



UNIVERSITAT POLITÈCNICA DE CATALUNYA

BARCELONATECH

Facultat d'Informàtica de Barcelona



# TETRIS: PROGRAMAR-LO O FER-LO PROGRAMAR?

POL RIBERA MORENO

Director/a

ENRIQUE ROMERO MERINO (Departament de Ciències de la Computació)

Titulació

Grau en Enginyeria Informàtica (Computació)

Memòria del treball de fi de grau

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

25/06/2025

## RESUM

Aquest treball de fi de grau explora una manera alternativa i creativa de programar: utilitzar el joc del Tetris com a motor de computació. El projecte es basa en el treball de Meat Fighter “Tetris is Capable of Universal Computation”, on es demostra que, amb certes modificacions, el Tetris és Turing complet. Això significa que, teòricament, es pot utilitzar per executar qualsevol càlcul computacional.

L'objectiu principal del projecte és fer accessible aquest sistema a qualsevol persona amb coneixements bàsics de programació. Per aconseguir-ho, es desenvolupa un compilador que tradueix un llenguatge de programació d'alt nivell a l'assemblador del simulador de Tetris dissenyat per l'autor original.

A més, es proporcionen diferents exemples de programes fets d'aquesta manera, comprovant que efectivament el Tetris no és tan sols un joc.

## RESUMEN

Este trabajo de fin de grado explora una forma alternativa y creativa de programar: utilizar el juego del Tetris como motor de computación. El proyecto se basa en el trabajo de Meat Fighter “Tetris is Capable of Universal Computation”, donde se muestra que, con ciertas modificaciones, el Tetris es Turing completo. Esto significa que, teóricamente, se puede utilizar para ejecutar cualquier cálculo computacional.

El objetivo principal del proyecto es hacer accesible este sistema a cualquier persona con conocimientos básicos de programación. Para lograrlo, se desarrolla un compilador que traduce un lenguaje de programación de alto nivel al ensamblador del simulador de Tetris diseñado por el autor original.

Además, se proporcionan distintos ejemplos de programas hechos de esta manera, demostrando que el Tetris no es simplemente un juego.

## ABSTRACT

This thesis explores an alternative and creative way of programming: using the game of Tetris as a computing engine. The project is based on Meat Fighter's work “Tetris is Capable of Universal Computation”, which demonstrates that, with certain modifications, Tetris is Turing complete. This means that, theoretically, it can be used to perform any computational calculation.

The main goal of the project is to make this system accessible to anyone with basic programming knowledge. To achieve this, a compiler is developed that translates a high-level programming language into the assembly language of the Tetris simulator designed by the original author.

Additionally, various example programs developed in this way are provided, showing that Tetris is indeed more than just a game.

# Índex

<b>1 Context i Planificació</b>	<b>4</b>
1.1 Introducció i contextualització . . . . .	4
1.1.1 Marc de la Facultat d'Informàtica de Barcelona . . . . .	5
1.1.2 Conceptes . . . . .	6
1.1.3 Formulació del projecte . . . . .	8
1.1.4 Actors implicats . . . . .	9
1.2 Justificació . . . . .	9
1.2.1 Productes alternatius . . . . .	9
1.2.2 Eines . . . . .	9
1.3 Abast . . . . .	10
1.3.1 Objectius . . . . .	10
1.3.2 Requeriments . . . . .	11
1.4 Obstacles i riscos . . . . .	11
1.5 Metodologia . . . . .	12
1.5.1 Eines . . . . .	12
1.5.2 Seguiment . . . . .	13
1.6 Planificació temporal . . . . .	13
1.6.1 Descripció de les tasques . . . . .	13
1.6.2 Recursos . . . . .	17
1.6.3 Gestió de risc . . . . .	18
1.6.4 Gantt . . . . .	19
1.7 Gestió econòmica . . . . .	20
1.7.1 Pressupost . . . . .	20
1.7.2 Control de gestió . . . . .	21
1.8 Sostenibilitat . . . . .	23
1.8.1 Autoavaluació . . . . .	23
1.8.2 Dimensió Econòmica . . . . .	23
1.8.3 Dimensió Ambiental . . . . .	24
1.8.4 Dimensió Social . . . . .	24
<b>2 Base Teòrica</b>	<b>24</b>
2.1 Resum del treball de Meat Fighter . . . . .	24
2.1.1 Principis bàsics . . . . .	24
2.1.2 TetrominoScript . . . . .	26
2.2 Programació d'un compilador amb Python utilitzant ANTLR4 . . .	41
2.3 Generació del compilador . . . . .	43
2.4 Primer contacte amb l'entorn i objectius inicials . . . . .	46

<b>3 Disseny</b>	<b>47</b>
3.1 Definició del llenguatge . . . . .	47
3.1.1 Permissivitat vs Restricció . . . . .	47
3.1.2 Originalitat vs Estàndards de la comunitat . . . . .	48
3.1.3 Generalització vs Especialització . . . . .	48
3.1.4 Funcionalitats essencials . . . . .	48
3.2 Classificació del llenguatge de programació . . . . .	49
<b>4 Implementació</b>	<b>50</b>
4.1 Funcionament bàsic del compilador . . . . .	50
4.2 Distribució de la memòria . . . . .	50
4.3 Funció funcioreturn . . . . .	53
4.4 La resta de codi . . . . .	54
4.5 Definicions lèxiques . . . . .	54
4.6 Construcció de la gramàtica . . . . .	55
4.7 Simples . . . . .	56
4.8 Aritmètica . . . . .	58
4.9 Boolean . . . . .	62
4.10 Logic . . . . .	63
4.11 Op . . . . .	65
4.12 Expressió . . . . .	69
4.13 Limitacions del sistema . . . . .	87
4.14 Creació d'interfície d'usuari . . . . .	88
<b>5 Resultats</b>	<b>100</b>
<b>6 Conclusions</b>	<b>101</b>
6.1 Conclusions tècniques . . . . .	101
6.2 Valoració personal . . . . .	102
<b>7 Bibliografia</b>	<b>102</b>

# 1 Context i Planificació

## 1.1 Introducció i contextualització

A principis del 2023 es va publicar el projecte "Tetris is Capable of Universal Computation", en el qual es basa tot aquest treball. El projecte, creat per Meat Fighter, parteix de la premissa que el Tetris, amb les modificacions adequades, és Turing complet i pot utilitzar-se com a motor de programació de propòsit general.

L'estudi comença mostrant com les peces del Tetris poden construir portes lògiques i gestionar memòria, per acabar desenvolupant un simulador fet exclusivament amb tetròminos capaç d'executar el propi joc dins seu. Es tracta d'una autèntica obra d'art: potser no gaire útil a nivell pràctic per qüestions d'eficiència, però com a curiositat científica i eina didàctica té un gran potencial. Qui no voldria veure un programa HELLO WORLD fet amb tetròminos?

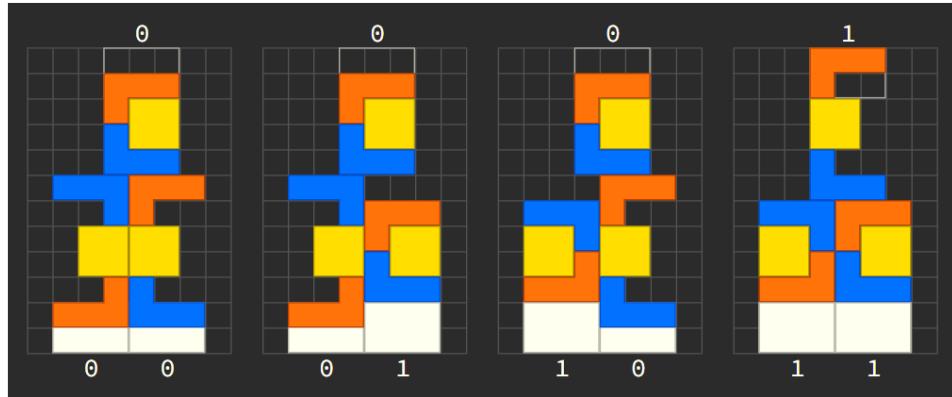


Figura 1: Porta AND feta a partir de tetròminos. (*Meat Fighter, 2023*)

Des que la computació ha estat una realitat, hi ha hagut un grup de persones que s'han dedicat a millorar-la, creant algoritmes, protocols i sistemes per fer-la més òptima i eficient. Mentrestant, un altre grup s'ha dedicat a posar a prova la seva pròpia creativitat i coneixements per tal d'exprémer fins l'última gota del que la computació pot arribar a fer, creant així ordinadors dintre de Minecraft o, com és el cas, utilitzar tetròminos per computar programes. Si un sistema és Turing complet, és probable que algú l'hagi explotat per aconseguir executar quelcom.



Figura 2: Execució del Tetris dintre del Minecraft. (*Youtube: sammyuri*)

Realment aquests tipus de projectes no tenen una utilitat pràctica en el món real, però gràcies a les millores del hardware són cada cop més populars i viables, ja que avui dia els programadors no estem obligats a optimitzar l'eficiència al màxim, fet que converteix la programació amb tetròminos en una realitat assumible i, per què no dir-ho, molt divertida.

L'únic inconvenient que hi ha és que, per programar en el simulador de Tetris is Capable of Universal Computation, cal dominar un assemblador especialitzat creat per l'autor, fet que pot dificultar-ne l'accés a moltes persones. Per això, l'objectiu principal en aquest TFG és fer accessible aquesta forma de programació a qualsevol persona amb coneixements bàsics de programació.

Per aconseguir-ho, es desenvoluparà un compilador que tradueixi d'un nou llenguatge d'alt nivell a l'assemblador creat per l'autor original. Tot i així, no serà senzill, degut a la complexitat de comprendre a fons el treball original i a que s'hauran de fer les modificacions necessàries per tal que tot funcioni correctament.

### 1.1.1 Marc de la Facultat d'Informàtica de Barcelona

Aquest projecte està estretament relacionat amb la FIB i la informàtica, ja que tracta sobre una forma alternativa d'executar codi, la base fonamental de tot el Grau en Enginyeria Informàtica.

A més, com s'ha mencionat anteriorment, l'objectiu principal és crear un compilador, fet que permetrà reafirmar i aplicar els coneixements adquirits en l'assignatura de Compiladors. També seran útils conceptes apresos a diferents assignatures del grau:

- Llenguatges de programació (LP), on s'estudien conceptes com la completenessa de Turing i les característiques intrínseqües dels diferents tipus de llenguatges de programació, fet que permetrà dissenyar de forma correcta el llenguatge d'alt nivell necessari per desenvolupar aquest projecte.

- Teoria de la computació (TC), que treballa amb màquines de Turing i les maneres més senzilles de computar, com autòmates i gramàtiques. Serà necessari en la creació de la gramàtica del llenguatge.
- Videojocs (VJ), on es tracta el desenvolupament de jocs 2D i que facilitarà la comprensió del funcionament del Tetris.
- Les assignatures de hardware bàsiques cursades durant la carrera: introducció als computadors (IC), estructura de computadors (EC), arquitectura de computadors (AC) i interfícies de computadors (IC), necessàries per a la comprensió del funcionament d'un ordinador i de llenguatges de baix nivell com assembler, el qual és un requisit bàsic sempre que es vol treballar a nivell de màquina.

El grau en Enginyeria Informàtica de la FIB ofereix 5 especialitats diferents als alumnes: Computació, Enginyeria de Computadors, Enginyeria del Software, Sistemes d'Informació i Tecnologies de la informació. Aquest treball està clarament arrelat a l'especialitat de Computació, que capacita per dissenyar sistemes informàtics complexos tenint en compte criteris crítics d'eficiència, fiabilitat i seguretat. Prepara per ser capaç d'avaluar aquests requeriments i recomanar les màquines, els llenguatges de programació i els mètodes algorísmics més adequats per dissenyar-ne una solució informàtica.

Les competències que abasta el treball són les següents:

- CCO1.1: Avaluar la complexitat computacional d'un problema, conèixer estratègies algorísmiques per resoldre'l i implementar la que millor rendiment ofereixi segons els requisits.
- CCO1.2: Demostrar coneixement dels fonaments teòrics dels llenguatges de programació i les tècniques de processament lèxic, sintàctic i semàntic, i saber aplicar-les en la creació i disseny de llenguatges.
- CCO3.1: Implementar codi crític seguint criteris d'eficiència, seguretat i temps d'execució.
- CCO3.2: Programar tenint en compte l'arquitectura hardware, tant en assemblador com en llenguatges d'alt nivell.

Un cop realitzat el projecte i donada la seva naturalesa didàctica, pot ser utilitzat per professors de la FIB de Compilació com a exemple i inspiració per als seus alumnes.

### 1.1.2 Conceptes

A continuació es defineixen els conceptes claus d'aquest treball.

## Tetris

És un videojoc de trencaclosques creat per Aleksei Pázhitnov el 1984, en què el jugador ha de col·locar en una graella rectangular de 10 quadrats per 20 quadrats tetròminos per completar línies horitzontals i evitar que la pantalla s'ompli en l'eix vertical, fet que implica la finalització de la partida.

Els tetròminos són les 7 possibles combinacions de col·locar 4 quadrats de tal manera que cap quedi lliure, és a dir, que cada quadrat ha de tenir almenys un costat compartit amb un altre quadrat. Aquestes peces van caient una a una des de la part superior de la graella, tenint el jugador la possibilitat de moure-les cap a qualsevol costat o rotar-les per tal de col·locar-les en la millor posició per poder formar una línia horitzontal, que desapareixerà de la pantalla quan es completi.

Tot i ser un joc aparentment senzill, s'ha estudiat en profunditat en l'àmbit de la computació per les seves propietats matemàtiques i algorítmiques.

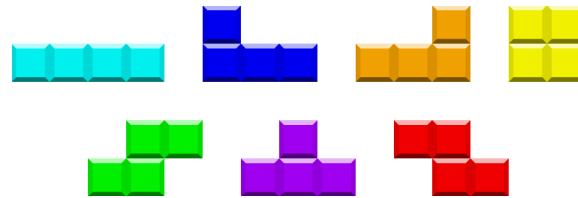


Figura 3: Els 7 tipus de tetròminos possibles. (*Tetris Wiki*)

## Turing Complet

Un sistema és Turing complet si pot realitzar qualsevol càlcul computacional possible, sempre que tingui recursos il·limitats (temps i memòria). Això vol dir que pot simular una màquina de Turing, el model teòric de computació definit per Alan Turing.

Una màquina de Turing consta d'un capçal de lectura i escriptura, que modifica i es mou -una posició cap a la dreta o una posició cap a l'esquerra- al llarg d'una cinta infinita dividida en cel·les que contenen símbols d'un alfabet. Aquest moviment depèn d'una funció de transició que indica com es pot anar d'un estat a un altre, a partir del símbol que hi ha a la cel·la i el propi estat.

La màquina comença en un estat inicial decidit a priori i el seu resultat dependrà de si acaba en un estat final o no. D'aquesta manera podem executar qualsevol algoritme amb la màquina.

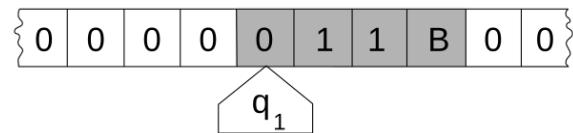


Figura 4: Representació d'una màquina de Turing. (*Wikipedia*)

## Compilador

Un compilador és un programa que tradueix el codi font escrit en un llenguatge de programació d'alt nivell a un llenguatge de baix nivell, com l'assemblador o el codi màquina, perquè pugui ser executat per un computador.

Les parts bàsiques d'un compilador són:

- Anàlisi lèxic: converteix el codi font en una seqüència de tokens (unitats bàsiques com paraules clau, identificadors i símbols).
- Anàlisi sintàctica: comprova l'estructura gramatical del codi i construeix l'arbre sintàctic.
- Anàlisi semàntica: verifica el significat del codi, com la compatibilitat de tipus i l'ús correcte de les variables.
- Generació de codi intermedi: transforma l'arbre sintàctic en una representació intermèdia, més propera al llenguatge màquina però encara independent del maquinari.

En aquest projecte, el compilador traduirà el nou llenguatge que es dissenyarà a l'assemblador del simulador de tetròminos.

### 1.1.3 Formulació del projecte

Aquest projecte vol fer accessible la programació en tetròminos a qualsevol persona que tingui un nivell bàsic de programació. Per poder resoldre aquest problema, s'ha de dividir en tres grans parts:

1. Comprensió del simulador creat per Meat Fighter (aprenentatge)

Caldrà entendre com els tetròminos són capaços d'executar codi, analitzar el funcionament intern del simulador i interioritzar l'assemblador desenvolupat per l'autor.

2. Creació del compilador (treball)

S'haurà de dissenyar un llenguatge d'alt nivell, desenvolupar un compilador que el tradueixi a l'assemblador i, si cal, ajustar alguns aspectes del simulador per garantir-ne el bon funcionament.

3. Divulgació i distribució (difusió)

Perquè aquest treball tingui un impacte en el món real, és important compartir-lo, fent-lo accessible a tothom per entretenir, ensenyar i inspirar programadors d'arreu del món.

#### **1.1.4 Actors implicats**

El projecte està dirigit, tal i com s'ha mencionat a l'apartat 1.1 d'aquest escrit, a qualsevol persona curiosa i amb uns mínims coneixements de programació, perquè a partir d'aquest treball pugui utilitzar el Tetris per executar els seus codis sense cap tipus d'esforç extra. A més, si es duen a terme tasques didàctiques i de divulgació, qualsevol persona amb interès podria beneficiar-se'n.

Cal remarcar que el treball és una extensió de "Tetris is Capable of Universal Computation", motiu pel qual un altre actor implicat i beneficiari és el seu autor, Meat Fighter.

A nivell personal i professional, aquest projecte no només em sembla curiós, divertit i emocionant, sinó que també crec que em permetrà aprofundir en els meus coneixements sobre computació. És un repte on hauré d'aplicar gran part del que he après durant la carrera, cosa que el fa encara més enriquidor.

### **1.2 Justificació**

En un món tan pragmàtic com l'actual, cada cop es publiquen menys treballs sobre curiositats. Tot i que aquests tipus de projectes poden semblar poc útils a primera vista, en realitat fomenten la creativitat col·lectiva, ens permeten innovar, compartir coneixements i tècniques, i explorar fins a l'últim detall d'una idea. Al cap i a la fi, són el veritable motor de la innovació.

#### **1.2.1 Productes alternatius**

El Tetris és un dels jocs més estudiats de la història, i al llarg dels anys s'han fet innombrables experiments i variacions amb ell. Tot i això, a causa de l'especificitat i complexitat d'aquest projecte, la seva escassa aplicabilitat en l'àmbit laboral i el fet que es basa en un treball ja existent, no és d'estranyar que no existeixi un estudi similar al que es vol dur a terme.

Es poden trobar moltes opcions si l'objectiu final només fos computar un algoritme. Però en aquest treball serà possible computar amb tetròminos, una forma totalment innovadora de fer-ho.

#### **1.2.2 Eines**

Per una banda, és necessari utilitzar totes les eines que Meat Fighter va emprar, com per exemple Java i l'assembler específic. No obstant això, per desenvolupar la part del compilador, s'utilitzarà ANTLR4, ja que pot ser una molt bona opció per crear un compilador i, a més, és l'eina que més s'usa durant el Grau en Enginyeria

Informàtica a la FIB.

També hi ha l'opció d'utilitzar Flex i Bison, però ANTLR4 té molts més avantatges, com que permet el suport a llenguatges moderns, unifica tot el procés i és autosuficient i permet l'automatització de molts processos tediosos, com per exemple la recursió cap a l'esquerra i la gestió d'ambigüïtats.

Per fer el compilador s'ha optat per usar Python, degut a la seva senzillesa i al suport que en proporciona la xarxa. Malgrat tot, haver utilitzat un altre llenguatge ens hagués donat molta més eficiència; però per flexibilitat i pragmatisme s'ha optat per Python.

## 1.3 Abast

En qualsevol projecte és molt important definir-ne l'abast, ja que el temps del qual es disposa per al seu desenvolupament és limitat i és necessari determinar clarament els objectius, els obstacles i els riscos del treball, que s'exposen a continuació.

### 1.3.1 Objectius

L'objectiu principal del projecte és aconseguir que qualsevol persona amb un nivell bàsic de programació pugui programar utilitzant els tetròminos. Això fa que s'hagin d'assolir els següents subobjectius, en l'ordre en què es presenten.

- Comprendre a la perfecció com és possible executar programes utilitzant només peces del Tetris. Aquest subobjectiu comporta un alt grau d'aprenentatge, ja que cal entendre com es creen les portes lògiques amb el Tetris, com funciona l'assembler específic de Meat Fighter i com ha creat el seu simulador. Si no s'assolís, no es podria continuar amb el treball, degut a que forma la base de desenvolupament del projecte.
- Dissenyar un llenguatge d'alt nivell que sigui el més senzill possible. Això comporta prendre moltes decisions, com per exemple, la seva extensió, la complexitat o el sistema de tipat. També cal tenir molt clara la forma en què s'emmagatzemaran les variables i com es cridaran les funcions.
- Crear un compilador capaç de traduir aquest llenguatge a l'assemblador del simulador. És el subobjectiu que requerirà més esforç per aconseguir el seu assoliment, ja que cal programar totes les parts del compilador, cadascuna de les quals pot comportar molts problemes per seguir endavant.
- Crear interfícies gràfiques que facilitin la programació del major nombre de funcionalitats possibles, amb exemples concrets de programes típics.

- Divulgar el treball perquè qualsevol persona curiosa o interessada en el tema hi pugui accedir. No és el subobjectiu més important, però tampoc és fàcil, ja que requereix de molta creativitat i constància.

### 1.3.2 Requeriments

No són molts els requeriments del projecte, si més no cal esmentar-ne alguns.

#### Requeriments funcionals

Els requeriments funcionals imprescindibles del projecte són els següents:

- El projecte està pensat per ser utilitzat en un sistema operatiu Windows amb totes les eines abans esmentades en l'apartat 2.2.
- Per a un bon funcionament també es requereix que l'usuari tingui nocions bàsiques d'ús de la línia de comandaments de Windows.

#### Requeriments no funcionals

Cal descriure uns requeriments no funcionals importants que també s'han de tenir en compte en el desenvolupament del projecte des del seu inici.

- La compilació ha de ser correcta per tal que no hi hagi errors ni comportaments inesperats.
- El sistema ha de ser eficient per tal que sigui assumible utilitzar-lo i que els usuaris no estiguin molt temps esperant.
- La simplicitat del projecte ha de tenir-se en compte per poder ser utilitzat per les màximes persones possibles.

## 1.4 Obstacles i riscos

Com en tots els projectes i, sobretot, de bon inici, s'entreveuen alguns obstacles i riscos, que poden impedir el bon desenvolupament del projecte. A continuació, s'esmenten els més importants.

- No aconseguir comprendre totalment el funcionament intern del simulador, la qual cosa impediria fer les simulacions adequades per adaptar-lo a la creació de qualsevol programa.
- Tenir problemes en la gestió dels inputs i outputs a baix nivell, ja que no permetria controlar bé amb l'assembler les trucades del sistema.
- Assolir un rendiment i eficiència massa baixos en la visualització d'un programa fet per tetròminos, de tal manera que fos inviable el fet de veure'ls degut a l'elevada quantitat que són necessaris per executar un programa.
- Esgotar el temps i no poder acabar el projecte. És el que més preocupa perquè hi ha molta feina individual a fer en poc temps.

## 1.5 Metodologia

El desenvolupament d'aquest treball està estretament vinculat amb la investigació de noves formes de programació basades en el Tetris, per la qual cosa és essencial comptar amb una metodologia flexible que permeti implementar funcionalitats, provar-les i avaluar si cal descartar-les, donar-les per finalitzades o explorar noves vies d'investigació.

Per aquest motiu, s'ha optat per utilitzar una metodologia àgil de desenvolupament de programari, que compleix aquests requisits. L'estrategia consisteix a definir cicles curts de desenvolupament (aproximadament una setmana), durant els quals es dissenyarà, implementarà i es validarà una nova funcionalitat.

Dins aquest marc, s'ha preferit una metodologia Kanban, ja que s'estableix l'organització seguint un sistema de «Per fer», «En procés» i «Fet». Aquest enfocament permetrà:

- Visualitzar clarament les tasques, facilitant-ne el seguiment.
- Millorar la gestió del temps, evitant sobrecàrrega de treball.
- Adaptar-se fàcilment als canvis, ajustant prioritats segons sigui necessari.
- Mantenir la motivació al llarg del projecte

Aquesta metodologia és ideal per aquest treball, ja que el seu abast pot créixer indefinidament, i cal un sistema que permeti flexibilitat i adaptació en funció del temps disponible.

### 1.5.1 Eines

Per facilitar el seguiment dels objectius, s'utilitzarà Trello, una eina en línia que permet establir tasques en forma de targetes, llistes i taulets virtuals. L'objectiu és fer servir un tauler específic per al projecte, on cada llista representi el conjunt de tasques a completar en una setmana, i cada targeta sigui una tasca concreta a realitzar.

El control de versions és una eina fonamental en qualsevol projecte informàtic, ja que permet fer un seguiment dels canvis realitzats i actua com a còpia de seguretat. Per aquest motiu, s'utilitzarà GitHub com a plataforma de control de versions, on es crearà un repositori específic per al projecte. Tots els canvis es pujaran al repositori en finalitzar cada tasca, facilitant així la traçabilitat i gestió del codi.

Finalment, per a la planificació de les tasques, s'utilitzarà Ganttter, una eina que permet la creació de diagrames de Gantt per fer un seguiment del temps dedicat a cada activitat. En les reunions setmanals, es revisarà el temps emprat en cada

tasca i s'actualitzarà el diagrama en conseqüència, assegurant així que el projecte avança dins els terminis establerts.

### **1.5.2 Seguiment**

Pel que fa a la validació del sistema, cada vegada que es completi una funcionalitat, es realitzaran proves pilot amb l'objectiu d'identificar ràpidament problemes i optimitzar el procés abans d'afegir noves funcionalitats. Aquestes proves seran fonamentals per garantir que el compilador funciona correctament i que les instruccions generades són compatibles amb l'entorn d'execució del simulador.

A més a més es realitzaran reunions setmanals amb el director del projecte per revisar els objectius assolits, identificar possibles bloquejos i redefinir la planificació segons l'estat del projecte. Aquestes reunions també serviran per establir noves tasques i assegurar-se que el desenvolupament avança de manera progressiva i efectiva.

## **1.6 Planificació temporal**

Amb l'objectiu de finalitzar aquest treball de fi de grau en la data estimada, i complir amb els objectius plantejats prèviament, en aquesta secció es realitza una planificació temporal del projecte dividida en tasques.

El treball comença el dia 2 de febrer de 2025 i la seva finalització està prevista per al dia 25 de juny de 2025. En total, el desenvolupament del projecte es durà a terme al llarg de 144 dies aproximadament i amb una durada estimada de 510 hores.

### **1.6.1 Descripció de les tasques**

#### **1. GEP – Gestió de projectes**

La gestió del projecte és essencial per planificar, definir i documentar la feina a realitzar. A més, inclou les reunions per a la validació i proposta d'objectius setmanals. S'estima que en total la gestió tindrà una durada de 135 hores.

#### **1.1. Abast**

Abans de començar amb el projecte és necessari delimitar el desenvolupament. Per aquest motiu, es dedica temps inicial a definir què es vol aconseguir amb la feina, què es desenvoluparà i quins mitjans seran necessaris. La durada és de 20 hores. A més, es contextualitzarà tot el projecte definint conceptes claus, entenent a qui va dirigit i l'estat de l'art.

## **1.2. Planificació**

Per complir amb els objectius proposats en la definició de l'abast del projecte, es realitza una planificació temporal, així com de recursos i requisits associats a cada tasca. A més, es defineixen els riscos i obstacles, i es plantegen tasques alternatives per solucionar-los. La realització d'aquesta fase requereix 10 hores.

## **1.3. Pressupost**

Es realitzarà un pressupost per quantificar el cost del projecte. Per això, s'in-dicaran partides per a cada tasca tenint en compte els costos de personal i equip, a més, es quantificaran els costos genèrics i les partides d'imprevistos. Atès que en el treball no es necessita ni material físic ni programes específics, es calcula una dedicació de 5 hores.

## **1.4. Informe de sostenibilitat**

S'analitzarà a partir d'un informe l'impacte mediambiental, econòmic i social del projecte, concretament, de les fases de planificació i desenvolupament. El temps estimat per realitzar l'informe és de 5 hores.

## **1.5. Entrega final**

En aquesta tasca s'agafarà tot el feedback donat pels tutors de l'assignatura gestió de projectes (GEP) per tal de millorar la part escrita. Per aconseguir que el treball quedi totalment polit, es calculen unes 10 hores.

## **1.6. Reunions**

La idea és fer una reunió virtual cada dues setmanes i igualment anar informant al tutor sobre com va el projecte, per tant no ocuparà més de 10h.

## **1.7. Documentació**

Aquesta és la part més robusta a nivell de gestió ja que s'ha d'explicar molt bé tot el treball, fet que portarà unes 50 hores.

## **1.8. Presentació**

La presentació també és una part que requereix molta atenció tant a l'hora de crear el guió, com per a memoritzar-lo i practicar-lo. A més a més s'haurà de crear suport visual i preparar-ho tot per a un funcionament excel · lent, per tant s'estima

que es trigarà unes 25 hores.

## **2. Treball previ**

Abans de començar qualsevol treball s'ha de realitzar una part teòrica, que avali que es tenen els coneixements necessaris per poder començar. En total seran 60 hores.

### **2.1. Lectura del treball**

La primera tasca constarà d'una lectura àmplia i profunda de la web del treball en què es basa aquest projecte, "Tetris is Capable of Universal Computation". Degut a la quantitat d'informació s'estimen unes 15h.

### **2.2. Comprensió del treball**

Un cop llegit el treball, s'haurà d'interioritzar com són l'assembler i el simulador creats pel desenvolupador, això requereix d'unes 10 hores per a cada un.

### **2.3. Primer Contacte**

Per confirmar que s'ha entès correctament com funciona tot, s'utilitzaran 5 hores per provar de forma pràctica certes funcionalitats.

### **2.4. Comprensió de com construir un compilador amb Python**

Aquesta tasca és independent de les altres, i consta de refreshar conceptes i entendre del tot com es fa un compilador amb Python. No s'extendrà més de 10 hores.

### **2.5. Primera Demo (Hello World)**

Acabades les tasques 2.3 i 2.4 es farà una demo per comprovar que tot vagi bé i perquè sigui una base sòlida a partir de la qual començar a construir el compilador. Es calcula que durarà unes 20h ja que probablement sorgiran molts errors de software.

## **3. Treballar**

Aquesta part forma el gruix de treball, ja que conté tota la realització del compilador. Temps estimat de 255h.

### **3.1. Modificacions del simulador**

Donat que el simulador està bruteforcejat per només permetre programar mostrant la graella del Tetris, si és possible s'hauran de fer les modificacions necessàries per aconseguir quelcom de més general. També és probable que s'hagin de fer altres modificacions importants, fet que pot comportar unes 45h. Aquesta tasca depèn de la tasca 2.3.

### **3.2. Definició del llenguatge**

Una de les parts més importants és pensar i dissenyar el llenguatge en què l'usuari programarà. El llenguatge ha de ser molt simple i tenir les principals utilitats per programar. Per fer això amb unes 10h n'hi haurà prou.

### **3.3. Creació del compilador**

Es tracta de fer el nucli del treball, allò pel qual un programador es prepara durant molt de temps, el compilador total, esperant no superar les 180h. El nom de cada subtasca és suficient explicativa. Cada una tardarà 20h i altres 60h.

- **Variables**
- **Operacions**
- **Comparadors**
- **For**
- **If**
- **Funcions**
- **Altres**

## **4. Treball posterior**

Cal posar a prova el compilador i poder crear codi amb aquest. S'estimen unes 120h.

### **4.1. Creació d'interfícies**

Com que tota la part d'interfícies es fa de forma externa, caldrà predisenyar alguns sistemes bàsics per tal que la gent menys avisada pugui fer alguna cosa. De moment les que es faràn serà una graella del Tetris on es podrà dibuixar, un panell amb números com entrada/sortida i un sistema de reconeixement de patrons en una gramàtica. Tot i així se'n faran els màxims possibles amb les 50h que s'hi dedicaran.

## 4.2. Creació de codi

Per tal de generar alguns exemples a seguir es programaran diferents codis on es veurà tot el potencial que té el Tetris com a computador. També se'n faran els màxims en 50h.

## 4.3. Polidesa

Probablement durant tot el projecte hi haurà petites coses que no seran òptimes del tot o que s'hauran deixat per millorar al final, per això aquestes 20h es dedicaran a testejar i polir les diferents ineficiències.

### 1.6.2 Recursos

**Recursos humans** En aquest projecte es poden definir 4 rols diferents: el cap de projecte, el programador, el tester i l'influencer. Malgrat això, com que aquest treball de fi de grau és realitzat per una sola persona, serà l'autor qui assumirà cadascun dels rols en funció de la tasca que es realitzi en cada moment. A la taula es poden veure les tasques assignades a cadascun dels rols.

A continuació es detalla la responsabilitat de cada rol.

- Cap de projecte: s'encarrega de la planificació del projecte, de liderar les reunions amb l'equip i d'escriure la documentació.
- Programador: és la persona encarregada de dissenyar el llenguatge i crear el compilador.
- Tester: s'ocupa de comprovar que no hi hagi errors i, si n'hi ha, de reportar-los per a poder solventar-los.
- Influencer: en aquest rol recau la responsabilitat de difondre el treball i exposar-lo al públic.

**Recursos materials** Com s'ha indicat anteriorment, es necessiten 2 tipus diferents de softwares, un per treballar i l'altre per organitzar.

El primer tipus inclou els llenguatges de programació (assembler, Python i Java), el sistema operatiu Windows i l'eina ANTLR4.

Els recursos que s'utilitzen per organitzar el projecte són Trello, Ganttter, Github i TeXworks. També és necessari un portàtil per executar tot el software.

**Recursos generals** Els recursos generals inclouen aquells elements necessaris per a la realització de qualsevol projecte, no tan sols per a projectes informàtics. Cal comptar doncs amb internet, un lloc físic on treballar (com un despatx) i disposar d'energia elèctrica.

### **1.6.3 Gestió de risc**

Els riscos són múltiples quan es desenvolupa un projecte com aquest. El més important i preocupant és la impossibilitat a l'hora de poder modificar el treball si no s'és capaç d'assolir el repte que suposa, ja sigui perquè la capacitat de l'autor no és suficient o perquè la tasca és realment costosa a nivell de temps. La solució principal a això serà reformular l'abast del projecte o àdhuc canviar-lo del tot.

L'altre risc principal és no disposar de temps suficient. Tot i haver planificat molt la gestió del temps, és possible que no s'ajusti perfectament a la realitat. És per aquest motiu que s'ha organitzat de tal manera que es disposa de 40 hores extres com a coixí, per si fan falta.

Els riscos continuen si es contemplen possibles errors i problemes de software, molt relacionats amb el risc anterior. Qualsevol problema a l'hora de programar el compilador es podria solucionar amb temps infinit, però com que això és totalment impossible, la solució seria demanar ajuda a actors externs.

Per últim cal tenir en compte el risc que el rendiment del programa sigui insuficient. Caldria aleshores buscar optimitzacions per millorar el rendiment o prescindir de certes funcionalitats. També hi ha la possibilitat que la computadora s'espantlli i se n'hagi de comprar una de nova.

#### 1.6.4 Gantt

	Nombre	Duración	Inicio	Fin	Predecesoras	Recursos
1	1.1	20horas	03/04/2025	03/06/2025		TeXworks,Portàtil
2	1.2	10horas	03/06/2025	03/07/2025	1	TeXworks,Portàtil
3	1.3	5horas	03/10/2025	03/10/2025	1	TeXworks,Portàtil
4	1.4	5horas	03/10/2025	03/10/2025	1	TeXworks,Portàtil
5	1.5	10horas	03/11/2025	03/12/2025	2,3,4	TeXworks,Portàtil
6	1.6	10horas	03/12/2025	03/13/2025		
7	1.7	50horas	05/20/2025	05/28/2025	19	TeXworks,Portàtil
8	1.8	25horas	05/28/2025	06/02/2025	7	Portàtil,Eina de presentacions
9	2.1	15horas	03/13/2025	03/14/2025		Portàtil
10	2.2	10horas	03/17/2025	03/18/2025	9	Portàtil,Assembler
11	2.3	5horas	03/18/2025	03/18/2025	10	Portàtil,Java,Assembler
12	2.4	10horas	03/19/2025	03/20/2025		Portàtil,Python,ANTLR4
13	2.5	20horas	03/24/2025	03/26/2025	11,12	Python,Portàtil,ANTLR4,Assembler
14	3.1	45horas	03/26/2025	04/02/2025		Portàtil,Java
15	3.2	10horas	04/02/2025	04/03/2025	11	
16	3.3.1	50horas	04/03/2025	04/11/2025	15	Portàtil,Python,ANTLR4
17	3.3.2	50horas	04/11/2025	04/21/2025	16	Portàtil,Python,ANTLR4
18	3.3.3	50horas	04/21/2025	04/29/2025	17	Portàtil,Python,ANTLR4
19	3.3.4	50horas	04/30/2025	05/08/2025	18	Portàtil,Python,ANTLR4,Assembler
20	4.1	20horas	05/09/2025	05/13/2025		Portàtil,Java,Assembler
21	4.2	20horas	05/13/2025	05/15/2025	19	Portàtil,Xarxes Socials
22	4.3	30horas	05/15/2025	05/20/2025	19	Portàtil

Figura 5: Taula on apareixen totes les tasques amb la seva informació. *Elaboració pròpia.*

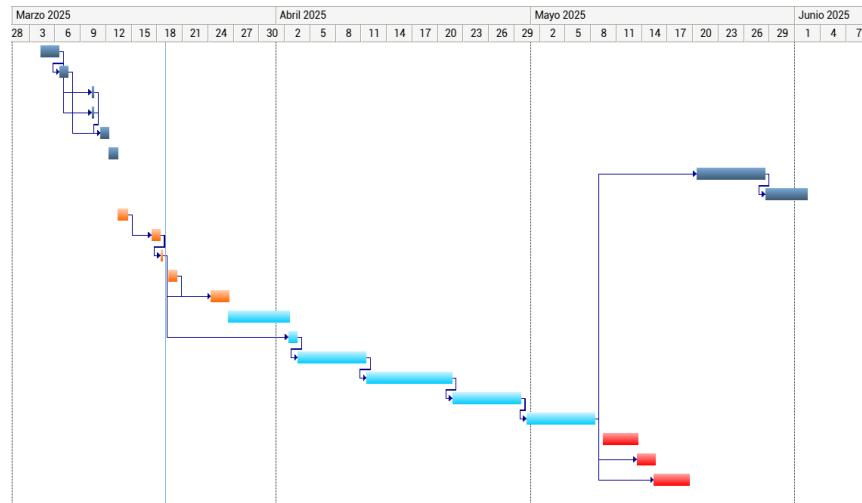


Figura 6: Gantt del projecte, creat amb Gantter. *Elaboració pròpia.*

## 1.7 Gestió econòmica

Un cop realitzada la planificació temporal del projecte, s'estimen els costos necessaris per al seu desenvolupament. S'identifiquen diferents tipus de costos associats al personal, a l'espai de treball i a les eines i dispositius utilitzats. A més, per superar els obstacles que puguin sorgir i assumir els costos no programats, es realitza un pla de contingència, una partida per a imprevistos i s'exposen mecanismes per controlar el pressupost.

### 1.7.1 Pressupost

**Costos de personal** A partir de la planificació per tasques, es calcula el cost de personal, tenint en compte els quatre rols definits anteriorment: cap de projecte (J), programador (P), influencer(I) i tester(T). A la taula es pot veure el cost per hora de cada rol.

Rol	Cost per hora
Cap de projecte	20€/h
Programador	12€/h
Influencer	18€/h
Tester	12€/h

Taula 1: Costos de personal, en base al BOE. *Elaboració pròpia.*

Tasca	Hores	Personal	preu/hora	total	Més SS
1.1	20	J	20	400,00 €	520,00 €
1.2	10	J	20	200,00 €	260,00 €
1.3	5	J	20	100,00 €	130,00 €
1.4	5	J	20	100,00 €	130,00 €
1.5	10	J	20	200,00 €	260,00 €
1.6	10	J+P+T	44	440,00 €	572,00 €
1.7	50	J	20	1.000,00 €	1.300,00 €
1.8	25	J	20	500,00 €	650,00 €
2.1	15	P	12	180,00 €	234,00 €
2.2	10	P	12	120,00 €	156,00 €
2.3	5	P	12	60,00 €	78,00 €
2.4	10	P	12	120,00 €	156,00 €
2.5	20	P+T	24	480,00 €	624,00 €
3.1	45	P	12	540,00 €	702,00 €
3.2	10	P	12	120,00 €	156,00 €
3.3.1	50	P	12	600,00 €	780,00 €
3.3.2	50	P	12	600,00 €	780,00 €
3.3.3	50	P	12	600,00 €	780,00 €
3.3.4	50	P	12	600,00 €	780,00 €
4.1	20	P	12	240,00 €	312,00 €
4.2	20	I	18	360,00 €	468,00 €
4.3	30	T + P	24	720,00 €	936,00 €
				8.280,00 €	10.764,00 €

Figura 7: Cost del personal. *Elaboració pròpia.*

Com es pot observar el cost total serà de 8280 euros, quantitat que es multiplica per 1,3 per tenir en compte els costos de la Seguretat Social. El resultat és de

10764 euros.

**Costos generals** Donat que tot el software que s'utilitza és gratuït, els costos vindran dels utensilis físics necessaris per treballar.

El lloc de treball és la casa de l'autor, per tant, el cost surt de multiplicar 400€/mes (el cost de llogar una habitació per la zona) \* 4 mesos de treball, amb un total de 1600€.

La mobilitat del personal per anar a les reunions té el cost derivat del transport públic. La T-jove té un cost de 44€, que fan un total de 132€, comptant les 3 persones que es reuneixen.

Com que la reunió té lloc al despatx del cap de projectes, no se'n deriva cap cost per al projecte.

Cal afegir l'energia necessària, la qual aproximem que seran 100 kWh consumits pel portàtil. Si assumim que el KWh costa 20 cèntims, obtindrem el cost de 20€.

Finalment cal calcular els costos d'amortització del portàtil, suposant una vida útil de 4 anys i que el portàtil costa 1200€. Tenint en compte que un any té 220 dies hàbils i la jornada de treball és de 8h/dia, el cost per hora es calcula com cost dispositiu / (vida útil \* 220 \* 8). El resultat seria de 87€.

La suma total dels costos generals ascendeix a 1839 euros.

**Contingència i cost total** En tot projecte és important afegir un sobre-cost per cobrir imprevistos. Donat que el projecte no té gaire risc s'han incrementat els costos en un 10 %; per tant, sumant-ho tot el cost total serà de 13.863€.

**Imprevistos** Per últim es calcula el cost dels obstacles que poden sorgir durant el desenvolupament del projecte, que ja s'han descrit a l'apartat de gestió de risc.

- Haver d'incrementar el temps de treball en 40h extres del programador. Això suposaria 480€. La probabilitat es considera del 20% perquè el risc és alt.
- Si cal demanar ajuda a un professional extern, el cost s'estima en 400€, però la probabilitat estimada és del 2%.
- En cas que la computadora s'espatlli, el cost de comprar-ne una de nova seria de 1200€, però és molt poc probable.

**Cost total** A continuació es resumeixen els costos, podent calcular que el cost total serà de 15943€.

### 1.7.2 Control de gestió

Després d'haver estimat els costos, cal definir els mecanismes necessaris per tal de controlar les desviacions que puguin aparèixer en relació amb el pressupost inicial,

Despesa	Cost
Personal	10764€
General	1839€
Contingència	1260€
Imprevistos	2080€
<b>Total</b>	<b>15943€</b>

Taula 2: Costos *Elaboració pròpia.*

així com els indicadors numèrics de càcul de les desviacions.

Quinzenalment s'actualitzarà el pressupost amb les hores reals de les tasques i es compararà amb les hores estimades.

També s'anotaran les despeses extres que es vagin produint per tal de controlar els imprevistos i s'aniran comparant amb la previsió d'imprevistos i contingència. D'aquesta manera es podrà detectar qualsevol desviació i predir si és necessari augmentar el pressupost.

A continuació es defineixen els càlculs dels indicadors numèrics per control de desviacions.

- Desviació del cost personal per tasca:

$$(\text{cost estimat} - \text{cost real}) * \text{hores reals}$$

- Desviació de la realització de tasques:

$$(\text{hores estimades} - \text{hores reals}) * \text{cost real}$$

- Desviació total en la realització de tasques:

$$\text{cost estimat total} - \text{cost real total}$$

- Desviació total de recursos (software, hardware, espai o personal):

$$\text{cost estimat total} - \text{cost real total}$$

- Desviació total del cost d'imprevistos:

$$\text{cost estimat imprevistos} - \text{cost real imprevistos}$$

- Desviació total d'hores:

$$\text{hores estimades} - \text{hores reals}$$

## 1.8 Sostenibilitat

En qualsevol projecte és important realitzar una anàlisi de sostenibilitat tenint en compte tres dimensions: l'econòmica, l'ambiental i la social.

A més, l'autor del TFG realitza a continuació una autoavaluació sobre el domini de la competència de la sostenibilitat.

### 1.8.1 Autoavaluació

M'he adonat que no era conscient del meu grau de coneixement entorn al tema de la sostenibilitat. Sempre he tingut present aquest concepte a la meva vida, però mai m'havia parat a reflexionar sobre la profunditat del que realment sabia. Això es deu, en gran part, a la meva criança, ja que des de petit m'han inculcat valors relacionats amb el respecte pel medi ambient. A casa sempre s'ha donat molta importància al reciclatge i a l'economia circular, fent que aquestes pràctiques fossin una rutina diària. Aquest entorn m'ha proporcionat unes nocions bàsiques que han estat fonamentals per al meu desenvolupament en aquest àmbit.

D'altra banda, la meva curiositat pel món de l'empresa m'ha portat a coneixer conceptes com el DAFO i altres eines estratègiques que es poden aplicar també a la sostenibilitat. Això m'ha permès entendre com les empreses poden contribuir a un món més sostenible mitjançant pràctiques responsables i innovadores. No obstant això, tot i tenir aquests coneixements previs, m'he quedat meravellat en descobrir la quantitat de coses que encara no sabia sobre aquest tema.

Hi ha moltes accions i iniciatives que es poden dur a terme per reduir l'impacte ambiental i fomentar un futur més sostenible. Aspectes com l'eficiència energètica, el consum responsable o les polítiques de reducció de residus són àmbits en els quals encara em queda molt per aprendre. Aquest procés de descoberta m'ha fet veure que la sostenibilitat no és només una qüestió de reciclatge, sinó un conjunt de pràctiques i decisions que poden transformar el nostre entorn i garantir un futur millor per a tothom.

### 1.8.2 Dimensió Econòmica

Realment és un projecte bastant car i res rendible, ja que no s'espera obtenir cap retorn, de moment, d'aquest. Molts d'aquests tipus de projectes a vegades estan finançats per centres de recerca o universitats, però el més comú és que siguin generats per la pròpia comunitat de forma lúdica en el seu temps lliure. Això és possible ja que la major part de la inversió necessària és en forma de temps, llavors la gent apassionada busca espais per poder generar i cobrir la seva curiositat. La meva solució a l'hora de computar programes, és molt menys eficient que la que hi ha al mercat actualment, per tant, serà més costosa a nivell de temps i energia.

### **1.8.3 Dimensió Ambiental**

En ser una forma lenta d'executar codi, requereix de més energia que de manera convencional sense obtenir un avantatge concret, i per tant tindrà un impacte ambiental negatiu. Tot i així en el moment de creació no té gaire impacte ambiental; així que si no s'utilitza, millor per la natura. El meu treball és com un article de luxe, ja que gastes innecessàriament recursos simplement pel fet de poder dir que ho has compilat amb el Tetris.

### **1.8.4 Dimensió Social**

Crec que aquest projecte m'ajudarà a consolidar conceptes i crec que és ideal per acabar la carrera, ja que toco idees de moltes assignatures i podré demostrar i repassar de forma pràctica tot el que he après. Tot i que no existeix una necessitat real del projecte, m'he proposat dedicar esforços en fer que sigui el màxim didàctic i divulgatiu possible, així que en certa manera serà contingut educatiu de qualitat i divertit.

## **2 Base Teòrica**

### **2.1 Resum del treball de Meat Fighter**

Aquest apartat conté un resum del treball desenvolupat per Meat Fighter, on s'introdueixen els conceptes bàsics necessaris per comprendre el projecte en la seva totalitat. Tot i això, es recomana la lectura del treball original, ja que inclou una explicació molt més detallada i és, sens dubte, un projecte que mereix ser examinat amb deteniment. S'intentarà presentar el contingut de la manera més senzilla i comprensible possible.

#### **2.1.1 Principis bàsics**

El primer concepte que cal entendre a l'hora de programar utilitzant el Tetris és que no s'utilitza una graella estàndard, sinó que aquesta ha estat substituïda per una graella infinita tant en alçada com en amplada. No obstant això, l'eix Y no pot prendre valors negatius; per tant, existeix un límit inferior que fa la funció de terra, sobre el qual es col·loquen les peces.

Les peces apareixen a una Y infinita i centrades respecte a l'eix X, és a dir, a la posició  $X = 0$ . Les accions definides per al jugador es basen en les d'un joc de Tetris convencional, que són les següents:

- Caiguda ràpida (Hard Drop): Permet moure la peça des de la posició inicial ( $Y$  infinita) fins a la superfície de col·locació. Si existeix una altra peça que impedeix el moviment, la nova peça es col·locarà just al damunt.

- Moviment lateral: Es tracta del moviment típic del joc Tetris, que permet desplaçar la peça horitzontalment per col·locar-la a la columna desitjada. Com que no hi ha límit temporal per a la caiguda, el jugador pot decidir amb precisió on situar-la i es pot arribar fins X molt elevades.
- Rotació: La peça gira seguint un patró predefinit Per norma general, les rotacions de les peces no s'utilitzen excepte al començament de la seva aparició. Per aquest motiu, es pot considerar cada possible rotació d'una peça com una entitat diferenciada, identificada pel nom que es mostra a la figura anterior.

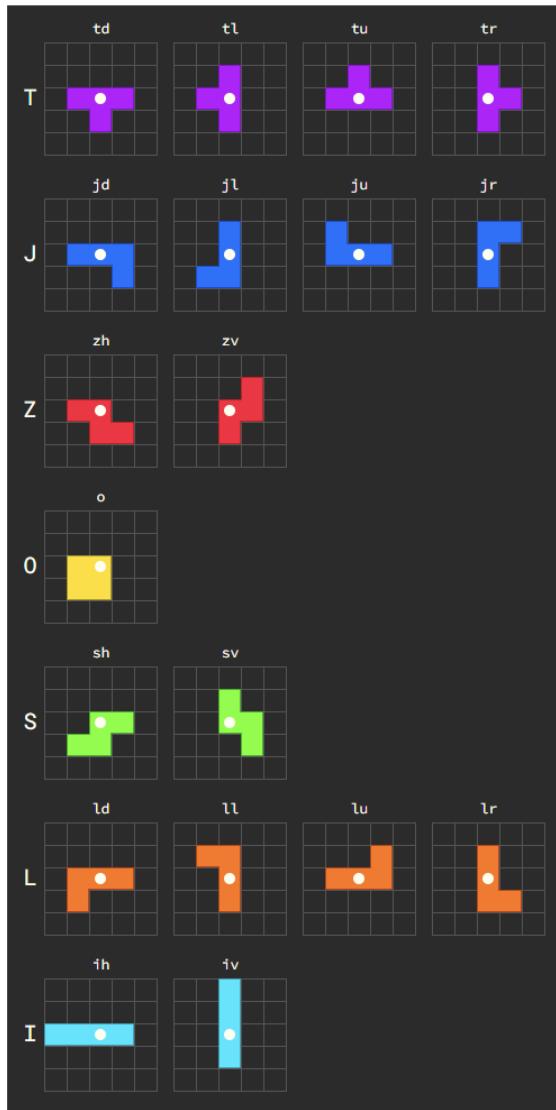


Figura 8: Rotacions dels tetròminos respecte el seu centre marcat amb un punt blanc. Cada rotació de cada peça té un nom. (*Treball Meat Fighter*)

- Semi-Hard Drop: Aquesta acció permet moure una peça des de la posició

inicial a Y infinita fins a una alçada Y determinada. Tot i això, la peça no quedarà col·locada en aquesta posició, excepte si troba un bloc que li impedeixi continuar descendint; en aquest cas, s'hi col·locarà a sobre. Això permet que la peça pugui ser objecte d'una altra acció posterior, simulant una caiguda automàtica fins a la Y desitjada.

Així, es pot representar una partida típica de Tetris (i per extensió, un joc amb una graella infinita) com un conjunt de sentències, on cada sentència inclou una peça i les accions aplicades sobre aquesta.

### 2.1.2 TetrominoScript

Com que el control del joc no el duu a terme una persona sinó un sistema automatitzat, s'ha desenvolupat un llenguatge d'accions que un agent automàtic pot executar. Aquest llenguatge s'anomena TetrominoScript (TS), i permet una conversió directa a un joc de Tetris real, sempre que es compleixin certes suposicions.

#### Instruccions TetrominoScript

A continuació, s'analitzen els tres tipus d'instruccions possibles:

##### Instrucció $m x_h$

Aquesta expressió permet col·locar una entitat concreta  $m$ , que indica la peça i la rotació que es vol utilitzar tal i com s'ha vist a la figura 8, a la posició horitzontal  $x_h$ . Aquesta acció és possible perquè es considera que es poden descartar peces (per exemple, mouent-les cap a l'esquerra i aplicant *Hard Drop*) fins a obtenir el tipus de peça desitjat. Un cop apareix la peça indicada, es rota segons calgui, es desplaça horitzontalment fins a  $x_h$ , i finalment s'executa la caiguda ràpida.

##### Instrucció $m x_s y_s x_h$

Es tracta d'una extensió de l'anterior, on es defineix primer una posició  $(x_s, y_s)$  en la qual es fa un *SemiHard Drop*, simulant la caiguda de la peça fins a certa altura. Des d'allà, es mou fins a  $x_h$  i es col·loca definitivament amb *Hard Drop*.

Aquest tipus d'instrucció resulta especialment útil per controlar amb precisió el comportament dinàmic de les peces abans de la col·locació, cosa que posteriorment serà fonamental pel projecte.

##### Instrucció $s x y$

Aquesta instrucció permet modularitzar el comportament mitjançant la reutilització d'scripts. L'script anomenat  $s$  s'executa com si el seu punt d'origen  $(0, 0)$

fos traslladat a les coordenades  $(x, y)$ . Això facilita l'organització, manteniment i reutilització de seqüències complexes.

Una qüestió important és com aconseguir la peça desitjada en cada moment. En aquest cas concret, gràcies a disposar d'un *playground* infinit, es poden llençar les peces que no interessen col·locant-les a la columna  $X = -2$ . Aquesta posició no interfereix en la resta del sistema, ja que mai no es farà ús de coordenades  $X$  negatives a l'hora de col·locar peces útils, per tant hem creat una brossa artificial.

Com que el sistema disposa d'infinites coordenades  $X$ , mai es completarà una línia, i com que tampoc hi ha cap límit superior, mai es finalitzarà la partida. Això assegura una execució contínua del sistema.

### Gestió d'informació a TetrominoScript

Cal entendre ara el model de representació de la informació dins d'aquest entorn. El sistema es basa en la codificació mitjançant l'altura d'una estructura. Per tant, els inputs es veuen tal que així:

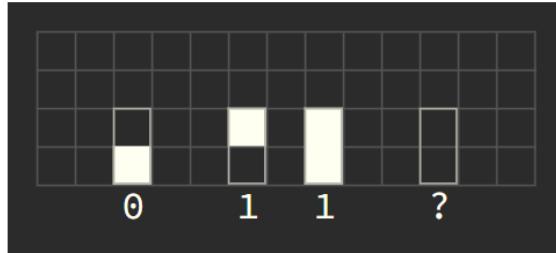


Figura 9: Possibles inputs d'un sistema. (*Treball Meat Fighter*)

Es defineix que una cel·la de color blanc indica la presència d'un bloc, i, en conseqüència, cap peça pot travessar aquesta posició: quedarà col·locada just a sobre.

Tot i que els inputs es poden generar fàcilment mitjançant peces del Tetris, normalment els primers es posen manualment. Aquests es representen mitjançant un rectangle de dues unitats d'alçada. La codificació d'un bit dins d'aquest rectangle és la següent:

- Si la cel·la superior està ocupada, es considera que el valor del bit és 1.
- Si només està ocupada la cel·la inferior, es considera que el valor és 0.

Els outputs segueixen una estructura molt similar. La diferència principal és que aquests no apareixen inicialment pintats. S'observen al final de l'execució per comprovar si han estat modificats o no, en funció de les peces col·locades. Aquest

comportament permet encadenar fàcilment diversos blocs d'entrada i sortida, creant així funcions compostes dins del sistema.

TetrominoScript incorpora una sintaxi específica per a la definició d'entrades i sortides. Aquesta es basa en la següent estructura:

**in**  $n$   $x_0$   $y_0$   $x_1$   $y_1$  ...  $x_{N-1}$   $y_{N-1}$  (1)

**out**  $n$   $x_0$   $y_0$   $x_1$   $y_1$  ...  $x_{N-1}$   $y_{N-1}$  (2)

On:

- **in** o **out** indiquen el tipus de node (entrada o sortida).
- $n$  és el nom identificador del node.
- $(x_i, y_i)$  són les coordenades de cada bit dins del node.

Ara bé, com es pot transformar una seqüència d'entrades binàries en una seqüència de sortides amb el valor desitjat?

La clau del funcionament rau en el control de l'**alçada**. Tant els inputs com els outputs es basen únicament en aquesta característica. Si es pot controlar l'alçada a la qual es col·loca una peça, llavors es pot controlar la informació binària codificada al sistema.

Aquest control es realitza aprofitant una mecànica fonamental del Tetris: si s'intenta moure una peça lateralment cap a una posició ja ocupada per un bloc, el moviment no és possible. Així doncs, es pot provocar un comportament condicional: es fa que una peça caigui amb *Semi-Hard Drop* fins a una altura determinada i després s'intenta desplaçar-la lateralment. Si una cel·la representa un 0, aquest moviment serà possible i per tant la alçada de la estructura augmentarà. En canvi, si la cel·la ja conté un 1, aquest moviment lateral serà bloquejat i l'estructura no podrà canviar tota l'alçada, que la peça li permetria.

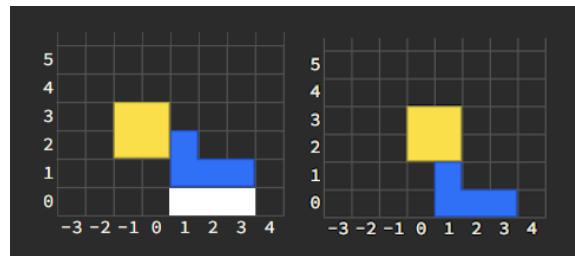


Figura 10: La peça O fa semi-hard drop fins  $Y = 2$  i intenta anar cap a la dreta. En la situació de l'esquerra no pot, ja que hi ha el bloc blanc representant l'input = 1 que provoca que la peça J estigui elevada, i per tant la peça O continuarà caient. En el cas de la dreta sí que pot moure's lateralment, cosa que canvia la X de la peça i que provoca que l'estructura final tingui més altura que l'altra. (Creació pròpia)

Això implica que amb el mateix codi es poden aconseguir resultats diferents en funció del valor d'entrada. D'aquesta manera, s'aconsegueix que dos inputs diferents generin dos outputs diferents utilitzant exactament la mateixa seqüència de TetrominoScript.

Finalment, aquest sistema pot escalar-se. És possible definir que l'alçada resultant depengui de més d'un input, i per tant, es poden construir funcions complexes amb múltiples entrades que controlin de forma conjunta l'alçada de sortida, i en conseqüència, el valor codificat a l'output.

### Portes lògiques amb TetrominoScript

#### Porta NOT

Una de les formes més simples i il·lustratives per entendre el funcionament computacional d'aquest sistema és mitjançant la implementació d'una porta lògica bàsica, com la porta NOT, que transforma un 0 en un 1 i viceversa.

A continuació s'analitza el codi de la porta `not.t`, que implementa aquest comportament:

#### `not.t`

```
lu 0
o 0 4 1
jd 0

in i -1..1 0
out o -1..1 5
```

En primer lloc, cau una peça de tipus L amb la rotació U. Aquesta peça actua com a element fixador de l'estructura base. Tot seguit, es col·loca una peça de tipus quadrat (0), que s'atura a l'alçada  $Y = 4$  i intenta desplaçar-se cap a la dreta.

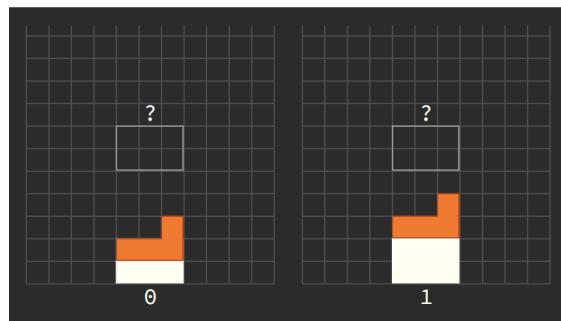


Figura 11: Col·locació peça L. A l'esquerra hi ha el cas de tenir com a input un 0 i a la dreta un 1. (*Treball Meat Fighter*)

Aquest moviment depèn del valor de l'input:

- Si l'input és 0, la peça taronja no bloqueja el moviment, de manera que la peça quadrada es desplaça correctament i s'eleva a una altura més gran. El resultat és una suma de dues unitats a l'alçada total.
- Si, en canvi, l'input és 1, la peça 0 no pot moure's lateralment, ja que la presència d'un bloc anterior ho impedeix. En aquest cas, la peça continua caient i només s'afegeix una unitat d'alçada a l'estructura. A més, es deixa un forat a la part dreta.

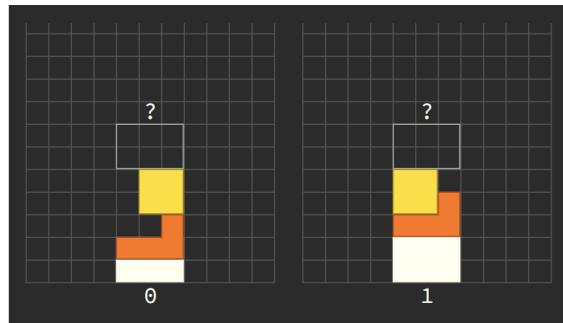


Figura 12: Col·locació peça O. (*Treball Meat Fighter*)

Aquest canvi d'alçada condiciona la col·locació de la peça següent, una J, que es posiciona una unitat més amunt en el cas en què l'input era 0. Aquest desfasament en l'alçada reflecteix el comportament esperat d'una porta NOT.

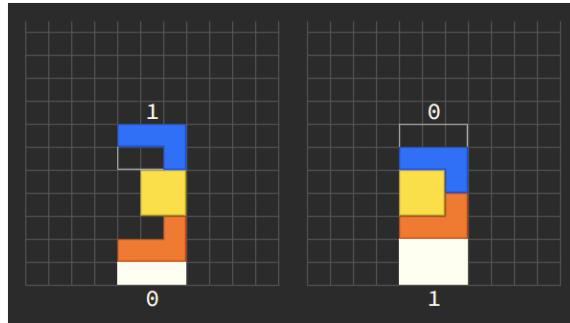


Figura 13: Resolució de l'estructura amb la peça J. (*Treball Meat Fighter*)

Tot i que existeixen moltes altres formes de construir aquesta porta, aquesta implementació destaca per la seva elegància i simplicitat.

### Porta NAND

També és possible construir portes amb múltiples entrades. Un exemple típic és la porta NAND. En aquest cas, s'aplica primer una negació als dos inputs mitjançant estructures similars a la porta NOT. A continuació, s'utilitza una peça llarga (I) per combinar els resultats.

Aquesta peça es col·loca a una alçada superior si almenys un dels inputs és 1 (ja que s'han aplicat negacions prèvies, això equival a que almenys un dels inputs originals era 0). Si ambdós inputs són 1, la peça cau més avall, representant un 0.

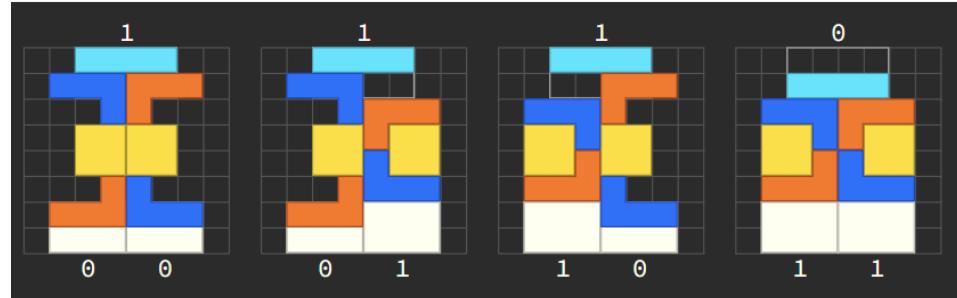


Figura 14: Les 4 combinacions de valors de la porta nand. (*Treball Meat Fighter*)

A partir d'aquestes portes bàsiques es poden construir totes les altres operacions lògiques. La combinació d'aquestes estructures permet la creació de circuits lògics complexos, com ara sumadors (*ADD*), multiplexors i altres components fonamentals en l'arquitectura computacional.

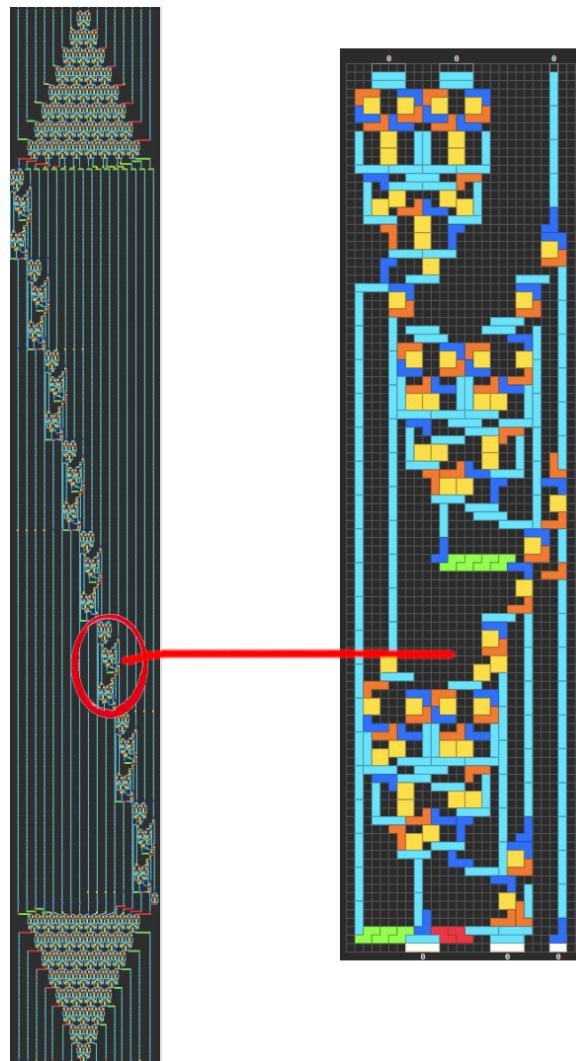


Figura 15: Component ADD fet amb tetròminos. (*Treball Meat Fighter*)

Aquestes representacions visuals serveixen per mostrar les dimensions i col·locacions relatives de les peces durant una operació elemental de suma binària, la qual es construeix encadenant diverses portes lògiques com les descrites anteriorment.

Function	Definition
CLEAR	$A \mapsto 0$
DEC	$A \mapsto A - 1$
DEC_16	$W \mapsto W - 1$
IDENTITY	$A \mapsto A$
INC	$A \mapsto A + 1$
INC_16	$W \mapsto W + 1$
LS2	$A \mapsto A \ll 2$
LS3	$A \mapsto A \ll 3$
LS4	$A \mapsto A \ll 4$
RS1	$A \mapsto A \ggg 1$
RS5	$A \mapsto A \ggg 5$
ADD_AB_FB	$[A, B] \mapsto [A + B, B]$
AND_AB_AF	$[A, B] \mapsto [A, A \& B]$
AND_AB_FB	$[A, B] \mapsto [A \& B, B]$
AND_NOT_AB_FB	$[D, E] \mapsto [D, D \text{ and not } E]$
COPY_A_B	$[A, B] \mapsto [A, A]$
COPY_B_A	$[A, B] \mapsto [B, B]$
INC_16_C	$[W, C] \mapsto [C ? W + 1 : W, C]$
OR_AB_FB	$[A, B] \mapsto [A   B, B]$
SUB_AB_FB	$[A, B] \mapsto [A - B, B]$
SWAP	$[A, B] \mapsto [B, A]$
XOR_AB_FB	$[A, B] \mapsto [A ^ B, B]$
AND_A_B_C	$[D, E, C] \mapsto [D, E, D \text{ and } E]$
C_AND_A_NOT_B	$[C, D, E] \mapsto [D \text{ and not } E, D, E]$
CMP_C	$[A, B, C] \mapsto [A, B, A = B]$
CMP_AND_C	$[A, B, C] \mapsto [A, B, (A = B) \text{ and } C]$
C_CMP	$[C, A, B] \mapsto [A = B, A, B]$
COPY_A_B_C	$[A, B, C] \mapsto [A, C ? A : B, C]$
COPY_B_A_C	$[A, B, C] \mapsto [C ? B : A, B, C]$
C_COPY_B_A	$[C, A, B] \mapsto [C, C ? B : A, B]$
C_COPY_A_B	$[C, A, B] \mapsto [C, A, C ? A : B]$

Figura 16: Catàleg de les funcions desenvolupades amb peces del Tetris. (*Treball Meat Fighter*)

## Memòria

La memòria dins del sistema es representa mitjançant words d'un byte, és a dir, conjunts de 8 bits. Cada bit continua utilitzant el mateix sistema d'*input/output* prèviament establert: un bloc 0 representa un 1 i un bloc I representa un 0.



Figura 17: Distribució d'un registre. (*Treball Meat Fighter*)

Els bytes (registres) es distribueixen uniformement sobre l'eix *X* del *playground* per garantir la coherència estructural. En canvi, l'eix *Y* representa la temporalitat. Això implica que qualsevol operació aplicada a registres incrementarà la seva alçada en l'eix *Y*, reflectint una nova *instància temporal* del registre.

Quan s'aplica una funció entre dos registres, aquesta s'implementa entre les seves respectives posicions. Tots els altres registres, per tal de no perdre sincronització temporal, reben una funció identitat, consistent bàsicament en la col·locació de peces I per conservar l'alçada.

Aquesta estratègia permet que, quan sigui necessari un nou registre, aquest es construeixi des del terra i se li apliquin successivament funcions identitat fins a assolir l'alçada temporal actual del programa. Això evita la creació innecessària de registres que mai s'utilitzaran.

És important destacar que les funcions només poden aplicar-se entre registres consecutius. Si això no es compleix, no es podrien aplicar funcions identitat adequadament i es trencaria la coherència en l'eix temporal. Per solucionar-ho, s'utilitza l'operació **SWAP**, que permet intercanviar registres i col·locar-los en la posició desitjada.

	0A	01	02	03	04
9					
8		SWAP	↑	↑	↑
7	01	0A	02	03	04
6	↑		SWAP	↑	↑
5	01	02	0A	03	04
4	↑	↑	↑		ADD_AB_FB
3	01	02	03	06	04
2	↑	↑	↑	SWAP	↑
1	01	02	03	03	04
0	00	01	02	03	04
	0	1	2	3	4

Figura 18: Execució d'un sumatori utilitzant funcions fetes amb tetròminos (*Treball Meat Fighter*)

A la Figura 18 es mostra un exemple de codi que suma els valors dels registres 1, 2, 3 i 4 al registre 0. Les fletxes verticals representen les funcions identitat que conserven l'alçada dels registres no implicats.

### Memory Code (MC)

Amb l'objectiu de facilitar la manipulació d'aquesta memòria funcional, s'ha creat un llenguatge específic anomenat **Memory Code (MC)**. Aquest llenguatge permet aplicar funcions sobre la memòria de forma senzilla i estructurada.

La seva sintaxi es basa en una única instrucció:

**f x**

On **f** representa una funció (o una composició de funcions definides prèviament) i **x** indica el primer registre sobre el qual s'aplica.

L'exemple anterior es veuria així:

```

sum.mc
COPY_B_A 0
ADD_AB_FB 1
SWAP 1
ADD_AB_FB 2
SWAP 2
ADD_AB_FB 3
SWAP 2
SWAP 1
SWAP 0

```

Aquest codi reflecteix una lògica seqüencial clara, on es fa servir la funció ADD\_AB\_FB per combinar registres, i l'ús de SWAP per mantenir els registres en posicions consecutives.

Es pot observar que qualsevol codi en MC es pot transformar en TetrominoScript (TS), ja que MC és simplement una versió estructurada i a gran escala de TS. Aquesta correspondència facilita la modularització del sistema i permet reutilitzar components ja implementats, millorant l'escalabilitat del projecte.

### Màquina de propòsit general (Registre de 22 bytes)

Finalment, es desenvolupa una màquina de Turing finita implementada dins del sistema. Tal com s'observa a la Figura 19, aquesta “màquina” està composta per tres parts principals: el codi màquina (Machine Code), un segment de *padding* de dos bytes, i un registre de 20 bytes anomenat State Register, que és responsable de gestionar l'estat de la màquina i executar totes les operacions necessàries.

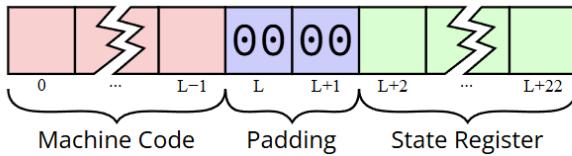


Figura 19: Les tres parts principals que formen la màquina amb propòsit general. (*Treball Meat Fighter*)

Durant l'execució, s'activen dos algoritmes escrits en Memory Code (MC) que tenen com a objectiu recórrer el State Register al llarg del codi màquina. Un d'aquests algoritmes avança cap a la dreta i l'altre cap a l'esquerra, alternant el sentit del recorregut per permetre una execució cíclica.

Durant el recorregut, aquests algoritmes gestionen la lectura i escriptura de la memòria necessària, com ara agafar la instrucció actual a executar. Un cop la informació pertinent ha estat carregada en el registre d'estat, s'executa la instrucció i es reinicia el recorregut en el sentit oposat.

Cal remarcar la dificultat d'implementar aquest comportament, ja que el llençatge MC no permet l'ús de bucles. Això obliga a definir manualment —o mitjançant scripts auxiliars— totes les instruccions **SWAP** implicades en cada cicle d'execució.

El segment de *padding* de dos bytes té com a finalitat garantir una correcta lectura dels bytes ubicats als extrems del codi màquina. Les adreces de memòria es representen amb dos bytes, cosa que permet gestionar un espai de memòria de fins a  $2^{16}$  bytes.

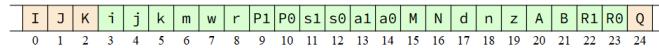


Figura 20: Composició del registre d'estat. (*Treball Meat Fighter*)

Tot i que el registre d'estat té una mida de 20 bytes, en el procés d'execució s'utilitzen fins a 24 bytes. Aquesta ampliació es deu al fet que els bytes anomenats I, J, K i Q formen part de la memòria, però estan inclosos ja que el programa MC els utilitza durant el recorregut. Aquests 4 bytes canvien a cada iteració ja que quan el registre es mou, aquests no ho fan i per tant quan es tornen a mirar són posicions de memòria diferents i, per tant, valors diferents.

A continuació, es descriuen els bytes més rellevants del registre:

- **a0, a1:** Apunten sempre a la posició de memòria on es troba el byte I.
- **P1, P2:** Representen el *program counter*. Aquest es veu incrementat al final de cada recorregut complet.
- **Condició:** Quan  $P == A$ , el valor dels bytes IJK es copia al registre ijk.
- **A, B, M, N:** Aquests quatre registres estan disponibles per al programador i s'utilitzen per dur a terme les operacions lògiques i aritmètiques principals.
- **R1, R0:** Representen les adreces de retorn de funcions o subrutines.
- **z:** Flag de zero, utilitzat per controlar condicions lògiques derivades de les instruccions.

Els registres no esmentats tenen funcions auxiliars o molt específiques, utilitzades en situacions concretes del sistema. Cal destacar que cada instrucció definida en el llenguatge assembler associat està dissenyada de forma precisa per ser executada sobre aquest registre d'estat, i està composta per diversos blocs de codi MC.

## Assembler

Finalment, es desenvolupa un llenguatge d'assemblador basat en mnemònics, dissenyat per ser interpretat per l'ordinador amb propòsit general. Aquest llenguatge

proporciona una capa d'abstracció que facilita l'escriptura de programes complexos, compilant-se en binari per poder ser executat directament sobre la màquina.

Cada instrucció d'aquest assembler està codificada mitjançant un únic byte d'operació (opcode), al qual poden seguir entre un i dos bytes addicionals, emprats per a paràmetres, com ara adreces de memòria o valors literals.

Aquest model simplificat d'arquitectura limita el nombre de registres que poden ser utilitzats en operacions:

- **A, B**: Registres generals. Són els únics registres sobre els quals es poden aplicar operacions aritmètiques, lògiques i de comparació.
- **M, N**: Registres reservats per al maneig d'adreces de memòria. Junts formen una paraula de 16 bits i s'utilitzen principalment per gestionar l'accés a memòria i el salt d'instruccions.

Value	Register
00	A
01	B
10	M
11	N

Figura 21: Registres. (*Treball Meat Fighter*)

Instruction	Name	Pseudocode	Opcode
TAB	Transfer A to B	B = A;	01
TAM	Transfer A to M	M = A;	02
TAN	Transfer A to N	N = A;	03
TBA	Transfer B to A	A = B;	04
TBM	Transfer B to M	M = B;	06
TBN	Transfer B to N	N = B;	07
TMA	Transfer M to A	A = M;	08
TMB	Transfer M to B	B = M;	09
TMN	Transfer M to N	N = M;	0B
TNA	Transfer N to A	A = N;	0C
TNB	Transfer N to B	B = N;	0D
TNM	Transfer N to M	M = N;	0E

Figura 22: Canvi de valors. (*Treball Meat Fighter*)

Instruction	Name	Pseudocode	Opcode
ADD	Add (without carry)	A += B;	10
AND	Bitwise AND	A &= B;	11
DEC	Decrement	--A;	12
INC	Increment	++A;	13
LS2	Logical Left Shift by 2	A <<= 2;	14
LS3	Logical Left Shift by 3	A <<= 3;	15
LS4	Logical Left Shift by 4	A <<= 4;	16
OR	Bitwise OR	A  = B;	17
RS1	Logical Right Shift by 1	A >>= 1;	18
RS5	Logical Right Shift by 5	A >>= 5;	19
SUB	Subtract (without borrow)	A -= B;	1A
XOR	Bitwise XOR	A ^= B;	1B

Figura 23: Operacions aritmèticologiques. (*Treball Meat Fighter*)

Instruction	Name	Pseudocode	Opcode
JMP label	Jump	P = label;	20
BNE label	Branch Not Equal	if (z == 0) P = label;	22
BEQ label	Branch Equal	if (z == 1) P = label;	23
BPL label	Branch Plus	if (n == 0) P = label;	24
BMI label	Branch Minus	if (n == 1) P = label;	25
JSR label	Jump Subroutine	R = P; P = label;	28
RTS	Return Subroutine	P = R;	70

Figura 24: Operacions de salts condicionals. (*Treball Meat Fighter*)

Instruction	Name	Pseudocode	Opcode
LDA	Load A	A = *MN;	40
LDB	Load B	B = *MN;	41

Figura 25: Lectures a memòria. (*Treball Meat Fighter*)

Instruction	Name	Pseudocode	Opcode
STA	Store A	*MN = A;	30
STB	Store B	*MN = B;	31

Figura 26: Escriptures a memòria. (*Treball Meat Fighter*)

Instruction	Name	Pseudocode	Opcode
SEA immediate	Set A	A = immediate;	50
SEB immediate	Set B	B = immediate;	51
SMN immediate	Set MN	MN = immediate;	2F

Figura 27: Assignar un literal a un registre. (*Treball Meat Fighter*)

```
SMN hex | label | label+offset | label-offset
```

Figura 28: Agafar la posició de memòria d'una etiqueta. (*Treball Meat Fighter*)

A més d'aquests conceptes també hi ha les etiquetes (nom\_etiqueta:), les definicions de constraints (define name value) i la alineació de memòria (segment address).

### Estructura final del sistema

Amb tota la informació i conceptes desenvolupats fins ara, podem descriure de forma simplificada però precisa l'estructura general i el flux d'execució del sistema. Aquest model representa un ordinador funcional implementat mitjançant lògica construïda exclusivament amb peces de *Tetris*, i es pot entendre com una arquitectura híbrida entre una màquina física simulada i un entorn de computació basada en codi.

El sistema es basa en els següents components:

1. Un ordinador físic executant un programa en **Java**.
2. Aquest codi Java gestiona:
  - La interfeície gràfica (visualització del tauler de joc).
  - Les entrades de teclat.
  - L'estat de la memòria i l'execució del codi MC.
3. El programador escriu el codi en assembler, el qual és compilat a binari.
4. Aquest binari, juntament amb el registre d'estat, constitueix la memòria de la màquina i és processat pel motor MC, que tradueix les instruccions a TetrominoScript (TS), i per tant es juga la partida.

A nivell conceptual, aquesta arquitectura es pot representar com:

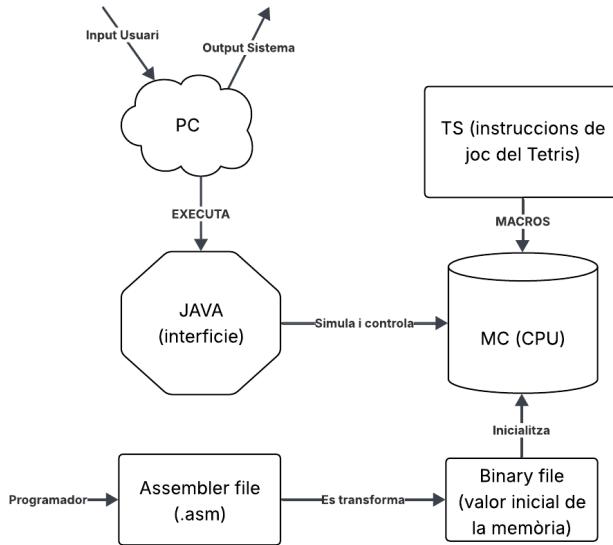


Figura 29: Esquema resum. (*Cració pròpia*)

### Optimitzacions i millores

Per tal de garantir una execució viable i eficient del sistema, s'han implementat diverses tècniques d'optimització:

- Lookup tables per simular portes lògiques. Aquestes taules han estat prèviament generades mitjançant simulació amb tetròminos i substitueixen la computació directa.
- Eliminació dinàmica de peces inactives o que ja no afecten el flux d'execució (per exemple, la base de la estructura).

### Flux d'execució del sistema

A continuació es detallen els passos necessaris per compilar, generar i executar un programa complet en aquest entorn:

#### 1. Compilar el projecte Java:

```
bash
mvn clean package
```

#### 2. Generar les Lookup Tables (només cal un cop):

```
java -cp target/tetromino-computer.jar
tetrominocomputer.ts.LutsGenerator
```

**3. Compilar el codi Assembler a binari:**

```
java -cp target/tetromino-computer.jar  
tetrominocomputer.asmAssembler  
-a example.asm -b example.bin
```

**4. Generar els cicles MC per processar la memòria:**

```
java -cp target/tetromino-computer.jar  
tetrominocomputer.mc.CycleProgramsGenerator  
-b example.bin  
-l CYCLE_LEFT.mc  
-r CYCLE_RIGHT.mc
```

**5. Executar el sistema complet:**

```
java -cp target/tetromino-computer.jar  
tetrominocomputer.gpc.app.GeneralPurposeComputer  
-c tetrominocomputer.gpc.app.MCProcessorAndMemory  
-b example.bin  
-l CYCLE_LEFT.mc  
-r CYCLE_RIGHT.mc
```

Aquest procés permet que el sistema llegeixi, interpreti i executi instruccions reals mitjançant una simulació física amb tetròminos, construint així una arquitectura funcional basada en la computació lògica.

## 2.2 Programació d'un compilador amb Python utilitzant ANTLR4

Una de les parts fonamentals del present treball és la creació d'un compilador. Tal com s'ha esmentat anteriorment, per dur a terme aquesta tasca s'utilitzarà el llenguatge de programació Python, juntament amb una de les eines més potents per al desenvolupament de compiladors: ANTLR4 (*Another Tool for Language Recognition*).

A continuació, es presenta una breu explicació del funcionament d'aquesta eina mitjançant un exemple pràctic. Per fer-ho, es requereixen dos fitxers: un amb extensió .g4, que conté la gramàtica, i un altre fitxer .py, que inclou el codi d'execució. De manera opcional, també es pot utilitzar un fitxer de text, com a input, amb l'expressió a analitzar.

Per il·lustrar el funcionament, s'utilitza com a exemple una calculadora molt bàsica, capaç d'analitzar i calcular el resultat de l'expressió següent:

2 + 3 \* 4 - 7

El fitxer `SimpleCalc.g4` conté la gramàtica de l'analitzador, és a dir, les regles que defineixen quines expressions són vàlides i quines no. En aquest cas, una expressió com `3 ++ 6` o `3 * a` no seria acceptada.

La gramàtica definida és la següent:

```
grammar SimpleCalc;

// Tokens lèxicos
NUMBER : [0-9]+ ;          // Nombres enters
PLUS   : '+' ;             // Operador suma
MINUS  : '-' ;             // Operador resta
TIMES  : '*' ;             // Operador multiplicació
DIV    : '/' ;             // Operador divisió
WS     : [ \t\r\n]+ -> skip ; // Espais en blanc ignorats

// Regles del parser
expr   : expr (TIMES | DIV) expr # MulDiv
        | expr (PLUS | MINUS) expr # AddSub
        | NUMBER                 # Num
        ;
```

A la part superior es defineixen els *tokens* bàsics, mentre que a la part inferior s'especifiquen les normes sintàctiques que estableixen com es poden combinar aquests elements per formar una expressió vàlida. Per exemple, es permet construir una expressió formada per dues subexpressions unides per un operador aritmètic (`+, -, *, /`), o bé una expressió pot ser simplement un número. En cas de ambigüïtat sobre quina norma aplicar, actua la de més amunt.

Aquestes regles permeten generar expressions de forma recursiva. A continuació, es mostra com es podria derivar l'expressió d'exemple a partir de les regles definides:

```
expr  (s'aplica MulDiv)
expr * expr  (s'aplica AddSub)
expr + expr * expr  (s'aplica AddSub)
expr + expr * expr - expr  (s'aplica Num per a cada operand)
2 + 3 * 4 - 7
```

Per tant, l'expressió utilitzada com a exemple és vàlida, ja que pot ser generada a partir de les regles anteriors (`MulDiv -> AddSub -> AddSub -> Num -> Num -> Num -> Num`). En canvi, no hi ha cap manera de generar l'expressió `3 ++ 6`, fet que confirma que no és sintàcticament correcta segons la gramàtica definida.

Cal destacar que, en casos més complexos, definir una gramàtica pot comportar dificultats relacionades amb l'ambigüïtat o la recursivitat a l'esquerra. Tot i això,

ANTLR4 és una eina prou robusta com per facilitar la gestió d'aquests problemes de forma automàtica o amb configuracions addicionals.

### 2.3 Generació del compilador

Un cop definida la gramàtica en el fitxer .g4, s'ha d'executar el següent comandament a la terminal, el qual generarà automàticament diversos fitxers necessaris per al funcionament del compilador:

```
antlr4 -Dlanguage=Python3 -no-listener -visitor SimpleCalc.g4
```

Aquest comandament indica a ANTLR4 que generi el codi corresponent en llenguatge Python 3, evitant l'ús de *listeners* i activant el patró *visitor*, que serà utilitzat posteriorment per a la interpretació de l'arbre sintàctic.

Un cop generats els fitxers, es pot crear un fitxer principal, anomenat `main.py`, que permet introduir una expressió, analitzar-la sintàcticament i obtenir-ne el resultat. A continuació, es mostra la seva implementació:

```

import sys
from antlr4 import *
from SimpleCalcLexer import SimpleCalcLexer
from SimpleCalcParser import SimpleCalcParser
from SimpleCalcVisitor import SimpleCalcVisitor

class EvalVisitor(SimpleCalcVisitor):
    def visitAddSub(self, ctx):
        left = self.visit(ctx.expr(0))
        right = self.visit(ctx.expr(1))
        if ctx.PLUS():
            return left + right
        elif ctx_MINUS():
            return left - right

    def visitMulDiv(self, ctx):
        left = self.visit(ctx.expr(0))
        right = self.visit(ctx.expr(1))
        if ctx.TIMES():
            return left * right
        elif ctx_DIV():
            return left / right

    def visitParens(self, ctx):
        return self.visit(ctx.expr())

    def visitNum(self, ctx):
        return int(ctx.NUMBER().getText())

def main():
    input_stream = InputStream(input("Ingrese una expresión: "))
    lexer = SimpleCalcLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = SimpleCalcParser(stream)
    tree = parser.expr()

    visitor = EvalVisitor()
    result = visitor.visit(tree)
    print("Resultado:", result)

if __name__ == '__main__':
    main()

```

Figura 30: Codi de la calculadora. (*Elaboració pròpia*)

La funció `main` és responsable d'iniciar tot el procés. Primer demana una expressió com a entrada (tot i que també podria llegir-se des d'un fitxer de text), després aplica el *lexer*, el qual transforma l'expressió en una seqüència de *tokens*. Per exemple, l'expressió  $2 + 3 * 4 - 7$  es tradueix en la següent llista de *tokens*:

```
[NUMBER, PLUS, NUMBER, TIMES, NUMBER, MINUS, NUMBER]
```

A continuació, el *parser* analitza la seqüència de *tokens* i construeix un arbre sintàctic basat en les regles definides a la gramàtica. L'arrel d'aquest arbre sempre serà l'expressió inicial (`expr`).

A tall d'exemple, l'arbre sintàtic per a l'expressió  $2 + 3 * 4 - 7$  és el següent:

```
expr (AddSub)
I-- expr (AddSub)
I   I-- expr (Num)
I   I   L-- '2'
I   I-- '+'
I   L-- expr (MulDiv)
I     I-- expr (Num)
I     I   L-- '3'
I     I-- '*'
I     L-- expr (Num)
I           L-- '4'
I-- '-'
L-- expr (Num)
L-- '7'
```

Finalment, el *visitor* recorre aquest arbre i executa accions concretes en funció de la regla gramatical identificada.

#### `visitAddSub`

Aquesta funció s'executa quan, durant el recorregut de l'arbre sintàctic, es detecta que s'ha aplicat la regla `AddSub`. Aquesta regla defineix operacions de suma i resta entre dues subexpressions.

El mètode rep com a paràmetre `ctx`, que representa el subarbre actual corresponent a l'operació detectada. Dins la funció, s'invoca recursivament la funció `visit` sobre cadascun dels operands (`ctx.expr(0)` i `ctx.expr(1)`), per tal de calcular-ne el valor.

Un cop obtinguts els resultats parcials de les dues subexpressions, es comprova si el símbol entre elles és una suma o una resta, i es retorna el resultat corresponent.

### **visitMulDiv**

Aquesta funció és anàloga a l'anterior, però tracta les operacions de multiplicació i divisió definides per la regla **MulDiv**.

De la mateixa manera, s'avaluen recursivament les dues subexpressions implicades, i després es realitza l'operació aritmètica pertinent segons si el símbol és una multiplicació (\*) o una divisió (/).

Aquesta estructura modular permet que el compilador sigui fàcilment extensible, afegint noves operacions o regles amb el mateix patró.

### **visitNum**

Finalment, aquesta funció és invocada quan s'arriba a una fulla de l'arbre sintàctic que representa un nombre enter. Aquesta fulla correspon a la regla **Num** de la gramàtica, la qual accepta una cadena de díigits.

La funció recupera el text associat al token **NUMBER** i el converteix a enter mitjançant la funció **int()**, retornant-ne el valor perquè pugui ser utilitzat en els càlculs superiors de l'arbre.

És important destacar que totes aquestes funcions es basen en un comportament recursiu: cada subexpressió és processada abans que es puguin combinar els resultats. Això es coneix com a recorregut en postordre (*bottom-up*), molt habitual en la implementació d'intèrprets i compiladors, ja que assegura que tots els valors necessaris es troben disponibles abans de realitzar una operació.

Aquesta estructura també permet una separació clara entre sintaxi i semàntica, facilitant la lectura, manteniment i escalabilitat del compilador.

Es pot observar la propagació de valors a través de l'arbre, amb els resultats parcials:

```
expr (AddSub) = 7  (7)
I-- expr (AddSub) = 14  (5)
I   I-- expr (Num) = 2  (4)
I   I-- '+'
I   L-- expr (MulDiv) = 12  (3)
I       I-- expr (Num) = 3  (1)
I       I-- '*'
I       L-- expr (Num) = 4  (2)
I-- '-'
L-- expr (Num) = 7  (6)
```

Aquesta estructura permet comprendre clarament com cada subexpressió és resolta i com es propaga el càlcul fins a obtenir el resultat final.

```

Run: python main
C:\Users\Polete\Desktop\ProvaCompilador\
Ingrese una expresión: 2+3*4-7
Resultado: 7
Process finished with exit code 0

```

Figura 31: Resultat després d'introduir  $2+3*4-7$ . (*Elaboració pròpia*)

## 2.4 Primer contacte amb l'entorn i objectius iniciais

El primer pas en el desenvolupament del projecte va consistir en clonar el repositori de Git on es trobaven tots els fitxers del projecte *Meat Fighter*. Un cop descarregat, es va procedir a l'execució de l'aplicació, la qual obria una finestra gràfica on es podia jugar a una versió funcional del joc *Tetris*. Aquesta primera execució tenia l'objectiu de verificar el funcionament correcte del sistema i familiaritzar-me amb el seu comportament visual i lògica interna.

Després d'una primera exploració del codi font, es va identificar que l'entorn es componia tant de fitxers Java com d'un arxiu en llenguatge assemblador, i que aquests estaven dissenyats específicament per fer funcionar el motor del joc sobre un sistema personalitzat —una mena d'ordinador fet amb tetròminos, que esdevé una abstracció visual del processador.

Durant aquesta fase inicial, es va dur a terme un període intens d'anàlisi del sistema, comprensió del funcionament intern i dels conceptes tècnics implicats, així com la realització de diversos experiments sobre el codi per verificar hipòtesis. Alguns dels canvis realitzats durant aquestes proves van incloure modificacions senzilles com ara:

- Canviar la posició inicial de les peces, fent que apareguessin des del centre del mapa en lloc d'aparèixer des de la part superior.
- Redefinir les tecles de control del joc.
- Pintar blocs arbitràriament al mapa mitjançant escriptura directa a la memòria —encara que aquest últim de manera forçada.

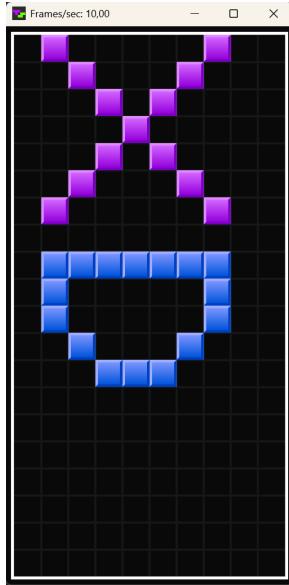


Figura 32: Una de les primeres proves que es van fer. (*Elaboració pròpia*)

Aquest conjunt de proves va permetre adquirir una comprensió suficient del sistema per tal d'encarar la següent etapa del projecte.

## 3 Disseny

### 3.1 Definició del llenguatge

Abans d'iniciar el disseny d'un llenguatge de programació, és imprescindible tenir molt clar quin objectiu es persegueix amb ell i quin tipus de funcionalitat ha d'oferir. Aquest exercici previ permet definir amb coherència la seva sintaxi, semàntica i abast, establint unes bases sólides que assegurin tant la consistència del llenguatge com la seva adequació a l'entorn on s'utilitzarà. Així doncs, cal prendre una sèrie de decisions fonamentals que no són ni correctes ni incorrectes de forma absoluta, però que condicionen fortament la naturalesa del llenguatge resultant.

#### 3.1.1 Permissivitat vs Restricció

Una de les primeres qüestions a resoldre és fins a quin punt el llenguatge ha de ser permisivi o restrictiu amb el codi que admet. Un llenguatge molt restrictiu limita la creativitat del programador, dificulta la implementació de certs algorismes i pot frustrar l'usuari avançat. Per contra, un llenguatge massa permisivi pot donar lloc a codi amb comportaments inesperats, errors semàntics difícils de detectar o dificultats a l'hora de mantenir el projecte.

En aquest cas, s'ha optat per una gramàtica relativament laxa i una gestió d'errors mínima. L'objectiu és oferir màxima llibertat als usuaris més avançats,

facilitant-los l'experimentació i la rapidesa en el desenvolupament. Per als usuaris menys experimentats, aquest document inclou recomanacions sobre bones pràctiques que, si se segueixen, minimitzen els riscos d'error. Tot i això, algunes instruccions d'alt nivell sí que s'han dissenyat amb cert grau de restricció per garantir una coherència estructural mínima i evitar usos inadequats.

### 3.1.2 Originalitat vs Estàndards de la comunitat

Un altre aspecte crític és trobar l'equilibri entre la innovació i l'adopció de convencions habituals en altres llenguatges. Un llenguatge massa original pot ser difícil d'aprendre i d'adoptar, mentre que un llenguatge que copia fil per randa patrons ja existents pot perdre personalitat i aportar poc valor afegit.

Amb aquest projecte s'ha intentat assolir un punt intermedi: s'introdueixen idees i formes pròpies, però mantenint sempre una lògica estructural comprensible i familiar per a qualsevol persona amb experiència en llenguatges imperatius. A més, s'ha optat per un enfocament clarament distintiu: tot el llenguatge està escrit en català. Aquest element, tot i no afectar directament a la funcionalitat, reforça la identitat pròpia del projecte i el fa únic dins el seu àmbit.

### 3.1.3 Generalització vs Especialització

Pel que fa a les instruccions del llenguatge, s'ha buscat un equilibri entre la generalitat i la funcionalitat. Un llenguatge massa especialitzat pot esdevenir rígid, mentre que un d'excessivament general pot requerir codificacions innecessàriament llargues per a tasques senzilles.

Per aquest motiu, s'ha estructurat el llenguatge amb un conjunt d'elements bàsics imprescindibles com assignacions, condicions, bucles i subrutines amb suport per a recursió (no només procediments simples, sinó també *no-leaf*). Amb aquestes estructures es pot construir pràcticament qualsevol funcionalitat. Addicionalment, per agilitzar el desenvolupament i fer un codi més intel·ligible, s'han afegit algunes instruccions específiques útils per treballar en l'entorn de la graella del Tetris, com per exemple poder pintar en una casella simplement indicant la seva posició cartesiana.

### 3.1.4 Funcionalitats essencials

El llenguatge proposat inclou les funcionalitats següents com a nucli:

- Declaració i assignació de variables de tipus enter (a partir d'aquests es poden crear booleans, chars i amb una mica d'imaginació floats).
- Suport per a llistes i accés per índex.
- Operacions aritmètiques bàsiques (+, -, \*, /, %), comparacions lògiques (==, !=, <, >, >=, <=) i operacions lògiques (i, o, no).

- Estructures condicionals (**si...sinó**).
- Bucles iteratius (**bucle...**, **per...**), amb instruccions de control (**surt**, **continua**).
- Definició i crida de subrutines (amb suport per a recursivitat).
- Instruccions específiques per interactuar amb el mapa i les entrades de l'usuari dins l'entorn del Tetris.

Aquest conjunt permet escriure programes complexos d'una manera relativament senzilla, tot mantenint el control directe sobre el codi assembler que es generarà en la compilació.

### 3.2 Classificació del llenguatge de programació

En aquest apartat es detalla la classificació del llenguatge de programació desenvolupat segons diversos criteris habituals en l'estudi dels llenguatges.

#### Nivell d'extracció

El llenguatge es troba en un punt intermedi entre el nivell mitjà i l'alt nivell, però aproximant-se més a l'alt nivell. Això es deu al fet que la majoria del temps el programador no s'ha de preocupar de com està la memòria i té instruccions que li gestionen pràcticament tot per ell. Tanmateix, el grau de llibertat que proporciona —com ara la inexistència de gestió d'errors— és més propi dels llenguatges de baix nivell. A més a més que hi ha vegades que s'ha de tenir en compte on es posen les variables i en la mida dels elements.

#### Paradigma

El paradigma seguit pel llenguatge és clarament imperatiu. El flux de control del programa es defineix mitjançant instruccions que alteren l'estat del sistema. Les estructures com bucles, assignacions i crides a funcions reflecteixen aquest model imperatiu tradicional.

#### Execució

El llenguatge és compilat. El codi font escrit per l'usuari és traduït a un codi intermedi o de baix nivell que pot ser executat per una màquina virtual especialitzada. Aquesta és la raó principal per la qual s'ha desenvolupat un compilador com a part del projecte.

#### Tipatge

El sistema de tipatge és dèbil. De fet, pràcticament no hi ha tipus en sentit estricte. Les variables poden contenir qualsevol valor numèric i només s'utilitza la

convenció de 0 i 1 per representar valors booleans. Aquesta simplicitat facilita la implementació però alhora redueix la seguretat del tipus.

### Segons el seu propòsit

Tot i que el llenguatge és, en certa manera, de propòsit general —es poden definir variables, llistes, funcions i condicions— el seu disseny està profundament lligat a un propòsit específic: generar codi per a una màquina virtual que simula l'execució de tetròminos, com en el joc Tetris. Per tant, pot considerar-se un llenguatge de propòsit específic a la pràctica.

## 4 Implementació

### 4.1 Funcionament bàsic del compilador

Per fer ús del compilador Tetris++, el programador ha de preparar dos fitxers d'entrada:

- `funcions.txt`: conté les definicions de funcions reutilitzables.
- `programa.txt`: conté el codi principal, que s'executarà de forma contínua com a bucle principal.

El compilador genera com a sortida un arxiu `.asm` dividit en tres seccions:

1. **Inicialització de la memòria:** codi pràcticament fix i comú a tots els programes, configura les adreces inicials perquè l'execució sigui estable.
2. **Definicions de funcions:** traducció del fitxer `funcions.txt` a codi assembler.
3. **Codi principal:** corresponent al contingut de `programa.txt`, que serà executat com a `main`.

El flux d'execució es gestiona mitjançant etiquetes i salts, propis del llenguatge assembler. Per evitar col·lisions entre etiquetes, el compilador genera automàticament noms únics mitjançant una variable global. Aquesta variable comença amb el valor `ta`, i cada nova etiqueta s'obté afegint-hi una `a`. En el cas de les condicions (`if`), que requereixen dues etiquetes, s'utilitza una `e` per diferenciar-les. Es fa d'aquesta manera ja que l'assembler no accepta etiquetes amb números.

### 4.2 Distribució de la memòria

La memòria està estructurada en diversos segments, cadascun amb una finalitat concreta:

### **Variables posicionades (0x0000 - 0x004F)**

Aquest espai està reservat per a variables que el programador pot assignar explícitament a una adreça concreta dins aquest rang. Principalment, aquestes variables es fan servir per gestionar entrada i sortida. El compilador manté una llista de les variables detectades en aquesta zona, registrades en l'ordre en què apareixen als fitxers d'entrada.

### **Variables globals (0x0050 - 0x00FF)**

Les variables globals no posicionades explícitament s'assignen automàticament de forma seqüencial a partir de l'adreça 0x0050. L'espai disponible permet definir fins a 80 variables.

Les últimes 5 posicions d'aquest rang estan reservades per a:

- `seedHigh` i `seedLow`: per a generació de nombres aleatoris.
- `nextBit`: auxiliar per al procés d'aleatorietat.
- `frameCounter` i `drawFrame`: control d'actualització de pantalla en l'entorn Java.

Totes les variables són globals excepte els paràmetres de funcions, que es gestioen a través de la pila.

### **Llistes (0x0100 - 0x026F)**

Les llistes segueixen una lògica similar a les variables, però poden ocupar múltiples posicions consecutives. És molt important assegurar-se que una llista no travessi la frontera entre els rangs 0x01XX i 0x02XX, ja que això pot generar comportaments erronis. Per evitar-ho, s'introdueixen llistes de *padding* si és necessari, per mantenir l'alineació.

### **Temporals (0x0270 - 0x02FF)**

Aquesta secció de memòria roman inalterada entre diferents programes, ja que és on s'inicialitzen les variables temporals i reservades necessàries per al funcionament intern del compilador i l'execució segura del codi. Aquestes variables inclouen `m`, `r`, `FUNCTIONS`, `rRETURN`, `aUX`, `cARRY` i `cOMP`. Totes aquestes es defineixen amb un valor inicial de 0.

Cada una d'aquestes variables es reserva per a funcionalitats específiques del compilador. Encara que el seu ús concret es detallarà més endavant, en general tenen com a finalitat protegir valors durant càlculs intermedis, per tal d'evitar pèrdues d'informació en instruccions encadenades.

Pel que fa a les variables temporals pròpiament dites, s'inicialitzen 20 espais de memòria amb noms seguint el patró `t` seguit d'una quantitat variable de lletres

P. El nombre de lletres P indica el número de temporal utilitzat. Per exemple, si s'han utilitzat els temporals 1 i 2, el següent serà tPPP, que representa el temporal número 3.

Aquest esquema permet identificar de manera sistemàtica cada variable temporal, la qual cosa és crucial per gestionar correctament càlculs intermedis. En llenguatges de baix nivell com l'assembler, les operacions es poden realitzar normalment només entre dos operands alhora, això significa que, per calcular expressions més complexes, és imprescindible conservar valors intermedis. Utilitzar sempre la mateixa variable per emmagatzemar aquests valors suposaria perdre resultats encara necessaris, per tant, cal gestionar múltiples temporals de manera dinàmica.

Per portar aquest control, el compilador manté una variable global que indica quin és el primer temporal lliure (és a dir, el primer que encara no ha estat assignat). Els temporals s'utilitzen en ordre ascendent (primer el temporal 1, després el 2, etc.), evitant així la sobreescritura de dades encara útils.

És fonamental, un cop finalitzada cada operació, restablir el valor del temporal a 0. En cas contrari, s'acabarien utilitzant tots els temporals disponibles i això provocaria conflictes a l'execució, com col·lisions de memòria o resultats incorrectes. Aquesta pràctica de “neteja” garanteix l'estabilitat i la reutilització segura dels recursos durant l'execució del programa.

### La pila (0x0300 – 0x03FF)

Per permetre subroutines amb crides recursives o amb crides entre funcions (no-leaf), s'ha implementat una pila virtual dins la memòria, ja que l'assembler no ho permetia. Cada crida a funció genera una “capa” a la pila, que es destrueix en finalitzar la funció. Cadascuna d'aquestes capes ocupa 8 bytes, distribuïts de la següent manera:

- **Bytes 1 i 2:** direcció de retorn (*high byte* primer). Degut a que no hi ha accés al *program counter*, per aconseguir-la s'ha de crear una etiqueta just després d'on s'ha fet la crida, llavors s'agafa aquella direcció.
- **Bytes 3 i 5 i 7:** paràmetres 1, 2 i 3.
- **Bytes 4, 6 i 8:** espais buits de separació per evitar col·lisions.

El compilador utilitza una variable interna per mantenir el nivell actual de crida a la pila, que funciona com un punter al *top* de la pila. Quan es crida una funció augmenta 8 unitats (+1 capa), i quan una retorna decrementa 8 unitats (-1 capa).

A partir de la direcció 0x0400, la memòria no té una distribució predefinida.

Abans d'introduir el codi de qualsevol funció definida pel programador, el compilador genera un salt automàtic cap a la funció `main`. Això assegura que, en el moment de l'execució, l'entrada al programa començi per la rutina principal i no per una funció auxiliar.

### 4.3 Funció funcioreturn

Quan el programa defineix una o més funcions, el compilador afegeix automàticament una rutina especial anomenada **funcioreturn**. Aquesta funció és essencial per implementar un sistema de retorn dinàmic en funcions no-leaf, ja que l'assembler base no suporta crides amb retorn gestionades de manera automàtica.

**funcioreturn** actua com a rutina de retorn compartida: cada funció, en lloc de tornar directament a la seva adreça de crida, salta a **funcioreturn**, que s'encarrega de reconstruir i executar el salt de retorn.

```
funcioreturn:  
    SMN fUNCTIONS  
    LDA  
    SMN pila  
    TNB  
    ADD  
    TAN  
    LDB  
  
    SMN notocar  
    TNA  
    INC  
    TAN  
    STB  
  
    SMN fUNCTIONS  
    LDA  
    SMN pila  
    TNB  
    ADD  
    INC  
    TAN  
    LDB  
    SMN notocar  
    TNA  
    INC  
    INC  
    TAN  
    STB  
  
notocar: JSR notocar
```

Aquesta funció té com a objectiu principal obtenir des de la pila els dos bytes de l'adreça de retorn de la capa actual (és a dir, de la funció que l'ha cridat) i modificar

en temps d'execució l'última instrucció de la pròpia funció `funcioreturn`, que és un JUMP. Aquesta instrucció està composta per tres bytes: l'*opcode*, el byte superior i el byte inferior de l'adreça de destí. Llavors es substituiran els dos bytes de l'adreça de destí pels que s'han agafat de la pila, que és on es vol retornar.

Per tal d'identificar la ubicació d'aquesta instrucció JUMP, s'utilitza una etiqueta anomenada `notocar`, la qual actua com a referència dins del codi. A partir de la seva adreça, s'aplica un increment o dos per accedir a l'adreça del byte superior o inferior respectivament, de la instrucció. Aquest increment ha de ser gestionat amb cura, ja que si el byte inferior de l'adreça és 0xFE o 0xFF, l'increment no genererà un *carry* i, per tant, es modificarà una posició de memòria que no és correcta. Tot i així, aquest cas concret no suposa un problema perquè la funció es troba al principi del codi i està correctament alineada, de manera que aquesta situació no es dona. A més de solucionar l'error, el fet de tenir aquesta única funció, estalvia molt d'espai ja que es reutilitza per a totes les funcions.

Aquest mecanisme permet modificar de forma dinàmica el flux d'execució en funció del context d'execució.

#### 4.4 La resta de codi

A partir d'aquí ja hi ha la traducció a assembler de totes les funcions declarades pel programador, seguit de la traducció de la funció `main` i un últim salt cap a l'inici del codi `main`, per assegurar l'execució contínua.

Tant el codi corresponent a les funcions com el de `main` es generen utilitzant la mateixa gramàtica i el mateix visitador. Això permet mantenir coherència en la traducció, reutilitzar estructures i garantir una semàntica unificada per a tot el llenguatge.

A continuació es presentaran les estructures gramaticals i el funcionament intern del compilador per a cada tipus d'instrucció del llenguatge Tetris++.

#### 4.5 Definicions lèxiques

La primera part de la gramàtica consisteix en la definició dels símbols lèxics (tokens) que formen les unitats bàsiques del llenguatge. Aquests es defineixen mitjançant expressions regulars i s'utilitzen com a blocs fonamentals per construir les regles sintàctiques. A continuació es descriuen els principals:

```
grammar TPlusPlus;

FUNC : ('A'...'Z')+;
HEX4 : [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F];
VAR   : ('a'...'z') ('A'...'Z')* ;
COMENTARI : ';' ('a'...'z' | 'A'...'Z' | ' ' | '0'...'9')* ;
NUMBER  : [0-9]+ ;
```

```

UNDOSTRES: '(' [1-3] ')' ;
LPAREN : '(' ;
RPAREN : ')' ;
COMA : ',' ;
WS      : [ \t\r\n]+ -> skip ;

```

- **FUNC**: Identificador de funcions. Consta d'una o més lletres en majúscula (d' A a Z). Aquest format permet diferenciar clarament les funcions de les variables, ja que el codi assembler utilitza etiquetes, les quals han de ser úniques.
- **HEX4**: Seqüència de quatre díigits hexadecimals. Aquest format s'utilitza, sobretot per identificar adreces de memòria.
- **VAR**: Identificador de variables i llistes. Està format per una lletra minúscula inicial seguida opcionalment de lletres en majúscula. Aquesta convenció serveix per distingir-les clarament de les funcions.
- **COMENTARI**: Comença amb un punt i coma (;) i segueix amb qualsevol combinació de caràcters alfanumèrics i espais. Es tracta d'una línia de comentari que el compilador ignora.
- **NUMBER**: Un o més díigits decimals. Representen valors enters literals.
- **UNDOSTRES**: Una estructura que representa un número entre 1 i 3, inclòs entre parèntesis. S'utilitza per representar paràmetres en certes instruccions del llenguatge.
- **LPAREN i RPAREN**: Símbols de parèntesis d'obertura i tancament, respectivament.
- **COMA**: El caràcter coma, utilitzat per separar arguments en instruccions compostes.
- **WS (Whitespace)**: Espais en blanc, tabulacions i salts de línia. Aquests són ignorats pel compilador mitjançant la directiva `-> skip`.

Aquest conjunt de definicions constitueix la base lèxica sobre la qual es construeix la gramàtica completa de Tetris++.

## 4.6 Construcció de la gramàtica

A partir dels símbols lèxics definits anteriorment, es procedeix a la definició de les regles sintàctiques que descriuen el llenguatge. L'enfocament seguit comença per les unitats més bàsiques (elements atòmics) i progressa cap a estructures més complexes, com ara instruccions, blocs de funcions i programes complets. Aquesta

metodologia garanteix una explicació progressiva i evita introduir conceptes que encara no han estat explicats.

En les seccions següents es detallaran les regles concretes de cada instrucció del llenguatge, així com el seu equivalent en codi assembler i el comportament esperat en temps d'execució. Per mantenir coherència en el codi, cada regla retorna el codi per executar-la i en el cas d'haver de calcular algun valor es guarda al registre A. El registre B en molts casos servirà com a suport per fer les operacions i inclús a vegades per guardar informació temporal, tot i que en combinar vàries regles és necessari utilitzar variables temporals. Sempre que es vulgui accedir a l'adreça d'una etiqueta s'utilitzarà el registre de 2 bytes MN.

## 4.7 Simples

### Gramàtica

```
simples : NUMBER # Num
         | VAR # Var
         | VAR '[' 'simples' ']' #Lista
         | 'parametre' UNDOSTRES #Param
;
```

La regla **simples** defineix les unitats mínimes amb significat dins del llenguatge Tetris++. Aquestes unitats representen valors concrets amb els quals el programa opera directament. Els elements reconeguts com a **simples** poden ser literals o referències a memòria, i s'utilitzen com a operands en instruccions més complexes.

Les quatre formes reconegudes són:

- **Número enter:** un valor decimal directe.  
*Exemple:* 22
- **Variable:** una referència a una posició de memòria global.  
*Exemple:* vARIABLE
- **Llista:** accés a un element d'una llista a partir d'un índex especificat amb un altre **simple**.  
*Exemple:* 1LISTA[vARIABLE]
- **Paràmetre:** accés a un paràmetre d'una funció, entès com una variable local que només existeix dins de l'àmbit d'aquesta. Per accedir-hi s'utilitza la paraula clau **parametre** seguida d'un número entre 1 i 3, indicant quin paràmetre es vol utilitzar.  
*Exemple:* parametre (2)

Tots aquests elements poden ser utilitzats com a operand directe en expressions, assignacions o com a entrada per a instruccions més complexes. Aquesta flexibilitat permet construir instruccions amb una sintaxi senzilla però potent.

## Comportament de les regles

Cada cas de la regla `simples` es tradueix a una seqüència d'instruccions assembler que permeten carregar un valor concret al registre A, que és on es fan habitualment les operacions.

- **Número:** el compilador transforma un nombre decimal en la seva representació hexadecimal de dos díigits i genera les instruccions necessàries per carregar aquest valor al registre A. És una càrrega directa d'un valor immediat.

```
...  
def visitNum(self, ctx):  
    num = hex(int(ctx.NUMBER().getText())[2:]).upper().zfill(2)  
    code = f"SEA {num}"  
    return code
```

Figura 33: Codi visitNum(). (*Elaboració pròpia*)

- **Variable:** el nom de la variable es tradueix a una posició de memòria fixa. L'instrucció generada accedeix a aquesta posició i en carrega el valor contingut al registre A. És una forma bàsica d'accés a memòria global.

```
...  
def visitVar(self, ctx):  
    code = f"""SMN {ctx.VAR().getText()}  
LDA"""  
    return code
```

Figura 34: Codi visitParam(). (*Elaboració pròpia*)

- **Llista:** primer s'avaluarà l'expressió que especifica l'índex (també una instrucció `simple`), i es carregarà al registre auxiliar. Aquest valor se suma a la posició base de la llista per accedir a l'element desitjat. A continuació, es carrega aquest valor al registre A. Això permet accedir a llistes de forma dinàmica.

```

def visitLista(self, ctx):
    code = self.visit(ctx.simples())
    code += f"""
SMN {ctx.VAR().getText()}
TNB
ADD
TAN
LDA"""
    return code

```

Figura 35: Codi visitLista(). (*Elaboració pròpia*)

- **Paràmetre:** per obtenir un paràmetre, es calcula l'adreça corresponent dins la capa activa de la pila (una àrea de memòria estructurada per crides a funcions). El sistema localitza la posició del paràmetre concret i n'estreu el valor, que després es carrega al registre A. El càclul de la posició depèn de quin paràmetre (1, 2 o 3) es demana, i es fa mitjançant desplaçaments controlats.

```

def visitParam(self, ctx):
    code = f"""
SMN FUNCTIONS
LDA
SMN pila
TNB
ADD
INC
"""
    for i in range((int(ctx.UNDOSTRES().getText()[1]) *2) - 1):
        code += "INC \n"
        code += """TAN
LDA
"""
    return code

```

Figura 36: Codi visitParam(). (*Elaboració pròpia*)

## 4.8 Aritmètica

### Gramàtica

```

aritmetica : '(' aritmetica ')'      #Parenarit
           | aritmetica '*' aritmetica #Mul
           | aritmetica '/' aritmetica #Div

```

```

| aritmetica '+' aritmetica #Suma
| aritmetica '-' aritmetica #Resta
| aritmetica '%' aritmetica #Modul
| simples           #Final

```

Aquesta regla defineix les operacions aritmètiques suportades pel llenguatge Tetris++. Les operacions inclouen suma, resta, multiplicació, divisió i mòdul. També es permeten parèntesis per alterar la precedència d'avaluació, i la part `simples` permet acabar la recursió aritmètica.

### Exemple d'expressió

`(a%2 + 3) /b`

### Comportament de les regles

Per garantir la correcció durant operacions encadenades, es fan servir variables temporals, ja que les limitacions del conjunt d'instruccions en assembler impedeixen treballar amb múltiples operacions simultànies sobre un sol registre.

- **Suma:** El resultat de la part esquerra de la suma es calcula primer i es guarda en una variable temporal per no perdre'l mentre s'avalua la part dreta. Un cop obtinguts els dos operands, es fa servir la instrucció ADD per sumar els registres A i B, i deixar el resultat en A.



```

def visitSuma(self, ctx):
    global num_temp
    num_temp += 1
    temp = 't'
    for i in range(num_temp):
        temp = temp + 'P'
    code = self.visit(ctx.aritmetica(0))
    code += f"""
SMN {temp}
STA
"""
    code += self.visit(ctx.aritmetica(1))
    code += f"""
TAB
SMN {temp}
LDA
ADD
"""
    return code

```

Figura 37: Codi `visitSuma()`. (*Elaboració pròpia*)

- **Resta:** El procés és pràcticament idèntic al de la suma, però s'utilitza **SUB** en lloc de **ADD**. A més, l'ordre dels operands és crític, ja que la resta no és commutativa.
- **Multiplicació:** La multiplicació  $x \cdot y$  es pot implementar com una suma repetida de  $y$ ,  $x$  vegades. Això és exactament el que fa el codi presentat. No s'utilitzen directament els registres **A** ni **B** per emmagatzemar dades, ja que aquests registres són necessaris per a les operacions internes, i utilitzar-los per a variables persistents faria perdre el context entre iteracions. Per dur a terme aquesta operació, es requereix una variable auxiliar **m**, que servirà com a comptador dins del bucle, a la qual inicialment se li carrega el valor de  $x$ . També es defineix una altra variable, **r**, que contindrà el resultat del que s'hagi calculat de la multiplicació fins al moment, la qual s'inicialitza a zero.

L'estructura del bucle és la següent: s'etiqueta l'inici del bucle **i**, al final, s'inclou una instrucció de salt condicional. Dins del bucle, primer es carrega el valor de  $y$  al registre **B**, després es carrega el valor de **r** al registre **A**, es realitza una suma amb **ADD**, i el resultat de **A** s'assiganya novament a **r**. Tot seguit, es decrementa el valor de **m** i es comprova si el seu valor ha arribat a zero; en cas contrari, es torna a saltar a l'inici del bucle. Finalment, un cop acabada l'operació, el valor de **r** es trasllada al registre **A** per ser utilitzat.

Cal tenir en compte que, en fer ús d'un salt condicional, no es poden utilitzar valors majors a 127, ja que aquests poden provocar comportaments incorrectes. De fet, multiplicar per un valor superior a aquest pot provocar un *overflow*, fent que l'operació no sigui fiable dins del límit imposat pel registre.

```

    def visitMul(self, ctx):
        global num_temp
        global etiqueta
        num_temp += 1
        temp = 't'
        for i in range(num_temp):
            temp = temp + 'P'
        actual = etiqueta
        etiqueta = etiqueta + 'a'

        code = self.visit(ctx.aritmetica(1))

        code += f"""
SMM {temp}
STA
"""

        code += self.visit(ctx.aritmetica(0))
        code += """
SMM m
DEC
STA
"""

        code += f"""
SMN r
SEA 00
STA

{actual}:
SMM {temp}
LDB

SMN r
LDA
ADD
STA

SMN m
LDA
DEC
STA
SEB 00
SUB
BPL {actual}
SMN r
LDA
"""

        return code

```

Figura 38: Codi visitMul(). (*Elaboració pròpia*)

- **Divisió:** Funciona com la inversa de la multiplicació. Es parteix del dividend i es va restant el divisor fins que ja no sigui possible. Cada resta compta com una unitat afegida al quocient, que és el resultat final.
- **Mòdul:** S'implementa de la mateixa manera que la divisió, amb l'única diferència que el resultat no és el quocient, sinó el residu que queda després de restar tantes vegades com sigui possible el divisor al dividend. El residu es té calculat sempre en la variable acumuladora, per tant simplement s'ha de carregar al final.
- **Parèntesi:** Simplement es força la recursió aritmètica a avaluar-se amb prioritat, respectant l'ordre natural matemàtic. No modifica l'operació, però assegura la jerarquia correcta.

## 4.9 Boolean

### Gramàtica

```
boolean :
    aritmetica '<' aritmetica      #Menor
    | aritmetica '>' aritmetica      #Mayor
    | aritmetica '<=' aritmetica     #Menorigual
    | aritmetica '>=' aritmetica     #Mayorigual
    | aritmetica '==' aritmetica      #Igual
    | aritmetica '!=' aritmetica      #Noigual
    | aritmetica                      #Simp
```

Aquesta regla defineix les comparacions booleanes disponibles. Permet comparar dues expressions aritmètiques mitjançant operadors com menor que, igual, diferent, etc. També es permet la conversió directa a un operació aritmètica.

### Exemple d'expressió

`3 + 4 == 7`

### Comportament de les regles

Totes les comparacions booleanes comparteixen una estructura bàsica comuna, ja que totes parteixen d'obtenir dos valors (esquerra i dreta) i comparar-los:

- Primer es calcula l'expressió de la dreta i es guarda en una variable especial anomenada `COMP`.
- Després es calcula l'expressió de l'esquerra, que queda carregada al registre `A`.
- Es carrega el valor de `COMP` al registre `B` i es fa una operació `SUB`.

- Segons el resultat del bit z, es modifica el valor del registre A a 1 o 0.

```

def visitIgual(self, ctx):
    global etiqueta
    actual = etiqueta
    etiqueta = etiqueta + 'a'
    code = self.visit(ctx.aritmetica(1))
    code += """
SMM cOMP
STA
"""
    code += self.visit(ctx.aritmetica(0))
    code += f"""
SMM cOMP
LDB
SUB
SEA 01
BEQ {actual}
SEA 00
{actual}:
"""
    return code

```

Figura 39: Codi visitIgual(). (*Elaboració pròpia*)

Aquest sistemaaprofita el fet que les operacions booleanes no poden estar imbricades dins d'altres booleanes, així que no cal fer ús de temporals com en les aritmètiques. Només cal una única variable auxiliar, cOMP, que mai serà sobreescrita dins la mateixa expressió.

La regla Simp simplementavaluarà una expressió aritmètica sense fer cap comparació.

## 4.10 Logic

### Gramàtica

```

logic :
  '(' logic ')'      #Paren
  | 'no' logic        #Not
  | logic 'i' logic   #And
  | logic 'o' logic   #Or
  | boolean            #Bool

```

Els operadors lògics disponibles són: **i** (AND), **o** (OR) i **no** (NOT). Aquestes operacions es poden fer sobre valors booleanos o altres operacions lògiques, ja que la

gramàtica és recursiva. També es poden utilitzar parèntesis per agrupar condicions i controlar l'ordre d'avaluació.

Tot i que la gramàtica permet expressions com `2 i 3`, aquestes no tenen gaire sentit lògic i és responsabilitat de l'usuari garantir que les operacions es facin sobre valors booleanos. Tot i així en aquest cas es retornarà una operació AND bit a bit, però el llenguatge no està pensat per això.

### Exemple d'expressió

```
no (a == 1 i b != 2) o 3 < 4
```

### Comportament de les regles

- **And / Or:** La idea és avaluar les dues subexpressions i aplicar l'operació binària corresponent al final. El patró és molt semblant al que s'utilitza amb les operacions aritmètiques:
  - Es calcula la primera expressió i es guarda en un temporal.
  - Es calcula la segona expressió.
  - S'aplica l'operació lògica (AND o OR) entre els dos resultats.



```
def visitAnd(self, ctx):  
    global num_temp  
    num_temp += 1  
    temp = 't'  
    for i in range(num_temp):  
        temp = temp + 'P'  
    code = self.visit(ctx.logic(0))  
    code += f"""\n    SMN {temp}\n    STA\n    """  
    code += self.visit(ctx.logic(1))  
    code += f"""\n    TAB\n    SMN {temp}\n    LDA\n    AND\n    """  
    return code
```

Figura 40: Codi visitAnd(). (*Elaboració pròpia*)

- **Not:** Aquesta operació només actua sobre una expressió. No necessita temporals ni registres addicionals. Simplement es fa una XOR amb la màscara 0x01, que inverteix el bit menys significatiu (el que representa el valor booleà).

```

def visitNot(self, ctx):
    code = self.visit(ctx.logic())
    code += """
SEB 01
XOR
"""
    return code

```

Figura 41: Codi visitNot(). (*Elaboració pròpia*)

- **Bool:** Aquesta regla finalitza la recursivitat i simplement visita una expressió booleana.

## 4.11 Op

A continuació es detallen les regles de les operacions, que són la forma més general de càcul de valors.

### Gramàtica

```

op : logic                      #Logistica
| 'mira' (simples, simples)   #Read
| 'peça' simples simples simples #Peça
| 'aleatori'                  #Random

```

### Comportament de les regles

- **Logística:** Aquesta regla simplement visita l'expressió lògica que conté.
- **Read:** Aquesta regla ha estat definida dins la gramàtica per a facilitar el desenvolupament de programes que interactuen amb l'entorn gràfic del Tetris proporcionat pel projecte *Meat Fighter*, ja que permet comprovar si una determinada casella del tauler ha estat pintada o no. L'expressió s'inicia amb la paraula clau **mira**, seguida de dos valors simples separats per una coma, que indiquen les coordenades  $(y, x)$  a comprovar. Aquests valors solament poden ser o una variable o un número literal.

## Codi generat

El codi generat per aquesta instrucció accedeix directament a la posició de memòria associada a la casella indicada i consulta el seu contingut. Recordem que el tauler de joc és una matriu de dimensions  $22 \times 11$ , per tant, per convertir les coordenades bidimensionals a una adreça lineal de memòria, és necessari multiplicar la coordenada  $y$  per 11 i sumar-hi la coordenada  $x$ .

- **Si la coordenada  $y$  és un valor literal:** es comprova que es trobi dins dels límits del tauler (entre 0 i 21). En cas afirmatiu, es realitza la multiplicació  $y \cdot 11$  durant la generació del codi i es genera directament el desplaçament correcte.
- **Si la coordenada  $y$  és una variable:** la multiplicació per 11 s'implementa mitjançant instruccions `assembler` durant l'execució. Per optimitzar el càlcul, primer es realitza un desplaçament a l'esquerra de 3 bits (equivalent a multiplicar per 8), i posteriorment se li afegeix tres vegades el valor original per aconseguir el resultat.

Un cop obtingut el desplaçament vertical, es carrega el valor de  $x$  i se li suma per obtenir l'índex total dins de la matriu. Posteriorment, s'hi afegeix una constant de  $0x16$  (22 en decimal) per compensar les dues primeres files que no són visibles a la pantalla però sí que estan representades a memòria.

Finalment, un cop determinada l'adreça de memòria corresponent a la casella  $(x, y)$ , es carrega el seu contingut al registre A, que indicarà si la casella està pintada (valor diferent de zero) o buida (valor zero).

```

def visitRead(self, ctx):
    left = ctx.simples(0).getText()
    if left.isdigit():
        left = int(left)
        if left > 19:
            raise Exception(f"Filà {left} no existent")
        h = hex(left * 11)[2:].upper().zfill(2)
        code = f"""
SEA {h}"""
    else:
        code = f"""
SMN {left}
LDA
LDB
LS3
ADD
ADD
ADD
"""

    right = ctx.simples(1).getText()
    if right.isdigit():
        right = int(right)
        if right > 9:
            raise Exception(f"Columna {right} no existent")
        h = hex(right)[2:].upper().zfill(2)
        code += f"""
SEB {h}"""
    else:
        code += f"""SMN {right}
LDB"""

    code += f"""
ADD
SEB 16
ADD"""
    code += f"""
SMN playfield
TAN
LDA
"""
    return code

```

Figura 42: Codi visitRead(). (*Elaboració pròpia*)

**Exemple:** `mira(4,7)`

Aquesta expressió mira el valor de la graella on  $y=4$  i  $x=7$

- **Peça:**

La instrucció `peça` constitueix una norma específica dissenyada per facilitar la manipulació de peces en el context del joc Tetris, que forma part de l'entorn desenvolupat. Aquesta construcció té com a objectiu recuperar el *mapa de bits* (bitmap) d'una peça concreta. L'estrucció de la instrucció consisteix en la paraula clau `peça` seguida de tres paràmetres:

- **Tipus de peça:** identifica la forma geomètrica.

- **Rotació:** especifica una de les rotacions possibles de la peça.
- **Byte seleccionat:** indica si es vol recuperar la primera o la segona meitat de la representació de la peça.

### Codi generat

Cada peça es representa mitjançant dues cel·les de memòria (dos bytes), i les diferents rotacions de cadascuna es troben emmagatzemades de forma contigua en una zona concreta del programa. Per accedir correctament a la porció desitjada de memòria, cal calcular un desplaçament sobre aquesta base segons els paràmetres proporcionats.

El càlcul de l'adreça es realitza com segueix:

$$\text{offset} = (\text{tipus\_peça} \times 8) + (\text{rotació} \times 2) + \text{byte\_index}$$

Aquest càlcul permet accedir directament a la representació de la peça, tenint en compte que:

- Cada peça té 4 rotacions.
- Cada rotació ocupa 2 bytes.
- Les peces estan ordenades primer pel tipus (T, Z, O, S, J, L, I) i després per rotació (És segueix el patró d'ordenació de la Figura 8, repetint rotacions a les peces que no tenen).

Per realitzar aquests operacions dins del llenguatge assemblador, es fa ús del registre B per emmagatzemar valors intermedis, cosa que limita la possibilitat de fer servir llistes o paràmetres més complexos en aquest punt del codi.

Aquest enfocament permet un accés ràpid i eficient a la informació gràfica d'una peça, mantenint una estructura compacta i optimitzada per a l'entorn limitat de l'arquitectura de baix nivell on s'executa el programa.

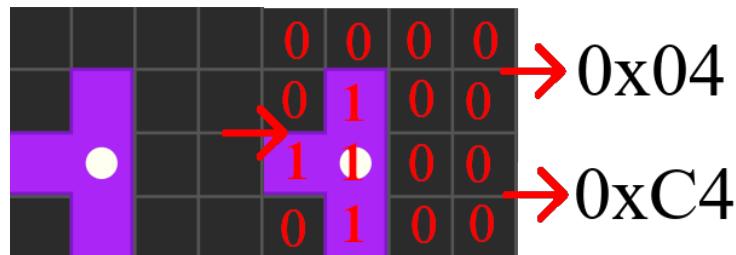


Figura 43: Codificació peça T en rotació 1. (*Elaboració pròpia*)

```

def visitPeça(self, ctx):
    code = self.visit(ctx.simples(1))
    code += f"""
TAB
ADD
TAB
"""
    code += self.visit(ctx.simples(0))
    code += f"""

LS3
ADD
TAB
"""
    code += self.visit(ctx.simples(2))
    code += f"""

ADD
SMN tETROMINOES
TAN
LDA
"""

    return code

```

Figura 44: Codi visitPeça(). (*Elaboració pròpia*)

**Exemple:** peça 6 3 1

S'agafa el byte inferior de la peça I en la posició vertical (iv).

- **Random (aleatori):** Proporciona un valor aleatori entre 0 i 6. El codi que s'utilitza ha estat extret del projecte Meat Fighter i implementa un generador de números pseudoaleatoris amb desplaçaments i operacions XOR.

**Nota:** L'explicació detallada del funcionament intern del generador es pot consultar al document original del projecte Meat Fighter.

## 4.12 Expressió

Les expressions són la forma més alta de la gramàtica, i permeten generar tot el conjunt d'instruccions que tindrà el codi. A continuació es mostra totes les parts de la seva gramàtica

### Gramàtica

```

expr      : VAR? 'ves' FUNC VAR? VAR? VAR? #Func
          | 'bucle' VAR expr 'I' #Bucle
          | 'per' VAR (NUMBER NUMBER| VAR) expr 'I' #For

```

```

| 'si' logic expr 'I'('sino' expr 'I') #If
| 'var' VAR ('=' NUMBER)? ('memoria' HEX4)? #Def
| 'llista' VAR NUMBER #Defllista
| VAR '=' op #Asig
| VAR '[' simples ']' '=' op #Asigllista
| 'parametre' UNDOSTRES '=' op #Paramasig
| expr expr # Expre
| 'surt' VAR #Break
| 'retorna' simples? #Return
| 'continua' VAR #Cont
| 'pinta' LPAREN simples COMA simples COMA simples RPAREN # Pintar
| 'actualitza' #Reset
| COMENTARI #Comenta
;

```

## Comportament de les regles

- **Crida a funció**

La crida a una funció es compon d'una variable (opcional) on es guardarà el resultat, seguida de la paraula clau **ves**, el nom de la funció (una ID en majúscules) i fins a tres paràmetres opcionals.

S'ha decidit que tant la variable de retorn com els paràmetres siguin sempre variables, ja que és més coherent dins el llenguatge. Si cal fer servir literals, es pot fer una assignació prèvia a una variable o posteriorment guardar el valor retornat on es vulgui.

## Codi generat

Cada funció ocupa 8 bytes a la pila. Per això, abans de fer la crida s'incrementa **fUNCTIONS** en 8. Aquesta posició indica l'espai reservat per la nova funció. Es col·loquen els valors dels paràmetres (si n'hi ha) dins l'espai assignat, i després es fa un salt a l'etiqueta que correspon a la funció.

Finalment, si s'ha indicat una variable de retorn, es copia el contingut de **rETURN** en aquesta variable i en tot cas es restaura el valor original de **fUNCTIONS**.

```

    ...
    def visitFunc(self, ctx):
        global etiqueta
        actual = etiqueta
        etiqueta = etiqueta + 'a'
        code = f"""
SMN FUNCTIONS
LDA
INC
INC
INC
INC
INC
INC
INC
INC
STA

SMN pila
TNB
ADD
SMN {actual}
TMB
SMN pila
TAN
STB

INC
SMN {actual}
TNB
SMN pila
TAN
STB"""

        if ctx.VAR(1):
            code += """
SMN FUNCTIONS
LDA
SMN pila
TNB
ADD
INC
INC
TAB
"""
            for i in range(1, 4):
                if ctx.VAR(i):
                    code += f"""SMN {ctx.VAR(i).getText()}"""
            LDA"""
            code += """
SMN pila
TBN
STA
TBA
INC
INC
TAB
"""
            code += f"""
JSR {ctx.FUNC().getText()}
{actual}:
"""

            if ctx.VAR(0):
                code += f"""
SMN rRETURN
LDA
SMN {ctx.VAR(0).getText()}
STA
"""
            code += f"""SMN FUNCTIONS
LDA
DEC
DEC
DEC
DEC
DEC
DEC
DEC
STA
STA
"""
        return code

```

Figura 45: Codi visitFunc(). (*Elaboració pròpia*)

Exemple: rRESULTAT ves MCM a b

Es va a la funció MCM on prametre(1) = a i parametre(2) = b. El valor que es retorna d'aquesta crida es guarda a rRESULTAT.

- **Bucle**

El bucle **bucle** és una estructura de repetició infinita. Està format per:

- La paraula clau **bucle**.
- Una variable que serveix únicament com a identificador del bucle (no emmagatzema cap valor).
- El bloc d'instruccions a repetir.
- La paraula clau **I**, que marca el final del bucle.

### Codi generat

Per implementar-lo, al igual que en el bucle for, es generen tres etiquetes associades amb la variable identificadora del bucle:

- Una etiqueta d'inici del bucle (**inicifor<var>**).
- Una etiqueta de *continue*, per controlar salts interns (**continuefor<var>**).
- Una etiqueta de final (**endfor<var>**), que serveix per trencar el bucle.

Després d'executar el bloc d'instruccions, es torna incondicionalment a l'etiqueta d'inici, creant així el comportament infinit.

Tot i que la variable usada no s'utilitza funcionalment dins el bucle, és important com a identificador per generar etiquetes úniques.

```
def visitBucle(self, ctx):
    global variables
    variable = ctx.VARO.getText()
    if variable.isdigit():
        raise Exception(f"No pots utilitzar un número com a ID en un bucle")

    body_code = self.visit(ctx.expr())

    variables += f"""
{variable}: 00
"""
    output = f"""

inicifor{variable}:
"""

    output += body_code
    output += f"continuefor{variable}:"
    output += f"""

JMP inicifor{variable}
endfor{variable}:
"""

    return output
```

Figura 46: Codi visitBucle(). (*Elaboració pròpia*)

```

Exemple:
bucle j
    instruccions
I

```

Aquest bucle repetirà les **instruccions** indefinidament, mentre no hi hagi cap instrucció de trencament.

- **For**

El bucle **per** (equivalent al *for* en anglès) permet iterar sobre un conjunt d'instruccions, modificant a cada iteració el valor d'una variable. Està format per:

- La paraula clau **per**.
- La variable iterativa, que com en la regla anterior serà el identificador de bucle.
- Dos nombres enters o bé una altra variable.
- El bloc d'instruccions a executar.
- La paraula clau **I** que marca el final del bucle.

### Codi generat

- Si es proporcionen dos nombres, la variable iteradora s'inicialitza amb el primer valor i s'incrementa o decrementa fins arribar al segon, canviant d'1 en 1. L'algorisme comprova si el valor inicial és més petit o més gran que el valor final, per saber si ha d'incrementar o decrementar la variable. També s'ajusten les instruccions de comparació que controlen la sortida del bucle segons aquest criteri.
- Si en lloc de dos nombres s'indica una variable, la variable iteradora s'inicia a 0 i s'incrementa fins arribar al valor d'aquesta.

S'han deixat aquestes dues variants perquè cobreixen els casos més habituals d'ús del bucle **for**. Tot i que es va considerar iterar sobre llistes, es va descartar per simplicitat i per mantenir la coherència amb la resta del llenguatge.

Per cada iteració:

- Es guarda la variable amb el valor actual.
- S'executa el bloc d'instruccions.
- S'aplica un increment o decrement.
- Es fa la comparació i es decideix si continuar o finalitzar.

És generen les mateixes etiquetes que en el bucle **bucle**, per a que el programador pugui manipular-lo a voluntat.

```
    ...
def visitFor(self, ctx):
    global variables
    variable = ctx.VAR(0).getText()
    if variable.isdigit():
        raise Exception(f"No pots utilitzar un número per iterar en un bucle")
    if ctx.VAR(1):
        inicial = 0
        final = 1
    else:
        inicial = int(ctx.NUMBER(0).getText())
        final = int(ctx.NUMBER(1).getText())
    body_code = self.visit(ctx.expr())
    augment = "DEC"
    compara = "BPL"
    if inicial < final:
        augment = "INC"
        compara = "BMI"
        final += 1
    inicial = hex(inicial)[2:].upper().zfill(2)
    final = hex(final)[2:].upper().zfill(2)

    if ctx.VAR(1):
        final = f"""SMN {ctx.VAR(1).getText()}"""
    LDB"""
    else:
        final = f"SEB {final}"

    variables += f"""
{variable}: 00
"""
    output = f"""
SMN {variable}
SEA {inicial}
STA
inicifor{variable}:
"""

    output += body_code
    output += f"continuefor{variable}:"
    output += f"""
{final}
SMN {variable}
LDA
{augment}
STA
SUB
{compara} inicifor{variable}
endfor{variable}:
"""
    return output

```

Figura 47: Codi visitFor(). (*Elaboració pròpia*)

**Exemple:**

```
per i 4 7
    instruccions
I
```

Aquest bucle assignarà a **i** els valors 4, 5, 6, 7 successivament i executarà les instruccions en cada iteració.

- **Si**

La instrucció condicional **si** és una de les estructures de control més habituals en programació. Permet executar un conjunt d'instruccions si una condició

lògica és certa, i opcionalment executar-ne un altre conjunt si la condició és falsa, mitjançant la clàusula **sino**.

L'estructura és la següent:

- La paraula clau **si**.
- Una expressió lògica (condició).
- Un bloc d'instruccions delimitat per la paraula clau **I**.
- Opcionalment, la paraula **sino**, seguida d'un altre bloc d'instruccions també acabat amb **I**.

### Codi generat

L'avaluació de la condició es fa sobre una expressió lògica que retorna un valor al registre **A**. Si el valor d'**A** és 1 (cert), s'executa el bloc de codi principal. Si és 0 (fals), s'executa el bloc del **sino**, si aquest existeix.

Per aconseguir això, s'utilitza un petit mecanisme de comparació:

- Es visita **i** es genera el codi la condició, cosa que deixa el valor resultant a **A**.
- Es resta 0 al valor de **A**.
- Això canvia el flag **Z** del processador (bit zero).
- Si **Z** és 0 (**A** era 1), significa que la condició és certa.
- Si **Z** és 1 (**A** era 0), la condició és falsa i se salta directament al bloc **sino**, si n'hi ha.

Aquest comportament es gestiona amb dues etiquetes:

- Una per saltar al final del bloc **si** si la condició és falsa.
- Una altra per saltar al final del bloc **sino** després d'executar el **si**, per evitar que s'executin ambdós blocs.

```

def visitIf(self, ctx):
    global etiqueta
    actual = etiqueta
    actual2 = etiqueta + 'e'
    etiqueta = etiqueta + 'a'
    code = self.visit(ctx.logic())
    body_code = self.visit(ctx.expr(0))
    body_code2 = ""
    if ctx.expr(1):
        body_code2 = self.visit(ctx.expr(1))
    code += f"""
SEB 00
SUB
BEQ {actual}
"""
    code += body_code
    code += f"""
JMP {actual2}
{actual}:
"""
    code += body_code2
    code += f"""
{actual2}:
"""
return code

```

Figura 48: Codi visitIf(). (*Elaboració pròpia*)

Exemple:

```

si condicio
    instruccions
I
sino
    instruccions2
I

```

Si la condició és certa s'executa instruccions, però si no ho és s'executa instruccions2

- **Definició**

Totes les variables que es volen utilitzar al programa han de ser declarades explícitament mitjançant la paraula clau **var**. Aquesta instrucció pot incloure opcionalment:

- Una assignació inicial mitjançant el símbol **=** i un número.
- Una localització concreta a la memòria mitjançant la paraula clau **memoria** i una adreça en format hexadecimal (4 dígits).

Per defecte, les variables s'inicialitzen a 0, i es col·loquen automàticament en una àrea de memòria gestionada pel compilador, a menys que s'especifiqui el contrari.

### Codi generat

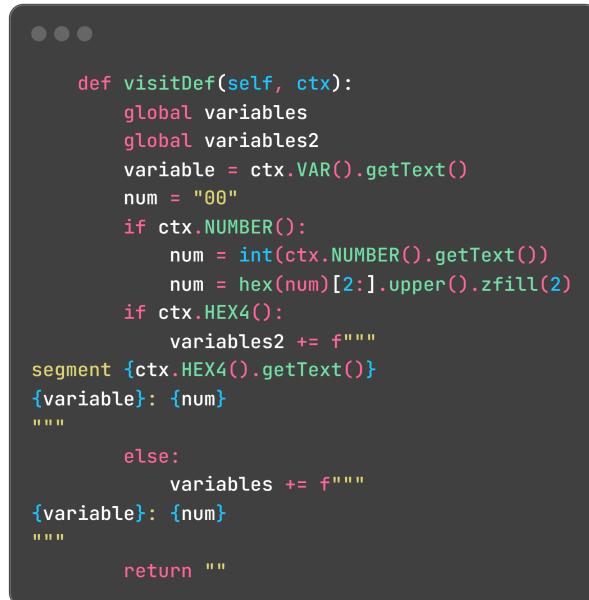
El compilador manté dues llistes de variables globals:

- **variables**: conté les declaracions de variables que no tenen una adreça de memòria especificada.
- **variables2**: conté les declaracions amb adreça fixa. Aquestes són útils, per exemple, per a gestió d'entrada/sortida, on és necessari saber exactament on es troba una variable.

Quan es compila una variable amb adreça fixa, es genera una secció de codi amb la directiva **segment** i l'adreça corresponent, això permet l'assembler gestionar la memòria.

En cas que no s'indiqui cap valor inicial, la variable és inicialitzada a 0x00.

No es permet inicialitzar una variable amb una altra variable, per simplificar la semàntica i evitar ambigüïtats, ja que aquesta pràctica no és habitual en declaracions. Si cal, es pot fer posteriorment amb una assignació.



```
def visitDef(self, ctx):  
    global variables  
    global variables2  
    variable = ctx.VAR().getText()  
    num = "00"  
    if ctx.NUMBER():  
        num = int(ctx.NUMBER().getText())  
        num = hex(num)[2:].upper().zfill(2)  
    if ctx.HEX4():  
        variables2 += f"""\nsegment {ctx.HEX4().getText()}\n{variable}: {num}\n"""\n    else:  
        variables += f"""\n{variable}: {num}\n"""\n    return ""
```

Figura 49: Codi visitDef(). (*Elaboració pròpia*)

```
Exemple: var oUTPUT = 5 memoria 0008
```

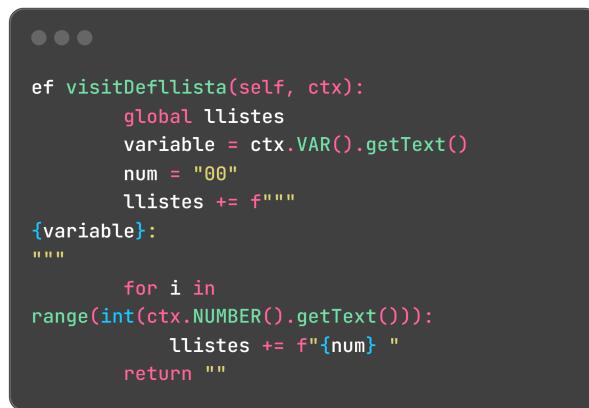
Aquesta declaració crea una variable anomenada `oUTPUT`, amb valor inicial 5, ubicada a l'adreça de memòria `0x0008`.

- **Definició llista**

Les llistes es declaren mitjançant la paraula clau `llista`, seguida del nom (`VAR`) i un número enter que indica la seva llargada. Tots els valors inicials dins la llista seran 0, i la memòria utilitzada es col·locarà en un segment especial destinat per a llistes.

### Codi generat

En memòria, una llista s'implementa com una seqüència contínua de bytes. El compilador simplement reserva espai escrivint una etiqueta amb el nom de la llista i afegint el nombre de valors 0 corresponents. Això permet accedir-hi mitjançant un desplaçament relatiu des del punt inicial, on està la etiqueta.



```
def visitDefllista(self, ctx):
    global llistes
    variable = ctx.VAR().getText()
    num = "00"
    llistes += f"""
{variable}:
"""
    for i in
range(int(ctx.NUMBER().getText())):
        llistes += f"{num} "
    return ""
```

Figura 50: Codi `visitDefllista()`. (Elaboració pròpia)

```
Exemple: llista a 10
```

Aquesta instrucció crea una llista anomenada `a` amb 10 posicions, totes inicialitzades a 0.

- **Assignació**

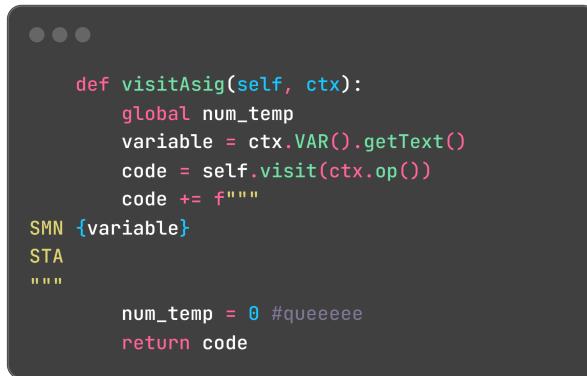
Permet assignar a una variable un valor calculat a partir d'una expressió, una instrucció essencial en qualsevol llenguatge. Realment es podria haver reutilitzat la gramàtica i el codi de les assignacions per a tots els casos fent `simples = op`, però en aquest cas només es permet assignar a variables, ja que sinó es permetria expressions com `4 = x`, que no tenen sentit.

## Codi generat

L'operació d'assignació es compon de dues fases:

1. Es calcula el valor de l'expressió que hi ha a la dreta del signe d'igual.  
Aquest valor quedarà guardat al registre A.
2. Es fa una crida a SMN amb el nom de la variable per establir el registre MN en l'adreça corresponent.
3. Es guarda el valor del registre A en aquella adreça mitjançant STA.

Com ja s'ha vist, s'utilitza la variable global `num_temp` per controlar els temporals. Després de fer una assignació es reinicia a zero ja que el càlcul aritmètic compost ja s'ha acabat i és necessari alliberar l'espai per poder tornar a utilitzar aquells temporals.



```
def visitAsig(self, ctx):  
    global num_temp  
    variable = ctx.VAR().getText()  
    code = self.visit(ctx.op())  
    code += f"""  
    SMN {variable}  
    STA  
    """  
    num_temp = 0 #queeeee  
    return code
```

Figura 51: Codi visitAsig(). (*Elaboració pròpia*)

Exemple: `b = 4`

S'assigna 4 a la variable b

- **Assignació llista**

Assigna un valor a una posició concreta d'una llista. L'índex dins dels claudàtors és un valor **simple**, i el valor a assignar pot ser una operació qualsevol, com en el cas de les variables.

## Codi generat

L'assignació en una llista implica calcular dues coses separades:

- L'índex de la posició dins la llista, el valor de `simples`.
- El valor que s'hi ha d'escriure, resultat de l'operació `op`.

Com que l'operació d'índex i l'operació de valor poden utilitzar els mateixos registres, es necessita una variable auxiliar, anomenada `m`, per emmagatzemar temporalment el resultat d'`op`. Després, s'agafa la adreça de la llista amb un SMN de la seva etiqueta i se li suma al bit inferior, el desplaçament de l'índex. Un cop trobada la direcció pertinent, s'hi escriu el valor que es tenia guardat a `m`. La variable auxiliar `m` és interna i no serà modificada per l'usuari ni per l'expressió `simples`, per tal de garantir la consistència del procés.

```

def visitAsigllista(self, ctx):
    global num_temp
    variable = ctx.VAR().getText()

    code = self.visit(ctx.op())
    code += f"""

SMM m
STA
"""
    code += self.visit(ctx.simples())
    code += f"""

SMM {variable}
TNB
ADD
SMM m
LDB
SMM {variable}
TAN
STB
"""
    num_temp = 0
    return code

```

Figura 52: Codi `visitAsigllista()`. (*Elaboració pròpia*)

**Exemple :** `LLISTA[3] = 4`

Aquesta línia assigna el valor 4 a la quarta posició (índex 3) de la llista `LLISTA`.

- **Assignació paràmetre**

Permet assignar un valor a un dels paràmetres d'una funció. La sintaxi utilitza la paraula `parametre` seguida d'un índex de l'1 al 3, inclosos, entre parèntesis.

## Codi generat

Els paràmetres de les funcions es troben a la pila, en una ubicació específica relativa al segment de pila on comença l'espai reservat per a la funció actual. Aquesta ubicació es determina mitjançant una referència a la variable **FUNCTIONS**, que indica la base actual dels paràmetres.

El procés d'assignació segueix els següents passos:

1. Es llegeix el valor de **FUNCTIONS** i se suma a la base de la **pila**.
2. S'incrementa el desplaçament per accedir al paràmetre concret (tenint en compte que cada paràmetre ocupa dues posicions a la pila).
3. Es guarda l'adreça resultant a una variable auxiliar anomenada **aUX**, perquè posteriorment es calcularà l'expressió de valor.
4. Un cop calculat el valor de l'expressió, es guarda aquest valor a la posició de memòria que apuntava **aUX**, accedint-hi a través de **pila** amb la direcció prèviament obtinguda.

L'ordre de les operacions i l'ús de la variable auxiliar són imprescindibles per evitar que l'avaluació de l'expressió **op** sobreescrigui el càlcul de l'adreça del paràmetre. És per això que **aUX** actua com una zona segura per emmagatzemar l'adreça temporal abans d'escriure-hi.

```

def visitParamasig(self, ctx):
    global num_temp

    code = """
SMN FUNCTIONS
LDA
SMN pila
TNB
ADD
INC
"""

    for i in range((int(ctx.UNDOSTRES().getText()[1]) * 2) - 1):
        code += "INC \n"

    code += """SMN aUX
STA
"""
    code += self.visit(ctx.op())
    code += """
SMN aUX
LDB
SMN pila
TBN
STA
"""
    num_temp = 0
    return code

```

Figura 53: Codi visitParamasig(). (*Elaboració pròpria*)

Exemple: parametre(3) = 7 + 4

Aquesta instrucció assigna el valor 11 al tercer paràmetre de la funció en execució.

- **Expressió**

Permet encadenar múltiples instruccions dins una mateixa expressió superior. Aquesta recursivitat és fonamental per construir seqüències d'instruccions en el llenguatge, una al costat de l'altra, formant programes complets.

### Codi generat

Cada crida a una expressió pot contenir dues expressions més. Això fa que el visitador processi primer la subexpressió de l'esquerra i després la de la dreta, unint el codi resultant en una sola cadena que serà executada de forma seqüencial.



```
def visitExpr(self, ctx):
    a = self.visit(ctx.expr(0))
    b = self.visit(ctx.expr(1))
    return a+b
```

Figura 54: Codi visitExpr(). (*Elaboració pròpia*)

Exemple:

```
b = 4
b = 3
```

Primer es posa la variable b a 4 i després a 3

- **Break**

Permet sortir immediatament d'un bucle. S'ha d'indicar el nom (ID) del bucle del qual es vol sortir, i es pot utilitzar tant en bucles infinitos com iteratius.

### Codi generat

El codi generat és simplement una instrucció de salt incondicional cap a l'etiqueta que marca el final del bucle corresponent. Això fa que s'abandoni immediatament l'execució del bucle.



```
def visitBreak(self, ctx):
    variable = ctx.VAR().getText()
    return f"JMP endfor{variable} \n"
```

Figura 55: Codi visitBreak(). (*Elaboració pròpia*)

Exemple: `surf i`

Es sort del bucle amb identificador i.

- **Return**

Permet retornar des d'una funció, opcionalment acompañat amb un valor simple, que és el que es guardarà a la variable de la instrucció que l'ha cridat.

#### Codi generat

1. Si hi ha un valor a retornar, aquest s'avalua i s'emmagatzema en la variable reservada `rRETURN`.
2. Tot seguit, es fa un salt a la subrutina especial anomenada `funcioreturn` que gestiona el retorn efectiu.



```
def visitReturn(self, ctx):
    a = ""
    if ctx.simples():
        a = self.visit(ctx.simples())
        a += f"""
SNN rRETURN
STA
"""
    code = a
    code += f"""
JSR funcioreturn
"""

    return code
```

Figura 56: Codi `visitReturn()`. (*Elaboració pròpia*)

Exemple: `retorna a`

La funció retorna el valor de la variable `a`.

- **Continue**

Permet finalitzar anticipadament la iteració actual d'un bucle i continuar amb la següent. Igual que amb `surt`, és necessari indicar a quin bucle s'aplica.

#### Codi generat

Es genera un salt directe cap a l'etiqueta `continueforX`, que ha estat definida just abans del salt al començament del bucle. Això fa que la resta de codi dins la iteració actual s'ometi i se'n comenci una nova.

```

    ...
def visitCont(self, ctx):
    variable = ctx.VAR().getText()
    return f"JMP continuefor{variable} \n"

```

Figura 57: Codi visitCont(). (*Elaboració pròpia*)

**Exemple:** continua i

El bucle amb identificador i, pasará a la següent iteració sense acabar la actual.

- **Pintar**

**pinta** és una altra instrucció específica per a l'entorn del **Tetris** i serveix per pintar amb el color indicat un bloc a una posició de la graella.

**pinta (Y, X, COLOR)**

On:

- Y és la fila (de 0 a 19)
- X és la columna (de 0 a 9)
- COLOR és un enter entre 0 i 6

### Codi generat

Utilitza el mateix sistema que el que s'ha vist a **mira** amb l'única diferència que un cop s'ha obtingut la posició de memòria que es vol modificar, s'hi guarda el valor del color, que pot ser un immediat o obtingut d'una variable.

El compilador llança excepcions si:

- La fila és major que 19
- La columna és major que 9
- El color és superior a 6

Aquest control evita escriure fora dels límits de la pantalla.

**Exemple:** pinta (7,2,3)

Es posa a la graella on y = 7 i x=2 el valor 3.

- **Actualitza**

La instrucció **actualitza** indica al simulador en Java que s'ha de redibuixar la pantalla gràfica amb els canvis realitzats en la memòria, especialment en l'àrea de **PLAYFIELD**.

**Codi generat**

La funció genera:

```
SMN drawFrame  
SEA 01  
STA
```

Aquest codi simplement posa a 1 la variable especial **drawFrame**, que el Java monitoritza constantment per saber quan ha d'actualitzar la interfície gràfica.

- **Comentari**

Aquesta regla permet inserir comentaris en el codi font que seran traslladats al codi generat. Això no afecta l'execució del programa.

**Exemple:** ; Aquest es un comentari

**Codi generat**

El compilador agafa directament el text del comentari amb el prefix ; inclòs i ho retorna com a codi.

- **Definició funcions**

Les funcions són blocs de codi reutilitzables definits per l'usuari. Comencen amb la paraula clau **funcio**, seguida del nom de la funció, les instruccions internes, i acaben amb el caràcter I, que marca el final de la definició.

**Codi generat**

- **visitFuncions**: concatena totes les funcions definides.
- **visitFuncio**: genera una etiqueta amb el nom de la funció i el cos del codi corresponent.

Aquest sistema permet crear tantes funcions com càpiguen en memòria.

```

def visitFuncions(self, ctx):
    code = ""
    for funcio_ctx in ctx.funcio():
        code += self.visit(funcio_ctx)
    return code

def visitFuncio(self, ctx):
    a = self.visit(ctx.expr())
    code = f"""
{ctx.FUNC().getText()}:
    """ + a
    return code

```

Figura 58: Codis visitFuncions() i visitFuncio(). (*Elaboració pròpia*)

Exemple:

```

funcio MCM
instruccions
I

```

Declaració de la funció MCM

#### 4.13 Limitacions del sistema

S'ha de tenir en consideració les següents coses pel bon funcionament del sistema:

- Els valors s'han de poder guardar en memòria (1 byte), per tant cap podrà superar el valor de 0xFF (255 en decimal)
- L'assembler no permet l'ús d'instruccions de branch amb valors superiors a 127, per tant totes les instruccions que utilitzen comparadors, com el **si** o el **per**, haurien d'actuar sobre valors inferiors.
- S'ha de respectar la quantitat de memòria assignada per cada cosa, per exemple no es crearan més de 80 variables, operacions combinades que requereixin més de 20 temporals, llistes no alineades o que s'estenguin de l'espai reservat per llistes, fer overflow de la pila (es recomana fer crides amb màxim 8 capes de profunditat) i les variables posicionades en memòria han d'estar ordenades de menor a major.
- S'ha de tenir molta cura en no repetir el nom de les variables ni utilitzar alguna paraula reservada per al funcionament del codi. Això implica noms d'etiquetes ja creades (ta, pila...), noms d'altres variables o temporals (FUNCTIONS, aUX, m, r...), noms de funcions, bucles o llistes (funcioreturn), símbols

de la pròpia gramàtica (i, o, si...) i tampoc noms de les intruccions assembler (INC, TAN).

#### 4.14 Creació d'interfícies d'usuari

Dins del directori `src\main\java\tetrominocomputer\gpc`, es troben els fitxers `.java` responsables de la simulació del computador. A continuació, s'analitzen els fitxers més rellevants, ja que seran aquells que es modificaran per a la creació d'una interfície d'usuari:

- **PlayfieldFrame:** fa ús de la llibreria `awt` de java per crear una finestra gràfica amb diversos components d'interacció amb l'usuari, com ara camps de text, botons, *labels* i gestió d'esdeveniments de teclat. Aquest projecte no se centra en el disseny gràfic d'interfícies, sinó en la gestió de dades i la seva relació amb la simulació, per la qual cosa no s'entrarà en profunditat en la llibreria `awt`.
- **PlayfieldModel:** actua com a intermediari entre *PlayfieldFrame* i *GeneralPurposeComputer*. El seu codi es basa principalment en un conjunt de mètodes `getter` i `setter`. El *PlayfieldFrame* desa la informació introduïda per l'usuari dins del model, i posteriorment, aquesta és recuperada per *GeneralPurposeComputer* per introduir-la dins del computador simulat. S'utilitza el fet que Java passa les llistes per referència, per poder guardar lectures de la memòria del simulador.
- **MCPProcessorAndMemory:** aquest component és l'encarregat d'executar instruccions i gestionar la memòria mitjançant el control dels cicles `Left` i `Right`. Manté una llista interna que representa la memòria del sistema, iniciatitzada amb el fitxer binari. Aquesta memòria es modifica durant l'execució per reflectir tant lectures com escriptures d'instruccions o dades.
- **GeneralPurposeComputer:** representa el nucli lògic de la simulació. Per una banda, interactua amb l'usuari a través del *PlayfieldModel*, i per altra, gestiona els accessos a la memòria mitjançant *MCPProcessorAndMemory*. Aquesta classe orquestra la comunicació entre la interfície gràfica i el funcionament intern del computador simulat.

#### Interfície gràfica del Tetris

La primera interfície implementada és una quadrícula de dimensions 10x20, en la qual es poden pintar blocs de fins a set colors diferents. Aquesta interfície es basa en la que es proporcionava per defecte en el projecte del *Meat Fighter*, tot i que s'han aplicat algunes modificacions lleugeres.

## Descripció de la interfície

Per a la gestió gràfica, s'utilitza una classe addicional anomenada *PlayfieldPanel*, la qual s'encarrega de representar visualment els blocs, aplicar els colors i en general millorar l'aspecte visual de la interfície. No obstant això, com que l'objectiu principal del treball no és el disseny gràfic, l'anàlisi se centrarà principalment en el comportament del *GeneralPurposeComputer*.

La funció `update`, present en aquesta classe i executada de manera cíclica, comença escrivint el valor 0 a la posició de memòria 0x00FD, i a continuació entra en un bucle que executa instruccions del codi binari fins que el contingut d'aquesta mateixa posició ja no sigui zero. Cal recordar que aquesta adreça de memòria està associada a la funció *drawFrame*, per la qual cosa aquest mecanisme és el que provoca que, quan el codi binari escriu un 1 a *drawFrame*, es produueixi una actualització de la pantalla. Aquest comportament és comú en totes les interfícies visuals utilitzades al projecte.

Un cop finalitzat el bucle d'execució, es recupera la matriu `cells` des de *PlayfieldModel*. Aquesta matriu és la que utilitza *PlayfieldFrame* per determinar quines cel·les han de ser pintades. Es realitza un recorregut per tota la matriu, llegint la memòria des de la posició 0x01F0 fins a la 0x0100, i assignant a cada posició de la matriu el valor de memòria corresponent. Com que en Java les llistes es passen per referència, qualsevol modificació en aquesta estructura afecta directament el component visual associat.

Tot seguit, el sistema força l'actualització del *PlayfieldFrame*, i escriu en memòria el resultat de la lectura d'esdeveniments del teclat. Concretament:

- A la posició 0x0000, s'escriu un valor booleà indicant si la fletxa esquerra ha estat premuda.
- A la posició 0x0001, el mateix però amb la tecla de la fletxa dreta.
- A la posició 0x0002, el mateix però amb la tecla de la fletxa amunt.



Figura 59: Imatge de la Interfície. (*Elaboració pròpia*)

### Implementació del Tetris

**Representació de les peces** Cada peça del joc està definida pels següents atributs:

- **X, Y:** Coordenades del centre de la peça dins del taulell.
- **TIPUS:** Valor enter que representa un dels 7 tipus de tetròminos (de 0 a 6).
- **ROTACIÓ:** Valor enter de 0 a 3, corresponent a les 4 possibles orientacions d'una peça.

Per facilitar el dibuix i la rotació, totes les peces estan centrades a la posició (1, 2) dins d'una graella de  $4 \times 4$  (Figura 8) .

Cada possible rotació d'una peça està codificada mitjançant dos *words* (4 bytes), que representen una matriu de  $4 \times 4$  (Figura 43) . Cada bit dins dels bytes indica si hi ha un bloc actiu (1) o buit (0). El bit amb més pes representa la posició (0, 0) (a dalt a l'esquerra), i es recorre de forma horitzontal i després vertical (ordre de lectura per files). Aquesta informació es guarda en la llista de 56 bytes de tamany, anomenada tETROMINOES.

La representació binaria de totes les peces és:

tetrominoes:

```
00 E4 ; T
04 C4
```

```
04 E0  
04 64  
  
00 C6 ; Z  
02 64  
00 C6  
02 64  
  
00 CC ; 0  
00 CC  
00 CC  
00 CC  
  
00 6C ; S  
04 62  
00 6C  
04 62  
  
00 E2 ; J  
04 4C  
08 E0  
06 44
```

```
00 E8 ; L  
0C 44  
02 E0  
04 46  
  
00 F0 ; I  
44 44  
00 F0  
44 44
```

La instrucció especial peça **TIPUS ROTACIÓ BYTE** permet obtenir la *word* corresponent a la part superior o inferior de la peça, segons el valor del tercer paràmetre (0 per dalt, 1 per baix).

Per pintar una peça, es recorre la seva codificació binària aplicant una màscara que comença amb valor 1 i es va desplaçant a l'esquerra (multiplicant per 2). A cada iteració es comprova si el bit és actiu; en aquest cas, es calcula la posició absoluta del bloc a partir de la posició actual de la peça i la posició relativa del bloc respecte al centre d'aquesta. Finalment es pinta al taulell.

Aquest procés es repeteix primer per la *word* superior i després per la inferior. L'ordre de pintat afecta les comprovacions posteriors de moviment i col·lisió.

## **Comprovació de col·lisions i moviments de la peça**

Durant el desenvolupament del control de les peces, es va plantejar un dilema respecte a com gestionar les comprovacions de col·lisió: realitzar-les només quan l'usuari intenta moure la peça, o bé fer-les en cada iteració del bucle de joc. Finalment, es va optar per la segona opció, ja que permet reaprofitar els càlculs ja efectuats durant la fase de pintat de la peça. Tot i que això introduceix un petit cost temporal per iteració, evita haver de duplicar càlculs cada cop que es vol moure una peça, fet que redundaria en una menor eficiència general.

### **Comprovació de moviment cap a l'esquerra**

Per determinar si una peça pot moure's cap a l'esquerra, s'aprofita el procés de pintat de cada bloc individual. Per a cada un dels quatre blocs que formen la peça, es comprova si la posició immediatament a l'esquerra ja està ocupada. Si alguna d'aquestes posicions està pintada, es considera que el moviment no és vàlid. S'utilitza una variable auxiliar, `vALIDE`, que comptabilitza quantes de les comprovacions han estat favorables. Només si el valor final d'aquesta variable és 4, es permet el desplaçament.

### **Comprovació de moviment cap a la dreta**

El cas del moviment cap a la dreta és més complex, ja que la peça es pinta de dreta a esquerra, i per tant no és possible determinar si un bloc adjacent a la dreta pertany a una altra peça o és part de la peça actual que ja ha estat pintada. Per resoldre aquest problema, es va implementar un sistema basat en una màscara auxiliar.

Aquesta màscara té una estructura idèntica a la que s'utilitza per pintar la peça, però amb la diferència que representa l'estat que tindrà la peça en la següent iteració de pintar. Això permet verificar si la posició adjacent a l'esquerra d'un bloc conté un altre bloc de la mateixa peça (hi ha el mateix número de blocs de la peça que tenen un bloc de la pròpia peça a la dreta, que de blocs que el tenen a l'esquerra). Per tant se sumarà 1 a la variable sempre que no hi hagi res a la dreta i sempre que a l'esquerra hi hagi un bloc de la pròpia peça (calculat gràcies a la màscara). Així, si la variable `vALIDD` arriba a 4, significa que tots els blocs tenen la posició de la dreta buida o tenen un altre bloc de la pròpia peça.

### **Comprovació de moviment cap avall**

El moviment cap avall es gestiona de manera molt similar al desplaçament cap a la dreta. Tanmateix, si s'hagués previst inicialment una codificació dels tetròminos pensada per pintar de dalt cap a baix, aquest moviment hauria pogut implementar-

se amb la mateixa simplicitat que el desplaçament cap a l'esquerra, tot i requerir un ajust general en els càlculs.

La màscara auxiliar, en aquest cas, comprova la posició immediatament superior per a cada bloc. Aquesta comprovació s'ha d'efectuar només en el byte inferior de cada *word*, ja que les peces estan codificades en dues parts. Per calcular la posició superior, es pot optar per multiplicar per 16 la màscara, o de manera equivalent, sumar quatre vegades el seu valor. Amb això, es determina si hi ha una col·lisió imminent.

No és necessari comprovar els límits del taulell, ja que aquests estan envoltats per blocs invisibles que actuen com a límit físic. A més, la pròpia lògica del programa i les condicions de límit incloses garanteixen que no es produueixin moviments il·legals.

```
playfield:
; 0 1 2 3 4 5 6 7 8 9 10
00 00 00 00 00 00 00 00 00 FF ; 0
00 00 00 00 00 00 00 00 00 FF ; 1
00 00 00 00 00 00 00 00 00 FF ; 2
00 00 00 00 00 00 00 00 00 FF ; 3
00 00 00 00 00 00 00 00 00 FF ; 4
00 00 00 00 00 00 00 00 00 FF ; 5
00 00 00 00 00 00 00 00 00 FF ; 6
00 00 00 00 00 00 00 00 00 FF ; 7
00 00 00 00 00 00 00 00 00 FF ; 8
00 00 00 00 00 00 00 00 00 FF ; 9
00 00 00 00 00 00 00 00 00 FF ; 10
00 00 00 00 00 00 00 00 00 FF ; 11
00 00 00 00 00 00 00 00 00 FF ; 12
00 00 00 00 00 00 00 00 00 FF ; 13
00 00 00 00 00 00 00 00 00 FF ; 14
00 00 00 00 00 00 00 00 00 FF ; 15
00 00 00 00 00 00 00 00 00 FF ; 16
00 00 00 00 00 00 00 00 00 FF ; 17
00 00 00 00 00 00 00 00 00 FF ; 18
00 00 00 00 00 00 00 00 00 FF ; 19
00 00 00 00 00 00 00 00 00 FF ; 20
00 00 00 00 00 00 00 00 00 FF ; 21
FF 00 ; 22
```

Figura 60: Representació de la graella en l'assembler. (*Treball Meat Fighter*)

Tot i que la solució que proposa la figura anterior és elegant, no es pot fer amb Tetris++ degut a la impossibilitat d'inicialitzar llistes, per tant s'haura d'inicialitzar els valors de pPLAYFIELD un per un.

### Comprovació de la rotació

Per validar la viabilitat d'una rotació, el sistema ha de comprovar que totes les caselles on aniria la nova peça girada estiguin buides o continguin blocs de la peça

actual. Això implica simular el pintat de la peça amb la nova rotació i comparar-lo amb la posició actual.

S'empren dues versions de la peça: la rotació actual i la rotació futura. S'aplica la màscara auxiliar per comprovar que les noves posicions no entrin en conflicte amb altres blocs, i es compta quantes coincidències són acceptables. Si totes quatre posicions són vàlides, la rotació es permet.

Aquesta comprovació puntual s'executa només quan l'usuari demana una rotació, a diferència dels desplaçaments que es comproven en cada iteració.

### Final de partida

El joc finalitza quan una nova peça no pot col·locar-se sense superar la Y màxima del taulell. En aquest cas, s'omple tot el taulell amb blocs del color de la peça que ha desbordat del taulell, com a senyal de fi de partida.

### Caiguda i fixació de peces

Un cop ja es disposa d'una variable que indica si la peça pot desplaçar-se una posició cap avall, només cal comprovar si el seu valor és 4. En cas afirmatiu, això indica que totes les posicions inferiors estan lliures, i per tant, la peça pot continuar descendint. En aquest cas, primer s'esborra la peça de la posició actual mitjançant el mateix procediment utilitzat per pintar-la, i posteriorment s'incrementa en una unitat la seva coordenada vertical. Això fa que en la següent iteració la peça es torni a pintar una posició més avall.

En cas contrari, és a dir, si la peça ja no pot continuar baixant, es considera que ha estat fixada al tauler. Entenem que una peça queda col·locada quan no pot moure's més cap avall, i per tant, no és necessari esborrar-la. Simplement es mantenen els blocs en la posició actual i es restableixen els atributs que defineixen la peça activa a la seva posició inicial. D'aquesta manera, es genera una nova peça de manera aleatòria, iniciant un nou cicle de moviment.

### Eliminació de línies

Un cop una peça ha estat col·locada al tauler, s'inicia un procés de comprovació per determinar si alguna de les quatre línies que pot haver ocupat ha estat completada. Si es detecta que una línia està completament ocupada per blocs, es procedirà a eliminar-la. La forma d'eliminació consisteix a desplaçar cap avall totes les línies superiors, copiant el contingut de cada línia des de la que té justament per sobre.

Aquest procés es fa bloc per bloc, comprovant el color del bloc situat just a sobre i assignant-lo a la posició actual. L'operació es repeteix per a totes les línies per sobre de la línia esborrada. Per optimitzar el rendiment, el bucle de còpia es finalitza quan es detecta una línia completament buida, ja que això implica que no cal continuar movent línies.

Per tal d'assegurar-se que s'eliminin correctament línies consecutives compleades (per exemple, si es completen dues o més línies de forma contigua), un cop s'ha eliminat una línia, es torna a revisar la mateixa posició en la següent iteració. Això garanteix que qualsevol nova línia que hagi passat a ocupar aquella posició també sigui revisada adequadament.

### Moviment

El control del moviment de la peça es realitza mitjançant una estructura condicional basada en les variables d'entrada especials, les quals estan associades a posicions de memòria específiques. Aquestes posicions són escrites pel programa Java quan es detecta una entrada de teclat, i permeten detectar ordres com moure la peça cap a l'esquerra, cap a la dreta o rotar-la.

La prioritat dels moviments es defineix explícitament en l'ordre següent: primer es comprova si s'ha premut la tecla de moviment cap a l'esquerra, després cap a la dreta i, finalment, la rotació. Aquesta jerarquia garanteix que només un moviment sigui processat per iteració de joc, evitant així conflictes o solapaments.

Abans d'executar qualsevol moviment, es verifica si aquest és vàlid. En el cas dels moviments laterals, aquesta comprovació ja s'ha dut a terme prèviament durant la iteració de càlcul de col·lisions, de manera que es pot procedir directament a modificar la coordenada horitzontal segons correspongui.

A més, després d'aplicar un moviment lateral vàlid, la coordenada vertical es reinicialitza al seu valor original al començament del *frame*. Aquesta mesura s'ha pres per evitar un comportament anomenat "diagonalitat", és a dir, el desplaçament simultani de la peça cap avall i cap als costats dins d'un mateix cicle de joc. Com que la validesa del moviment només s'ha calculat tenint en compte la posició vertical original de la peça, moure-la cap avall durant el mateix cicle podria portar-la a una posició no verificada.

### Flux de programa

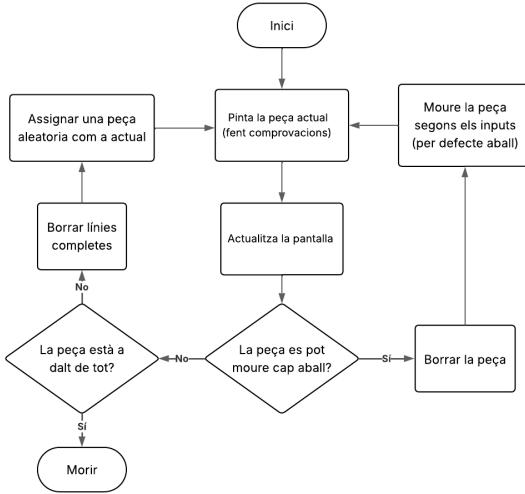


Figura 61: Diagrama de flux. (*Elaboració pròpia*)

### Consola

Aquesta interfície ha estat dissenyada per facilitar l'execució de qualsevol codi que tingui com a entrada i sortida únicament valors numèrics. Es pot entendre com una calculadora avançada capaç d'executar funcions computacionals arbitràries, o com una terminal simplificada orientada al processament numèric.

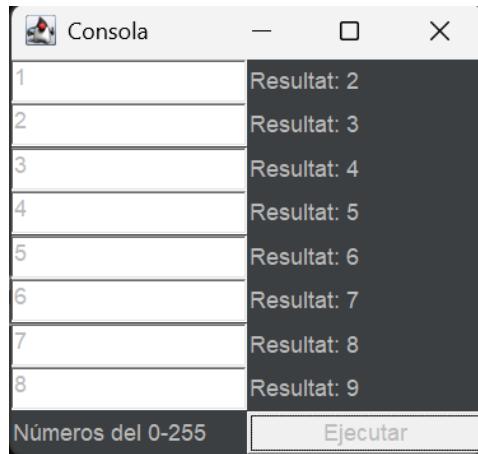


Figura 62: Programa que suma una unitat a tots els inputs . (*Elaboració pròpia*)

**Funcionament** La interfície consta de vuit camps de text que funcionen com a entrades, un botó per executar, i vuit etiquetes (*labels*) per mostrar les sortides. Els camps de text només accepten valors decimals entre 0 i 255, límit imposat pel fet que aquests valors han de ser codificats en un sol byte (8 bits) abans de ser escrits a la memòria del computador simulat.

Les sortides, de la mateixa manera, són valors numèrics que es llegeixen d'adreses de memòria específiques i es mostren als corresponents *labels* de sortida.

**Mecanisme d'execució** Quan l'usuari fa clic al botó d'execució del *Frame*, els valors introduïts als camps de text s'envien al *Model*. A continuació, el *General-PurposeComputer* escriu aquests valors a la memòria, concretament a les posicions compreses entre 0x0008 i 0x000F.

Després de l'execució del codi binari, les sortides són recollides de les posicions de memòria de 0x0000 a 0x0007 i es mostren a les etiquetes de sortida.

Cal destacar que, degut al funcionament intern del *GeneralPurposeComputer*, primer s'escriuen les dades d'entrada i just després es llegeixen les dades de sortida per mostrar-les per pantalla. Això pot provocar que, si el càlcul és lleugerament llarg, es mostrin dades desactualitzades. Per tal de mitigar aquest efecte i fer el comportament del sistema més predictable, s'ha optat per actualitzar els camps de text de sortida només quan es prem el botó d'execució. Així, s'assegura que sempre que es premi el botó es mostrin les dades corresponents a l'operació anterior, sempre que aquesta hagi finalitzat.

**Aplicacions desenvolupades** Aquesta interfície és la més versàtil del projecte, i s'han implementat nombrosos programes (fibonacci, potències, nombres aleatoris...), que poden trobar-se als anexos. A continuació es mostren dos exemples representatius.

- **Descomposició en divisors**

Aquest programa rep un número com a entrada i retorna tots els seus divisors a les sortides.

Tot i que el programa hauria estat molt més senzill si les entrades i sortides es gestionessin com a llistes (es podria haver utilitzat l'espai a partir de 0x0100 per fer-ho), en aquest cas s'ha decidit utilitzar variables individuals per mostrar una implementació alternativa. En la següent interfície s'utilitzarà el sistema de llistes. Donat això el codi serà tediós i repetitiu.

El codi comença inicialitzant les variables.

```
var a
var oUTPUT memoria 0000
var oUTPUTT memoria 0001
var oUTPUTTT memoria 0002
var oUTPUTTTT memoria 0003
var oUTPUTTTTT memoria 0004
var oUTPUTTTTTT memoria 0005
var oUTPUTTTTTTT memoria 0006
var oUTPUTTTTTTTT memoria 0007
```

```
var iNPUT memoria 0008
```

- Es llegeix el valor d'entrada i s'emmagatzema a **a**.
- Es prova per força bruta (no es la millor manera) un valor **x** que divideixi exactament **a**.
- Quan se'n troba un, s'escriu a la sortida corresponent i es divideix **a** per **x**.
- Si **a** és més gran que 1, s'entra a un bucle exactament igual que l'anterior però que guarda el resultat en un altre output.
- Un cop acabat (**a == 1**), s'actualitza la pantalla.
- Finalment, es posen totes les sortides a 1 per reinicialitzar-les. Això evita que, en càlculs posteriors amb menys divisoris, es mantinguin valors antics a les posicions no sobreescrites.

#### • Factorialització amb Recursió

L'objectiu d'aquest exercici és resoldre el problema P12509 del jutge, que consisteix a calcular el factorial d'un nombre utilitzant recursitat. Per fer-ho, s'utilitza únicament la posició d'entrada 0x0000 (input 0) i la de sortida 0x0000 (output 0).

El codi principal simplement fa una crida a la funció FACTORIAL, que s'ha implementat de manera recursiva, enviant com a paràmetre el valor del qual es vol calcular el factorial.

```
funcio FACTORIAL
    si parametre (1) > 1

        d = parametre (1) - 1
        b ves FACTORIAL d
        rES = b * parametre (1)
        retorna rES
    I
    sino
        retorna 1
    I
    retorna 0
I
```

Es calcula el factorial de  $n - 1$  fins arribar al cas base, que és 1, que llavors retorna 1. Aleshores, el resultat de la crida es multiplica pel valor actual del paràmetre, retornant així el resultat del factorial de  $n$ .

El últim retorna, tot i no fer res, és necessari per a que funcioni el codi.

Tot i que també es podria prendre com a cas base el valor 2, l'ús de 1 és més fidel a la definició matemàtica del factorial.

Potser pròximament veurem Tetris++ al jutge? ;)

## DFA – Autòmat Determinista Finit

La interfície és similar a la de la **Consola**, però en aquest cas s'ha dissenyat específicament per simular el comportament d'un autòmat determinista finit. L'entrada accepta paraules formades a partir d'un alfabet (per defecte {a, b}), i la sortida indica si la paraula és acceptada o no per l'autòmat definit.

**Funcionament de la interfície** L'usuari introduceix una paraula en un camp de text. Un cop premut el botó d'execució:

- La paraula es converteix en una llista d'enters corresponents al valor ASCII de cada lletra.
- Aquesta llista s'escriu a memòria a partir de la posició 0x0100, la qual ha estat reservada per a les llistes. Cada lletra ocupa un byte.
- A la posició 0x0009 s'emmagatzema la longitud de la paraula.
- A la posició 0x0008 s'escriu un senyal que indica que el botó ha estat premut, de manera que deixa la possibilitat que el programa només s'executi quan es detecta aquesta condició.
- El resultat (acceptació o no de la paraula) s'escriu a la posició 0x0000. El valor 1 indica acceptació, i el valor 2, rebuig. En la interfície aquests números es converteixen en un *string* entenedor.

**Consideracions sobre l'eficiència** Tot i que una possible millora en eficiència seria representar les lletres amb un sol bit (per exemple, a = 0, b = 1), aquesta implementació opta per utilitzar el valor ASCII complet. Això permet major versatilitat a l'hora de canviar o ampliar l'alfabet, a costa d'un major ús de memòria (cada lletra ocupa un byte, vuit vegades més del que podria ocupar).

**Implementació del DFA** Aquest projecte no es limita a l'objectiu específic de reconèixer una paraula mitjançant un comptador, sinó que vol mostrar com es pot implementar qualsevol DFA utilitzant el model de memòria i execució dissenyats. Per tant, en una variable està emmagatzemat l'estat en què es troba el programa actualment i a partir d'una seqüència de condicionals podem imitar les transicions. Si apareix una 'a', es canvia d'estat segons l'estat actual; si apareix una 'b', es va cap a un altre.

S'han implementat els dos DFA següents.

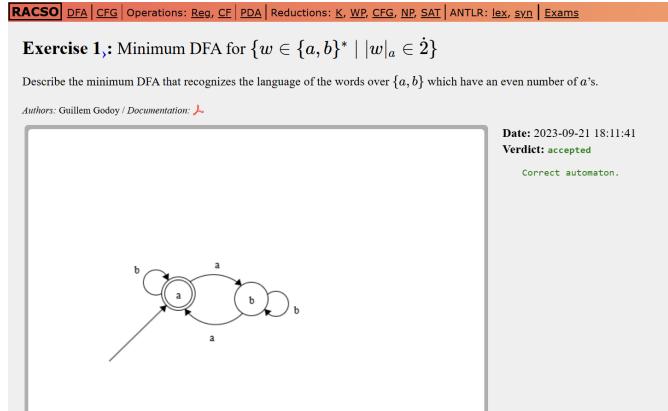


Figura 63: Exercici 1 del RACSO. (RACSO)

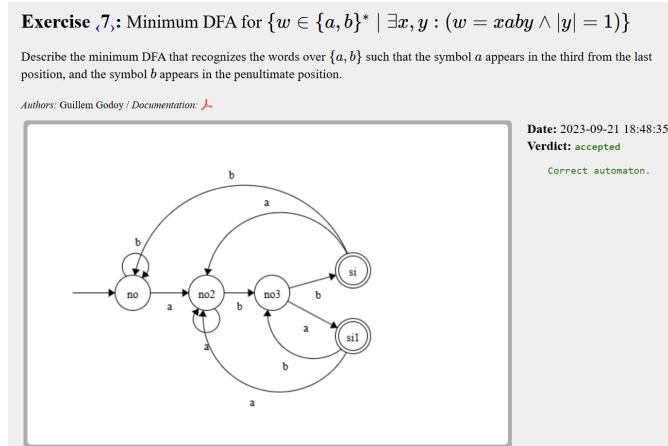


Figura 64: Exercici 7 del RACSO. (RACSO)

## 5 Resultats

Al llarg d'aquest treball s'ha entès amb un alt nivell de profunditat com és possible programar utilitzant el Tetris com a motor de computació. Aquest coneixement, a més, ha estat sintetitzat i explicat de manera que qualsevol persona amb nocions bàsiques de programació pugui entendre com funciona aquest sistema tan particular.

L'objectiu principal del projecte s'ha assolit amb èxit: s'ha desenvolupat una eina funcional que permet programar amb tetròminos d'una manera accessible i àgil. El llenguatge dissenyat i el compilador construït han demostrat ser capaços de suportar pràcticament totes les estructures i instruccions típiques dels llenguatges d'alt nivell, els quals permeten crear una gran varietat de codis dins del simulador.

Tot i això, cal destacar que la gestió d'errors ha estat una part poc desenvolupada del projecte. A posteriori, s'ha fet palès que hagués estat beneficiós incorporar una capa mínima de robustesa o algun sistema d'ajuda per identificar i corregir

errors de manera més intuïtiva per a l'usuari.

A més, s'han creat tres interfícies molt diferents que han permès expandir els límits de la programació amb tetròminos, donant peu a crear una gran varietat de programes adaptats a cada entorn. Un dels punts culminants ha estat la programació del propi joc del Tetris dins del simulador, un repte tècnic considerable que, tot i la seva lentitud, demostra les possibilitats reals del sistema.

Pel que fa a la difusió, aquest ha estat probablement l'objectiu menys assolit. Tanmateix, es preveu publicar el projecte a GitHub properament per tal que pugui ser consultat, utilitzat i ampliat per qualsevol persona interessada. Estic convençut que aquest treball té un gran potencial didàctic, ja que ofereix una manera original i entretinguda d'aprendre sobre computació a baix nivell, disseny de llenguatges i construcció de compiladors.

## 6 Conclusions

### 6.1 Conclusions tècniques

La primera conclusió, i la més evident, és que Tetris és capaç d'executar programes. Tot i això, s'ha comprovat que ho fa de manera extremadament lenta, llavors no sembla una bona opció per a usos pràctics.

El desenvolupament ha estat totalment condicionat per l'assembler creat per Meat Fighter, ja que tota la implementació es basava en ell. A més, aquest assembler només permetia accedir als registres A, B i MN, fet que impedia l'accés directe al *program counter* o a bits importants com el *zero flag* (Z).

Una altra conclusió rellevant és que és molt complicat controlar variables en temps d'execució sense restringir l'usuari ni definir tipus de dades.

També s'ha posat de manifest la complexitat tècnica del projecte, especialment per la necessitat d'una comprensió profunda de l'assembler. Per exemple, l'ús correcte dels registres A i B requeria un gran coneixement de com funciona el sistema, per evitar perdre el context. A més, calia intentar utilitzar el mínim nombre de *labels* i variables temporals per reduir l'accés a memòria, tot i que en molts casos ha estat impossible evitar-ne l'ús.

No s'ha implementat cap sistema d'optimització del compilador, ni cap detector d'errors, dues funcionalitats que haurien estat molt útils. No obstant això, com que el projecte ha estat desenvolupat per una sola persona, aquestes millores s'han considerat fora de l'abast actual.

De cara al futur, seria interessant que es pogués crear la interfície des del mateix llenguatge Tetris++. Aquesta idea es va considerar, però finalment es va descartar, ja que, en tractar-se d'un llenguatge d'alt nivell, el programador pot implementar la interfície pel seu compte. Per tant, l'esforç necessari no compensava els beneficis que s'obtindrien.

## 6.2 Valoració personal

Aquest treball m'ha deixat un gust agre dolç pel que fa a la satisfacció. Tot i haver-hi dedicat moltes hores i molta passió, i haver aconseguit molt més del que m'esperava, tinc la sensació que encara hi ha aspectes que podria millorar si tingués més temps. Però, com va dir Leonardo da Vinci: “Una obra d'art mai no s'acaba, només s'abandona”, i ha arribat el moment d'abandonar aquesta.

Tot i així, no me'n vaig amb les mans buides. Aquest TFG m'ha aportat un gran bagatge de coneixements. Encara que és cert que algunes coses ja les sabia, i que ja havia creat un compilador a l'assignatura de CL, aquesta ha estat una experiència totalment diferent. Enfrontar-me cara a cara amb tots els problemes que comporta crear un compilador, i haver de treballar sempre a un nivell tan baix, ha estat tot un repte.

Una de les descobertes més sorprenents i estimulants del projecte ha estat la possibilitat de crear codi autoprogramable, és a dir, fragments de codi que generen o modifiquen altres parts del programa de manera automàtica.

Es pot dir que aquest treball ha estat molt particular i ambicions, però ha estat precisament això el que m'ha motivat a seguir. Tot i que al principi em semblava del tot impossible, a poc a poc he anat aprenent i, després de molts errors, les coses han començat a sortir.

Espero que us hagi agradat, o que almenys us hagi aportat alguna cosa.

## 7 Bibliografia

Meat Fighter. (2023). Tetromino computer. Meat Fighter. <https://meatfighter.com/tetromino-computer/>

BOE. (2023). Conveni empreses enginyeria. [https://www.boe.es/eli/es/res/2023/02/27/\(6\)](https://www.boe.es/eli/es/res/2023/02/27/(6))

Turing, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 42(2), 230-265.

Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.

OpenAI. (2025). ChatGPT: Applications and Limitations in Artificial Intelligence Research. OpenAI. <https://openai.com>

Facultat d'Informàtica de Barcelona. (2025). Documents de l'assignatura de Gestió de Projectes (GEP). Universitat Politècnica de Catalunya.

Facultat d'Informàtica de Barcelona. (2025). Documents de l'assignatura de Compilació (CL). Universitat Politècnica de Catalunya.

ANTLR. (s. f.). ANTLR (ANother Tool for Language Recognition). <https://www.antlr.org/>

AWL de Java. (s. f.). Abstract Window Toolkit - Java Platform.  
<https://docs.oracle.com/javase/tutorial/uiswing/>

Universitat Politècnica de Catalunya. (s. f.). RACSO – Jutge automàtic UPC.  
<https://racso.cs.upc.edu/juezwsgi/index>

Jutge.org. (s. f.). Jutge.org: Online judge for programming. <https://jutge.org/>

Ray.so. (s. f.). Ray.so – Beautiful images of your code. <https://ray.so/>