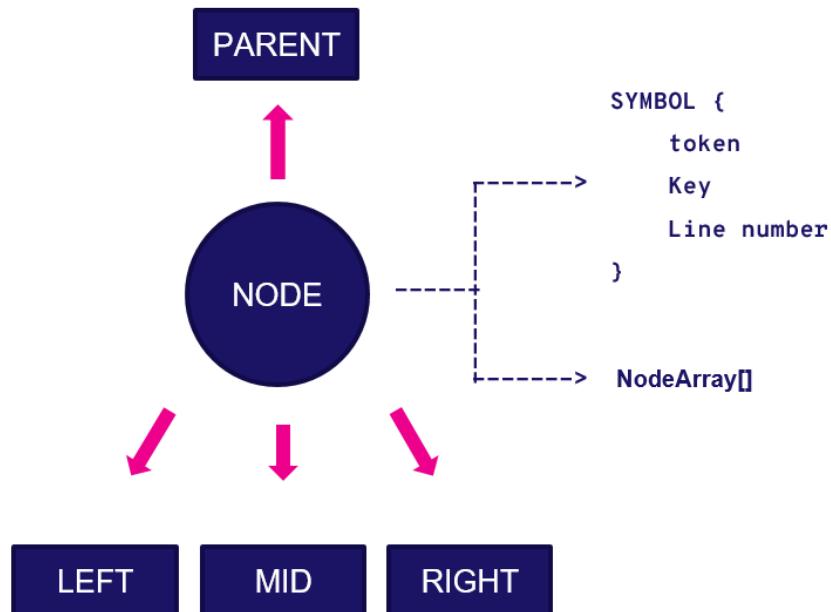
En la següent imatge es mostra una petita fracció del diccionari estàtic. Aquest es troba en un fitxer en format <token : clau>.

```
#define:DEFINE  
int:INT  
float:FLOAT  
int64:INT_64  
float64:FLOAT_64  
string:STRING  
bool:BOOL  
void:VOID  
main:MAIN  
true:TRUE  
false:FALSE  
//:COMMENT_LINE  
/*:COMMENT_BLOCK_OPEN  
*/:COMMENT_BLOCK_CLOSE  
+:ADD  
-:SUBTRACT  
*:MULTIPLICATION  
/:DIVISION
```

ParseTree node

El node del ParseTree que es genera durant l'anàlisi sintàctic del compilador és una classe. Aquest node es compon d'un pare, el qual és un altre node, i tres fills que representen el fill esquerre, mig i dreta, cadascun un node també. D'aquesta manera aconseguim per les properes fases recorre l'arbre top-down o bottom-up, segons ens convingui en cada cas. Cada node té tres fills amb l'objectiu de facilitar la implementació del Three Address Code, per contra, haguéssim hagut de rebalancejar l'arbre, una tasca que comporta un cost computacional elevat. Un node

també està format per la classe Symbol, la qual cosa ens permet identificar-lo segons la producció de la gramàtica que conte. Per acabar, un Node també està format per una llista d'altres nodes, això ens és útil en el cas del punt d'entrada del programa on podem trobar variables globals o funcions i pel cas de totes les possibles produccions que poden haver-hi dintre d'una funció, if, while, etc. i volem que es trobin en el mateix nivell.



SymbolTable

La taula de símbols emmagatzema informació sobre les instàncies de vàries entitats, noms de variables i funcions, etc.

L'estruktura de dades utilitzada per la taula de símbols conté un array dels scopes que es troben dintre d'un altre scope. Aquests scopes emmagatzemen Symbols, que poden ser variables o funcions, en una taula de hash. La gestió d'aquests scopes és realitzada cada vegada que es troba un '{' i es tanca quan es troba '}'. Els scopes també emmagatzemen el seu pare, d'aquesta manera quan es fa una cerca d'una variable dins d'un scope podem buscar a scopes superiors o scopes inferiors.

```

func main ( ) -> void {
    int a = 3
    if ( a < 5 ) {
        a = 20
    }
}
  
```

```

    ✓  ⌂ result = {Scope@1372}
        ⌂ father = null
    ✓  ⌂ child = {ArrayList@1373} size = 1
        ✓  ⌂ 0 = {Scope@1377}
            >  ⌂ father = {Scope@1372}
            >  ⌂ child = {ArrayList@1378} size = 1
                ⌂ currentChild = 1
                ⌂ depth = 1
            >  ⌂ scopeName = "main"
            >  ⌂ localScopeSymbols = {HashMap@1380} size = 1
                >  ⌂ "a" -> {Variable@1403} "Symbol{token='a', regex='NAME', lineNumber='3'}"
                ⌂ currentChild = 1
                ⌂ depth = 0
            >  ⌂ scopeName = "global"
        ✓  ⌂ localScopeSymbols = {HashMap@1375} size = 1
            >  ⌂ "main" -> {Function@1384} "Symbol{token='main', regex='MAIN', lineNumber='2'}"

```

Quadrupletes

Una llista de quadrupletes són implementació del “Intermediate Code”. Aquestes són una de les multiples implementacions del “Three Adress Code” possibles. L’estructura d’aquestes es la seguent.

```

QUADRUPLE {
    result
    operator
    argument 1
    argument 2
}

```

El camp “operator” conte la operació que es vol realitzar. En el camp “result” el resultat d’aquesta operació. I els arguments són els quals se sotmetran a la operació especificada. A continuació es mostra un exemple de quadrupletes.

```
Quadruple{arg1='null', arg2='null', op='VARIABLE', tmp='a'}
Quadruple{arg1='null', arg2='null', op='MIPS_TEXT', tmp='text'}
Quadruple{arg1='null', arg2='null', op='SECTION', tmp='main'}
Quadruple{arg1='6', arg2='null', op='EQUAL', tmp='t1'}
Quadruple{arg1='fb', arg2='t1', op='CALL_FUNC', tmp='t2'}
Quadruple{arg1='t2', arg2='null', op='EQUAL', tmp='a'}
Quadruple{arg1='null', arg2='null', op='END_PROGRAM', tmp='null'}
Quadruple{arg1='null', arg2='null', op='SECTION', tmp='fb'}
Quadruple{arg1='a0', arg2='null', op='EQUAL', tmp='num'}
Quadruple{arg1='1', arg2='null', op='EQUAL', tmp='t3'}
Quadruple{arg1='t3', arg2='null', op='EQUAL', tmp='prev'}
Quadruple{arg1='1', arg2='null', op='EQUAL', tmp='t4'}
Quadruple{arg1='t4', arg2='null', op='EQUAL', tmp='count'}
Quadruple{arg1='null', arg2='null', op='SECTION', tmp='WHILE_LABEL_1_INIT'}
Quadruple{arg1='count', arg2='num', op='LOWER_THAN', tmp='t5'}
Quadruple{arg1='t5', arg2='WHILE_LABEL_1_END', op='WHILE', tmp='null'}
Quadruple{arg1='null', arg2='null', op='SECTION', tmp='while'}
Quadruple{arg1='prevPrev', arg2='prev', op='ADD', tmp='t6'}
Quadruple{arg1='t6', arg2='null', op='EQUAL', tmp='res'}
Quadruple{arg1='prev', arg2='null', op='EQUAL', tmp='t7'}
Quadruple{arg1='t7', arg2='null', op='EQUAL', tmp='prevPrev'}
```

S'ha implementat fent us de Quadruples ja que era una de les implementacions que requerien de menys complexitat a la hora d'optimitzar. En afegit llegint la llista de quadruples es pot interpretar les accions molt fàcilment.

La facilitat que ens dona al optimitzar i que no ens donen altres estructures de dades es deu a que cada quadrupleta és independent de la resta. Per tant mentre no creus “SECTIONS” o l’assignació d’una mateixa variable amb l’ús d’aquesta, cosa que resultaria en que el codi T@C no sigui fidel al codi original. Amb aquestes excepcions pots reordenar quadruples així com expandir constants sense gran complexitat. Per contra és una estructura que en comparació amb d’altres implementacions del T@C o del propi “Intermediate Code” ocupa més espai i es mes costosa al processar ja que has de recorre la llista múltiples vegades per a cada optimització. Un altre inconvenient és que fa us d'un gran nombre de variables temporals per a poder representar el codi.

8. CONCLUSIONS

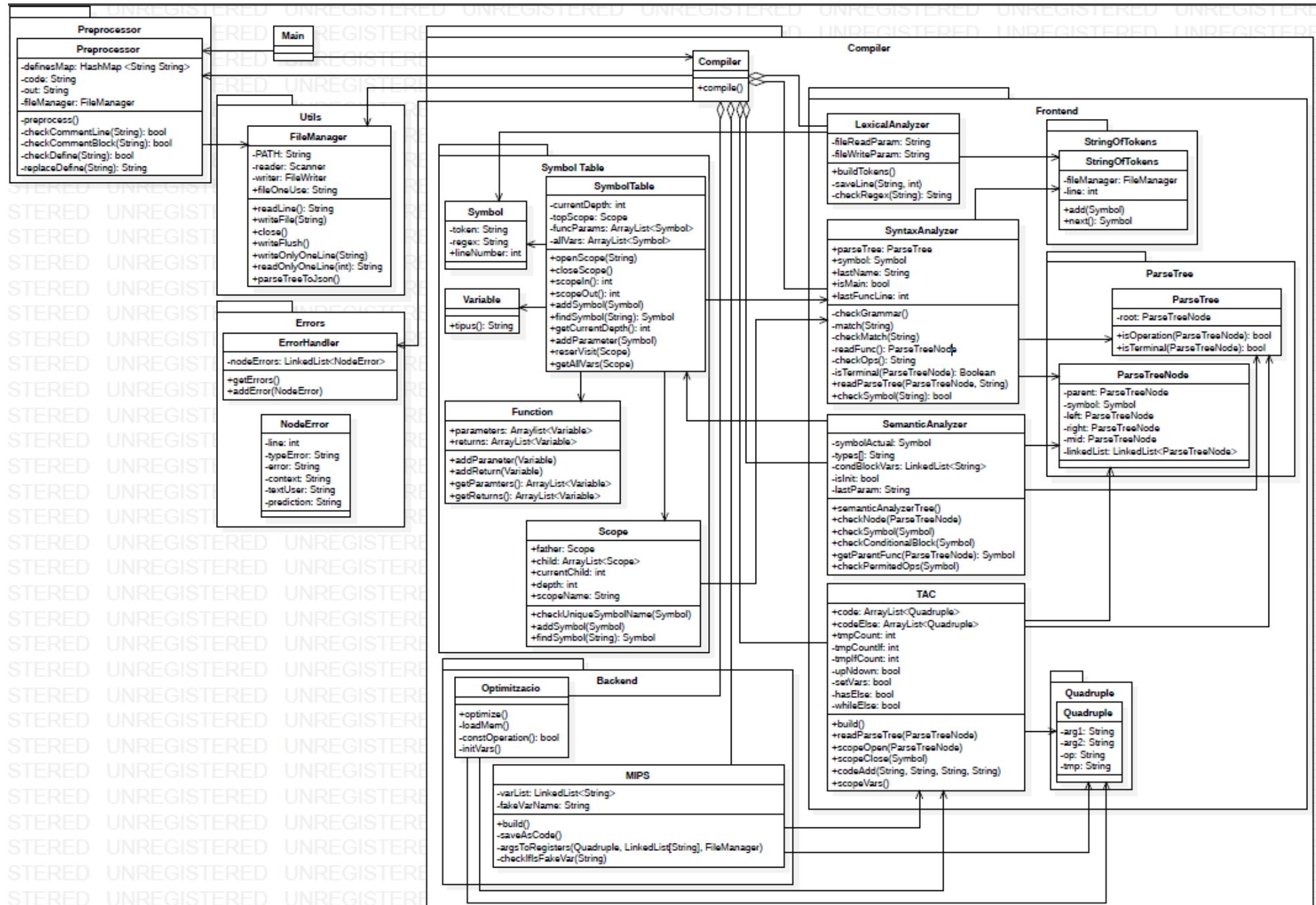
Aquesta pràctica ens ha servit a tots els membres del grup per a poder aplicar de forma pràctica tota la teoria que hem anat veient durant el semestre en la assignatura. Ha estat un aprenentatge incremental, en el qual anàvem aprenent setmana a setmana un concepte teòric nou i el haver de aplicar-lo al projecte ha estat un aprenentatge progressiu que ens ha semblat molt adient ja que si no hagués estat així, la idea de crear un llenguatge de programació o un compilador de principi a fi era molt complicat de plantejar per part nostre.

Relacionat amb això, ha estat molt útil treballar en SPRINTS, STORIES i TASKs ja que, com hem comentat, al ser un projecte de grans dimensions, ens hem pogut organitzar i estructurar el treball progressivament per cada setmana planificar el que havíem de fer per a, finalment, arribar al nostre objectiu que era tenir en funcionament el nostre compilador. A diferencia d'altres assignatures en aquesta hem pogut fer servir aquestes eines en una situació que sembla molt realista, en quant a tenir un equip de treball i un concepte a desenvolupar, on cada membre es fa responsable d'una part i finalment les hem de anar ajuntant per a crear un producte.

Tot això ha estat també gràcies a la feina en equip, ja que en aquets sprints i en aquest projecte, ens hem hagut de dividir el treball i treballar en equip per tal de arribar a les dates. Hi ha hagut molta feina relacionada amb la comunicació i organització, en la qual en el nostre grup hem anat treballar tant conjuntament com individualment. Primer de tot parlàvem conjuntament en com podíem dividir la feina, després, cada membre (o parella de membres) avançava parts diferents del treball i finalment concretàvem una reunió per a que cada membre pogués explicar com havia plantejar i avançant la seva part del treball. Òbviament tots els membres ens anàvem ajudant quan algun no estava segur de com continuar la seva part. També aquesta qüestió ha estat útil per a que tots fem comentaris de més qualitat ja que el company que els llegís havia de entendre que feia aquell codi. També el tema de les pull-request per a assegurar-nos en tot moment de que el codi que avançava en el projecte principal complís amb el objectiu que havia funcionar.

Finalment comentar que a nivell de dificultat, tots els membres del grup ens hem estat d'acord en quant que cap de nosaltres ens havíem plantejat totes les fases per les que passa un codi per a ser compilat i realitzant aquest treball hem pogut entendre amb més profunditat el funcionament dels compiladors, la seva importància i també la rellevància de escriure un bon codi i utilitzar correctament el llenguatge que s'utilitza en cada moment.

9. DIAGRAMA DE CLASSES



10. BIBLIOGRAFIA

Mozota, M.A. (2009). *Teoria de la Compilació; Anàlisi Lèxic i Sintàctic.*

<https://estudy.salle.url.edu/mod/resource/view.php?id=740109>

Mozota, M.A. (2009). *Teoria de la Compilació; Anàlisi Semántica i Sintàctic.*

<https://estudy.salle.url.edu/mod/resource/view.php?id=740110>

TAC + First and Follow reviewFitxer.

<https://estudy.salle.url.edu/mod/resource/view.php?id=761945>

Exemple TAC + basic blocsFitxer

<https://estudy.salle.url.edu/mod/resource/view.php?id=761947>

Gestió de memòria.

<https://estudy.salle.url.edu/mod/resource/view.php?id=761949>

Parsing step by step.

<https://estudy.salle.url.edu/mod/resource/view.php?id=761950>

BNF Grammars – CS 61 2020. (2020). Harvard.Edu.

<https://cs61.seas.harvard.edu/site/2020/BNFGrammars/>

CFG Developer. (2014). Stanford.

<https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg/>

About BNF notation. (s. f.). What is BNF notation?

<http://cui.unige.ch/isi/bnf/AboutBNF.html>

Grammar: The language of languages (BNF, EBNF, ABNF and more). (2015). Matt.might.

<https://matt.might.net/articles/grammars-bnf-ebnf/>

RegExr: Learn, Build, & Test RegEx. (2018). *RegExr*.

<https://regexr.com/>

MIPS32® Instruction Set Quick Reference.

<https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00565-2B-MIPS32-QRC-01.01.pdf>

MIPS Quick Tutorial. (s. f.). minnie.tuhs.org.

https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html

Online JSON to Tree Diagram Converter. (2014). vanya.jp.net.

<https://vanya.jp.net/vtree/>

Tak, T. (2018, 20 enero). *Java Tree Data Structure.* Java Code Gists.

<https://www.javagists.com/java-tree-data-structure#:~:text=Java%20Tree%20Implementation&text=In%20Java%20Tree%2C~%20each%20node,defined%20as%20a%20generic%20type>

What is Implementation of Three Address Code Statements? (s. f.). TutorialsPoint.Com.

<https://www.tutorialspoint.com/what-is-implementation-of-three-address-code-statements>