POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Abilitare nuove e future applicazioni tramite QR code eseguibili

Politecnico di Torino

Relatori

Dr. Stefano Scanzio

Dr. Gianluca Cena

Candidato

Matteo Rosani

Luglio 2023

Alla mia famiglia,
per avermi sempre incoraggiato e supportato

"Non ho talenti speciali. Sono solo appassionatamente curioso."

Albert Einstein

Sommario

I codici QR sono codici a barre a matrice solitamente utilizzati per memorizzare informazioni come URL, testi, numeri di telefono e altro che verranno poi letti con appositi lettori ottici o addirittura uno smartphone. Possono contenere una quantità limitata di dati e, per questo motivo, richiedono spesso una connessione a Internet per fornire un servizio più complesso visualizzato su un sito Web accessibile tramite un URL.

A volte, tuttavia, non è disponibile una connessione Internet. Si pensi, ad esempio, a luoghi come l'alta montagna, o addirittura luoghi dove la connessione Internet è volutamente limitata, come in un impianto industriale. In questi casi, come fornire un servizio?

In questa tesi è stata progettata e sviluppata una tecnica per inserire la logica di elaborazione all'interno del codice QR. In particolare sono stati formalizzati due linguaggi di programmazione di facile utilizzo, che possono essere compilati, memorizzati all'interno del crittogramma e successivamente interpretati su un dispositivo dotato di macchina virtuale.

Il primo linguaggio di programmazione che abbiamo formalizzato si chiama QRtree. Ci permette di descrivere alberi decisionali con una sintassi semplice che ricorda il linguaggio di programmazione Python. Gli alberi decisionali sono utili quando è necessario guidare l'utente nel processo decisionale. Con questa tecnologia, possiamo integrare un semplice albero decisionale all'interno di un codice QR per aiutare gli utenti nella decisione di cosa fare in situazioni in cui non è disponibile una connessione Internet.

Il secondo linguaggio di programmazione che abbiamo formalizzato si chiama QRprog. Ci permette di descrivere un programma più generale con una sintassi che ricorda il linguaggio di programmazione C. Questo linguaggio è stato sviluppato per dare la possibilità di esprimere algoritmi più complessi che un albero decisionale non può rappresentare. Introduce la possibilità di loop, di creare funzioni e, soprattutto,

di suddividere il codice su più codici QR senza la necessità di scansionarli tutti prima di iniziare l'esecuzione.

Il codice memorizzato interagisce con l'utente senza richiedere una connessione e può fornire un aiuto in situazioni in cui non è possibile accedere alle informazioni sulla rete.

Il lavoro di tesi è stato sviluppato da me e Mattia Scamuzzi. Ci siamo concentrati su diversi aspetti durante lo sviluppo. Si tenga presente che il design dei linguaggi assembly di basso livello, che è il vero focus di questo progetto di tesi, è stato fatto insieme per avere una base comune per lo sviluppo.

In particolare, per il primo linguaggio proposto, QRtree, ho lavorato principalmente sul linguaggio ad alto livello sviluppando la toolchain di compilazione per trasformare il codice di alto livello in codice di tipo assembly di basso livello e l'interprete per l'esecuzione dell'assembly. Mattia Scamuzzi si è concentrato sullo sviluppo della toolchain di compilazione per trasformare il codice di basso livello in binario e viceversa.

Per il secondo linguaggio, QRprog, ho lavorato principalmente sul linguaggio di basso livello sviluppando la toolchain di compilazione per trasformarlo in binario, gestendo riferimenti e simboli definiti su più file, e anche la trasformazione inversa. Mattia si è concentrato sullo sviluppo della toolchain di compilazione per trasformare il codice di alto livello in codice assembly e un interprete per eseguirlo.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

La tecnologia sviluppata in questa tesi permette di incorporare intelligenza all'interno di codici QR. Questi possono successivamente essere scansionati ed eseguiti anche in assenza di una connessione ad Internet. Questo è un vantaggio in quanto in molti luoghi non è disponibile o non si vuole rendere disponibile una connessione alla rete, ma si ha la necessità di accedere a delle informazioni aggiuntive per poter completare un'operazione.

Una possibile applicazione è attraverso il *Decision Tree Dialect* fornire una guida alla risoluzione dei problemi di funzionamento di un macchinario in un ambiente industriale o guidare l'utente nel prendere una decisione su quale percorso di montagna scegliere in base alle sue capacità.

La configurazione dei parametri di funzionamento di reti industriali di elevata complessità [1, 2], sia wireless [3, 4] che cablate [5, 6], può essere facilita dall'utilizzo di codici QR. In particolare, a causa della loro eterogeneità, sia per quanto riguarda i servizi (ad es., sincronizzazione [7, 8]) che per quanto riguarda i protocolli di comunicazione (IEEE 802.11 [9], IEEE 802.15.4 [10], etc.), le operazioni di configurazione possono risultare molto complesse.

Altri dialetti specifici possono essere sviluppati in base alle esigenze specifiche di un ambito applicativo, come ultimo ripiego un *General Purpose Dialect* fornisce le funzionalità di un linguaggio di programmazione generico al costo di avere una minore compattezza del codice binario da inserire all'interno del codice QR.

1.1 Struttura

Obiettivo principale è quello di sviluppare un metodo per codificare all'interno di un codice QR un programma eseguibile e con cui l'utente può interagire attraverso una "macchina virtuale" installabile su un generico dispositivo.

Il problema principale è il riuscire a produrre un codice eseguibile il più compatto possibile, ma che allo stesso tempo fornisca la maggior espressività possibile.

Per risolvere questo problema, l'approccio scelto è di dividere l'implementazione in vari dialetti specifici per un ambito di applicazione. I dialetti rappresentano specifici linguaggi di programmazione con caratteristiche peculiari in termini di occupazione e di espressività. In questo modo riducendo le possibilità a uno specifico campo di applicazione possiamo ottenere la massima compressione possibile.

Un altro approccio che abbiamo previsto è la possibilità di concatenare più codici QR per avere una maggiore capacità di memorizzazione.

Le possibili applicazioni di questa tecnologia sono molteplici, ma hanno tutte in comune un ambiente in cui la connessione a Internet è assente o non si vuole avere un server web dedicato.

Questa tesi è stata divisa nei seguenti capitoli:

- Sezione 1, per dare un'introduzione generale al problema e alle possibili applicazioni
- Sezione 2, in cui è presentata la soluzione proposta, la struttura di essa e parti comuni alle sezioni successive
- Sezione 3, in cui è presentato il *General Purpose Dialect*
- Sezione 4, in cui è presentato il *Decition Tree Dialect*
- Sezione 5, dove un esempio di codice eseguibile viene mostrato e trasformato in codice QR e eseguito
- Sezione 6, in cui sono presentate le conclusioni

1.2 Codici QR

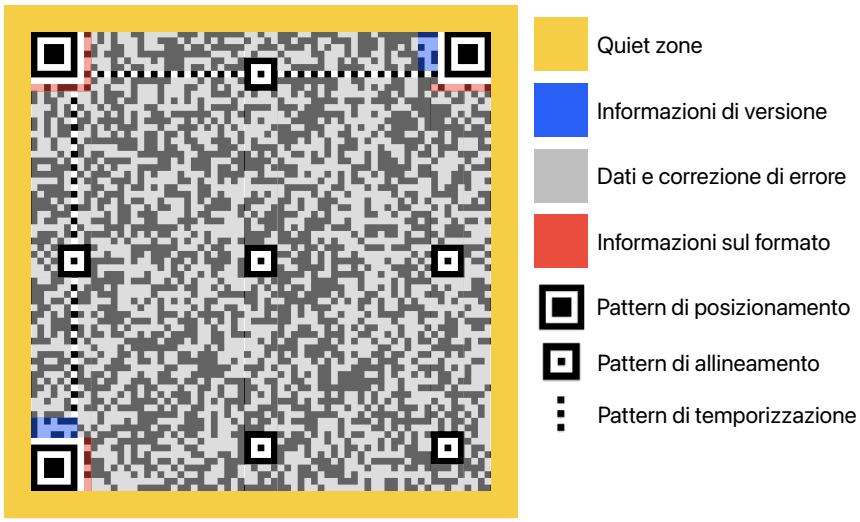

Figura 1.1: Struttura di un codice QR

Un codice QR è un codice a barre bidimensionale di forma quadrata composto da quadrati neri su uno sfondo bianco. Sviluppato inizialmente in Giappone nel 1994 per tracciare i pezzi sulla linea di produzione delle automobili Toyota. Distribuita successivamente sotto licenza libera questa tecnologia ha avuto un crescente utilizzo grazie anche all'avvento di smartphones con fotocamere in grado di decodificare il crittogramma.

È stato standardizzato secondo lo standard ISO/IEC 18004 la cui ultima revisione risale al 2015 [11].

All'interno possono essere memorizzati vari tipi di dati (numerici, alfanumerici, binari, Kanji/Kana) con diverse capacità. Inoltre se i tipi di dati sono misti si ha una divisione in blocchi e per ogni blocco un identificatore di tipo è specificato.

Incide sulla capacità anche il numero di *versione* che indica lo spazio fisico di memorizzazione. Si hanno dalla versione 1 alla versione 40 con un crescente numero di moduli, si parte da un quadrato 21x21 fino ad arrivare a 177x177.

Inoltre, per aumentare l'affidabilità della lettura si usa un codice di correzione di errore [12] che può essere impostato su 4 livelli (L, M, Q, H), con ognuno di questi livelli una parte del crittogramma viene utilizzata per permettere la lettura anche nel caso in cui l'immagine sia macchiata o graffiata. Si riesce a ottenere, con

il grado più alto di correzione di errore, una lettura anche con il 30% del codice QR danneggiato al costo di una sostanziale riduzione della capacità.

Per la lettura del crittogramma sono posti ai quattro angoli dei segnaposti per posizionare e allineare il lettore correttamente, in versioni più grandi possono essere presenti in maggior numero. Inoltre il codice deve essere circondato da una *zona di guardia* di altro colore per delimitare l'area di lettura.

Essendo un codice bidimensionale è necessario inserire una *sequenza di timing* per sincronizzare il lettore sulle due direzioni di lettura, questa sequenza è composta da un alternarsi costante di moduli neri e bianchi presente su entrambe le direzioni.

La codifica di dati all'interno di un codice QR può avvenire in diversi modi in base a che tipo di dato si vuole memorizzare, detta *modalità di input*. Diverse modalità hanno diverse densità di memorizzazione.

Inoltre nei codici QR è utilizzato un codice per la rilevazione e correzione di errore che permette di leggere un codice QR in parte danneggiato e di recuperare fino a una certa percentuale dei dati persi in base a che *livello di correzione* si è scelto nella fase di creazione.

Diverse *modalità di input*:

- *numerico*, memorizza caratteri numerici (0-9) fino a un massimo di 7089 caratteri
- *alfanumerico*, memorizza caratteri alfanumerici (0-9, A-Z, spazio, '$', '%', '*', '+', '-', '.', '/', ':') fino a un massimo di 4296 caratteri
- *binario*, memorizza byte grezzi fino a un massimo di 2953 bytes
- *kanji/kana*, memorizza caratteri dell'alfabeto giapponese fino a un massimo di 1817 caratteri

Diversi *livelli di correzione*:

- *Livello L*, permette di ripristinare circa il 7% dei dati
- *Livello M*, permette di ripristinare circa il 15% dei dati
- *Livello Q*, permette di ripristinare circa il 25% dei dati
- *Livello H*, permette di ripristinare circa il 30% dei dati

Attualmente i codici QR sono utilizzati in una varietà di contesti applicativi. Nella Tabella 1.1 sono citati alcuni riferimenti per i vari campi in cui sono utilizzati i codici QR.

Ambito	Riferimento	Descrizione
Anticontraffazione	[13]	Implementare e migliorare la qualità di un sistema anticontraffazione attraverso l'utilizzo di codici QR
Realtà aumentata	[14]	Codici QR per la personalizzazione di oggetti visualizzati dalla realtà virtuale
Configurazione automatica	[15]	Codici QR utilizzati per configurare automaticamente le reti di sensori IoT nell'industria
Velocità di lettura	[16]	Applicazione mobile che consente di leggere codici QR densamente posizionati
Identificazione	[17]	Codici QR e bracci robotici per gestire automaticamente i libri in una biblioteca
Localizzazione	[18]	Codici QR utilizzati per migliorare la localizzazione dei robot mobili
Manutenzione	[19]	Codici QR utilizzati per eseguire la manutenzione predittiva e l'autocalibrazione di un braccio robotico
Sanità	[20]	Incorporamento di un segnale ECG all'interno di un codice QR
Pagamento	[21]	Come viene percepito dagli utenti l'utilizzo dei codici QR per i pagamenti mobili
Riciclo	[22]	Uso combinato di codici QR e blockchain per costruire una piattaforma di riciclaggio
Sicurezza	[23]	Gestione della sicurezza in cantiere tramite codici QR
Sicurezza informatica	[24]	Uno studio completo sulle applicazioni dei codici QR dal punto di vista della sicurezza e della privacy
Insegnamento	[25]	Integrazione di codici QR nel materiale didattico e nelle aule per migliorare la qualità dell'istruzione
Tracciabilità	[26]	Uso di blockchain, intelligenza artificiale spiegabile e codici QR per la tracciabilità degli alimenti
Turismo	[27]	Codici QR per esplorare l'Etna (vulcano)

Tabella 1.1: Contesti applicativi principali in cui vengono sfruttati i codici QR

Nello stesso tempo i codici QR sono oggetto di ricerca per migliorare la tecnologia ed espandere i suoi utilizzi. Nella Tabella 1.2 sono citati alcuni riferimenti per i vari campi in cui attualmente è attiva la ricerca sui codici QR.

Ambito	Riferimento	Descrizione
Programmabilità	[**]	Incorporare un programma in un codice QR per renderlo eseguibile
Anticontraffazione	[28]	Inserimento di informazioni nascoste all'interno del codice QR per impedirne la copia/falsificazione
Migliorare la bellezza	[29]	Generazione di codici QR artistici, incorporati in un'immagine
Migliorare altre tecnologie	[30]	Integrazione di codici QR e tecnologie RFID per ridurre la complessità dei circuiti RFID
Migliorare il riconoscimento	[31]	Utilizzo di un metodo adattivo basato sulla binarizzazione per migliorare la qualità del riconoscimento
Aumentare la capacità	[32]	Aumentare la capacità di un codice QR mediante compressione senza perdita di dati
Molteplici informazioni	[33]	Incorporamento di tre livelli di informazioni all'interno di un codice QR
Tipo di stampa	[34]	Codici QR stampati con inchiostro sensibile al pH per il monitoraggio della qualità della freschezza degli alimenti
Sicurezza	[35]	Incorporamento di un segreto all'interno del codice QR utilizzando i colori

Tabella 1.2: Principali tendenze di ricerca sui codici QR (il riferimento [**] rappresenta il paper prodotto durante lo sviluppo di questa tesi)

1.3 Tecnologie utilizzate

Il progetto è stato sviluppato nel linguaggio `Python` usando librerie open-source per analizzare il linguaggio sorgente e convertirlo in una immagine contenente il codice QR e anche per eseguire il processo inverso ed eseguire il codice in una interfaccia web.

Per l'analisi della grammatica è stata usata la libreria `PLY` che si compone di due parti fondamentali: lo scanner e il parser.

1.3.1 Scanner

Lo scanner si occupa di estrapolare gli elementi sintattici dal sorgente scorrendo il file in ingresso e utilizzando delle espressioni regolari identifica i vari elementi della grammatica come parole chiave del linguaggio, identificatori e valori letterali per citarne alcuni.

Le espressioni regolari (RegEx) permettono di descrivere una grammatica regolare, ma non sono sufficientemente potenti per descrivere un linguaggio context-free. Però sono molto potenti e veloci perché possono essere convertite in una macchina a stati, detta Deterministic Finite Automaton, che in un numero deterministico e finito di transizioni porta a riconoscere se una stringa appartiene alla grammatica di una espressione regolare.

Lo scanner prende tutte le espressioni regolari per i vari componenti essenziali del linguaggio e le combina in una unica espressione valutando che non vi siano conflitti. Questa master expression è poi convertita in un DFA per avere la maggior velocità possibile.

Le stringhe che vengono riconosciute vengono passate poi al parser come token, questi compongono la sintassi del linguaggio. Il parser si occuperà di costruire con i vari tipi di token una grammatica context-free e di valutare la correttezza sintattica e semantica.

In particolare lo strumento utilizzato è il lexer offerto da `PLY` (Lex) con questo possiamo definire i vari token come proprietà di una classe o metodi di una classe.

Nel primo caso il valore della proprietà sarà l'espressione regolare che riconosce quel tipo di token.

Nel secondo caso la stringa di documentazione del metodo contiene l'espressione regolare e nel corpo del metodo possono essere eseguite delle azioni e anche decidere se scartare il token, utile per eliminare i commenti nel sorgente, che in ogni caso non sono necessari per la grammatica in quanto non contengono istruzioni. Ad esempio possiamo definire un token per riconoscere i valori numerici e nell'azione corrispondente fare un cast della stringa per ottenere il valore numerico.

1.3.2 Parser

Il parser, anche in questo caso `PLY` (YACC), si occupa di riconoscere la grammatica context-free del linguaggio.

Usando i token prodotti dallo scanner come elementi base del linguaggio e attraverso delle regole si effettuano delle riduzioni che portano a identificare costrutti più complessi composti da più token semplici e riducendo ulteriormente fino ad arrivare a un unico simbolo detto di partenza per la grammatica.

Essenzialmente, vedendo il problema al contrario si parte da un simbolo che rappresenta l'intero sorgente e lo si spezza nei costrutti di alto livello essenziali e si continua ricorsivamente così fino ad arrivare ai componenti sintattici di base, i token.

YACC implementa un parser LR che permette di interpretare grammatiche context-free leggendo l'input da sinistra (Left) verso destra (Right) e producendo una derivazione destra con un processo bottom-up.

Durante la compilazione il parser costruisce in memoria un `AST` (`Abstract Syntax Tree`), ovvero una rappresentazione astratta dell'intero programma. Con questa rappresentazione può effettuare controlli sulla sintassi e semantica del linguaggio, come ad esempio verificare che una variabile sia dichiarata prima di un'assegnazione o che a una variabile dichiarata di un certo tipo venga assegnato un valore di quel tipo.

La rappresentazione in memoria è usata anche per ottimizzare il codice. Questo primo livello di ottimizzazione è indipendente dal target di compilazione (ovvero dal linguaggio di basso livello) e principalmente rimuove codice non raggiungibile e altre operazioni non necessarie allo scopo di ridurre la dimensione ed aumentare la velocità di esecuzione.

Dopo questi passaggi il parser produce il codice di basso livello su un file testuale per comodità. Verrà poi prelevato dal'*assembler* che produrrà il binario eseguibile.

Nella Fig. 1.2 è presente una raffigurazione molto schematica dell'intero processo.

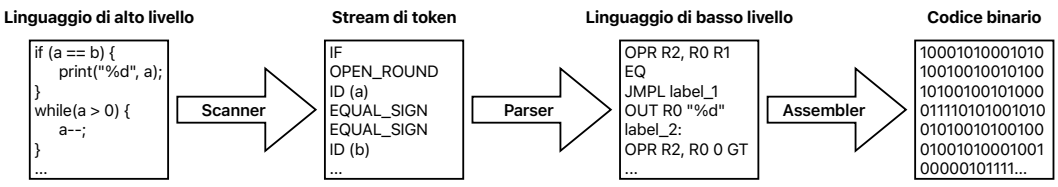

Figura 1.2: Catena di compilazione

1.4 Terminologia

Vengono ora elencati alcuni termini usati successivamente:

- `linguaggio di alto livello`, indica un linguaggio di programmazione più semplice da usare e comprendere da parte di un essere umano
- `linguaggio di basso livello`, indica un linguaggio di programmazione simile al linguaggio macchina e quindi difficile da comprendere da parte di un essere umano
- `eQR code`, è un codice QR eseguibile (**e**QR code), è il prodotto della compilazione
- `eQRbytecode`, è la codifica binaria prodotta dalla compilazione del linguaggio di basso livello
- `QR code`: tecnologia utilizzata per codificare gli eQR code.

Capitolo 2

QRscript

2.1 Introduzione

Il linguaggio in sè è composto da vari dialetti che offrono funzionalità specifiche per un campo di applicazioni ristretto. Questo è fatto principalmente per questioni di spazio, il codice QR non riuscirebbe a contenere il codice sorgente come stringa di testo, quindi viene convertito secondo una grammatica definita in un codice binario.

Per fare questa conversione si passa attraverso una fase intermedia in cui ogni istruzione è rappresentata come *Three-address Code* in un formato testuale che ricorda il codice assembly.

Il Three-Address Code è una rappresentazione intermedia indipendente dall'architettura hardware molto usata dai compilatori per rappresentare e ottimizzare il codice prima di convertirlo in linguaggio macchina. Il Three-Address Code permette di rappresentare qualunque tipo di istruzione in memoria e di ottimizzare le istruzioni eliminando quelle non necessarie riducendo ulteriormente lo spazio occupato.

Il Three-Address Code ha diverse categorie di istruzioni:

- Assegnazione, in cui un operatore binario (o unario) combina due operandi (o uno) e salva il risultato in un terzo operando;

- Salto, operazioni di controllo del flusso, possono essere condizionate o incondizionate.

Il linguaggio di alto livello sviluppato si concentra sulla semplicità di utilizzo e sulla compatibilià con le istruzioni disponibili al livello più basso e in base alle funzionalità che vogliamo offrire in questo dialetto.

Il Three-Address Code viene poi convertito dalla sua rappresentazione in memoria in un formato testuale e salvato su un file. Questo viene fatto per permetterci di ispezionare il codice e debuggarlo e anche per poter disaccoppiare il compilatore del linguaggio di alto livello dal compilatore per il linguaggio di basso livello. Queste due tipologie di linguaggio verranno definite in seguito.

Siccome è necessaria la maggiore compressione possibile le istruzioni disponibili a livello di Three-Address Code cambiano a seconda del dialetto scelto. Questo in alcuni casi porta ad avere una forte similarità tra codice ad alto livello e codice a basso livello. Questo è fondamentale per tradurre nel minor numero di istruzioni possibile le complesse istruzioni del linguaggio di alto livello.

Per esempio, nel Decision-Three Dialect tutte le istruzioni della categoria *assegnazione* sono rimosse in quanto non necessarie per come è strutturato il dialetto. Inoltre è introdotta l'istruzione *IFC* per eseguire un confronto che differisce dal normale salto condizionato in quanto incorpora il confronto che normalmente andrebbe svolto da un'istruzione a parte.

In questo progetto ci siamo appoggiati a `PLY` (Python Lex-Yacc) per costruire il compilatore. È un'implementazione in `Python` di Lex e YACC, strumenti per usati per analizzare la grammatica di un linguaggio originariamente scritti per il linguaggio C.

2.2 Codifica binaria

La codifica binaria del codice assembly deve essere la più compatta possibile, quindi per minimizzare lo spazio utilizzato si adottano strategie di codifica estendibile per i valori letterali contenuti nelle istruzioni.

Alcune codifiche sono spesso riutilizzate in più parti del codice, quindi sono elencate in seguito.

2.2.1 Codifica esponenziale di un numero intero

Dato un intero senza segno X la codifica esponenziale fa in modo che lo spazio occupato dal valore binario cresca esponenzialmente nel numero di bit.

Su una porzione di n bit il valore $2^n - 1$ indica che il valore da memorizzare è troppo grande e che nei successivi n bit sarà presente la differenza $X - (2^n - 1)$, in questo modo la lunghezza totale sarà di $2n$ bit.

Questo processo viene ripetuto raddoppiando ogni volta n ($n_i = 2n_{i-1}$) partendo da $n_0 = 4$ fino a che la disuguaglianza

$$X - \sum_{i=n_0 \ldots n} 2^i - 1 < 2^{n_0 + (\frac{n}{2} - n_0)\mathcal{H}(n - n_0 - 1)} - 1$$

risulta vera.

Dove $\mathcal{H}$ è la funzione gradino di Heaviside:

$$\mathcal{H}(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Si ottengono così numeri codificati su 4 bit, 8 bit, 16 bit, eccetera.

Risolvendo per n si ottiene la lunghezza in bit della codifica esponenziale del numero.

Per un numero intero con segno si usa la rappresentazione segno più modulo, dove il segno è rappresentato da un bit (0 se $X > 0$ e 1 altrimenti) e il modulo è rappresentato usando la codifica esponenziale presentata sopra.

Questo ci permette di avere una codifica estendibile anche per numeri interi con segno.

Alcuni esempi di codifica sono presenti nella Tabella 2.1.

Valore Intero	Segno 1 bit	Base 4 bit	1^a estensione 4 bit	2^a estensione 8 bit	3^a estensione 16 bit
Numeri senza segno					
12		1100			
14		1110			
15		1111	0000		
120		1111	1111	01011010	
300		1111	1111	11111111	0000000000001111
Numeri con segno					
-234	1	1111	1111	11001100	
+17	0	1111	0010		
-503	1	1111	1111	11111111	0000000011011010
+129	0	1111	1111	01100011	
-30	1	1111	1111	00000000	

Tabella 2.1: Esempi di codifica binaria di numeri interi con e senza segno

Per la conversione nel verso opposto si procede in modo iterativo, prelevando dal binario 4 bit, se questi corrispondono al valore massimo (1111) preleviamo altri 4 bit, se anche con questi ultimi la stringa corrisponde al valore massimo (11111111) preleviamo altri 8 bit e così via raddoppiando ogni volta il numero di bit prelevati. Il processo si ferma quando incontra una sequenza di bit che non rappresenta il valore massimo composto da tutti uni.

Questa struttura ci permette di affiancare due numeri senza la necessità di separatori avendo in questo modo la flessibilità di una codifica infinitamente estendibile.

2.2.2 Codifica di un numero a virgola mobile

La codifica di un numero a virgola mobile può essere scelta dal programmatore ponendo dei suffissi al valore letterale.

Tre codifiche sono supportate:

- f16, codifica su 16 bit detta *half-precision* secondo lo standard IEEE 704-2008 [36] (1 bit di segno, 5 bit di esponente e 11 bit di mantissa)
- f32, codifica su 32 bit detta *single-precision* secondo lo standard IEEE 704-2008 [36] (1 bit di segno, 8 bit di esponente e 24 bit di mantissa)
- f64, codifica su 32 bit detta *double-precision* secondo lo standard IEEE 704-2008 [36] (1 bit di segno, 11 bit di esponente e 53 bit di mantissa)

Questo viene fatto per dare maggiore controllo al programmatore e ridurre lo spazio occupato quando una maggiore precisione non è necessaria.

Alcuni esempi di codifica sono presenti nella Tabella 2.2.

Valore	Segno	Esponente	Mantissa
11.234f16	0	10010	0110011110
-34.0009f32	1	10000100	00010000000000011101100
2345.0000001f64	0	10000001010	0010010100100000000000000000000000011010110101111111110

Tabella 2.2: Esempi di codifica binaria di numeri a virgola mobile

2.2.3 Codifica di stringhe di caratteri

La codifica di stringhe di caratteri testuali avviene principalmente in tre modi:

- 7 bit ASCII, secondo lo standard ISO/IEC 646 [37]
- UTF-8, secondo lo standard Unicode o RFC 3629 [38]
- Dizionario

La scelta tra queste tre opzioni avviene in modo automatico per le prime due, mentre per l'opzione *dizionario* il programmatore dovrà specificare quale dizionario utilizzare e quale chiave usare.

Nel primo e secondo caso, quando si fa la traduzione tra linguaggio di alto livello e Three-Address Code, la scelta avviene automaticamente in base a che caratteri sono presenti nella stringa, se sono presenti caratteri non codificabili in 7 *bit ASCII* verrà usata la codifica *UTF-8*. In entrambi i casi la stringa viene terminata dal carattere *ETX* (end of text).

Il tipo dizionario viene usato per risparmiare spazio usando un riferimento alla stringa anzichè la stringa stessa.

Vengono definiti tre tipi di dizionari: *globale*, *specifico* e *locale*.

- *globale*, contiene stringhe definite al di fuori del codice QR e non specifiche per un'applicazione, ma così generali da poter essere usati in molte applicazioni diverse (per esempio "sì", "no", ...)
- *specifico*, contiene stringhe definite al di fuori del codice QR che sono specifiche per una applicazione o ambito applicativo
- *locale*, contiene stringhe definite all'interno del codice QR nella sezione iniziale, può essere utilizzato per esempio per definire stringhe ricorrenti nel programma, ma che non appartengono a uno specifico ambito applicativo

I dizionari *globale* e *specifico* possono essere definiti seguendo uno schema YAML e resi disponibili tramite un server web in modo che l'utente possa accedervi e scaricarli per l'utilizzo in un secondo momento. In un file di dizionario possono essere presenti più lingue dello stesso dizionario, l'applicazione sceglierà la lingua da utilizzare in base alle impostazioni del dispositivo dell'utente finale.

Nel caso in cui la lingua del dispositivo dell'utente non sia definita nel dizionario, la lingua inglese sarà utilizzata. Se anche questa non fosse definita allora la prima lingua definita nel dizionario verrà selezionata dall'applicazione.

Lo schema usato per definire un dizionario è il seguente:

```
"\$schema": http://json-schema.org/draft-07/schema
definitions:
  dictionary:
    type: object
    properties:
      language:
        type: string
        minLength: 2
      words:
        type: array
        items:
          type: string
oneOf:
- "\$ref": "#/definitions/dictionary"
```

```
- type: array
  items:
    "\$ref": "#/definitions/dictionary"
```

Esempio

Un possibile esempio di utilizzo è il seguente. Si supponga un dizionario costituito dalle seguenti parole con due lingue disponibili:

```
- language: it
  words:
  - "Si"
  - "No"
- language: en
  words:
  - "Yes"
  - "No"
```

Si noti che l'ordine in cui sono presenti le parole è importante, deve esserci una corrispondenza tra una parola e la sua traduzione in un'altra lingua.

Una parola sarà referenziata nel codice binario utilizzando il suo indice a partire dall'inizio dell'elenco. La codifica dell'indice in binario differisce da dialetto a dialetto, ma una scelta comune è utilizzare la codifica esponenziale per numeri interi senza segno (vedi 2.2.1).

2.3 Codifica nel codice QR

Per codificare il binario generato dal codice a basso livello preponiamo un *header* che contiene informazioni comuni a tutti i dialetti e che si occupa di cinque aspetti fondamentali: *Padding*, *Continuazione*, *Sicurezza*, *URL* e *Dialetto*.

Per inserire il binario generato all'interno di un codice QR (vedi 1.2) sono state fatte alcune scelte di design in base ai casi d'uso previsti.

Per prima cosa è chiaro che è meglio avere il massimo *livello di correzione* possibile anche al costo di dover usare una *versione* maggiore. Questa scelta è stata fatta perché nei casi d'uso di questa tecnologia graffi e piccoli danni al codice QR stampato sono plausibili e probabili, quindi è meglio avere una maggiore tolleranza ai danni al posto di una dimensione fisica minore.

In secondo luogo, dato che il eQRbytecode è prodotto usando singoli bit e non considerando l'allineamento al byte un modo per aggiungere padding in modo deterministico per ottenere l'allineamento è necessario, il binario in questo modo allineato può essere convertito in una sequenza di byte e inserito all'interno del codice QR usanto la *modalità di input binaria* (vedi 1.2).

2.3.1 Header

Le sezioni di questo header sono comuni a tutti i dialetti, devono quindi occuparsi di quelle funzionalità di cui ogni dialetto ha bisogno.

Le cinque sezioni non hanno una lunghezza fissa nel numero di bit, ma ognuna ha una sintassi che permette di determinarne la fine.

L'ordine delle sezioni è presentato in Fig. 2.1.

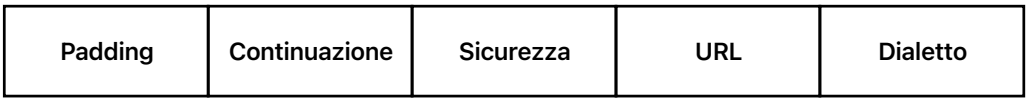

Figura 2.1: Visione generale dell'header

Padding

Figura 2.2: Sezione di padding nell'header

Il padding necessario per l'allineamento al byte è richiesto indipendentemente da quale dialetto si è scelto di usare. Perciò per una questione di semplicità e di coerenza si è deciso di posizionarlo alla testa dell'header di QRscript.

Il padding è composto da tanti bit a 0 (da un minimo di 0 a un massimo di 7) terminati da un bit a 1. Quest'ultimo è necessario per identificare la terminazione del padding in modo che possa essere rimosso prima di procedere con ulteriori trasformazioni.

Il numero di zeri dipende da quanti bit rimangono liberi nell'utlimo byte incompleto meno un bit (il bit a 1 utilizzato per terminare la sezione di padding). La seguente formula permette di calcolare il numero di zeri necessari.

$$n_{zeri} = 8 - (n_{bit\ codice} + n_{bit\ header} + 1) \mod 8$$

Continuazione

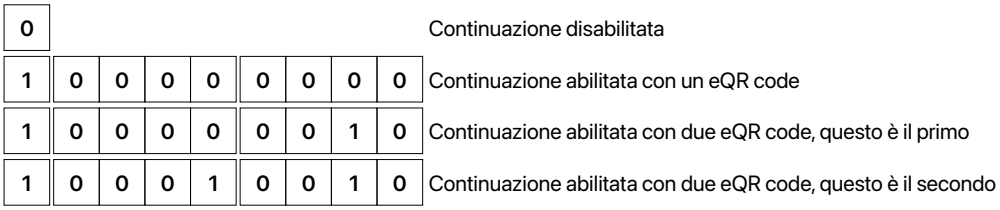

Figura 2.3: Sezione di continuazione nell'header

Per sopperire alla limitazione di spazio è prevista la possibilità di spezzare un programma su più codici QR (detti segmenti) che andranno poi scansionati e concatenati per permetterne l'esecuzione.

La *continuazione* è abilitata da un bit impostato a `1`. Questo bit è poi seguito dal *numero di sequenza* e dal *numero totale di segmenti*. Entrambi questi campi sono numeri interi senza segno codificati con la codifica esponenziale (vedi 2.2.1).

Al momento della lettura il *numero di sequenza* verrà utilizzato per riordinare i segmenti e concatenarli nell'ordine corretto, il *numero totale di segmenti* serve per capire quando tutti i segmenti sono stati scansionati.

Sicurezza

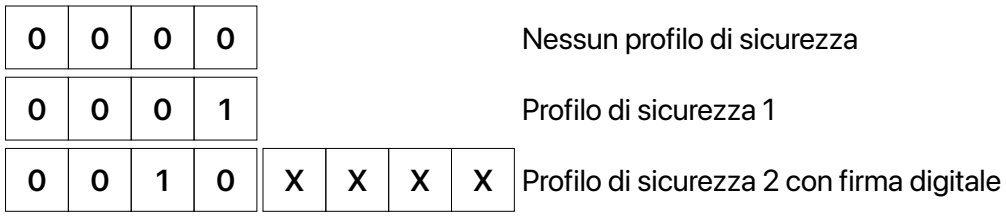

Figura 2.4: Sezione di sicurezza nell'header

La sezione di *sicurezza* gestisce *autenticazione*, *integrità* e possibilmente anche la *riservatezza* del codice QR. La sezione è composta da un numero intero senza segno codificato con la codifica esponenziale (vedi 2.2.1) che specifica il *profilo di sicurezza* scelto. Il valore 0 corrisponde all'assenza di sicurezza.

Un *profilo di sicurezza* indica:

- quali misure di sicurezza sono messe in atto
- quali algoritmi sono utilizzati per gestire la sicurezza
- la firma digitale (opzionale) preceduta dalla sua lunghezza in bits per garantire *autenticazione* e *integrità*

Non è definita dallo standard l'associazione tra il numero del *profilo di sicurezza* e le effettive misure associate al suddetto *profilo di sicurezza* (a parte il profilo 0), ciò significa che diverse applicazioni possono avere diverse associazioni.

Le associazioni devono essere definite dal programmatore e configurate nel compilatore e nell'interprete per permettere una corretta esecuzione.

URL

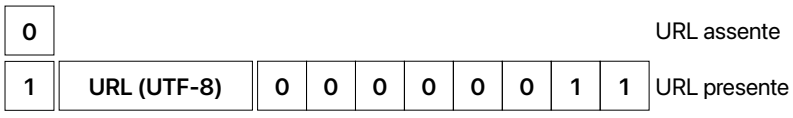

Figura 2.5: Sezione di URL nell'header

Nel caso in cui il dispositivo dell'utente abbia una connessione a Internet, o una connessione locale, è possibile collegarsi all'applicazione in esecuzione su un server remoto tramite un *URL*. Questo è utile i quei casi in cui si vuole migliorare l'interazione con l'utente ad esempio includendo immagini o video che sarebbero difficili da incorporare nel eQR code per via di limitazioni di spazio. L'URL presente nell'header serve proprio a questo.

Un bit impostato a 0 indica l'assenza di un URL, mentre invece se è importato a 1 indica che a seguire è presente una stringa di caratteri UTF-8 [38] terminati dal carattere *EXT* (end of text).

L'applicazione in esecuzione sul dispositivo dell'utente può programmare una particolare risposta dopo aver richiesto l'URL per continuare l'esecuzione del eQR code. Questo permette di disattivare l'URL anche dopo la creazione del eQR code.

Si noti che questa sezione non è obbligatoriamente implementata, ciò significa che un'implementazione aderente allo standard potrebbe non implementarla.

Dialetto

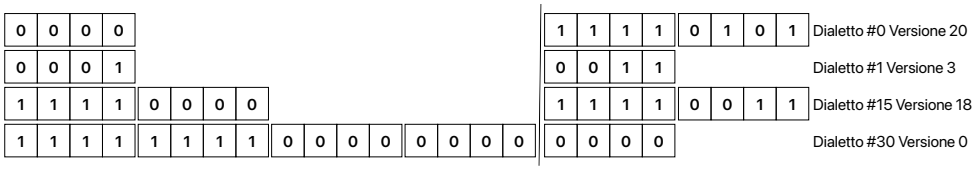

Figura 2.6: Sezione di dialetto nell'header

Il dialetto è identificato da un numero intero senza segno codificato con la codifica esponenziale (vedi 2.2.1). I dialetti, come menzionato in precedenza, sono linguaggi pensati per scopi diversi, con un set di istruzioni specifico per il campo di applicazione e codifica binaria diversa e con diversa dimensione.

Il numero di versione è anch'esso identificato da un numero intero senza segno codificato con la codifica esponenziale (vedi 2.2.1). Il numero di versione serve per permettere di aggiornare un dialetto senza dover per forza creare un nuovo dialetto per ogni nuova versione.

Il lavoro pubblicato nell'articolo "*QRscript: Embedding a Programming Language in codice QRs to support Decision and Management*" [39] è, a seguito di

un'approfondita analisi della letteratura scientifica, il primo a presentare un linguaggio di programmazione inserito all'interno di un codice QR. Ciò non esclude che precedenti implementazioni possano essere incluse come uno specifico dialetto.

Al momento della stesura di questo documento i dialetti disponibili sono:

- Decision Tree Dialect, codice 0
- General Purpose Dialect, codice 1

2.3.2 Istruzioni

Le istruzioni codificate dipendono dal dialetto scelto, ma in generale si può dire che a ogni istruzione verrà assegnato un identificativo numerico (che verrà codificato in binario) e a seguire l'elenco degli argomenti di questa istruzione.

Si può scegliere tra definire tante istruzioni semplici che compiono piccole operazioni o scegliere poche istruzioni complesse che svolgono compiti più ampi. Questo è un compromesso da fare in base alle trasformazioni che un dialetto può fare sui dati sempre per ridurre la dimensione del binario.

Alcuni dialetti implementano a loro volta un header all'interno della sezione per le istruzioni che modifica il funzionamento dell'interprete durante l'esecuzione e/o l'interpreteazione della rappresentazione binaria. La gestione di questo ulteriore header è demandata all'implementazione del dialetto.

Capitolo 3

QRprog

3.1 Introduzione

Il dialetto *QRprog* è pensato come un linguaggio di programmazione generico che può rappresentare qualunque tipo di linguaggio.

Una delle funzionalità più importanti è la possibilità di poter richiamare funzioni presenti in altri codici QR (dello stesso progetto) che è diverso dalla *continuazione* presente a priori in *QRscript*. Questo dà la possibilità al programmatore di spostare specifiche parti del codice che vengono usate solamente in certe occasioni in codici QR separati senza poi doverli scansionare tutti per eseguire il programma come nel caso della *continuazione*.

In questo dialetto mi sono concentrato principalmente sul linguaggio di basso livello, mentre il mio collega Mattia Scamuzzi si è occupato di progettare il linguaggio di alto livello. Segue comunque una breve descrizione del lavoro svolto dal mio collega.

3.2 Linguaggio di alto livello

Il linguaggio di alto livello sviluppato per *QRprog* è un linguaggio fortemente tipizzato. I tipi di base sono: **int**, **hfloat** (half float), **float** e **double**.

Il linguaggio ricorda molto il **C** in quanto utilizza le parentesi graffe per delimitare i blocchi di istruzioni e il punto e virgola per separare le istruzioni l'una dall'altra.

Il programma può essere diviso in *moduli*. Ogni modulo è un file separato contenente *funzioni* che possono essere importate in base alle necessità in altri moduli. Ogni modulo produrrà uno (o più) codici QR che sarà necessario scansionare nel momento in cui una funzione di tale modulo è richiamata dal programma in esecuzione. Questo ci permette di dividere il programma in porzioni da caricare solo se utilizzate e quindi riducendo la dimensione dei singoli codici QR.

Un programma è diviso in due sezioni principali: la sezione degli `import` e la lista di `funzioni`. Nella sezione degli import vengono definiti i simboli importati da altri moduli. Si possono anche rinominare i simboli importati per evitare conflitti. Nella lista di funzioni invece sono definiti i vari metodi. Una funzione particolare è quella chiamata `main` perché sarà il punto di ingresso per l'esecuzione del programma.

3.2.1 Struttura dei moduli e `import`

I moduli definiti dall'utente non devono necessariamente essere strutturati in alcun modo. È possibile avere cartelle separate per i vari moduli o averli tutti sullo stesso livello, non ha importanza in quanto quando saranno importati bisognerà specificare il path relativo al modulo.

La differenza principale tra un *file eseguibile* e un modulo è che quest'ultimo non ha un punto di ingresso per l'esecuzione, ovvero non ha una funzione `main`.

Senza il punto di ingresso quando il codice QR generato da un modulo verrà scansionato ed eseguito, l'esecuzione terminerà immediatamente senza produrre alcun risultato.

Per importare una funzione da un modulo esterno bisogna utilizzare il costrutto `import-from`. Non è possibile importare tutte le funzioni di un modulo automaticamente, ma solamente specificare espressamente quelle che si vogliono nella prima sezione del costrutto. È anche possibile rinominare le funzioni importate usando la parola chiave `as`.

Seguono degli esempi:

```
import { func1, func2 } from "./module1";
import { func1 as f1, func3 } from "./module2/submodule1";
```

Possiamo notare come nel secondo esempio sia possibile importare il simbolo `func1`, nonostante sia già definito dall'import precedente, rinominandolo come `f1`.

È da tenere presente che al momento della compilazione tutti i moduli utilizzati devono essere passati al compilatore in modo che i riferimenti possano essere

verificati e i rispettivi codici QR vengano generati. Notasi che l'ordine con cui si passano i vari moduli al compilatore è importante: in base a questo verrà assegnato un identificativo numerico a ogni codice QR generato ed è questo identificativo che permette di fare riferimento al codice QR da un altro.

3.2.2 Funzioni

Il cuore del linguaggio sono le funzioni. Ogni funzione è introdotta dalla parola chiave `fn` seguita dall'identificativo della funzione e tra parentesi i vari argomenti per la funzione.

Ogni funzione ha un corpo composto da istruzioni e può produrre oppure no un risultato.

Una funzione che ha un trattamento particolare è la funzione `main` che non accetta argomenti e non ritorna nulla. Questa funzione è il punto di ingresso per l'esecuzione, ovvero il punto in cui l'esecuzione del programma incomincia.

Le funzioni possono essere richiamate indicando il nome della funzione e a seguire tra parentesi gli argomenti. Una funzione può avere zero o più valori di ritorno che verranno trasferiti al chiamate una volta che l'esecuzione della funzione è terminata. Il chiamante potrà utilizzare i valori ritornati assegnandoli a delle variabili locali usando l'operatore di assegnazione (=) preceduto dalle variabili a cui assegnare i valori.

Segue un esempio.

```
fn fun1(int a, int b) {
    return a * b, a / b;
}

fn main() {
    int mul, div;

    mul, div = fun1(2, 3);
}
```

3.2.3 Controllo del flusso

All'interno di un blocco di istruzioni si può controllare il flusso di esecuzione attraverso i comuni costrutti di controllo del flusso: **if-then-else**, **for-loop** e **while-loop**.

Il costrutto **if-then-else** permette di testare una condizione e decidere in base a quella quale ramo dell'esecuzione intraprendere. È introdotto dalla parola chiave **if** e può usare la parola chiave **else** per indicare un blocco di codice da eseguire se la condizione risulta falsa. È possibile avere più di due rami con più condizioni da testare ponendo la parola chiave **if** subito dopo la parola chiave **else**. Al più un blocco *else* è permesso per il costrutto.

Segue uno schema.

```
if (cond1) {
    # Blocco 1
} else if (cond2) {
    # Blocco 2
} else if (cond3) {
    # Blocco 3
} else {
    # Blocco else
}
```

Il costrutto **for-loop** permette di creare un ciclo strutturato. È introdotto dalla parola chiave **for** seguita da parentesi tonde che contengono tre parti: una parte di *inizializzazione* eseguita una sola volta all'inizio del ciclo, una parte di *controllo* eseguita all'inizio di ogni iterazione che verifica se continuare a ciclare o uscire dal ciclo ed infine una parte di *incremento* eseguita al termine di ogni iterazione. Queste tre parti sono separate da un punto e virgola.

Segue il blocco di istruzioni del ciclo, detto *corpo*, che verrà eseguito ad ogni iterazione. Le iterazioni continueranno fino a che la parte di *controllo* produce un valore vero.

Segue uno schema.

```
for (<inizializzazione>;<controllo>;<incremento>) {
    # Corpo del ciclo
}
```

Il costrutto **while-loop** permette di creare un ciclo non strutturato. È introdotto dalla parola chiave **while** seguita da parentesi tonde, contenenti una condizione che verrà testata all'inizio di ogni iterazione, e un blocco di istruzioni, detto *corpo*, che verrà eseguito ad ogni iterazione. Le iterazioni continueranno fino a che la condizione risulta vera.

Segue uno schema.

```
while (cond) {
    # Corpo del ciclo
}
```

3.2.4 Variabili ed espressioni

È possibile definire variabili attribuendogli un tipo e assegnandogli un valore, le variabili possono anche essere array multidimensionali dove le dimensioni sono parte integrante del tipo e vanno specificate in fase di dichiarazione attraverso numeri interi tra parentesi quadre.

Sono disponibili i comuni operatori aritmetici (‘+’, ‘-’, ‘*’, ‘/’, ‘^’, ‘%’) con cui è possibile definire operazioni aritmetiche tra variabili e valori letterali. Sono disponibili operatori di confronto (‘>’, ‘>=’, ‘<’, ‘<=’, ‘==’, ‘! =’) con cui è possibile confrontare espressioni aritmetiche tra di loro e produrre risultati booleani che possono essere ulteriormente combinati utilizzando i comuni operatori booleani *and* (&), *or* (|) e *not* (!).

Data l'assenza di un tipo booleano, le espressioni booleane producono un valore intero. Il valore 0 indica che l'espressione è falsa, qualunque altro valore diverso da 0 (tipicamente 1) indica che l'espressione è vera.

3.2.5 Input e output

Si può interagire con l'utente richidendo un input da memorizzare in una variabile utilizzando la parola chiave `input` seguita dall'identificativo di una variabile. Il tipo del valore richiesto all'utente è dettato dal tipo della variabile ricevente.

Per produrre un output per l'utente sono disponibili due opzioni. La prima consiste nell'istruzione `print` che permette di produrre l'output di un singolo elemento, sia variabile sia valore letterale. Il formato dell'output è quello di default per il tipo

di dato e non è modificabile. La seconda opzione consiste nell'utilizzo dell'istruzione `printf` che permette di produrre l'output di multipli elementi specificandone il formato attraverso una stringa di formato stile C.

La stringa di formato contiene dei segnaposti identificati dal carattere % seguito da una serie di opzioni per la formattazione ed infine il tipo del dato.

```
%[flags][width][.precision][length]type
```

- `flags`, specifica opzioni oper l'allineamento (a destra o sinistra) e il riempimento degli spazi non utilizzati
- `width`, specifica il numero minimo di caratteri da produrre in output
- `precision`, specifica il numero massimo di caratteri da produrre in output e per numeri a vigola mobile specifica quante cifre decimali mostrare
- `length`, specifica la lunghezza del dato da formattare
- `type`, specifica il tipo di dato da formattare, è l'unica opzione obbligatoria

In Tabella 3.1 sono presenti alcuni esempi di segnaposti.

Segnaposto	Flags	Width	Precision	Length	Type	Descrizione
%%					%	Produce il carattere '%'
%3.2f		3	2		f (float)	Produce un numero a virgola mobile allineato a destra con un minimo di tre caratteri e due cifre dopo la virgola
%04d	0	4			d (decimal)	Produce un numero intero decimale allineato a destra con un minimo di quattro caratteri riempiti da zeri
%.4s			4		s (string)	Produce una stringa composta dai primi quattro caratteri della stringa passata come argomento
%-2X	-	2			X (exadecimal)	Produce una stringa allineata a sinistra con un minimo di due caratteri riempiti da spazi, il valore numerico viene convertito in esadecimale con lettere maiuscole

Tabella 3.1: Esempi di stringa di segnaposti

3.3 Linguaggio di basso livello

Volendo fornire le potenzialità di un linguaggio di programmazione generico le istruzioni che dobbiamo mettere a disposizione devono poter svolgere un'ampia gamma di operazioni. Per fare ciò possiamo ispirarci ai comuni set di istruzioni presenti nei processori reali come ad esempio *ARM* e *Intel x86*.

Però questi set di istruzioni sono molto complessi, con migliaia di istruzioni e semplicemente produrrebbero binari troppo grandi per le capacità di un codice QR, ma sono un buon punto di partenza.

La prima considerazione che andremo a fare è sulla *memoria*, nei processori fisici il numero di registri disponibili è limitato per ragioni di costo e di area di silicio disponibile, ma nel nostro caso queste limitazioni non si applicano in quanto nella macchina virtuale che andrà a eseguire questo codice potremmo creare un numero elevato di registri senza alcun problema. Questo ci permette di eliminare del tutto

le istruzioni di accesso alla memoria (*Load e Store*). Questi registri possono essere di una dimensione qualsiasi e contenere qualunque tipo di informazione (intero, virgola mobile, stringa, ...) questo semplifica ulteriormente il set di istruzioni necessarie.

Un altro aspetto da considerare sono le *operazioni sui registri*, con questo intendiamo gli operatori aritmetici, di confronto e booleani. Nei processori fisici ognuna di queste è un'istruzione a se stante che può essere combinata con le altre. Questo aumenta la dimensione necessaria per identificare le singole istruzioni, perciò abbiamo deciso di condensare tutte le operazioni tra registri in un'unica istruzione che potrà eseguire una serie di operazioni che coinvolgono registri e valori letterali. Per fare ciò abbiamo pero dovuto distinguere tra registri che contengono valori semplici e registri che contengono valori complessi come i vettori; per poter accede agli elementi del vettore è necessario un indice, perciò una istruzione dedicata a operare con i vettori viene definita parallelamente all'istruzione per operare con i registri semplici.

L'ultima criticità da gestire è il richiamo a funzioni con il passaggio di parametri e valori di ritorno. Su un processore fisico si segue uno standard preciso detto ABI (Application Binary Interface) che descrive l'ordine per il passaggio dei parametri, se sono passati tramite registri o stack o entrambi, quali registri la funzione chiamata deve preservare per il chiamante e come il valore di ritorno è restituito al chiamante.

Nel nostro caso abbiamo scelto di passare tutti i parametri e i valori di ritorno (sono possibili valori di ritorno multipli) tramite lo stack, l'ordine per passare i parametri è lo stesso del codice ad alto livello (quindi dovranno essere estratti al contrario) e la funzione chiamante non deve preoccuparsi di preservare alcun registro. Queste scelte sono state fatte per permettere alle funzioni di usare sempre i registri bassi (identificati da numeri più piccoli) e in questo modo ridurre il numero delle istruzioni necessarie per una chiamata a funzione occupando meno spazio nella codifica binaria.

3.3.1 Valori letterali

Si è scelto di riservare il registro numero 0 per segnalare che a seguire è presente un *valore letterale*. Dopo la codifica binaria del registro 0 è presente l'identificativo del tipo codificato su 3 bit.

I tipi disponibili sono:

- **INT**, formattato seguendo la codifica esponenziale (vedi 2.2.1)
- **FLOAT_16**, **FLOAT_32**, **FLOAT_64** per valori a virgola mobile
- **STRING_7_BIT**, **STRING_UTF_8**, **STRING_DICT** per stringhe

La codifica di ogni valore letterale è composta da $R0$, tipo e valore, come mostrato in figura.

Figura 3.1: Formato dei valori letterali

Ogni tipo è codificato su 3 bit secondo la Tabella 3.2.

Tipo	Codifica
INT	000
FLOAT_16	001
FLOAT_32	010
FLOAT_64	011
STRING_7_BIT	100
STRING_UTF_8	101
STRING_DICT	110

Tabella 3.2: Codifica binaria dei tipi

La lunghezza del *valore* dipende dal tipo. I valori a virgola mobile hanno lunghezze predefinite (16, 32 e 64), i valori interi hanno una codifica estendibile, ma per le stringhe bisogna usare un *carattere di terminazione* o preporre alla stringa la

lunghezza della stessa. Abbiamo scelto la prima opzione usando il carattere **ETX** (End-of-Text) per indicare la fine di una stringa in quanto permette di ottenere rappresentazioni più compatte.

3.3.2 Label

Una label è un identificativo univoco nel codice che può essere utilizzato come destinazione per un salto nelle istruzioni di controllo del flusso.

La definizione di una label in un file che contiene la rappresentazione intermedia avviene nel seguente modo:

```
    istruzioni...
label_0: istruzione
    istruzioni...
```

Potrà essere utilizzata referenziando il nome univoco `label_0` all'interno dello stesso file. Una label può anche essere utilizzata in un altro file dello stesso progetto come label esterna. Per fare ciò basterà preporre il nome del file esterno separato da un due punti (`file1:label_0`).

Le label vengono codificate, in base a dove sono referenziate, come il numero di istruzioni da saltare in avanti o indietro dalla posizione corrente. Per label esterne la codifica consiste in due numeri interi senza segno, il primo identifica il file a cui si fa riferimento (cioè il numero assegnato al eQR code per identificarlo) e il secondo è il numero di istruzioni da saltare a partire dall'inizio del file. Queste due codifiche vengono distinte da un bit iniziale impostato a 0 per indicare una label interna al file o impostato a 1 per indicare una label esterna.

Un'altra rappresentazione usata principalmente durante la decodifica di label esterne è di usare direttamente l'identificativo del file e il numero di istruzioni da saltare dall'inizio del file decodificati come label (vedi Fig. 3.2). É possibile utilizzare questo formato nel proprio codice, ma è fortemente sconsigliato in quanto difficile da mantenere.

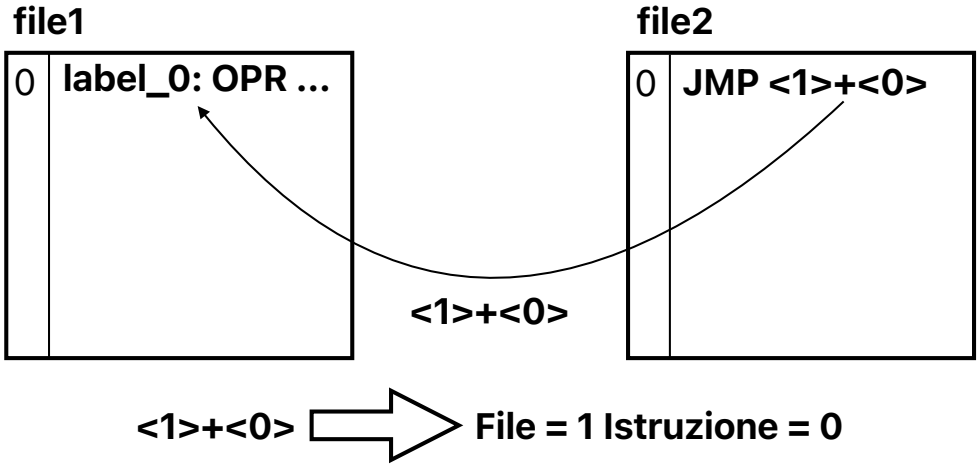

Figura 3.2: Esempio label con identificativo e numero di istruzioni

3.3.3 QRprog header

Il linguaggio QRprog prevede un header molto semplice. Contiene un numero intero senza segno codificato con la codifica esponenziale presentata nella Sezione 2.2.1.

Questo numero identifica il eQR code all'interno del gruppo e permette di effettuare salti a funzioni contenute in questo eQR code da altri eQR code.

3.3.4 Istruzioni di basso livello

Le istruzioni di basso livello disponibili sono 12 e vengono codificate su 4 bit ciascuna (più i parametri).

Istruzione	Codifica	Istruzione	Codifica
OPR	0000	RET	0110
OPV	0001	IN	0111
JMP	0010	OUT	1000
JMPL	0011	EXIT	1001
JMPF	0100	PUSH	1010
JMPR	0101	POP	1011

Tabella 3.3: Codifica binaria delle istruzioni

Di seguito verrà descritto il funzionamento di ogni singola istruzione.

OPR (0000)

Questa istruzione esegue operazioni tra registri semplici, ovvero che non contengono vettori. Questa scelta di separare operazioni tra registri semplici e registri con vettori è fatta allo scopo di ridurre lo spazio occupato dalle operazioni tra registri semplici. Può eseguire più di una singola operazione usando la notazione polacca inversa.

Con la notazione polacca inversa si può rappresentare qualunque serie di operazioni senza dover usare parentesi e senza avere problemi con la precedenza degli operatori.

Per esempio:

$$5 \ + \ (10 \ * \ 2) \rightarrow 5 \ 10 \ 2 \ * \ +$$

Gli operatori disponibili sono presentati nella Tabella 3.4 con la loro codifica su 4 bit.

Operazione	Codifica	Operazione	Codifica
PLUS	0000	LE	1000
MINUS	0001	GT	1001
STAR	0010	GE	1010
DIV	0011	AND	1011
POWER	0100	OR	1100
EQ	0101	NOT	1101
NEQ	0110	MOD	1110
LT	0111		

Tabella 3.4: Codifica binaria delle operazioni

Il formato testuale di questa istruzione può essere espresso come `OPR <registro>, <operazione>`. Dove il `registro` specificato come primo parametro è la destinazione dell'`operazione`.

Un'`operazione` può essere una `costante` o un `registro` oppure un'espressione in notazione polacca inversa `<operazione> <operazione> <operando>`.

Per permetterci di distinguere tra operando e operatore nella catena di operazioni viene utilizzato un bit impostato a 0 per segnalare un operando e impostato a 1 per segnalare un operatore. La catena di operazioni è terminata usando il registro riservato numero 1.

Un esempio di codifica può essere visionato nella Fig. 3.3.

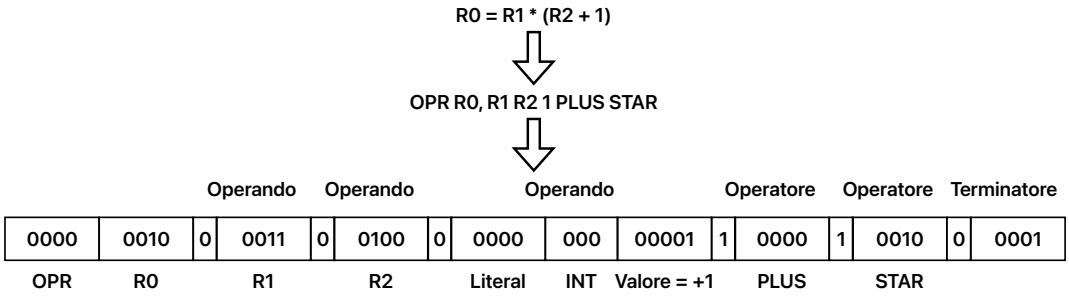

Figura 3.3: Esempio codifica OPR

OPV (0001)

Questa istruzione esegue operazioni tra registri contenenti vettori. Il formato è simile alla istruzione OPR presentata precedentemente, ma ogni operando ha un indice per accedere al vettore contenuto nel registro.

Gli operatori disponibili sono stati già presentati in Tabella 3.4.

Il formato testuale di questa istruzione può essere espresso come `OPV <rigistro_vettoriale>, <operazione>`. Dove il `registro_vettoriale` specificato come primo parametro è la destinazione dell'`operazione`.

Un `registro_vettoriale` è un `registro` con in indice indicato tra parentesi (`<registro>[<indice>]`) dove l'indice può essere un altro `registro` o una `costante`.

Un'`operazione` può essere una `costante` o un `registro` o un `registro_vettoriale` oppure un'espressione in notazione polacca inversa `<operazione> <operazione> <operando>`.

Per permetterci di distinguere tra operando e operatore nella catena di operazioni viene utilizzato un bit impostato a 0 per segnalare un operando e impostato a 1 per segnalare un operatore. La catena di operazioni è terminata usando il registro riservato numero 1. Un registro semplice può essere usato all'interno della catena di operazioni e verrà trattato come un registro contenente un vettore di lunghezza 1, il registro riservato numero 1 è usato per indicare questo indice. Un registro contenente un vettore può usare come indice un altro registro oppure un valore letterale. Non vi sono restrizioni sul tipo dell'indice, questo offre la possibilità di

avere *vettori associativi* aventi come chiave una stringa di caratteri, numeri interi, numeri a virgola mobile e anche una combinazione di essi.

Un esempio di codifica può essere visionato nella Fig. 3.4.

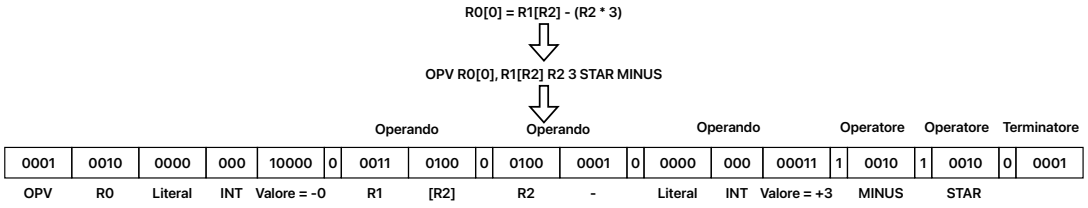

Figura 3.4: Esempio codifica OPV

JMP (0010)

Questa istruzione esegue un salto incondizionato a una *label* passata come parametro.

Nella rappresentazione testuale corrisponde a `JMP <label>`. Da notare che in fase di compilazione la validità dei riferimenti è verificata e se la label referenziata non esista la compilazione fallirà.

Nella rappresentazione binaria la label verrà sostituita con il numero relativo di istruzioni da saltare. Questo numero può essere anche negativo per permettere salti all'indietro, utili per creare cicli, ed è codificato secondo la codifica esponenziale presentata nella Sezione 2.2.1.

JMPL (0011)

Questa istruzione esegue un salto condizionato al risultato dell'ultima operazione verso una *label* passata come parametro. Se il valore dell'ultima espressione valutata è vero (numero diverso da 0 o stringa non vuota) allora il salto verrà effettuato.

Nella rappresentazione testuale corrisponde a `JMPL <label>`. Da notare che in fase di compilazione la validità dei riferimenti è verificata e se la label referenziata non esista la compilazione fallirà.

Nella rappresentazione binaria la label verrà sostituita con il numero relativo di istruzioni da saltare. Questo numero può essere anche negativo per permettere salti all'indietro, utili per creare cicli, ed è codificato secondo la codifica esponenziale presentata nella Sezione 2.2.1.

JMPF (0100)

Questa istruzione esegue un salto incondizionato a una funzione identificata da una *label* passata come parametro. A differenza di un normale salto incondizionato questa istruzione salva in una struttura di tipo stack il punto di ritorno per quando la funzione termina la propria esecuzione.

Nella rappresentazione testuale corrisponde a `JMPF <label>`. Da notare che in fase di compilazione la validità dei riferimenti è verificata e se la label referenziata non esista la compilazione fallirà.

Nel caso di funzioni esterne sarà necessario specificare in quale file questa funzione è presente e sarà anche necessario passare al compilatore questo file per poter verificare la correttezza dei riferimenti.

Nella rappresentazione binaria la label verrà sostituita con il numero relativo di istruzioni da saltare. Questo numero può essere anche negativo per permettere salti all'indietro, utili per creare cicli, ed è codificato secondo la codifica esponenziale presentata nella Sezione 2.2.1.

Nel caso di funzioni esterne la label verrà rappresentata usando due numeri interi senza segno codificati secondo la codifica esponenziale presentata nella Sezione 2.2.1. Il primo numero identifica il eQR code in cui è presente la funzione richiesta, il secondo rappresenta il numero di istruzioni da saltare partendo dall'inizio del codice presente nel eQR code.

JMPR (0101)

Questa istruzione esegue un salto condizionato al registro passato come primo parametro verso una *label* passata come secondo parametro. Se il valore del registro è vero (numero diverso da 0 o stringa non vuota) allora il salto verrà effettuato.

Nella rappresentazione testuale corrisponde a `JMPR <registro>, <label>`. Da notare che in fase di compilazione la validità dei riferimenti è verificata e se la label referenziata non esista la compilazione fallirà.

Nella rappresentazione binaria la label verrà sostituita con il numero relativo di istruzioni da saltare. Questo numero può essere anche negativo per permettere salti all'indietro, utili per creare cicli, ed è codificato secondo la codifica esponenziale presentata nella Sezione 2.2.1.

RET (0110)

Questa istruzione termina l'esecuzione di una funzione facendo in modo che l'esecuzione riprenda dall'istruzione successiva alla **JMPF** che ha lanciato la corrente. Non accetta argomenti.

Nella rappresentazione testuale corrisponde a `RET`.

IN (0111)

Questa istruzione richiede l'input dell'utente. Il valore immesso viene salvato all'interno del registro passato come parametro. Il tipo di dato che si richiede all'utente è definito dal tipo di dato contenuto nel registro al momento dell'esecuzione.

Nella rappresentazione testuale corrisponde a `IN <destinazione>`. Dove per `destinazione` si intende un registro semplice o un registro vettoriale con indice.

Nella codifica binaria il registro di destinazione, che può essere un registro semplice o vettoriale, è codificato usando un bit a 0 per indicare come destinazione un registro semplice o un bit impostato a 1 per indicare come destinazione un registro vettoriale con indice.

OUT (1000)

Questa istruzione produce un output su schermo stampando gli argomenti passati come parametri. Accetta una lista di registri, registri vettoriali e valori letterali che possono essere opzionalmente formattati passando un ultimo argomento che non è altro che una stringa contenente dei segnaposto in cui posizionare i valori.

Nella rappresentazione testuale conrrisponde a `OUT <arg1>[, <arg2>[, ...]] [<formato>]`. Dove ogni argomento può essere un registro semplice, un registro vettoriale con indice o un valore letterale separati da una virgola. L'ultimo argomento (`<formato>`), opzionale, è separato solamente da uno spazio ed è una stringa letterale.

Nella codifica binaria ognuno degli argomenti viene differenziato in base al tipo in due categorie: valori letterali o registri e registri vetoriali. Questo è fatto perché un valore letterale può essere interpretato come il registro riservato 0 a cui segue il valore letterale, ma i registri vettoriali hanno bisogno di un indice oltre al numero di registro. Le due codifiche si sovrapporrebbero quindi un bit è usato per discriminare quale delle due categorie di argomento segue: 0 significa valore letterale o registro semplice, 1 significa registro vettoriale.

Nella codifica binaria la lista di argomenti è terminata usando il registro riservato numero 1 interpretato come registro semplice (quindi preceduto da un bit a 0), ma se invece vuole essere interpretato come un registro vettoriale (quindi preceduto da un bit a 1) indica che a seguire è presente la stringa di formattazione.

La stringa di formattazione è codificata, in base a che caratteri contiene, in **STRING_7_BIT** o **STRING_UTF_8**. Per differenziare la codifica un bit è utilizzato (0 implica **STRING_7_BIT**, mentre 1 implica **STRING_UTF_8**).

In Fig. 3.5 e Fig. 3.6 si può notare la differenza di formato dell'istruzione nel caso sia presente o no la stringa di formato.

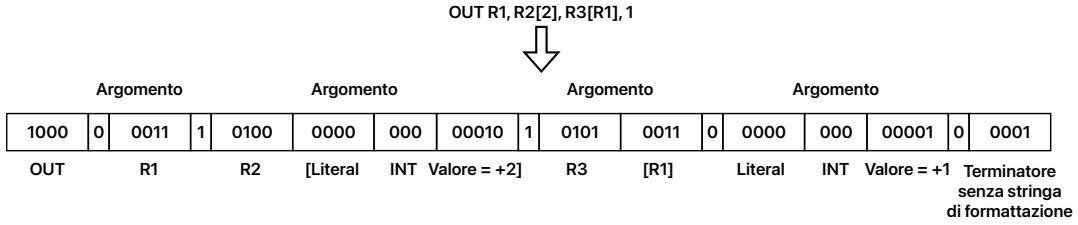

Figura 3.5: Esempio codifica OUT senza formattazione

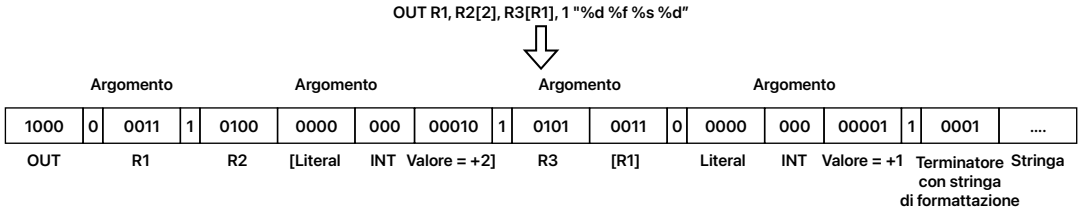

Figura 3.6: Esempio codifica OUT con formattazione

Le stringhe di formattazione usano dei segnaposti all'interno della stringa. Ogni segnaposto è preceduto dal simbolo % e successivamente da varie opzioni per configurare la formattazione testuale (opzionali) ed infine il tipo di dato che si intende stampare a video.

EXIT (1001)

Questa istruzione termina l'esecuzione dell'intero programma da qualunque posizione nel codice venga chiamata. Non accetta argomenti.

Nella rappresentazione testuale corrisponde a `EXIT`

PUSH (1010)

Questa istruzione permette di inserire valori nello stack dei dati. Accetta una lista di argomenti separati da virgola che possono essere registri, valori letterali o anche registri vettoriali.

Nella rappresentazione testuale conrrisponde a `PUSH <arg1>[, <arg2>[, ...]]`. Dove ogni argomento può essere un registro semplice, un registro vettoriale con indice o un valore letterale separati da una virgola.

Nella codifica binaria ognuno degli argomenti viene differenziato in base al tipo in due categorie: valori letterali o registri e registri vetoriali. Questo è fatto perché un valore letterale può essere interpretato come il registro riservato 0 a cui segue il valore letterale, ma i registri vettoriali hanno bisogno di un indice oltre al numero di registro. Le due codifiche si sovrapporrebbero quindi un bit è usato per discriminare quale delle due categorie di argomento segue: 0 significa valore letterale o registro semplice, 1 significa registro vettoriale.

Nella codifica binaria la lista di argomenti è terminata usando il registro riservato numero 1 interpretato come registro semplice (quindi preceduto da un bit a 0).

In Fig. 3.7 si ha un esempio di codifica.

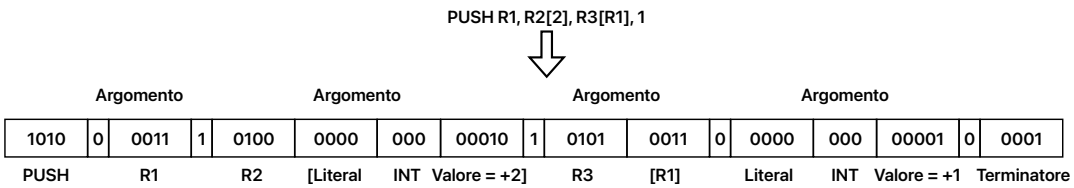

Figura 3.7: Esempio codifica PUSH

POP (1011)

Questa istruzione permette di estrarre valori dallo stack dei dati. Accetta una lista di argomenti separati da virgola che possono essere soltanti registri o registri vettoriali.

Nella rappresentazione testuale conrrisponde a `POP <arg1>[, <arg2>[, ...]]`. Dove ogni argomento può essere un registro semplice o un registro vettoriale con indice separati da una virgola.

Nella codifica binaria ognuno degli argomenti viene differenziato in base al tipo in due categorie: registri e registri vetoriali. Questo è fatto perché i registri vettoriali hanno bisogno di un indice oltre al numero di registro. Per discriminare quale delle due categorie di argomento segue: 0 significa registro semplice, 1 significa registro vettoriale.

Nella codifica binaria la lista di argomenti è terminata usando il registro riservato numero 1 interpretato come registro semplice (quindi preceduto da un bit a 0).

In Fig. 3.8 si ha un esempio di codifica.

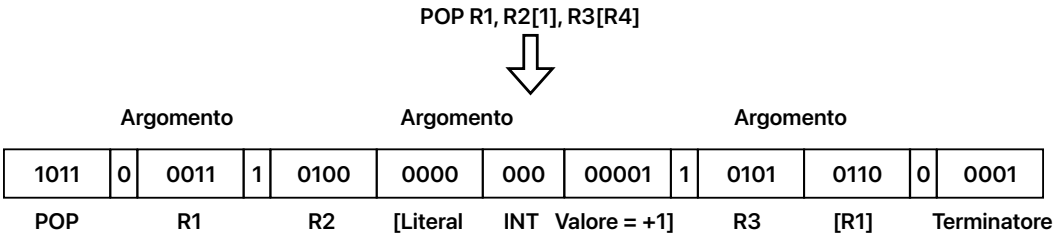

Figura 3.8: Esempio codifica POP

Capitolo 4

QRtree

4.1 Introduzione

Il dialetto *QRtree* è pensato come un linguaggio di programmazione che si concentra sul descrivere un albero di decisione. L'obiettivo è guidare l'utente attraverso una serie di scelte multiple o aperte per portarlo a una conclusione.

Una delle peculiarità di questo dialetto è l'assenza totale di variabili. Un'unica variabile nascosta è usata per contenere l'ultimo valore immesso dall'utente e su cui è possibile testare delle condizioni.

La più importante qualità di questo dialetto è la facilità di utilizzo e la vastità degli impieghi possibili: può essere utilizzato per guidare un operatore nella risoluzione di un problema tecnico in un'area industriale dove spesso una connessione Internet è assente o aiutare degli escursionisti a scegliere quale tracciato percorrere in base alle loro capacità, tempo disponibile, etc.

In questo dialetto mi sono concentrato principalmente sul linguaggio di alto livello, mentre il mio collega Mattia Scamuzzi si è occupato di progettare il linguaggio di basso livello. Segue comunque una breve descrizione del lavoro svolto dal mio collega.

4.2 Linguaggio di alto livello

Il linguaggio di alto livello è strutturato come un *linguaggio di scripting*, ovvero il punto di ingresso dell'esecuzione è sempre la prima istruzione invece che una funzione "*main*", anche perché le funzioni non sono contemplate in questo dialetto.

4.2.1 Valori letterali

All'interno del codice possono essere utilizzati valori letterali come stringhe per comunicare con l'utente o numeri interi o a virgola mobile per effettuare confronti.

Per i numeri interi è sufficiente il valore, mentre per i numeri a virgola mobile è a discrezione del programmatore scegliere quanta precisione è necessaria. Usando un suffisso si indica se memorizzare il valore come *half-precision* (**f16**) o *single-precision* (**f32**).

Per le stringhe si faccia riferimento alla sezione 2.2.3.

4.2.2 Referenze

Dato che in questo dialetto l'interazione con l'utente è fortemente presente, capita spesso per richiedere un input dell'utente o produrre un output di voler usare una stringa che dovrà per forza di cose essere memorizzata all'interno del codice QR.

Una soluzione alternativa consiste nell'usare dei numeri che corrispondono a una stringa. Una legenda contenente la corrispondenza tra numero e stringa viene allegata al codice QR ad esempio stampandola sullo stesso foglio o cartellone. In questo modo si possono avere stringhe anche di grandi dimensioni senza doverle salvare all'interno del codice QR, quindi aumentando la quantità di istruzioni che si possono utilizzare.

Notasi che questo non significa che non è possibile usare stringhe all'interno del codice, ma soltanto che esiste la possibilità di estrarre le stringhe che occupano maggiore spazio. La scelta è a discrezione del programmatore.

4.2.3 Parole chiave

Il linguaggio è composto di 7 parole chiave ed utilizza il livello di indentazione per determinare l'inizio e la fine di un blocco di istruzioni, in questo ci siamo ispirati al linguaggio Python.

Analizeremo ora le varie parole chiave.

input

Indica che un input deve essere prodotto dall'utente. Accetta un parametro che può essere una *stringa* o una *referenza* che rappresenta un'indicazione per l'utente su cosa è richiesto che immetta.

Con questa parola chiave si introduce una *scelta multipla*, le possibili scelte vengono dettate dalle successive istruzioni di controllo (**if**).

Il valore immesso viene memorizzato in una variabile nascosta e verrà mantenuto fino alla prossima richiesta di input, in questo modo tutte le successive istruzioni di controllo potranno accedere al valore immesso.

inputs

Indica che un input deve essere prodotto dall'utente. Accetta un parametro che può essere una *stringa* o una *referenza* che rappresenta un'indicazione per l'utente su cosa è richiesto che immetta.

Con questa parola chiave si introduce una *scelta aperta*, l'utente immetterà una stringa di testo, che potrà essere interpretata come valore numerico in base alle successive istruzioni di controllo (**ifc**).

Il valore immesso viene memorizzato in una variabile nascosta e verrà mantenuto fino alla prossima richiesta di input, in questo modo tutte le successive istruzioni di controllo potranno accedere al valore immesso.

if

Indica una diramazione dell'albero di decisione in base all'ultimo input fornito dall'utente. L'unico confronto possibile è di uguaglianza tra stringhe. Usando questa istruzione assieme alla parola chiave **else** si possono costruire diramazioni multiple.

Il formato è il seguente:

```
if "caso1":
    # Blocco indentato
else if "caso2":
    # Blocco indentato
else:
    # Blocco indentato
```

ifc

Indica una diramazione dell'albero di decisione in base all'ultimo input fornito dall'utente. È possibile con questa istruzione eseguire *confronti* utilizzando i comuni

operatori di confronto. Non è però possibile avere multipli confronti combinandoli con operatori booleani (AND, OR , NOT, ...), ma è comunque possibile ottenere gli stessi risultati con condizioni annidate.

Gli operatori disponibili sono <, <=, >, >=, ==, ! = e sono preposti al valore di confronto che può essere un valore intero o a virgola mobile (su 16 o 32 bit).

Usando questa istruzione assieme alla parola chiave **else** si possono costruire diramazioni multiple.

Il formato è il seguente:

```
ifc <= 10:
    # Blocco indentato
else ifc > 100:
    # Blocco indentato
else:
    # Blocco indentato
```

else

Indica una diramazione dell'albero di decisione in base all'ultimo input fornito dall'utente. Il ramo indicato da questa parola chiave verrà percorso se nessuna delle precedenti condizioni ha avuto esito positivo.

print

Indica la produzione di un output da parte del programma. Accetta un parametro che può essere una *stringa* o una *referenza* che rappresenta ciò che si vuole mostrare all'utente.

Esempi:

```
print "Messaggio" # Messaggio
print 1           # Look at reference 1
```

exit

Questa parola chiave indica la terminazione dell'esecuzione. Non accetta parametri.

4.3 Linguaggio di basso livello

La codifica a basso livello si divide in due parti: il *QRtree header* (opzionale) e il codice vero e proprio.

4.3.1 QRtree header

Dopo all'intestazione generale del *QRscript header* segue opzionalmente un header specifico per il dialetto *QRtree*. Un bit impostato a 1 segnala la presenza dell'header, è altrimenti impostato a 0 se assente.

Seguono dei comandi codificati su 3 bit, con la possibilità di essere estesi usando la codifica esponenziale (vedi 2.2.1).

HEADER_END (000)

Questo comando indica la fine del *QRtree header*. Questo è necessario in quanto non è noto a priori quanti comandi sono presenti nell'header.

INT_TYPE (001) e FLOAT_TYPE (010)

Questi comandi servono a indicare la dimensione di memorizzazione di numeri interi e a virgola mobile.

In particolare per il comando `INT_TYPE`, se il bit successivo è impostato a 0 tutti i numeri interi presenti sono da considerarsi su 16 bit, alternativamente se il bit successivo è impostato a 1 tutti i numeri interi presenti sono da considerarsi su 32 bit.

Per il comando `FLOAT_TYPE`, se il bit successivo è impostato a 0 tutti i numeri reali presenti sono da considerarsi su 16 bit (half-precision), alternativamente se il bit successivo è impostato a 1 tutti i numeri reali presenti sono da considerarsi su 32 bit (single-precision).

DICT_TYPES (011)

Questo comando è utilizzato per attivare dizionari di tipo *globale* e *specifico*. I successivi due bit identificano quale dizionario è attivato. Si ricorda che di default entrambi i dizionari sono attivi.

Con questo si intende che nessun tipo di ottimizzazione, basata sull'assenza di uno dei due dizionari, per elidere uno o entrambi i bit utilizzati per identificare il tipo di dizionario utilizzato per rappresentare una stringa nella sezione di codice è attiva.

Nella Tabella 4.1 sono riportate le configurazioni dei due bit successivi e il loro significato.

Bits	*globale*	*specifico*
00	NO_DICT_GLOBAL	NO_DICT_SPEC
01	NO_DICT_GLOBAL	DICT_SPEC
10	DICT_GLOBAL	NO_DICT_SPEC
11	DICT_GLOBAL	DICT_SPEC

Tabella 4.1: Bit per l'attivazione dei dizionari *globale* e/o *specifico*

DICT_SPEC_TYPE (100)

Questo comando è utilizzato per specificare quale *dizionario specifico* deve essere utilizzato. I successivi bit rappresentano il numero intero che identifica il *dizionario specifico*. Questo comando può essere ripetuto più volte per caricare più *dizionari specifici*. Quando vengono caricati ai dizionari è assegnato un indice crescente che verrà utilizzato nel codice per accedere alle voci del dizionario.

DICT_LOCAL (101)

Questo comando permette di definire un *dizionario locale* le cui voci sono salvate all'interno del eQRcode stesso. Questo può essere utile per stringhe che si ripetono spesso definendole una volta sola e usando poi una referenza numerica.

Al comando segue un numero intero senza segno codificato su 3 bit con estensione esponenziale che identifica la lingua del dizionario. La lingua `000` è detta di *default*. Successivamente è presente un numero intero senza segno codificato su 4 bit con estensione esponenziale che indica il numero di voci del dizionario. A seguire è presente una lista di voci con lunghezza corrispondente al numero di voci definito precedentemente.

Ogni voce usa un bit per identificare il tipo di codifica (`0` per ASCII-7 e `1` per UTF-8) e successivamente la stringa che compone la voce terminata dal carattere `EXT`.

USER_DEF (110)

Questo comando non è associato a nessuna azione specifica. Può essere quindi utilizzato a discrezione del programmatore per aggiungere nuove funzionalità. È responsabilità dello sviluppatore la gestione di un corretto parsing di questo comando nell'applicazione client che si occuperà di eseguire l'eQR code.

4.3.2 Sezione del codice

Dopo al QRtree header (opzionale) è presente la parte da effettivamente eseguire composta da una lista di istruzioni. Il set di istruzioni fornito dal linguaggio di basso livello è composto da 7 istruzioni codificate su 3 bit ognuna. L'ultimo valore disponibile `111` è stato lasciato di proposito senza un'istruzione assegnata per permettere l'estensione in futuro del numero di istruzioni componenti il linguaggio secondo il solito metodo esponenziale (vedi 2.2.1) anche se a partire da 3 bit e quindi procedendo con 6, 12, 24, eccetera.

Il set di istruzioni di basso livello è fortemente legato al linguaggio di alto livello, lo si nota dalla similarità dei nomi delle varie istruzioni.

input (000)

Questa istruzione introduce un input dell'utente guidato da una scelta multipla. Il formato dell'istruzione è il seguente `input <constante>`.

La `<costante>` è una stringa o una referenza da stampare per presentare all'utente la richiesta di un valore. Un bit è utilizzato per distinguere le due casistiche, se impostato a 0 segue una stringa, se impostato a 1 segue una referenza.

Le opzioni da cui l'utente può scegliere sono dettate dai successivi casi proposti dal costrutto **if-then-else** ad alto livello e dall'istruzione **if** a basso livello.

inputs (001)

Questa istruzione introduce un input dell'utente libero, nel senso che l'utente può immettere una stringa testuale qualunque. Il formato dell'istruzione è il seguente `inputs <constante>`.

La `<costante>` è una stringa o una referenza da stampare per presentare all'utente la richiesta di un valore. Un bit è utilizzato per distinguere le due casistiche, se impostato a 0 segue una stringa, se impostato a 1 segue una referenza.

Il valore immesso viene memorizzato in una variabile ed usato nei successivi contronti con l'istruzione **ifc**.

print (010)

Questa istruzione produce un output per l'utente. Il formato dell'istruzione è il seguente `print <constante>`.

La `<costante>` è una stringa o una referenza da stampare prima di terminare l'esecuzione. Un bit è utilizzato per distinguere le due casistiche, se impostato a 0 segue una stringa, se impostato a 1 segue una referenza.

printex (011)

Questa istruzione produce un output per l'utente e termina l'esecuzione. Il formato dell'istruzione è il seguente `printex <constante>`.

La `<costante>` è una stringa o una referenza da stampare prima di terminare l'esecuzione. Un bit è utilizzato per distinguere le due casistiche, se impostato a 0 segue una stringa, se impostato a 1 segue una referenza.

La scelta di accorpare l'istruzione di terminazione con l'istruzione per produrre un output è stata fatta perché si è notato un pattern negli alberi decisionali, ovvero che spesso arrivati alla foglia dell'albero si avevano sempre due comportamenti: la produzione di un output e la terminazione dell'esecuzione. Accorpando le due istruzioni si risparmia un notevole spazio nella codifica binaria (3 bit per ogni foglia dell'albero).

goto (100)

Questa istruzione (che non ha nessuna controparte nel linguaggio di alto livello) è utilizzata per effettuare salti incondizionati nel codice. Il formato dell'istruzione è

il seguente `goto <salto>`.

Il `<salto>` è il numero dell'istruzione a cui saltare è espresso come un numero intero senza segno, in quanto non possono esistere salti all'indietro in questo dialetto, ma solo salti in avanti. Il numero è codificato in binario con la codifica esponenziale (vedi 2.2.1). Per risparmiare spazio non è codificato il numero dell'istruzione assoluto, ma relativo all'istruzione corrente, in breve è usato il numero di istruzioni da saltare partendo dall'istruzione attuale.

if (101)

Questa istruzione comporta una scelta multipla per l'utente al seguito di un'istruzione **input**. Il formato dell'istruzione è il seguente `if <costante> <salto>`.

La `<costante>` è il valore della scelta multipla espresso come una stringa o una referenza. Un bit è utilizzato per distinguere le due casistiche, se impostato a 0 segue una stringa, se impostato a 1 segue una referenza.

Il `<salto>` è il numero dell'istruzione a cui saltare è espresso come un numero intero senza segno, in quanto non possono esistere salti all'indietro in questo dialetto, ma solo salti in avanti. Il numero è codificato in binario con la codifica esponenziale (vedi 2.2.1). Per risparmiare spazio non è codificato il numero dell'istruzione assoluto, ma relativo all'istruzione corrente, in breve è usato il numero di istruzioni da saltare partendo dall'istruzione attuale.

ifc (110)

Questa istruzione esegue un confronto con l'ultimo valore immesso dall'utente al seguito di un'istruzione **inputs**. Il formato dell'istruzione è il seguente `ifc <operatore> <costante> <salto>`.

L'`<operatore>` è un operatore di confronto che può essere <, <=, >, >=, == o != codificato su 3 bit (vedi Tabella 4.2). Notasi che le combinazioni di bit `110` e `111` sono attualmente non utilizzate e sono disponibili per future estensioni del linguaggio.

Operatore	Binario	Operatore	Binario
==	000	>	011
!=	001	>	100
<=	010	Non usato	110
>=	011	Non usato	111

Tabella 4.2: Codifica binaria degli operatori relazionali per l'istruzione **ifc**

La `<costante>` può essere un valore intero o reale. Nella codifica binaria due bit sono utilizzati per distinguere il tipo del valore costante. Se il primo bit è impostato a 0 indica che la costante è un valore intero e il seguente bit indica la lunghezza del valore intero (0 indica 16 bit, 1 indica 32 bit). Se invece il primo bit è impostato a 1 significa che la costante è un valore reale e il seguente bit indica la lunghezza del valore reale (0 indica 16 bit half-precision, 1 indica 32 bit single-precision) secondo lo standard IEEE 704-2008 [36].

Il `<salto>` è il numero dell'istruzione a cui saltare ed è espresso come un numero intero senza segno, in quanto non possono esistere salti all'indietro in questo dialetto, ma solo salti in avanti. Il numero è codificato in binario con la codifica esponenziale (vedi 2.2.1). Per risparmiare spazio non è codificato il numero dell'istruzione assoluto, ma relativo all'istruzione corrente, in breve è usato il numero di istruzioni da saltare partendo dall'istruzione attuale.

4.4 Interprete

Nel design di un interprete per questo dialetto un considerazione che possiamo fare è: come avviene l'interazione con l'utente? Questa domanda è molto importante perché, specialmente per questo dialetto, l'interazione con l'utente è fondamentale. È al cuore di questo dialetto il ricevere input dall'utente per poterlo guidare attraverso l'albero decisionale.

La soluzione da noi scelta si basa sull'utilizzo del browser, attenzione però, sempre senza una connessione a Internet. La nostra scelta si basa sul fatto che al giorno d'oggi ogni dispositivo ha a disposizione un browser che può eseguire del codice Javascript e l'HTML è appositamente progettato per le interazioni con

l'utente attraverso bottoni, caselle di testo e messaggi stampati a schermo che è tutto di ciò di cui abbiamo bisogno.

4.4.1 Generazione della pagina

Quando si vuole eseguire un eQR code, dopo averlo scansionato ed estratto il codice binario, viene eseguito il parsing del codice binario estratto tenendo in conto QRscript header (vedi 2.3.1) e l'header del dialetto (vedi 4.3.1) producendo un file contenente le istruzioni del linguaggio di basso livello.

Partendo dalla lista di istruzioni di basso livello viene generata una pagina HTML contenente tutti gli elementi necessari per eseguire ogni ramo dell'albero decisionale e uno script Javascript in grado di reagire alle interazioni dell'utente e di tenere traccia dello stato dell'esecuzione attraverso una macchina a stati.

Ogni istruzione è convertita in una sezione di codice HTML in modo statico. L'unico costrutto che richiede un trattamento speciale è l'**else** in quanto non è un'istruzione, ma si riconosce dall'assenza di un'istruzione `goto` dopo una serie di istruzioni `if`.

Tutte le istruzioni producono un elemento `div` con due attributi sempre presenti (`data-line` per indicare il numero dell'istruzione e `data-type` per indicare il tipo dell'istruzione) e altri opzionali che al suo interno contiene altri elementi che dipendono dal tipo dell'istruzione.

Istruzioni come `input`, `print` e `printex` producono un elemento che mostra il messaggio o la referenza a schermo.

L'istruzione `goto` produce un elemento vuoto con un attributo (`data-par1`) per indicare l'istruzione a cui saltare quando il salto verrà eseguito.

L'istruzione `inputs` produce un elemento che mostra il messaggio o la referenza a schermo e contiene un elemento per ricevere l'input dell'utente con un bottone per confermare l'inserimento.

L'istruzione `if` produce un elemento che contiene un bottone che ha come testo la costante presente nell'istruzione. Un elemento simile viene prodotto per un ramo **else**.

L'istruzione `ifc` produce un elemento vuoto con tre attributi `data-par1-cmp`, `data-par1-val` e `data-par2` che contengono rispettivamente l'operatore di confronto, il valore con cui confrontare l'input dell'utente e l'istruzione a cui saltare se il confronto è positivo.

Ai componenti generati dalle istruzioni si aggiungono alcuni stili basandoci sulla libreria Bootstrap ed alcuni stili specifici per nascondere le istruzioni non ancora eseguite. La distinzione tra istruzioni eseguite e non è fatta utilizzando un attributo sugli elementi delle istruzioni già processate (`data-executed`) che attiva una regola di stile per mostrare il contenuto.

Viene anche inserito nella pagina lo script Javascript responsabile dell'esecuzione. Questo script si attiverà al caricamento della pagina andando a configurare la macchina a stati necessaria per l'esecuzione.

4.4.2 Esecuzione

Dopo che la pagina è stata generata con tutti i componenti dati dalle istruzioni, non appena verrà aperta in un browser qualunque si avvierà l'esecuzione. Un contatore è utilizzato per memorizzare la prossima istruzione da eseguire. Viene usato un evento custom chiamato `execute` per scatenare l'esecuzione di una specifica istruzione.

Prima di iniziare l'esecuzione ad ognuno degli elementi generati per ogni istruzione è agganciato un ascoltatore di eventi per lo specifico evento di tipo `execute`. Ogni tipologia di istruzione ha una gestione specifica:

- `print` e `input` producono l'output a schermo e continuano l'esecuzione,
- `inputs` mostra la casella di testo, poi attende che l'utente la riempia e prema il bottone prima di continuare l'esecuzione,
- `printex` produce l'output a schermo e termina l'esecuzione,
- `if` mostrano il bottone e continuano l'esecuzione fino a che l'istruzione successiva continua ad essere di tipo `if`,
- `ifc` preleva il valore immesso dall'utente nell'ultima istruzione `inputs`, effettua il confronto e decide se proseguire l'esecuzione o effettuare un salto.

Dopo quest ultimo passo l'esecuzione può cominciare a partire dalla prima istruzione.

Capitolo 5

Esempi

5.1 QRprog: Da codice di alto livello a codice QR

Per questo esempio useremo il dialetto QRprog, ma il procedimento è lo stesso per altri dialetti.

Questo procedimento con i passi intermedi è svolto da uno script Python che si occupa di mediare con i vari moduli per le singole conversioni da linguaggio ad alto livello a linguaggio di basso livello, da linguaggio di basso livello a rappresentazione binaria e infine da rappresentazione binaria a codice QR.

Prendiamo un semplice codice di esempio e salviamolo in un file "`test.txt`":

```
fn main() {
  int dischi, da, a;
  print "Questo programma risolve una torre di Hanoi";
  print "Quanti dischi ci sono?";
  input dischi;
  print "In quale paletto si trovano? (0, 1, 2)";
  input da;
  print "In quale paletto vuoi spostali? (0, 1, 2)";
  input a;
  if (da != a) {
    hanoi(dischi, da, a);
  }
```

```
  print "Risolto!";
}

fn hanoi(int dischi, int da, int a) {
  if (dischi == 1) {
    printf "Muovi un disco dal posto %d al posto %d", da, a;
  } else {
    int ausiliario;
    ausiliario = 3 - (da + a);
    dischi = dischi - 1;
    hanoi(dischi, da, ausiliario);
    hanoi(1, da, a);
    hanoi(dischi, ausiliario, a);
  }
}
```

Notasi che avendo la funzione `main` questo codice può essere eseguito appena scansionato dal dispositivo.

Passando questo file al compilatore questo produrrà l'immagine contenente il codice QR assieme ad alcuni file contenenti rappresentazioni intermedie.

Uno dei file prodotti è "`test.qr`", questo file contiene il codice di basso livello in cui sono presenti gli identificatori delle funzioni e le label per eseguire i salti.

In questo esempio il file conterrà:

```
main:   OPR R0, 0
  OPR R1, 0
  OPR R2, 0
  OUT "Questo programma risolve una torre di Hanoi"
  OUT "Quanti dischi ci sono?"
  IN R0
  OUT "In quale paletto si trovano? (0, 1, 2)"
  IN R1
  OUT "In quale paletto vuoi spostali? (0, 1, 2)"
  IN R2
  OPR R3, R1 R2 EQ
  JMPL L1
```

```
  PUSH R0, R1, R2
  JMPF hanoi
L1:  OUT "Risolto!"
  EXIT
hanoi:   POP R0, R1, R2
  OPR R3, R0 1 NEQ
  JMPL L2
  OUT R1, R2 "Muovi un disco dal posto %d al posto %d"
  JMP L3
L2:  OPR R4, 0
  OPR R4, 3 R1 R2 PLUS MINUS
  OPR R0, R0 1 MINUS
  PUSH R0, R1, R4
  JMPF hanoi
  PUSH 1, R1, R2
  JMPF hanoi
  PUSH R0, R4, R2
  JMPF hanoi
L3:  RET
```

Un altro file prodotto è "`test.bin`", questo file contiene la rappresentazione binaria delle istruzioni in cui sono già stati sostituite le label per i salti con il numero relativo di istruzioni da saltare.

In questo esempio il file conterrà:

```
10000000000000001000000000010000000001000000001000000110000000001000000
00010000010000000000100000000110000000010010100011110101110010111100111
01111101001101111010000011100001110010110111111001111110010110000111011
01101110110111000010100000111001011010011110011110111111011001110110110
11001010100000111010111011101100001010000011101001101111111001011100101
10110010101000001100100110100101000001001000110000111011101101111101001
10010000011000011000000001001010001111010111000011101111011100110100101
01010000011001001101001111001111000111101000110100101000001100011110010
10010100000111001111011111011101101111011111100000110000101110001010000
00000000100100100111011100100000111000111101011100001110110011001010010
10000011100001100001110110011001011110100111010011011110100000111001
```

```
11101001010000011101001110010110111111101101100001110111011011110111
11101000000101000011000001011000100000011000101011000100000011001001
01001000001100001011100011100000000100100100111011100100000111000111
10101110000111011001100101010000011100001100001110110011001011110100
11101001101111010000011101101110101110111111010010100000111001111100
00110111111100111111010011000011101100110100101111110100000010100001 1
00000101100010000001100010101100010000001100100101001000001100001011
10010000000101000110010010101000010011000011101000010000110010000001
01000000111000000000100101001011010011110011110111111011001110100 1101
11101000010000011000011001101100010000110010000001000001010001000000
00000001101100000100110000111000000110010010001010011011110101110111
11110110110100101000001110101110111001000001100100110100111100111100
01111011110100000110010011000011101100010000011100001101111111001111
10100110111101000000100101110010001000001100001110110001000001110000
11011111110011111010011011110100000010010111001000000011001000101000
00011000000000100000000100000110000000000000110001100100100000100010 00
01000000100001000000000000001100010000110100001000011001100000101000 1
10011010000000000000100011001000000101000110111010000100011000100000
0101000111010110
```

Si noti che in questo binario non è contenuto l'header di QRscript che verrà aggiunto in seguito.

L'immagine generata è rappresentata in Fig. 5.1.

Figura 5.1: Esempio di eQR code

In questo caso il codice QR generato è di dimensioni contenute, ma nel caso di un sorgente contenente più istruzioni verrà prodotto un codice QR di maggiori dimensioni o addirittura verrà utilizzata la *continuazione* per dividere il binario su due codici QR che dovranno essere tutti scansionati prima di poter ricostruire il binario originale e poter quindi continuare con l'esecuzione.

5.2 QRtree: Esecuzione del codice QR

Per questo esempio useremo il dialetto QRtree, ma il procedimento è simile per altri dialetti.

Questo procedimento che comprende tutti i passi intermedi è svolto da uno script Python che si occupa di coordinare i vari moduli per le singole conversioni da codice QR a rappresentazione binaria, da rappresentazione binaria a linguaggio di basso livello. Infine il linguaggio di basso livello ottenuto verrà eseguito da un interprete.

Prendiamo un eQR code contenente un programma QRtree.

Figura 5.2: Esempio di eQR code

Eseguendo la scansione dell'immagine si potrà notare il formato binario nella codifica dei dati. Una volta estratti i dati viene processato l'header QRscript. Viene rimosso il *padding*. La *continuazione* viene gestita e se non sono presenti tutti i segmenti questi ultimi verranno richiesti prima di poter passare al passo successivo. La *sicurezza* viene gestita: azioni specifiche vengono intraprese in base a quale profilo di sicurezza è stato scelto. Nel caso del profilo 0 in cui la sicurezza è disabilitata si avranno le istruzioni in chiaro. Viene processata la sezione riguardante l'*URL*, se presente e una connessione a alla rete disponibile, l'esecuzione proseguirà visitando l'URL. Il *dialetto* e *versione* vengono riconosciuti e comunicati al prossimo step per la decodifica delle istruzioni. Dopo di che la parte di istruzioni del dialetto viene estratta e salvata in un file come binario.

00000010000111101000110010101000001100011110111111001001101001110001
11100101010000011001001101001010000011001011110010111001011011111110
01011001010100000111001011010011110000110111111100101110100110000101
00000110110011000010100000110110011000011110110110000111101001110010
11010011100011110010101111110000011101000101000001100010000011011010
10001010000011001000000111000101000101000001100110000011100110100010
10000011010000000111010010001010100010111010101100101011100110111010
00110111100100000011000110110111101100100011010010110001101100101001
00000011011100110111101101110001000001100001110101000001000000110011
10110010101110011011101000110100101110100011011110010000001100100011
00001011011000010000001110000011100100110111101100111011100100110000
10110110101101101011000010010111000000011011101011001101010001010011
00001001110110111101100010011011001100001110110010001000000110111001
10111101101110001000001100001110101000001000000110001101101000011010
01011101010111001101101111001011100000001101110001100101001000010011
00110111101000001110011110001111000011110010110100111000111101111010
00001100100110010111011001101100010011111000011100011111000111101011
10000101000001110010110100111100111110101110110011101001100001010000
01101111111001111101001110010111010111010011110100110111101011100000
01101110010010000100110001001111101001110111011001111110010110010111
10011111001111011110100000110010011001011101100110110001001111100001
11000111110001111010111000010100000110111011011111110111001000001100 1
10111010111011101111010110100111011111101110110000101011100000011011
10011001000101000111101011100001110111011101001101001010000011000111
10100011010011101100110100101000001100100110100101000001110110110010
11110011111010011010011110100110100101000001110011110111111011101101
11101000001110000111001011001011110011110010111011101110100110100101
00000110111011001011101100010000011000111100101111001111101001100101
11011001101100110111101111110000011110101000000000000000110001001000
10100100101101100001000000111000001110010011011110110001001101100011
00101011011010110000100100000011011100110111101101110001000000111000
00111010111000011101100100010000001100101011100110111001101100101011
10010011001010010000001110010011010010111001101101111011011000111010
00110111100101110000000110111010101000010100111110101111000011001011
11001011000011111010011011110100000110100111011000100000111000110010

```
11110011110111101000001101101110000111100111110011110100111011011101
11101011100000011011101000
```

Questo binario viene successivamente prelevato e parsificato per ottenere il linguaggio di basso livello che lo ha generato.

```
(0) input "Che codice di errore riporta la lavatrice?"
(1) if "P1" (8)
(2) if "P2" (11)
(3) if "P3" (13)
(4) if "P4" (15)
(5) print "Questo codice non è gestito dal programma."
(6) printex 5
(7) goto (21)
(8) print "L'oblò non è chiuso."
(9) printex 1
(10) goto (21)
(11) print "Lo scarico dell'acqua risulta ostruito."
(12) printex 2
(13) print "L'ingresso dell'acqua non funziona."
(14) printex 3
(15) inputs "Quanti chili di vestiti sono presenti nel cestello?"
(16) ifc > 6 (19)
(17) print "Il problema non può essere risolto."
(18) printex 5
(19) print "Superato il peso massimo."
(20) printex 4
```

Il codice intermedio viene salvato su un file per poter essere processato in seguito dall'interprete.

L'interprete preleva il codice intermedio e si occupa di eseguirlo interagendo con l'utente e processando le istruzioni. Il modo in cui questo accade dipende dal

dialetto utilizzato. In questo esempio, l'interprete del dialetto QRtree produce una pagina HTML contenente del codice Javascript con cui l'utente può interagire.

In Fig. 5.3 si può vedere un esempio di esecuzione.

Figura 5.3: Esempio di pagina generata dall'interprete QRtree

Il codice HTML generato viene salvato in un file e automaticamente aperto nel browser di default del dispositivo per avviare l'esecuzione.

Capitolo 6

Conclusioni

In questa tesi, è stata formalizzata e implementata una tecnologia per inserire intelligenza all'interno di codici QR che permette di eseguire operazioni complesse anche in totale assenza di connessione a Internet. Questa tecnologia trova applicazione in molti ambiti dal turismo, ad esempio per guidare i visitatori durante una visita, all'industria, per valutare e risolvere un problema di un macchinario.

Durante lo sviluppo di questa tesi sono stati prodotti due dialetti, QRtree e QRprog, ognuno con il proprio compilatore e interprete che permettono di realizzare tutti i passi richiesti per utilizzare questa tecnologia. Ognuno ha il proprio ambito applicativo: QRtree è utile per guidare l'utente nel prendere una decisione, mentre QRprog è utile per la più generale esecuzione di un algoritmo.

Il binario prodotto da entrambi i dialetti è estremamente compatto ed è stata la maggiore sfida durante lo sviluppo. Infatti la possibilità di *continuazione*, cioè di concatenare più codici QR, nasce per sopperire a questa difficoltà e potrà essere utile in futuro per dialetti che non riusciranno a essere altrettanto ottimizzati.

La toolchain di compilazione presentata in questa tesi può facilmente essere usata e può anche essere la base per futuri dialetti. In particolare, durante la formalizzazione di QRscript, è stata prestata attenzione alla estendibilità del formato, utilizzando codifiche binarie estendibili per permettere di creare con facilità successive versioni ed implementazioni.

Nuovi dialetti possono facilmente essere sviluppati per ambiti applicativi non ancora completamente supportati dai dialetti proposti. Un processo di pubblicazione dei risultati ottenuti durante questo lavoro di tesi è in corso, e sperabilmente si concluderà con la pubblicazione di un articolo contenente le specifiche di QRscript

e dei due dialetti già sviluppati.

Bibliografia

[1] Stefano Scanzio, Lukasz Wisniewski, and Piotr Gaj. Heterogeneous and dependable networks in industry – A survey. *Computers in Industry*, 125:103388, 2021.

[2] Gianluca Cena, Stefano Scanzio, Adriano Valenzano, and Claudio Zunino. Performance evaluation of the EtherCAT distributed clock algorithm. In *2010 IEEE International Symposium on Industrial Electronics*, pages 3398–3403, 2010.

[3] Gianluca Cena, Stefano Scanzio, Mohammad Ghazi Vakili, Claudio Giovanni Demartini, and Adriano Valenzano. Assessing the Effectiveness of Channel Hopping in IEEE 802.15.4 TSCH Networks. *IEEE Open Journal of the Industrial Electronics Society*, pages 1–17, 2023.

[4] Stefano Scanzio, Francesco Xia, Gianluca Cena, and Adriano Valenzano. Predicting Wi-Fi link quality through artificial neural networks. *Internet Technology Letters*, 5(2):e326, 2022.

[5] Gianluca Cena, Stefano Scanzio, Adriano Valenzano, and Claudio Zunino. A distribute-merge switch for EtherCAT networks. In *2010 IEEE International Workshop on Factory Communication Systems Proceedings*, pages 121–130, 2010.

[6] Gianluca Cena, Marco Cereia, Ivan Cibrario Bertolotti, and Stefano Scanzio. A MODBUS extension for inexpensive distributed embedded systems. In *2010 IEEE International Workshop on Factory Communication Systems Proceedings*, pages 251–260, 2010.

[7] Maurizio Mongelli and Stefano Scanzio. Approximating Optimal Estimation of Time Offset Synchronization With Temperature Variations. *IEEE Transactions on Instrumentation and Measurement*, 63(12):2872–2881, 2014.

[8] Maurizio Mongelli and Stefano Scanzio. A neural approach to synchronization in wireless networks with heterogeneous sources of noise. *Ad Hoc Networks*, 49:1–16, 2016.

[9] Gianluca Cena, Stefano Scanzio, and Adriano Valenzano. SDMAC: A Software-Defined MAC for Wi-Fi to Ease Implementation of Soft Real-Time Applications. *IEEE Transactions on Industrial Informatics*, 15(6):3143–3154, 2019.

[10] Stefano Scanzio, Gianluca Cena, and Adriano Valenzano. Enhanced Energy-Saving Mechanisms in TSCH Networks for the IIoT: The PRIL Approach. *IEEE Transactions on Industrial Informatics*, 19(6):7445–7455, 2023.

[11] ISO/IEC. Information technology — Automatic identification and data capture techniques — QR Code bar code symbology specification. *ISO/IEC Std 18004-2015*, pages 1–117, Feb 2015.

[12] Gustave Reed, Irving S.; Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[13] Yulong Yan, Zhuo Zou, Hui Xie, Yu Gao, and Lirong Zheng. An IoT-Based Anti-Counterfeiting System Using Visual Features on QR Code. *IEEE Internet of Things Journal*, 8(8):6789–6799, 2021.

[14] Pei-Yu Lin, Wen-Chuan Wu, and Jen-Ho Yang. A QR Code-Based Approach to Differentiating the Display of Augmented Reality Content, journal = Applied Sciences. 11(24), 2021.

[15] Simon Soele Madsen, Athila Quaresma Santos, and Bo Nørregaard Jørgensen. A QR code based framework for auto-configuration of IoT sensor networks in buildings. *Energy Informatics*, pages 46–volume = 4, 2021.

[16] Xiaohua Tian, Shaofei Qin, Binyao Jiang, Yuan Gao, and Xinbing Wang. Fast Batch Reading Densely Deployed QR Codes. *IEEE Transactions on Mobile Computing*, 22(3):1507–1520, 2023.

[17] Xiaojun Yu, Zeming Fan, Hao Wan, Yuye He, Junye Du, Nan Li, Zhaohui Yuan, and Gaoxi Xiao. Positioning, Navigation, and Book Accessing/Returning in an Autonomous Library Robot using Integrated Binocular Vision and QR Code Identification Systems. *Sensors*, 19(4), 2019.

[18] Sy-Hung Bach, Phan-Bui Khoi, and Soo-Yeong Yi. Application of QR Code for Localization and Navigation of Indoor Mobile Robot. *IEEE Access*, 11:28384–28390, 2023.

[19] Yizheng Zhang, Wangshu Zhu, and Andre Rosendo. QR Code-Based Self-Calibration for a Fault-Tolerant Industrial Robot Arm. *IEEE Access*, 7:73349–73356, 2019.

[20] P. Mathivanan and A. Balaji Ganesh. QR code based color image cryptography for the secured transmission of ECG signal. *Multimedia Tools and Applications*, 78:6763–6786, 2019.

[21] Berrin Arzu Eren. QR code m-payment from a customer experience perspective. *Journal of Financial Services Marketing*, 2022.

[22] Emin Borandag. A Blockchain-Based Recycling Platform Using Image Processing, QR Codes, and IoT System. *Sustainability*, 15(7), 2023.

[23] Joon-Soo Kim, Chang-Yong Yi, and Young-Jun Park. Image Processing and QR Code Application Method for Construction Safety Management. *Applied Sciences*, 11(10), 2021.

[24] Heider A. M. Wahsheh and Flaminia L. Luccio. Security and Privacy of QR Code Applications: A Comprehensive Study, General Guidelines and Solutions. *Information*, 11(4), 2020.

[25] Siti Nazleen Abdul Rabu, Haniza Hussin, and Brandford Bervell. QR code utilization in a large classroom: Higher education students' initial perceptions. *Education and Information Technologies*, 24:359–384, 2019.

[26] Surbhi Bhatia and Abdulaziz Saad Albarrak. A Blockchain-Driven Food Supply Chain Management Using QR Code and XAI-Faster RCNN Architecture. *Sustainability*, 15(3), 2023.

[27] Federico Pasquaré Mariotto, Fabio Luca Bonali, Alessandro Tibaldi, Emanuela De Beni, Noemi Corti, Elena Russo, Luca Fallati, Massimo Cantarero, and Marco Neri. A New Way to Explore Volcanic Areas: QR-Code-Based Virtual Geotrail at Mt. Etna Volcano, Italy. *Land*, 11(3), 2022.

[28] Tianyu Wang, Hong Zheng, Changhui You, and Jianping Ju. A Texture-Hidden Anti-Counterfeiting QR Code and Authentication Method. *Sensors*, 23(2), 2023.

[29] Mingliang Xu, Hao Su, Yafei Li, Xi Li, Jing Liao, Jianwei Niu, Pei Lv, and Bing Zhou. Stylized Aesthetic QR Code. *IEEE Transactions on Multimedia*, 21(8):1960–1970, 2019.

[30] Junfeng Sun, Kiran Shrestha, Hyejin Park, Pravesh Yadav, Sajjan Parajuli, Seunggun Lee, Sagar Shrestha, Gyan Raj Koirala, Yushin Kim, Kale Amol Marotrao, Bijendra Bishow Maskey, Ojo Charles Olaoluwa, Jinhwa Park, Hyewon Jang, Namsoo Lim, Younsu Jung, and Gyoujin Cho. Bridging R2R Printed Wireless 1 Bit-Code Generator with an Electrophoretic QR Code Acting as WORM for NFC Carrier Enabled Authentication Label. *Advanced Materials Technologies*, 5(2):1900935, 2020.

[31] Rongjun Chen, Weijie Li, Kailin Lan, Jinghui Xiao, Leijun Wang, and Xu Lu. Fast Adaptive Binarization of QR Code Images for Automatic Sorting in Logistics Systems. *Electronics*, 12(2), 2023.

[32] Ammar Mohammed Ali and Alaa Kadhim Farhan. Enhancement of QR Code Capacity by Encrypted Lossless Compression Technology for Verification of Secure E-Document. *IEEE Access*, 8:27448–27458, 2020.

[33] Sijia Liu, Zhengxin Fu, and Bin Yu. Rich QR Codes With Three-Layer Information Using Hamming Code. *IEEE Access*, 7:78640–78651, 2019.

[34] Yuan Xu, Zhangming Liu, Rui Liu, Mengxue Luo, Qi Wang, Liqin Cao, and Shuangli Ye. Inkjet-printed pH-sensitive QR code labels for real-time food freshness monitoring. *Journal of Materials Science*, 56:18453–18462, 2021.

[35] Jeng-Shyang Pan, Tao Liu, Bin Yan, Hong-Mei Yang, and Shu-Chuan Chu. Using color QR codes for QR code secret sharing. *Multimedia Tools and Applications*, 81:15545–15563, 2022.

[36] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[37] ISO/IEC. Information technology — ISO 7-bit coded character set for information interchange. *ISO/IEC Std 646-1991*, pages 1–15, Dec 1991.

[38] François Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629, November 2003.

[39] Stefano Scanzio, Gianluca Cena, and Adriano Valenzano. QRscript: Embedding a Programming Language in QR codes to support Decision and Management. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2022.