POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Incorporare linguaggi di programmazione in QR code eseguibili

Politecnico di Torino

Relatori

Dr. Stefano Scanzio

Dr. Gianluca Cena

Candidato

Mattia Scamuzzi

Luglio 2023

Alla mia famiglia
e a tutte le splendide persone
che ho incontrato durante questo viaggio

"Ogni enigma ha una soluzione"
Professor Hershel Layton

Sommario

Negli ultimi anni, l'uso dei QR code è aumentato in maniera considerevole, tanto che, ad oggi, essi vengono impiegati in moltissimi campi diversi. Essi sono generalmente usati per contenere esclusivamente una stringa che rappresenta un URL, al quale l'utente accede al momento della scansione. Tuttavia, pensando in maniera più generale, il QR code non contiene al suo interno la stringa vera e propria, ma una serie di bit che codifica la stringa stessa. È quindi possibile estendere questa logica in modo che esso possa contenere al suo interno altri tipi di dato che non siano semplici stringhe, fino ad arrivare ad inserire al suo interno un vero e proprio programma che, una volta scannerizzato il QR Code, possa essere eseguito all'interno di una macchina virtuale o con l'aiuto di un interprete.
Tuttavia, è fondamentale notare come lo spazio interno ad un QR code sia estremamente limitato, anche nelle versioni più grandi di esso (il più grande arriva infatti a contenere appena 2953 bytes); occorre quindi definire un formato molto compatto di conversione del codice del programma in binario, così che si possa inserire la più grande quantità possibile di dati all'interno di un singolo QR code.
QRscript definisce questo formato compatto per due tipi di programmi: un programma che rappresenta un albero delle decisioni ed un programma general purpose scritto in un linguaggio pseudo-C. Il lavoro svolto in questa tesi consiste nella progettazione di un linguaggio intermedio fra linguaggio ad alto livello e binario e nella successiva implementazione, sia per l'albero decisionale che per il programma general purpose, di una serie di compilatori in grado di tradurre, per prima cosa, il linguaggio ad alto livello in linguaggio intermedio e viceversa e successivamente di tradurre questo linguaggio intermedio in binario e viceversa. Attraverso l'implementazione di questi software, è stato possibile trasformare programmi in linguaggio ad alto livello dapprima in sequenze di bit e quindi in QR Code particolari, denominati *eQR code* e successivamente, leggendo questi eQR code, riottenere ed eseguire il codice della rappresentazione intermedia.

Indice

Elenco delle figure

Elenco delle tabelle

Listati

Capitolo 1

Introduzione

La tecnologia *Quick Response Code*, più comunemente conosciuta come QR code, consiste in un barcode bidimensionale e fu inventata nel 1994 dalla Denso Wave, un'azienda giapponese nel settore automotive. In origine, aveva lo scopo di tracciare i veicoli durante il processo di produzione. Questa nuova tecnologia vantava due principali pregi rispetto ai suoi predecessori, i quali erano invece monodimensionali: una migliore velocità di riconoscimento ma, soprattutto, la possibilità di contenere una dimensione decisamente più elevata di dati al suo interno.
Nonostante ciò, la loro dimensione risulta comunque essere abbastanza ridotta in un contesto come quello attuale: per questo motivo, ad oggi, essi sono prevalentemente utilizzati per codificare informazioni testuali, come URL a pagine web o informazioni per accedere a reti Wi-Fi.
Quando si parla quindi di eseguire un programma scannerizzando un QR code, quello che in realtà succede è che un dispositivo connesso ad Internet legge un URL, mentre il programma vero e proprio viene comunque eseguito su un server remoto. Tuttavia, come sottolineato in precedenza, ciò richiede che il dispositivo che effettua la scansione sia connesso ad Internet, così che possa connettersi al server che esegue il programma. Ciò, però, non è sempre possibile: basti pensare a quelle aree rurali dove la connessione ad Internet è spesso molto debole o addirittura assente, oppure a tutte quelle attività industriali che, per questioni di costi o di sicurezza, rinunciano oppure limitano la copertura della loro connessione ad Internet.
In questi casi, al fine di avere un funzionamento analogo a quello atteso, è necessario che il programma da eseguire sia direttamente codificato all'interno del QR code stesso. Tuttavia, ciò fa sorgere un problema non indifferente: il programma deve essere fisicamente inserito all'interno di uno o più QR code e, perché tutto il procedimento possa essere ottenuto nella pratica, esso deve essere il più compatto possibile, così che il numero di bit necessari per la sua codifica sia il più basso possibile.

QRscript definisce questo formatto compatto, permettendo di passare da un programma al rispettivo QR code e viceversa, passando attraverso una rappresentazione intermedia, simile all'Assembly.
Il lavoro svolto in questa tesi, basato su quanto proposto in [1], ha l'obiettivo di progettare ed implementare QRscript per due diversi tipi di dialetti: il primo in grado di codificare un albero decisionale; il secondo in grado invece di codificare un linguaggio general-purpose. Mentre il secondo caso è quello più generale, si è pensato di sviluppare una variante di QRscript anche per gli alberi delle decisioni per due principali motivi: un albero decisionale ha una struttura più semplice rispetto ad un linguaggio general-purpose, per cui poteva essere sfruttato come base per lo sviluppo di quest'ultimo e inoltre, la struttura dell'albero decisionale si presta estremamente bene per alcuni dei principali contesti applicativi individuati: per esempio, un albero decisionale potrebbe, con estrema facilità, scegliere quale, tra i percorsi disponibili, sia il migliore per una comitiva che si trova nel bel mezzo di un'escursione in montagna, solamente facendo loro alcune domande, oppure sarebbe altrettanto efficiente nel diagnosticare quale potrebbe essere il problema di un apparecchio danneggiato all'interno di una fabbrica che non dispone di una connessione Internet, chiedendo all'utente di inserire alcuni dati facilmente reperibili.

Capitolo 2

QR code

2.1 Applicazioni

I QR code sono diventati una tecnologia sempre più popolare negli ultimi anni, grazie alla loro capacità di memorizzare una molto maggiore quantità di informazioni rispetto ad i loro predecessori in uno spazio relativamente piccolo. Sebbene fossero stati inizialmente utilizzati per il tracciamento delle parti nella produzione di veicoli, i QR code vengono, ad oggi, utilizzati in una gamma molto più ampia di applicazioni e la loro diffusione è notevolmente aumentata dall'inizio della pandemia da Covid-19.

Nella maggior parte dei casi, il QR code contiene un URL che fa riferimento ad un sito web contenente informazioni più dettagliate sull'oggetto a cui il codice fa riferimento: in questo caso, ad oggi, il loro utilizzo più comune si ha nel campo del marketing e della pubblicità [2], dove sono spesso usati per fornire informazioni aggiuntive ai clienti, come offerte speciali, sconti e promozioni oppure per portare il cliente direttamente alla pagina dell'acquisto. Il loro uso massiccio deriva da due principali motivazioni: per prima cosa, essi possono essere comodamente stampati su volantini e manifesti ed è inoltre stato notato che, oltre ad attrarre più volentieri i potenziali clienti, riducono il loro sforzo e la latenza tra l'inizio della ricerca ed il momento in cui la risposta attesa viene trovata, aumentando quello che in economia viene chiamato *tasso di conversione*, cioè la possibilità che il contatto fra un potenziale cliente e l'annuncio si converta in una vendita [3].

Un altro degli usi più comuni è collegato al settore alimentare, in particolare per quanto riguarda i menù dei ristoranti, a cui i clienti possono avere immediato accesso dai loro dispositivi scansionando il codice sul loro tavolo.

Un uso molto simile a quest'ultimo si trova sempre più spesso nel settore dei trasporti [4], dove i passeggeri possono scansionare i QR code sui loro biglietti per accedere rapidamente ad informazioni importanti sul viaggio, come il percorso, gli orari e le fermate; discorsi equivalenti possono essere fatti per l'assistenza sanitaria [5], per avere informazioni complete riguardanti i pazienti, per l'istruzione [6], dove

i QR code vengono sempre più utilizzati come supporto all'insegnamento su libri di testo e materiale didattico, fornendo link a siti web specializzati che permettono di approfondire gli argomenti trattati e per tutti gli eventi in generale, dove la scansione di un QR code può portare ad un sito web che specifica tutte le informazioni a riguardo.

In altri casi, invece, il QR code contiene direttamente delle informazioni alfanumeriche che possono essere usate dall'utente che ha scannerizzato il codice oppure che vengono usate per particolari operazioni automatiche: è questo il caso, ad esempio, in cui il QR code contiene una stringa con un formato particolare che permette di connettere il dispositivo usato per la scansione alla rete Wi-Fi corrispondente ai dati contenuti nel codice.

Uno degli usi che, invece, sta molto aumentando di popolarità in questi ultimi anni è quello nel settore dei pagamenti online [7]. Quando i clienti scansionano il QR code sul terminale di pagamento, l'applicazione elabora automaticamente il pagamento, consentendo di evitare la necessità di inserire manualmente i dati della carta di credito.

I QR code vengono anche usati per questioni di controllo degli accessi ad aree riservate, nelle quali la loro scansione permette di tenere traccia delle entrate e delle uscite.

Un ulteriore possibile utilizzo dei QR code è per supportare la configurazione di reti di comunicazione wireless [8, 9] e cablate [10, 11], specialmente quelle industriali [12, 13], che a causa della loro eterogeneità sia come tipologia di protocollo di comunicazione (IEEE 802.11 [14], IEEE 802.15.4 [15], etc.) sia come servizi (ad esempio, sincronizzazione [16, 17]) richiedono complesse operazioni di configurazione.

Oltre a tutti questi campi di utilizzo, che si sono diffusi solamente in seguito alla grande diffusione della tecnologia dei QR code, essi sono ancora usati, soprattutto in campo industriale, per il tracciamento [18] e la gestione degli inventari ed anche delle consegne. In questi casi i QR code identificano i prodotti oppure i pacchi, e ne possono anche segnalare la posizione esatta e lo stato.

Si può quindi capire come i QR code siano una tecnologia estremamente versatile, che si presta in modo ottimale a molte applicazioni in settori molto vari. Soprattutto per questo, essi sono il fulcro di molte ricerche recenti, principalmente riguardanti il campo della sicurezza [19, 20]. Con la sempre crescente popolarità dei dispositivi mobili, è molto probabile che anche il loro uso dei continui ad aumentare nei prossimi anni, così come la vastità dei loro campi di utilizzo.

2.2 Standard

Lo standard ISO/IEC 18004, pubblicato per la prima volta nel 2000, si occupa, in generale, di simboli grafici da utilizzare nei disegni tecnici delle tecnologie dell'informazione e delle telecomunicazioni e delle tecniche per la loro identificazione

automatica e l'acquisizione dei dati in loro contenuti.
Nel corso degli anni lo standard è stato aggiornato un paio di volte e si è sempre più concentrato sui QR code. Infatti, l'ultima versione dello standard, al momento della stesura di questa tesi (standard ISO/IEC 18004:2015 [21]), definisce i requisiti per la simbologia dei QR code, ne specifica le caratteristiche, i metodi di codifica dei caratteri dei dati, i formati dei simboli, le caratteristiche dimensionali, le regole di correzione degli errori, l'algoritmo di decodifica dei riferimenti, i requisiti di qualità della produzione e i parametri dell'applicazione selezionabili dall'utente.

2.2.1 Codifica dei dati

Un QR code può codificare un insieme ben definito di caratteri al suo interno:

- *dati numerici*: tutte le cifre, fino ad un massimo di 7089 caratteri.
- *dati alfanumerici*: tutte le cifre, le lettere e nove altri caratteri speciali (spazio, $, %, *, +, -, ., /, :) fino ad un massimo di 4296 caratteri.
- *byte data*: dati di tipo binario, con un massimo di 2953 bytes.
- *caratteri Kanji*: caratteri giapponesi che vengono compattati in stream da 13 bit (fino ad un massimo, quindi, di 1817 caratteri).

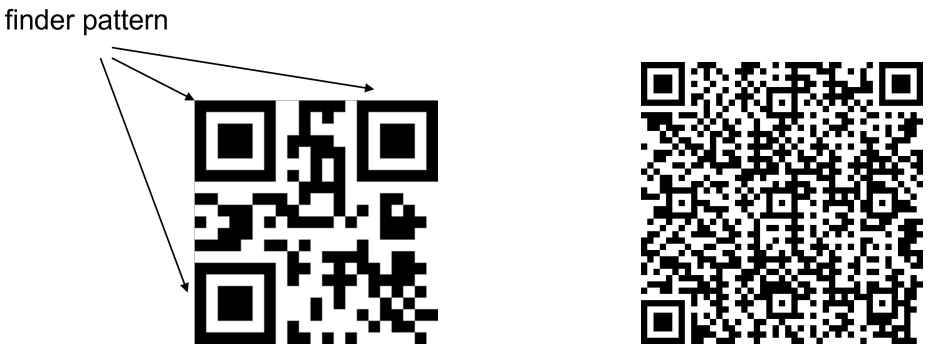
Tutti questi diversi tipi di dati vengono codificati in stream binari ed inseriti all'interno del QR code. Lo standard definisce un modulo come nero se corrispondente ad un bit 1, mentre definisce un modulo come bianco se corrispondente ad un bit 0. Tuttavia, è anche previsto dallo standard il caso, denominato *reflectance*, in cui i colori corrispondenti ad ogni bit siano invertiti.

2.2.2 Versioni

I QR code sono simboli matriciali che consistono in array di moduli quadrati disposti in uno schema quadrato generale, che include un pattern unico, detto *finder*, posto in tre dei quattro angoli del simbolo (di norma i due angoli in alto e l'angolo in basso a sinistra). Questo *finder pattern* è pensato per semplificare la localizzazione del QR code, ed il riconoscimento della sua dimensione e di una eventuale inclinazione.

All'interno dello standard sono state definite 40 diverse dimensioni di QR code, denominate Versioni e numerate dalla versione 1 alla versione 40. La Versione 1 ha una dimensione di 21 moduli x 21 moduli; ad ogni aumento di versione, la dimensione è aumentata di 4 moduli x 4 moduli, fino ad arrivare alla dimensione di 177 moduli x 177 moduli della Versione 40.

Inoltre, le versioni dalla numero 2 in poi contengono un altro tipo di pattern, denominato *alignment pattern*, simile al finder pattern, usato per il riconoscimento



(a) QR code Version 1

(b) QR code Version 7

Figura 2.1: Esempi di versioni di QR code

della versione in seguito alla scansione del QR code. Un esempio di *alignment pattern* può essere trovato in Figura 2.2

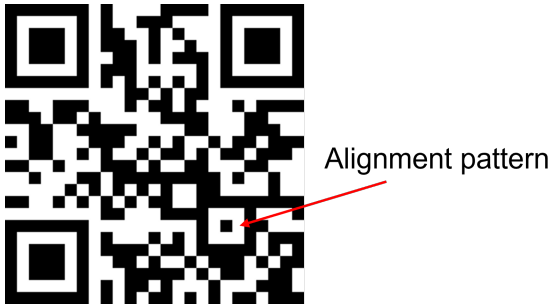


Figura 2.2: QR code Version 2

2.2.3 Livelli di correzione

Non è sempre possibile mantenere qualitativamente inalterato lo stato fisico di QR code, in quanto essi vengono spesso lasciati esposti pubblicamente, dove è possibile che vengano danneggiati in molti modi, o più semplicemente che si consumano. Tuttavia, molto spesso, nonostante un QR code appaia sporco, danneggiato oppure parte di esso sia ostruito da altri oggetti durante la scansione, quest'ultima va a buon fine e la lettura del codice avviene in modo corretto. Ciò è dovuto alla funzione di correzione degli errori dei QR code, che consente il ripristino dei dati anche quando essi siano stati in qualche modo danneggiati.

La correzione degli errori avviene mediante l'implementazione del Codice Reed-Solomon: durante il processo di codifica dei dati, i dati da codificare vengono suddivisi in blocchi di codewords. Per ciascuna di queste codewords, vengono calcolate le necessarie e corrispondenti codewords per la correzione degli errori. Questi nuovi blocchi di informazioni di correzione degli errori vengono calcolati in modo tale da garantire che ogni possibile errore di lettura o di decodifica possa essere corretto tramite il ricalcolo dei dati originali e, infine, vengono memorizzate insieme ad i dati nel QR code, aumentandone la densità.

Lo standard offre quattro diversi livelli di correzione, indicati in tabella 2.1, che permettono di recuperare la corrispondente quantità di danni:

Livello di correzione degli errori	**Capacità di recupero in %**
L	7
M	15
Q	25
H	30

Tabella 2.1: Livelli di correzione degli errori

Aumentare il livello significa migliorare la correzione degli errori.

Le codewords per la correzione degli errori sono in grado di correggere due tipi di codewords errate: le cancellature (codewords errate in luoghi noti) e gli errori (codewords errate in luoghi non noti). Una cancellatura rappresenta un simbolo o un carattere che non è stato possibile scannerizzare o decodificare, mentre un errore è un simbolo che è stato decodificato in modo errato.

Oltre a fare in modo che il QR code venga scannerizzato correttamente, la correzione degli errori permette anche variazioni fantasiose dei codici stessi. Infatti, è grazie a questa caratteristica che è possibile aggiungere dei loghi all'interno dei QR code: per aggiungere il logo, infatti, è necessario rimuovere una porzione di dati per fargli spazio.

Alcuni fattori devono essere tenuti in considerazione nella scelta del livello di correzione degli errori che viene applicato ad un QR code: è infatti importante notare che, sebbene esso fornisca una crescente resistenza ai danni, più il livello scelto è alto, più sono i dati che devono essere contenuti nel QR code.

A causa di queste considerazioni, è consigliato usare:

- i livelli L ed M in caso sia necessario mantenere il QR code il meno denso possibile (come, ad esempio, nel caso in cui lo stesso debba essere stampato su un'area piccola) oppure nel caso in cui sia necessario contenere la quantità di dati da inserire al suo interno (come ad esempio quando i dati da inserire abbiano già una grandezza prossima a quella limite per la versione 40)

- i livelli Q ed H nel caso in cui si prevede che i codici saranno soggetti a molti danni ed imperfezioni. Un tipico esempio di ciò è il contesto industriale, in cui è altamente probabile che il QR code si sporchi frequentemente.

Capitolo 3

Compilatori

3.1 Introduzione

Si dice compilatore un programma che traduce una porzione di codice scritta in un particolare linguaggio di programmazione (il cosiddetto *linguaggio sorgente*) in un altro linguaggio di programmazione (il cosiddetto *linguaggio target*). Il nome *compilatore* è principalmente usato per denotare quei programmi che traducono codice da linguaggi di alto livello a basso livello (ad esempio, C -> Assembly). La struttura di un compilatore può essere più o meno complessa, in relazione alle funzioni necessarie durante il suo utilizzo. Essa può comunque essere semplificata come in Figura 3.1.

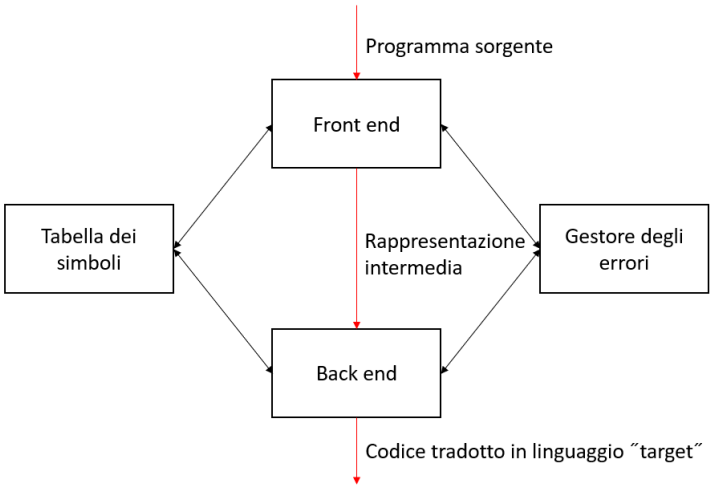


Figura 3.1: Struttura base di un compilatore

A loro volta, le fasi relativi ai moduli chiamati *front end* e *back end* possono essere suddivise in varie sottofasi, che rappresentano le operazioni atomiche che il compilatore può svolgere.

La struttura del front end è rappresentata in Figura 3.2.

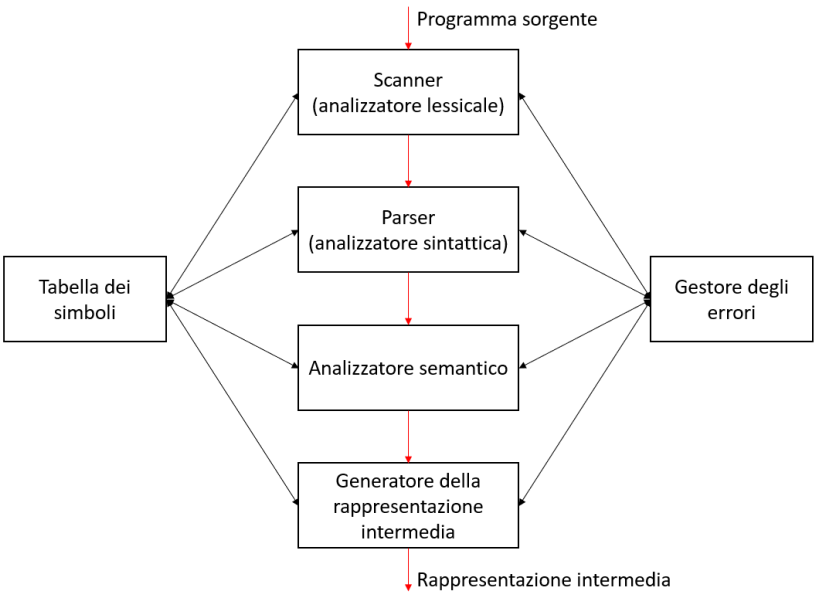


Figura 3.2: Fase front end

In questa fase, il compilatore esegue tutte o alcune fra le seguenti operazioni:

1. *Analisi lessicale*: il compilatore legge il codice sorgente e lo converte in un insieme di stream fondamentali, detti *token*. Questi simboli devono essere definiti dal programmatore del compilatore all'interno di un suo componente speciale, denominato *Scanner* e rappresentano le parti più piccole e fondamentali del linguaggio. Tipici esempi di token sono le *keyword* di un linguaggio, gli *identificatori*, gli *operatori* e tutti *i simboli di punteggiatura*.

2. *Analisi sintattica o Parsing*: il compilatore controlla lo stream di simboli generato dal passo precedente e si assicura che esso segua le regole del linguaggio sorgente, cioè le cosiddette *regole grammaticali*. Questo insieme di regole viene definito dal programmatore del compilatore all'interno di un suo componente speciale, denominato *Parser*. Grazie a questo insieme di regole, il compilatore è in grado di generare un *albero strutturale* (AST), che rappresenta la struttura del programma in un grafo ad albero.

3. *Analisi semantica*: il compilatore, durante questa fase, si assicura che il codice processato abbia effettivamente un significato. Esso si occupa di trovare errori di natura logica, come ad esempio l'uso di una variabile non dichiarata.

4. *Generazione del codice*: in questo ultimo passo del front end, il compilatore usa l'AST generato durante la fase di Parsing per tradurre il codice sorgente. Anche questa operazione viene solitamente effettuata all'interno del Parser, tramite la definizione di operazioni da svolgere (cioè la stampa della porzione di codice di codice tradotta), corrispondenti a ciascuna delle regole definite durante la fase di Parsing.

Generalmente, alla fine del front end, l'output è una rappresentazione intermedia del codice sorgente.

La struttura del back end è invece rappresentata in Figura 3.3.

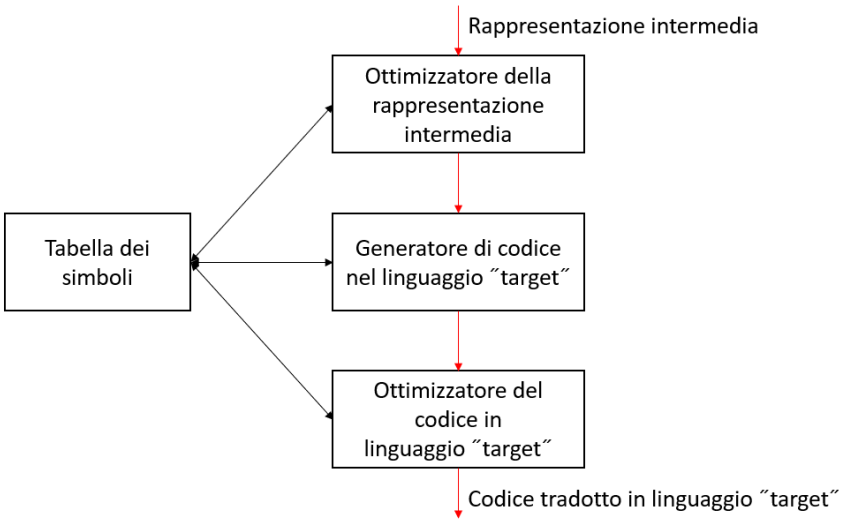


Figura 3.3: Fase back end

In questa fase, il compilatore esegue tutte o alcune fra le seguenti operazioni:

1. *Ottimizzazione della rappresentazione intermedia*

2. *Traduzione della rappresentazione intermedia nel linguaggio target*, usando tecniche simili a quelle definite nella fase front end

3. *Ottimizzazione del codice generato al punto precedente*

Il fatto che la struttura di un compilatore sia così modulare è molto importante per gli sviluppatori, in quanto, per malfunzionamenti dovuti ad errori interni al

compilatore, è molto più facile riuscire a risalire alla causa. Questa modularità, quindi, permette di controllare in modo efficiente gli output di ogni fase e di risalire ad eventuali errori in modo più semplice.
Tuttavia, è necessario ed importante notare come i compilatori non siano gli unici strumenti usati per eseguire operazioni di comprensione e trasformazione di un linguaggio. Gli *interpreti*, a differenza dei compilatori, eseguono il codice sorgente direttamente anziché tradurlo nel linguaggio target. Pertanto, il codice viene eseguito mentre viene letto, senza la necessità di essere tradotto in precedenza. In questo modo, l'interprete è in grado di individuare errori sintattici o logici nel codice sorgente in tempo reale, mentre il compilatore richiede una fase aggiuntiva di compilazione prima dell'esecuzione. L'interprete, tuttavia, può essere più lento del compilatore, poiché il codice viene eseguito durante la sua esecuzione. Gli interpreti vengono spesso utilizzati in situazioni in cui la portabilità e la rapidità di sviluppo sono prioritarie rispetto alle prestazioni.

3.2 Compilatori in Python: PLY

PLY realizza in Python gli strumenti di costruzione di compilatori *Lex* e *YACC*, già preesistenti per il C.

PLY consiste in due moduli separati: lex.py e yacc.py. Il modulo lex è usato per scomporre il testo di input in una collezione di token, i quali corrispondono a espressioni regolari. Il modulo yacc è invece usato per il riconoscimento della sintassi del linguaggi, specificata nella forma di una *Context Free Grammar (CFG)*.

I due moduli lavorano insieme. Nello specifico, lex fornisce l'interfaccia per la produzione dei token che verranno ricevuti da yacc ed usati per il controllo delle regole grammaticali. L'output del modulo yacc è un AST. Tuttavia, è anche possibile usare yacc per l'implementazione di *compilatori one-pass*. I *compilatori one-pass* sono un tipo di compilatore che cerca di eseguire tutte le fasi di analisi del codice in un solo passaggio. Questo tipo di compilatore è generalmente utilizzato per compilare programmi che richiedono una rilevazione di errori molto semplice. Essi sono veloci e leggeri, il che li rende adatti per l'utilizzo su sistemi con risorse limitate.

Così come i suoi equivalenti per altri linguaggi, il modulo yacc di Python fornisce tutti gli strumenti principali per la parte di Parsing, quali validazione della grammatica, controllo degli errori, supporto di regole vuote e per token di errore, e la risoluzione di eventuali ambiguità nella grammatica attraverso la definizione di regole di precedenza. Un'*ambiguità sintattica* si verifica quando una grammatica, detta *ambigua* può generare due o più *AST*, ovvero due o più possibili strutture sintattiche valide, a partire dalla stessa stringa di input. Ciò significa che il compilatore non è in grado di riconoscere inequivocabilmente quale sia la struttura sintattica corretta, portando a comportamenti imprevedibili.

3.2.1 LEX

Come specificato anche in precedenza, il modulo LEX consiste nella definizione di una serie di corrispondenze, dette *token*, che associano un particolare simbolo o un'espressione regolare ad un nome, che verrà usato dal modulo yacc per il riconoscimento della grammatica legata al programma di input.

Un esempio di un semplice modulo lex è riportato nella seguente porzione di codice.

```
1  import ply.lex as lex
2
3  tokens = (
4      'NUMBER',
5      'PLUS', 'MINUS', 'STAR', 'DIVIDE',
6      'RO','RC',
7  )
8
9  t_PLUS    = r'\+'
10 t_MINUS   = r'-'
11 t_STAR    = r'\*'
12 t_DIVIDE  = r'/'
13 t_RO      = r'\('
14 t_RC      = r'\)'
15
16 def t_NUMBER(t):
17     r'\d+'
18     t.value = int(t.value)
19     return t
20
21 def t_newline(t):
22     r'\n+'
23     t.lexer.lineno += 1
24
25 def t_error(t):
26     print("Illegal character '%s'" % t.value[0])
27     t.lexer.skip(1)
```

Listing 3.1: Esempio di modulo lex

Come facilmente intuibile, lo Scanner precedente è un semplice modulo che accetta come input numeri interi, come token denominati `NUMBER`, i principali operatori matematici (i.e., +, -, *, /) e le parantesi tonde. Inoltre, questo modulo, usa il simbolo *newline* per contare il numero di righe di cui il testo in input è composto, come si può vedere dalla definizione del token relativo. L'ultimo token definito, denominato token di errore, è un particolare tipo di token che dovrebbe essere sempre presente all'interno di uno Scanner e che permette semplicemente di ignorare caratteri non specificati in nessun altro token.

La particolarità di questo modulo è rappresentata dalla lista di token definita in linea (3). Essa è necessaria in quanto contiene tutti i token che si desidera passare

al modulo yacc, cioè al Parser. Tenere aggiornata questa lista rispetto alle necessità di cui si ha bisogno è molto importante in quanto il modulo yacc riconoscerà un token che gli viene passato in input come valido se e solo se esso è presente in questa lista.

3.2.2 YACC

Il modulo YACC consiste nella definizione di una serie di regole, dette *regole di Parsing* che, abbinate a delle azioni a loro correlate e, se necessarie, ad alcune regole di precedenza, permettono di interpretare e tradurre correttamente l'output dello Scanner, producendo come output la traduzione richiesta. Un esempio di un semplice modulo yacc, corrispondente al modulo lex riportato nella porzione di codice 3.1 è presentato di seguito.

```
1  import ply.yacc as yacc
2
3  # Get the token map from the lexer.  This is required.
4  from calclex import tokens
5
6  def p_expression_plus(p):
7      'expression : expression PLUS term'
8      p[0] = p[1] + p[3]
9
10 def p_expression_minus(p):
11     'expression : expression MINUS term'
12     p[0] = p[1] - p[3]
13
14 def p_expression_term(p):
15     'expression : term'
16     p[0] = p[1]
17
18 def p_term_times(p):
19     'term : term TIMES factor'
20     p[0] = p[1] * p[3]
21
22 def p_term_div(p):
23     'term : term DIVIDE factor'
24     p[0] = p[1] / p[3]
25
26 def p_term_factor(p):
27     'term : factor'
28     p[0] = p[1]
29
30 def p_factor_num(p):
31     'factor : NUMBER'
32     p[0] = p[1]
33
34 def p_factor_expr(p):
35     'factor : LPAREN expression RPAREN'
```

```
36      p[0] = p[2]
37
38  # Error rule for syntax errors
39  def p_error(p):
40      print("Syntax error in input!")
```

Listing 3.2: Esempio di modulo yacc

Questo Parser definisce le regole di una grammatica con la quale è possibile definire catene di operazioni fondamentali (i.e., +, -, *, /) fra numeri interi, con il possibile utilizzo di parentesi. Così come per il modulo lex, anche il modulo yacc si conclude con una generica regola di errore, che viene usata per matchare tutte quelle sequenze di token che non rispettano le regole della grammatica definita. Come accennato in precedenza in Sezione 3.2.1, si può notare come, alla riga (4), sia necessario che il Parser riceva dallo Scanner la lista di token riconosciuti dalla grammatica, così che essi possano essere usati come base per la definizione ed il riconoscimento delle sequenze che formano la grammatica.

Regole di Parsing

Come si può vedere dalla porzione di codice 3.2, ogni regola grammaticale è definita come una funzione Python, che contiene una stringa rappresentante la sequenza di simboli necessaria per produrre il risultato a cui si fa riferimento. Per esempio, possiamo notare come un'espressione possa derivare da molte combinazioni diverse di simboli, la prima delle quali è formata da un'altra espressione (la regola `expression` verrà visitata ricorsivamente dal Parser) seguita da un token `PLUS` e dalla regola `term`, che a sua volta può essere rappresentata da diverse sequenze di simboli.

Ad ogni regola grammaticale possono inoltre essere associate alcune azioni, quali ad esempio stampe di valori o assegnazioni a seguito di un qualche calcolo, che vengono eseguite al momento della riduzione della regola stessa. Queste azioni fanno spesso riferimento ad un elemento che ogni funzione accetta come argomento, chiamato `p`, che è un array che rappresenta la sequenza dei valori che ogni simbolo specificato nella regola assume. I valori assunti da ogni `p[i]` sono mappati nei simboli della regola come segue.

```
1  def p_expression_plus(p):
2      'expression : expression PLUS term'
3      #   ^            ^        ^    ^
4      #  p[0]          p[1]     p[2] p[3]
```

Listing 3.3: Mapping del parametro p

Per i token, il valore di `p[i]` è quello corrispondente definito nel modulo Lex. Mentre per i valori cosiddetti *non terminali* (i.e., tutto ciò che non è un token, ma è definito da un'altra regola grammaticale), il valore assunto all'interno della funzione è determinato dal processo di Parsing avvenuto fino al momento della riduzione

della regola specifica; esso, infatti, corrisponde al valore inserito all'interno di `p[0]` durante la riduzione della regola corrispondente. Prendiamo, ad esempio, la regola sulla riga (26), che definisce un `term` come un `factor`; la regola successiva, sulla riga (30), definisce un `factor` come un semplice token `NUMBER`, definito nel modulo lex. Quando il Parser trova un token `NUMBER`, quest'ultima regola dovrà essere ridotta e, all'interno di `p[0]` viene inserito `p[1]`, ovvero il valore del `factor` diventa lo stesso che è stato ritornato dallo Scanner nel riconoscimento del numero. Di conseguenza, il prossimo passo del Parser sarà la riduzione della regola `term : factor`. Il valore interno a `p[1]`, che verrà inserito in `p[0]`, sarà uguale a quello derivante dalla riduzione della regola precedente.

È importante, inoltre, notare come sia possibile l'uso di indici negativi all'interno del parametro `p`: essi vengono usati quando, all'interno di una regola, è necessario avere il valore di un simbolo che non fa parte della regola, ma che, ad esempio, è quello precedente all'interno dell'AST. In questo caso, si può ottenere quel valore richiamando `p[-1]`.

La prima regola definita nel modulo yacc determina il simbolo di inizio della grammatica, che rappresenta la generalizzazione più ampia della grammatica. Pertanto, è molto comune che la prima regola definita all'interno di un modulo yacc abbia una forma simile a quella riportata nella porzione di codice seguente.

```
1 def p_prog(self, p):
2     'prog : function_list | '
3     p[0] = p[1]
```

Listing 3.4: Esempio di prima regola di una grammatica

La regola precedente presenta una definizione molto particolare: un `prog`, infatti, oltre a poter corrispondere ad una `function_list`, può anche essere rappresentata attraverso la regola vuota ('| ', oppure '| `empty`'). Questo perché, solitamente, un programma è una lista di elementi ripetuti 0 o più volte. Il valore contenuto in `p[0]` dopo la riduzione di questa regola rappresenta l'output del Parser.

Regole di precedenza

La grammatica presentata nella porzione di codice 3.2 è stata scritta in modo tale che non presentasse ambiguità. Tuttavia, il formato non è molto leggibile o immediatamente comprensibile. Sarebbe infatti molto più comprensibile e compatto rappresentare un'espressione come segue.

```
1 def p_expression(p):
2     'expression : expression PLUS expression
3             | expression MINUS expression
4             | expression TIMES expression
5             | expression DIVIDE expression
6             | LPAREN expression RPAREN
7             | NUMBER
```

```
8       '
```

Listing 3.5: Esempio di regole che presenta ambiguità

Tuttavia, definire una regola del genere creerebbe delle ambiguità all'interno della grammatica. Il Parser, infatti non ha alcuno strumento per sapere come i vari operatori debbano essere raggruppati. Per esempio, una sequenza di token del tipo `5 + 3 * 2`, senza alcuna indicazione, può essere vista come `(5 + 3) * 2` oppure come `5 + (3 * 2)`. È necessario, quindi, trovare un modo per indirizzare il Parser alla rappresentazione giusta. Per farlo, è sufficiente fornire al Parser una serie di regole di precedenza per disambiguare la grammatica.

Ciò può essere fatto definendo, all'interno del modulo, una variabile `precedence`, contentente la lista dei token in ordine **descrescente** di precedenza.

```
precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
)
```

Listing 3.6: Esempio di regole di precedenza

Questo semplice accorgimento fornisce al Parser due informazioni: le operazioni di prodotto e divisione sono prioritarie rispetto all'addizione e alla sottrazione e, a parità di priorità, le operazioni trovate devono essere eseguite (e le loro regole ridotte) da sinistra verso destra (*'left'*).

Un caso particolare riscontrabile quando si lavora con le precedenze è quello dei numeri negativi: infatti, il simbolo `MINUS` è lo stesso di quello usato per la sottrazione. Tuttavia, esso ha una maggiore priorità anche rispetto ai simboli di prodotto e divisione. Per convenzione, in questi casi, si definisce il tipo di precedenza detto `UMINUS`, che solitamente ha il livello più alto in assoluto e si aggiunge, a tutte le regole che ne hanno bisogno, il suffisso `%prec UMINUS`, che indica al Parser che quella regola ha precedenza maggiore rispetto a tutte le altre regole.

```
[...]

precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),            # Unary minus operator
)

[...]

def p_expr_uminus(p):
    'expression : MINUS expression %prec UMINUS'
    p[0] = -p[2]
```

Listing 3.7: Esempio di uso della precedenza di tipo UMINUS

Ad esempio, una regola grammaticale come quella definita nella porzione di codice precedente avrà il livello di precedenza più alto possibile e sarà analizzata dal Parser, se in conflitto con altre regole, per prima.

Capitolo 4

QRscript

4.1 Introduzione

Nonostante i QR code siano, ad oggi, principalmente usati per contenere semplici dati testuali, quali semplici informazioni o URL per l'accesso a siti web, il modo in cui essi sono definiti permette, potenzialmente, di inserire al loro interno uno qualsiasi dei formati elencati in Sezione 2.2.1. È quindi anche possibile inserire al loro interno interi programmi eseguibili, purché essi siano correttamente codificati. Tuttavia, è altresì necessario, data la limitata dimensione di dati che ogni QR code può contenere, fare in modo che il formato che viene inserito all'interno del codice stesso sia il più compatto possibile, in modo che venga inserito, all'interno di un singolo QR code, il maggior numero di dati possibili.

QRscript si occupa della definizione di una serie di regole che permettono di tradurre un programma eseguibile in un formato binario compatto destinato ad essere inserito all'interno di un *eQR code* (Executable QR code); un eQR code è un qualsiasi QR code contenente del codice eseguibile che è stato generato conformemente alle regole di QRscript.

La catena di funzionamento della tecnologia QRscript può essere riassunta nello schema presentato in Figura 4.1, che divide l'intero processo in due fasi principali: la fase di generazione, che porta, a partire dal programma eseguibile scritto in linguaggio di alto livello, alla generazione di un eQR code; e la fase di esecuzione, che consiste nella scansione dell'eQR code e nell'esecuzione del programma contenuto al suo interno attraverso un interprete della rappresentazione intermedia.

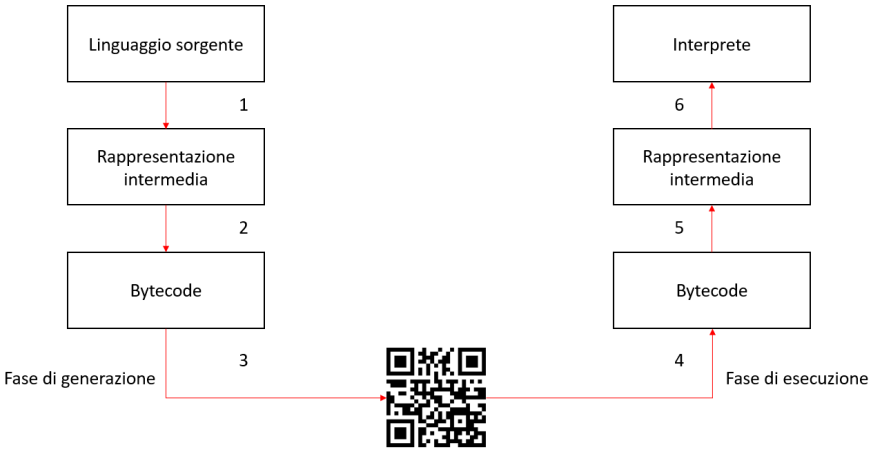


Figura 4.1: Processo di funzionamento di QRScript

La fase di generazione è a sua volta composta da tre passaggi principali:

1. Un *linguaggio di alto livello* viene tradotto in una *rappresentazione intermedia* attraverso un compilatore. Ciò fa in modo che il passaggio successivo, a parità di rappresentazione intermedia, sia un passaggio standard ed indipendente dal linguaggio di alto livello usato; quindi, per aggiungere il supporto ad un nuovo linguaggio di alto livello, l'unica cosa necessaria è adattare questo passaggio al nuovo linguaggio.

2. La rappresentazione intermedia è a sua volta processata da un altro compilatore che la trasforma in un formato binario, denominato *QRbytecode*. Questa è la porzione di dati che verrà inserita all'interno dell'eQR code.

3. Il QRbytecode viene trasformato in un *eQR code*.

Anche la fase di esecuzione è composta da tre passaggi principali:

4. L'*eQR code* viene scannerizzato, ottenendo quindi i dati al suo interno, che sono rappresentati dal QRbytecode generato nella fase di generazione.

5. Il QRbytecode viene tradotto nella rappresentazione intermedia. Questo passaggio fa uso di un compilatore che svolge un ruolo sostanzialmente opposto a quello introdotto al punto 2 della fase di generazione.

6. La rappresentazione intermedia ottenuta al punto precedente viene eseguita, tramite una *virtual machine* o un *interprete*.

Siccome il linguaggio di alto livello che viene utilizzato per la generazione di un eQR code non può essere conosciuto a priori, QRscript si occupa principalmente della definizione delle specifiche riguardanti il punto 2 della fase di generazione ed il punto 5 della fase di esecuzione. In questa ottica, risulta quindi fondamentale la definizione del formato denominato *QRbytecode*, che verrà dettagliata in Sezione 4.2. Tuttavia, il lavoro svolto in questa tesi non si limita ai soli punti sottolineati in precedenza e si concentra invece nell'intero processo di funzionamento di QRscript: parte del lavoro, infatti, è consistito nella definizione di due rappresentazioni intermedie distinte, denominate *QRprog* e *QRtree*, che permettesse di fornire un esempio della flessibilità di QRscript. Infatti, permettendo che QRscript supporti varie rappresentazioni intermedie, anche definibili dall'utente, si fa in modo che, per ogni caso di applicazione e per ogni diverso linguaggio di alto livello utilizzato si possa trovare e definire la più efficiente rappresentazione intermedia possibile, aumentando di conseguenza anche l'efficienza dell'eQR code.

Queste varie rappresentazioni intermedie sono state denominate *dialetti* e saranno approfondite nei Capitoli 5 e 6.

4.2 QRbytecode

QRbytecode è la rappresentazione binaria fondamentale per il corretto funzionamento di QRscript. Essa, infatti, fa in modo che il programma scritto in linguaggio di alto livello sia inserito all'interno di un eQR code in un modo molto compatto ed inoltre permette la generazione di quest'ultimo in modo tale che esso possa essere letto e decodificato correttamente durante la fase di esecuzione di QRscript.

Il QRbytecode è formato da due parti principali:

1. Una parte iniziale, detta *QRscript header*, il quale contiene informazioni generali per la corretta decodifica dell'eQR code.

2. Il contenuto vero e proprio dell'eQR code, cioè tutti quei dati binari che sono stati generati durante la fase di generazione; alcuni sono dipendenti dal funzionamento richiesto all'eQR code e dal dialetto adottato, mentre la parte più cospicua è quella ottenuta traducendo la rappresentazione intermedia ottenuta partendo dal programma scritto in linguaggio di alto livello. Questa parte di *QRbytecode* è denominata *Code*.

Mentre il punto 2 verrà trattato in modo approfondito nei capitoli riguardanti i *dialetti*, la Sezione (Sezione 4.2.1) seguente si occuperà di approfondire il contenuto delle parti principali che formano l'header del QRbytecode.

4.2.1 QRbytecode header

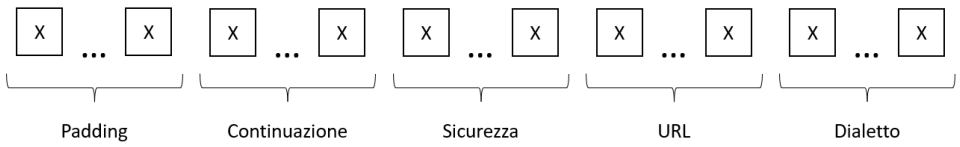


Figura 4.2: Struttura del QRbytecode header

La struttura dell'header del QRbytecode è sintetizzata in Figura 4.2, la quale mostra le cinque parti principali che lo compongono:

1. *Padding*, che serve per allineare il QRbytecode da scrivere nell'eQR code, nel caso la codifica selezionata sia quella binaria e il numero di bit di cui è composta non è un multiplo di 8.

2. *Continuazione*, fondamentale in quanto, date le ridotte dimensioni di un QR code rispetto ad un possibile programma che si voglia inserire al suo interno, è stato deciso di estendere QRscript in modo che fosse possibile avere il programma diviso in più QR code, che devono quindi essere scannerizzati in gruppo in modo tale che sia possibile ricostruire l'intero QRbytecode da trasformare nella rappresentazione intermedia eseguibile tramite virtual machine o interprete. Questa parte dell'header è opzionale, per cui un'implementazione conforme alle regole dettate da QRscript può anche non implementarla.

3. *Sicurezza*: è la parte dell'header che si occupa della gestione dell'autenticità, dell'integrità ed eventualmente anche della crittografia del contenuto dell'eQR code. Questa parte dell'header è opzionale, per cui un'implementazione conforme alle regole dettate da QRscript può anche non implementarla.

4. *URL*: questa parte è usata per permettere la connessione ad un URL esterno, in modo che ciò a cui esso corrisponde venga eseguito al posto del programma contenuto all'interno dell'eQR code. Ciò permette di eseguire un'applicazione con più contenuti (e anche più interattivi, come video o immagini), rispetto a quelli che è possibile inserire all'interno di un eQR code. Questa parte dell'header è opzionale, per cui un'implementazione conforme alle regole dettate da QRscript può anche non implementarla.

5. *Dialetto*: è la sezione che identifica il dialetto usato durante la fase di generazione ed è quindi fondamentale per il corretto funzionamento della fase di esecuzione. Un'implementazione conforme alle regole dettate da QRscript deve implementare il supporto a questa parte di *QRscript header*.

Padding

Il QRbytecode header inizia con una serie di bit che indicano se è presente o meno del padding, richiesto solamente nel caso in cui la modalità di input scelta fra quelle disponibili sia quella binaria e la lunghezza del QRbytecode non sia divisibile per 8. Ciò in quanto questo particolare tipo di codifica richiede uno stream di dati che possano essere divisi in gruppi di byte. A differenza di quanto accade nei protocolli di comunicazione, in questo contesto, il padding è stato inserito nell'header (e non nella coda del QRbytecode) perché non è una cosa dipendente dal dialetto codificato all'interno dell'eQR code. In particolare, il QRbytecode inizia con un bit settato ad `1` in caso non sia necessario ulteriore padding, `01` nel caso di un bit di padding, `001` in caso di 2 bit e così via. Nel caso in cui venga scelta la codifica di tipo numerico, invece, la quantità di bit di padding inserita non influenzerà la conversione in quanto i bit settati a 0 prima del primo 1 non saranno considerati significativi. Vari esempi di padding sono mostrati in Figura 4.3.

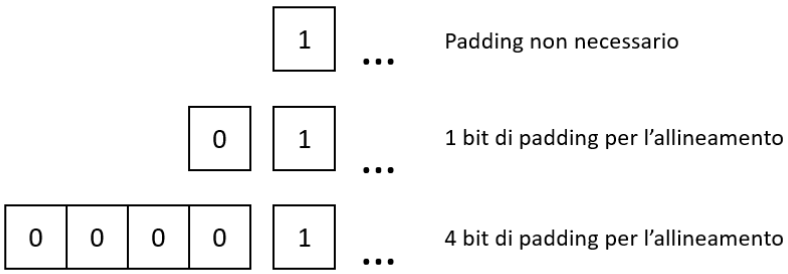


Figura 4.3: Esempi di padding

Continuazione

Nonostante i QR code debbano la loro fama e diffusione al fatto che è possibile inserire al loro interno una quantità di dati di molto maggiore rispetto ad i loro predecessori (quali ad esempio i barcode), la loro dimensione rimane comunque piuttosto limitata per gli standard dei giorni nostri. Per esempio, la versione più grande di QR code, a cui viene aggiunta la minor quantità possibile di dati accessori per la correzione degli errori (Versione 40 con livello "L" di correzione), che è quella che in assoluto può contenere la maggior quantità di dati, può contenere appena 2953 bytes. Al fine di superare questa limitazione, QRscript offre la possibilità di concatenare varie porzioni di codice binario ottenuto dalla scansione individuale e sequenziale di più eQR code (chiamati *frammenti*) in un unico QRbytecode. Questa funzionalità è stata denominata *continuazione*. Essa è abilitata

se il primo bit contenuto nel QRbytecode è settato ad 1. Questo bit è poi seguito dal numero progressivo di sequenza assegnato allo specifico eQR code e dal numero totale di frammenti che compongono l'intero programma di partenza (usando una numerazione che parte dallo zero; quindi, ad esempio, se dovesse essere abilitata la continuazione per un programma composto da un solo frammento, sia il numero di sequenza assegnato al frammento che il numero totale di frammenti sarebbe pari a 0).

Inoltre, sia il numero di sequenza che il numero totale di frammenti sono codificati in 4 bit, ma sono estendibili seguendo la notazione esponenziale definita in Appendice A.

Durante la lettura degli eQR code e la conseguente ricostruzione del QRbytecode, l'applicazione che si occupa della lettura userà il numero di sequenza per concatenare in ordine corretto i dati binari ottenuti. Il numero totale di frammenti che compongono il programma, invece, può essere usato per comunicare all'utente quanti e quali siano gli eQR code che devono essere ancora scannerizzati per ricostruire correttamente il programma di partenza.

Un'implementazione di QRscript che si occupa di programmi composti da un solo frammento può gestire la continuazione in due modi diversi:

1. Non implementandola, settando quindi semplicemente il primo bit a 0

2. Implementandola come continuazione composta da un solo frammento con numero di sequenza 0; in questo caso i primi bit che formano il QRbytecode saranno 100000000.

Alcuni esempi di codifica di casi particolari di uso della continuazione sono riportati in Figura 4.4.

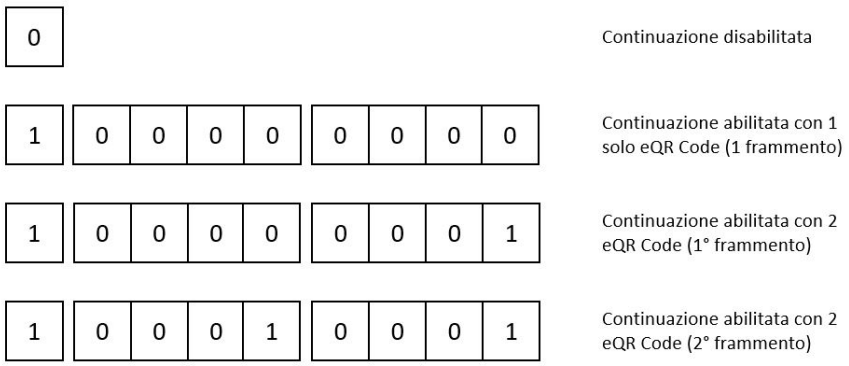


Figura 4.4: Esempi di possibili continuazioni comuni

Sicurezza

La parte dell'header relativa alla sicurezza si occupa della gestione dell'autenticità, dell'integrità e della crittografia ed è opzionale; quindi, un'implementazione conforme a QRscript non è obbligata a supportarla. In seguito ad i bit relativi alla continuazione, si trovano i 4 bit relativi alla sicurezza. Anche questi 4 bit possono però essere estesi seguendo la notazione esponenziale definita in Appendice A. La sequenza 0000 rappresenta l'assenza di uso di meccanismi di sicurezza, mentre le altre possibili sequenze rappresentano possibili profili di sicurezza.
Un profilo indica:

1. Quali meccanismi di sicurezza sono attivi per lo specifico eQR code.

2. Quali parti dell'eQR code sono soggette ai meccanismi di sicurezza attivi: infatti, per quanto possa sembrare molto utile proteggere l'intero eQR code attraverso tutti i 3 meccanismi citati in precedenza, dal punto di vista dell'efficienza della fase di lettura potrebbe essere ottimale lasciare in chiaro la parte relativa alla continuazione. Questo perché, in questo modo, l'applicazione che si occupa della lettura sarebbe in grado di identificare il numero di sequenza di ogni eQR code, permettendo così la corretta concatenazione dei frammenti, senza il bisogno di decriptarli uno per uno.

3. L'algoritmo usato per la gestione della sicurezza (e.g., RSA).

4. La lunghezza della firma digitale, che può essere opzionalmente usata per la gestione dell'autenticità e dell'integrità, in bit. Questo dato è molto importante per il corretto riconoscimento dei bit successivi come quelli della firma digitale, permettendo così un corretto allineamento con la parte successiva del QRbytecode.

Nel caso in cui sia presente la parte di sicurezza, lo sviluppatore dovrà definire le coppie chiavi-valori che rappresentano una mappa fra i bit che si trovano in questa sezione dell'header (chiavi) ed i profili definiti per l'applicazione con i loro dettagli (valori).

Questa parte dell'header può essere seguita dalla firma digitale oppure, in sua assenza, direttamente dalla parte dedicata all'URL.

Alcuni esempi di codifica di casi particolari di uso della sicurezza sono riportati in Figura 4.5.

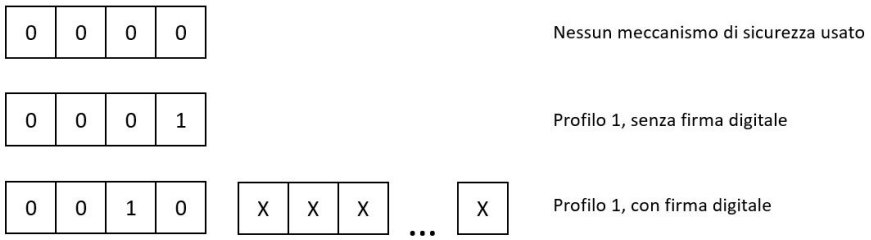


Figura 4.5: Esempi di possibili codifiche della sicurezza

URL

In alcune situazioni ed alcuni contesti, se il dispositivo dell'utente finale ha accesso ad Internet, potrebbe essere più appropriato connettersi ad un server remoto ed eseguire l'applicazione in esso contenuta, piuttosto che eseguire il programma codificato all'interno dell'eQR code. Il vantaggio principale di questo approccio è dato sicuramente dall'incremento dell'interattività che può essere fornita all'utente. Infatti, a causa del fatto che l'applicazione viene eseguita su un server remoto, essa può contenere una gamma più ampia di informazioni, tra le quali spiccano certamente quelle multimediali, come immagini e video, che sono molto difficili da inserire all'interno di un eQR code per ragioni di spazio. Per questo motivo, è stata introdotta la possibilità di inserire, all'interno del QRbytecode header, questo URL. In particolare, l'utilizzo di questa sezione viene segnalato attraverso un bit: se esso è pari ad 1, l'URL è presente e decodificato nei bit immediatamente successivi utilizzando la codifica UTF-8 [22]. Il carattere *End-Of-Text* (EXT), corrispondente alla sequenza di bit `00000011`, è il carattere usato come terminatore di stringa. Se invece il bit è pari a 0, significa che questa sezione non è utilizzata ed i bit successivi corrispondono alla sezione seguente dell'header. È curioso notare come, anche se l'URL è presente all'interno dell'eQR code, sia possibile avere una specie di *disattivazione* di quest'ultimo anche in seguito alla fase di generazione dell'eQR code, programmando, come risposta alla richiesta di esecuzione dell'URL, un riferimento all'esecuzione dell'eQR code. I due possibili tipi di codifica della sezione URL sono riportati, in maniera generale, in Figura 4.6.

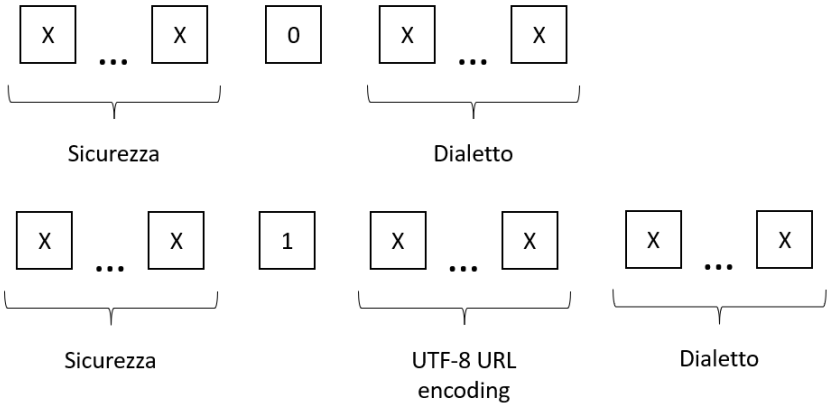


Figura 4.6: Codifica della sezione URL

Dialetto

Questa parte dell'header di QRbytecode si occupa della corretta identificazione del dialetto (e della sua versione) usato per la rappresentazione del programma scritto in linguaggio di alto livello. Essa è, quindi, formata da due campi, uno che identifica il dialetto e l'altro che identifica la versione di quest'ultimo. Entrambi questi campi sono formati da 4 bit e sono estensibili secondo la notazione esponenziale definita in Appendice A. Ciò viene fatto in quanto ogni dialetto può avere diverse caratteristiche anche dal punto di vista dell'instruction set, ed è quindi possibile, in base ai bisogni specifici, definire dialetti più efficienti ad un particolare tipo di programma, che permettano di avere QRbytecode più compatti.

Alcuni esempi di codifica di casi particolari di dialetti sono riportati in Figura 4.7.

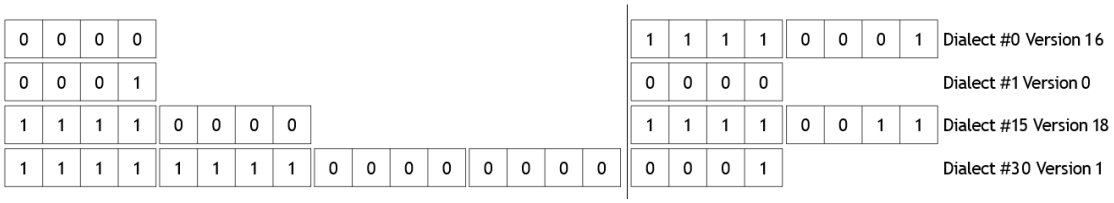


Figura 4.7: Esempi di possibili codifiche di dialetti

I Capitoli 5 e 6 si occuperanno di definire nello specifico i due dialetti che sono stati oggetto del lavoro di questa tesi:

1. Decision Tree Dialect, con codice 0000, che ha permesso lo sviluppo della variante di QRscript denominata *QRtree*.

2. General Purpose Dialect, con codice 0001, che ha permesso lo sviluppo della variante di QRscript denominata *QRprog*.

Capitolo 5

QRtree

5.1 Introduzione

Come già specificato in Sezione 4.2, QRscript è definito in modo che sia in grado di supportare la conversione in e da eQR code di diversi linguaggi di alto livello, facendo uso di diverse rappresentazioni intermedie, dette *dialetti*.

Il primo dialetto che è stato definito e sviluppato durante il lavoro di questa tesi è stato denominato QRtree, ed è un dialetto specializzato nella traduzione di programmi con la struttura di alberi decisionali.

Un programma è detto ad albero decisionale se il suo diagramma di flusso è un diagramma ad albero decisionale. Un diagramma ad albero decisionale è un semplice diagramma di flusso che inizia con un singolo nodo radice, che rappresenta il punto di inizio del programma, nonché una decisione, sia essa binaria o multipla. Le varie possibilità derivanti dalla decisione iniziale fanno sì che il grafico si ramifichi in più possibili flussi i quali, in modo equivalente al flusso originale, consisteranno in una decisione che ramificherà ulteriormente il diagramma. Questa ramificazione continua finché non si raggiunge un cosiddetto *nodo foglia* o *nodo finale*, che rappresenta il risultato finale del flusso, ed è rappresentativo tutte le decisioni che sono state prese durante il flusso di esecuzione del diagramma.

È stato deciso di sviluppare questo tipo di dialetto come prima implementazione di QRscript in quanto la struttura ad albero decisionale è una delle strutture più semplici che un programma possa avere, mantenendo comunque moltissime possibili applicazioni pratiche. Basti pensare, come già accennato in breve nella sezione introduttiva, alla diagnostica di un qualche problema ad un macchinario posizionato all'interno di una costruzione industriale isolata dalla rete Internet: un programma ad albero decisionale sarebbe in grado, attraverso semplici domande sullo stato attuale del macchinario ed i valori di particolari parametri, di fornire una diagnosi e, potenzialmente, anche una soluzione in un tempo molto breve rispetto a quello che dovrebbe invece essere impiegato per rintracciare ed attendere che un tecnico venga fisicamente a controllare il macchinario; è altrettanto semplice, come ulteriore

esempio, pensare ad una comitiva in escursione in una zona di montagna senza una copertura di rete particolarmente efficace che vuole sapere quale possa essere il percorso più adatto alle sue preferenze per raggiungere una determinata destinazione. Anche in questo caso, tramite semplici domande, il programma potrebbe essere in grado di indicare quale tra i percorsi possibili sia il più adatto in un tempo molto rapido.

Le Sezioni successive si occuperanno della definizione di tutti le parti principali che compongono le specifiche di QRtree. In particolare, la Sezione 5.2 definirà le modalità di codifica dei tipi di dato supportati dal dialetto, mentre la Sezione 5.3 approfondirà la parte iniziale che compone i dati che verranno inseriti all'interno dell'eQR code, cioè il cosiddetto QRtree header.

La Sezione 5.4, invece, si occuperà della definizione formale dei componenti principali che compongono QRtree, che possono essere riassunti in due punti fondamentali:

1. Una rappresentazione testuale intermedia, denominata *QRtreeAssembly*, che permette di rappresentare in modo efficiente e più compatto possibile un qualsiasi programma di tipo albero decisionale.

2. Una rappresentazione binaria, caso particolare di QRbytecode, che per questo dialetto è stata denominata *QRtreebytecode*, che permette di compattare ulteriormente la rappresentazione QRtreeAssembly e che verrà usata come interfaccia per la generazione e per la lettura dell'eQR code.

Parte fondamentale della Sezione sarà, quindi, la definizione di tutte le regole necessarie per la conversione da QRtreeAssembly a QRtreebytecode e viceversa.

È importante anche notare che, nonostante QRtree sia stato pensato per lavorare specificamente con programmi ad albero decisionale, esso è in realtà in grado di lavorare con tutti quei programmi che possono essere convertiti in QRtreeAssembly.

5.2 Tipi di dato QRtree

I formati di dati costanti che possono essere utilizzati in QRtree sono stringhe, numeri interi e numeri reali.

5.2.1 Stringhe

La codifica delle stringhe, in questo campo applicativo, è fondamentale, per due motivi principali: eccezion fatta per i numeri, le stringhe sono sicuramente il tipo di dato più usato in assoluto nella maggior parte dei programmi ed in particolare ciò si riscontra in maniera ancor più accentuata negli alberi decisionali, data la presenza

di quei nodi che devono porre le domande e di quei nodi che, invece, forniscono le risposte (*nodi foglie*). Inoltre, se comparate con i numeri, le stringhe occupano uno spazio decisamente maggiore: per cui, in un contesto in cui si lavora con uno spazio di archiviazione molto limitato, come nel caso dei QR code, è fondamentale cercare di ottimizzare il più possibile questa codifica.

Al fine di cercare una codifica efficiente delle stringhe, sono stati proposti tre diversi tipi di codifica: ASCII-7, UTF-8 e DICT.

Questi tre tipi di codifica richiedono l'introduzione di due bit precedenti ogni stringa, che permetta di identificarne il tipo usato permettendo di usare la codifica più efficiente stringa per stringa, con il risultato che, nella maggior parte dei casi, il QRtreebytecode risultante risulta più compatto rispetto al caso in cui venga usata la codifica più generale (in questo caso UTF-8) per tutte le stringhe.

La corrispondenza tra il tipo di codifica ed il codice di due bit ad essa assegnato è la seguente:

- ASCII-7: 00
- UTF-8: 01
- DICT: 10

Mentre il codice 11 è stato lasciato come possibile futura estensione di questi tipi di codifica.

- **ASCII-7**
  La codifica di tipo ASCII-7 è identificata dal codice 00, seguito dalla lista di caratteri che compongono la stringa e terminata da un carattere speciale di fine stringa. Il carattere scelto come terminatore di stringa è stato il carattere ASCII-7 *end-of-text (ETX)*, quarto carattere dello standard ASCII-7 [23] e pertanto identificato dalla sequenza di bit `0000011`. Questo tipo di codifica permette, rispetto all'UTF-8, di risparmiare un bit per ogni carattere che compone la stringa; chiaramente, però, i caratteri codificabili attraverso questa codifica sono ridotti. Ad esempio, non è possibile codificare i caratteri accentati. Da queste considerazioni si evince che questo tipo di codifica è quello più conveniente fra i due tipi standard ed è quindi particolarmente consigliato per tutte le stringhe i cui caratteri rientrano fra quelli supportati nello standard.

- **UTF-8**
  La codifica di tipo UTF-8 è identificata dal codice 01, seguito dalla lista di caratteri che compongono la stringa e terminata da un carattere speciale di fine stringa. Il carattere scelto come terminatore di stringa è stato il carattere UTF-8 *end-of-text (ETX)*, quarto carattere dello standard UTF-8 [22] e pertanto identificato dalla sequenza di bit `00000011`. Questo tipo di codifica

permette, rispetto ad ASCII-7, di codificare un numero più ampio di caratteri (il doppio); tuttavia, richiede un bit in più per ogni carattere, per cui è adatta per quelle stringhe che non possono essere codificate in ASCII-7.

- **DICT**
  A differenza delle prime due codifiche, che derivano da standard già esistenti e molto diffusi, il tipo DICT è stato definito appositamente per aumentare l'efficienza della codifica delle stringhe in QRtree. Il motivo di questa scelta è molto semplice: in un contesto applicativo come quello di un albero decisionale, è molto probabile che molte delle stringhe che si incontrano durante un programma siano uguali, in quanto molto usate nel campo in cui esso viene impiegato (basti pensare semplicemente a stringhe come "sì" o "no" oppure, più nello specifico, alla parola "dispositivo" in contesti quali diagnostica di problemi per dispositivi elettronici). In casi come questo, sarebbe molto inefficiente codificare la stessa stringa usando una delle due codifiche precedenti ogni volta che essa viene incontrata. È molto più efficiente, invece, fare in modo di avere a disposizione un dizionario, in cui ognuna di queste stringhe comuni sia abbinata ad un codice, grazie al quale sia possibile riconoscerla immediatamente. In particolare, sono stati definiti 3 tipi di stringhe di tipo DICT: *globali*, *specifici* e *locali*.

  I dizionari di tipo *globale* sono definiti al di fuori degli eQR code, in quanto essi non sono specifici per l'applicazione, ma sono così generali che risultano efficaci per una gamma molto ampia di contesti (e.g., "sì", "no"). Anche i dizionari di tipo *specifico* sono definiti al di fuori dell'eQR code, ma essi sono specifici per ogni applicazione. In caso di utilizzo di questo tipo di dizionario, quello da usare fra tutti quelli definiti è selezionabile attraverso il comando `DICT_SPEC_TYPE` del *QRtree header* (i dettagli su questo comando sono riportati in Sezione 5.3.4). Nel caso in cui siano presenti più dizionari specifici, il comando `DICT_SPEC_TYPE` specifica il numero del dizionario da usare in base all'ordine in cui essi sono stati caricati (partendo da 0 per il primo dizionario caricato). È possibile caricare più dizionari specifici usando il comando `DICT_SPEC_TYPE` più volte all'interno del *QRtree header*.

  Sia i dizionari di tipo globale che quelli di tipo specifico sono definiti usando il formato `YAML` [24], seguendo le specifiche riportate in Appendice C. In particolare, gli URL che fanno riferimento al dizionario globale oppure ad uno o più dizionari specifici sono salvati all'interno dell'applicazione.

  I dizionari di tipo *locale*, invece, sono definiti all'interno del *QRtree header* usando il comando `DICT_LOCAL` (i dettagli su questo comando sono riportati in Sezione 5.3.5). Dato che, come già discusso molte volte, le stringhe occupano molto spazio, è opportuno definire come stringhe di tipo locale solamente quelle che compaiono più frequentemente all'interno dell'eQR code. La scelta

di quali stringhe inserire all'interno del dizionario locale può essere, ad esempio, automatizzata attraverso un'analisi automatica delle stringhe contenute nel programma e da inserire all'interno dell'eQR code.

Nel caso in cui siano presenti dizionari con diverse versioni (che differiscono tra di loro per la lingua usata), l'applicazione selezionerà il dizionario con la stessa lingua di configurazione dell'applicazione. Se invece, la lingua non è specificata, l'applicazione sceglierà il dizionario di default (ad esempio, quello in lingua inglese).

La codifica di tipo DICT è identificata dal codice 10. I bit successivi, denominati di *dictionary_type*, specificano i tipi di dizionari (*globale*, *specifico* o *locale*) usati. Il valore di questi bit dipende dalla configurazione di alcune costanti. La Tabella 5.1 mostra le varie combinazioni possibili di configurazione di questi bit in base ai tipi di dizionari che si intendono utilizzare. In particolare, la configurazione di default identifica la presenza di un dizionario *globale* e di uno *specifico*, con l'assenza di quello *locale*.

Se un dizionario di tipo *locale* è stato definito usando il comando `DICT_LOCAL` (Sezione 5.3.5), la constante `DICT_LOCAL` viene attivata. Invece, il comando `DICT_TYPES` (Sezione 5.3.3) può essere usato per disabilitare l'uso del dizionario *globale*, settando la costante `NO_DICT_GLOBAL` o l'uso del dizionario *specifico*, settando la costante `NO_DICT_SPEC`.

Infine, la codifica di tipo DICT è terminata dai bit che rappresentano la parola contenuta all'interno del dizionario a cui si vuole fare riferimento (sia esso di qualsiasi tipo). La parola viene selezionata usando la sua posizione all'interno del dizionario, usando un numero di bit uguale al minor numero di bit necessari per rappresentare tutte le parole contenute all'interno del dizionario. Per esempio, se un dizionario contiene 18 parole, tutte le parole contenute al suo interno saranno codificate usando 5 bit.

Nel caso in cui siano presenti più dizionari di tipo *specifico*, essi vengono concatenati usando l'ordine in cui i relativi comandi `DICT_SPEC_TYPE` compaiono all'interno del *QRtree header* e, equivalentemente al caso precedente, la parola viene selezionata usando la sua posizione all'interno del dizionario *specifico* formato dalla precedente concatenazione, usando la stessa logica per il numero di bit da usare per la sua rappresentazione.

Dictionary Options	*global*	*specific*	*local*
DICT_GLOBAL DICT_SPEC DICT_LOCAL	00	01	1
NO_DICT_GLOBAL NO_DICT_SPEC NO_DICT_LOCAL	-	-	-
DICT_GLOBAL NO_DICT_SPEC DICT_LOCAL	0	-	1
NO_DICT_GLOBAL DICT_SPEC DICT_LOCAL	-	0	1
DICT_GLOBAL DICT_SPEC NO_DICT_LOCAL	0	1	-
DICT_GLOBAL NO_DICT_SPEC NO_DICT_LOCAL	*	-	-
NO_DICT_GLOBAL DICT_SPEC NO_DICT_LOCAL	-	*	-
NO_DICT_GLOBAL NO_DICT_SPEC DICT_LOCAL	-	-	*

Tabella 5.1: Bit usati per identificare i tipi di dizionari (*globale*, *locale*, *specifico*) definiti. Il simbolo - indica che quel tipo di dizionario non può essere utilizzato, mentre il simbolo * indica che il dizionario può essere usato senza scrivere alcun bit in quanto è l'unico tipo configurato per lo specifico eQR code. La configurazione di default è `DICT_GLOBAL`, `DICT_SPEC` e `NO_DICT_LOCAL`

5.2.2 Interi: INT16 e INT32

La codifica degli interi è basata su due tipi di numeri interi, quelli a 16 bit e quelli a 32 bit. Entrambe le codifiche sono inoltre basate sul formato *Complemento a due* [25]. Per distinguere i due casi di codifica, sia tra di loro, sia con i casi relativi ad i numeri reali, le codifiche dei numeri interi sono introdotti da un codice di due

bit, corrispondente a 00 nel caso della codifica INT16 e a 01 nel caso della codifica INT32.

5.2.3 Reali: FP16 e FP32

Il formato FP16 prevede la codifica del numero su 16 bit usando la codifica half-precision floating-point secondo lo standard IEEE Standard for Floating-Point Arithmetic (IEEE 754) [26] che prevede 1 bit di segno, 5 bit di esponente e 10 bit di frazione. Il formato FP32 prevede la codifica del numero su 32 bit usando la codifica single-precision floating-point secondo lo standard IEEE Standard for Floating-Point Arithmetic (IEEE 754) [26] che prevede 1 bit di segno, 8 bit di esponente e 23 bit di frazione. Per distinguere i due casi di codifica, sia tra di loro, sia con i casi relativi ad i numeri interi, le codifiche dei numeri reali sono introdotti da un codice di due bit, corrispondente a 10 nel caso della codifica FP16 e a 11 nel caso della codifica FP32.

5.3 QRtree header

Indipendentemente dal dialetto usato per la codifica del programma, il QRbytecode è introdotto sempre dal QRscript header, introdotto in Sezione 4.2.1. Esso è però, opzionalmente, seguito da un ulteriore header specifico per il dialetto usato. Nel caso di QRtree, questo header è denominato *QRtree header*.

La presenza di questo ulteriore header è denotato dal primo bit dopo il QRscript header settato ad 1. In alternativa, il primo bit settato a 0 ne notifica la sua assenza.

Il QRtree header è costituito da una serie di comandi codificati su 3 bit estensibili con il formato esponenziale definito in Appendice A. Questi comandi sono trattati in maniera approfondita nelle sottosezioni seguenti.

5.3.1 HEADER_END: 000

Il comando HEADER_END, identificato con il codice 000, identifica la fine della sezione riguardante il QRtree header. Questo particolare comando è necessario in quanto questo header è una lista di comandi di cui non si ha alcuna indicazione riguardante la lunghezza. Per questo motivo è necessario sapere dove esso ha la sua fine: una possibile alternativa sarebbe avere come primo dato dell'header il numero di comandi presenti. Tuttavia, pare subito evidente come questo dato possa molto facilmente andare ad occupare uno spazio di molto superiore ad i 3 bit occupati da questo comando risultando, nella maggior parte dei casi, del tutto inefficiente. Si è deciso, per cui, di riservare un comando specifico per questa operazione.

5.3.2 INT_TYPE: 001 E FLOAT_TYPE: 010

I comandi INT_TYPE, identificato con il codice 001 ed il suo analogo per i numeri reali, FLOAT_TYPE, identificato con il codice 010 sono utilizzati per definire, in casi particolari, la dimensione di memorizzazione dei numeri interi e reali, rispettivamente. In particolare, per quanto riguarda il comando INT_TYPE, se il bit che lo segue è settato a 0, tutti i numeri interi presenti nel QRtreebytecode corrispondente sono da interpretare come codificati usando il formato INT16, mentre se il bit è settato ad 1, essi sono da considerarsi come codificati usando il formato INT32. Analogamente, per quanto riguarda il comando FLOAT_TYPE, se il bit che lo segue è settato a 0, tutti i numeri reali presenti nel QRtreebytecode corrispondente sono da interpretare come codificati usando il formato FP16, mentre se il bit è settato ad 1, essi sono da considerarsi come codificati usando il formato FP32. Questi due comandi sono particolarmente utili nel caso in cui tutti numeri interi e/o tutti i numeri reali all'interno del programma di alto livello siano tutti codificabili usando lo stesso formato. Infatti, come visto in Sezione 5.2.2 ed in Sezione 5.2.3, le codifiche dei numeri sono preceduti da una coppia di bit che identificano il tipo di numero e la codifica con cui esso è stato codificato e questa operazione è necessaria per ogni singolo numero che viene incontrato. Se però tutti i numeri interi e/o reali possono essere codificati usando lo stesso formato, è possibile omettere uno dei due bit per ognuno dei numeri di quel tipo, aggiungendo in questo comando l'informazione riguardante il tipo di codifica da usare, ottenendo, nella maggior parte dei casi, una diminuzione dello spazio necessario per codificare il programma.

5.3.3 DICT_TYPES: 011

Il comando DICT_TYPE, identificato con il codice 011 permette di identificare quali tipi di dizionari tra quelli esterni definiti in Sezione 5.2.1 sono attivati per il corrispondente eQR code. In particolare, i due bit che seguono il comando specificano se i dizionari di tipo *globale* e di tipo *specifico* sono attivati oppure no. La rappresentazione ed il significato di questi due bit è riportata in Tabella 5.2, dalla quale si può notare come un bit settato a 0 indichi l'assenza dell'utilizzo del relativo dizionario, mentre un bit settato a 1 indichi l'attivazione del corrispondente tipo di dizionario.

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 5.2: Bit per l'attivazione dei dizionari esterni

In assenza di questo comando all'interno di QRtree header, la configurazione di default è quella in cui entrambi i dizionari sono attivi. È quindi utile notare come, nonostante la sequenza di bit 11 sia presente in tabella, essa non dovrebbe mai essere utilizzata, in quanto rappresenterebbe solamente uno spreco di spazio e non comporterebbe alcuna differenza rispetto alla sua assenza.

5.3.4 DICT_SPEC_TYPE: 100

Il comando DICT_SPEC_TYPE, identificato con il codice 100, permette di identificare quale dizionario di tipo specifico deve essere usato per l'attuale eQR code. I bit che seguono questo comando rappresentano l'indice intero che identifica il dizionario da usare. Infatti, quando dizionari di questo tipo vengono caricati dall'applicazione, essi vengono caricati con un particolare ordine, che permette quindi di assegnare un identificatore unifico (che parte da 0) ad ogni dizionario. Nel caso in cui si vogliano caricare più dizionari ti tipo specifico, il comando DICT_SPEC_TYPE deve essere ripetuto una volta per ogni dizionario che si vuole caricare.

5.3.5 DICT_LOCAL: 101

Come evidenziato anche in precedenza, i dati di tipo stringa occupano molto spazio. Un dizionario di tipo locale permette di definire alcune stringhe (tipicamente stringhe che non sono molto comuni, ma che si trovano ripetute molte volte all'interno del particolare programma da caricare all'interno dell'eQR code), dandone una numerazione. Il comando DICT_LOCAL, identificato dal codice 101, permette la definizione di un dizionario locale, il cui contenuto verrà salvato all'interno dell'eQR code. I primi tre bit che seguono di comando, estensibili secondo il formato definito in Appendice A, rappresentano la lingua associata al dizionario che viene definito (e.g. 000 per la lingua 1, 001 per la lingua 2, ecc...). Se un dizionario è definito per più lingue, è necessario ripetere il comando un numero di volte pari al numero di lingue. L'associazione fra lingua e numero identificativo corrispondente è definito all'interno dell'applicazione, fatta eccezione per la lingua con indice 000, che è sempre associata con la lingua di *default*. In seguito a questi 3 bit, ci sono altri

4 bit (sempre estensibili usando il formato esponenziale definito in Appendice A), che identificano il *numero di parole* contenute all'interno del dizionario. Successivamente, è presente una lista di *parole*. Ognuna di queste parole è rappresentata dal suo tipo di codifica (`0` per ASCII-7, e `1` per UTF-8) e dalla relativa codifica binaria. Se all'interno di un dizionario di una lingua diversa da quella di default, una parola viene codificata con la stringa vuota (i.e., “”), sarà stampata la corrispondente parola del dizionario di default. Nel caso in cui a una parola del dizionario di default sia associata la stringa vuota, quando questa parola dovesse essere stampata sarà coerentemente stampata la stringa vuota.

5.3.6 USER_DEF: 110

Il comando USER_DEF, identificato con il codice 110, non è associato a nessuna azione particolare. Esso può dunque essere usato liberamente dallo sviluppatore dell'applicazione per aggiungere nuove funzionalità associate ad alcuni dati inseriti all'interno dell'header.

5.4 Dialetto

All'interno di QRbytecode, in seguito alle due sezioni corrispondenti a *QRscript header* e, opzionalmente, a *QRtree header*, si trova la parte principale di QRbytecode, contenente la traduzione del programma di alto livello da eseguire.

Questa sezione e le sue relative sottosezioni descriveranno in modo dettagliato le regole di conversione da QRtreeAssembly a QRtreebytecode (corrispondente al passaggio 2 in Figura 4.1). La conversione opposta, cioè quella da QRtreebytecode a QRtreeAssembly, passaggio fondamentale per la fase di esecuzione, segue esattamente le stesse regole, ma in modo inverso.

Nel passaggio 1 rappresentato in Figura 4.1, il programma scritto in linguaggio di alto livello viene tradotto in una rappresentazione intermedia che consiste in una lista di istruzioni. In particolare, per quanto riguarda QRtree, questa rappresentazione intermedia, denominata *QRtreeAssembly*, è composta da 7 tipi di istruzioni fondamentali, identificati da un codice di tre bit. La corrispondenza tra le istruzioni definite ed i loro codici è riportata in Tabella 5.3

Il codice 111 è stato lasciato senza una corrispondenza in quanto, in caso si voglia estendere l'instruction set di QRtreeAssembly, è possibile farlo usando il formato esponenziale definito in Appendice A, per cui tutte le nuove istruzioni definite saranno introdotte da questa particolare sequenza di bit seguita dal loro codice.

Nome dell'istruzione	Codice binario
`input`	000
`inputs`	001
`print`	010
`printex`	011
`goto`	100
`if`	101
`ifc`	110

Tabella 5.3: Instruction set di QRtreeAssembly

5.4.1 Tipi di input

Come riportato in Tabella 5.3, QRtree definisce due tipi di istruzioni diverse per le operazioni di input. Questo perché, in QRtree, vengono differenziati due casi: gli input cosiddetti *diretti*, a cui è associata l'istruzione `inputs`, e quelli *indiretti*, a cui è associata l'istruzione `input`.
Un input di tipo diretto è rappresentato da una stringa di testo che viene inserita dall'utente, mentre un input di tipo indiretto è un input in cui le scelte possibili sono ristrette ad un dominio predefinito e molto limitato e che, quindi, possono essere rappresentate, ad esempio, da uno o più bottoni corrispondenti. I valori di queste possibili scelte predefinite vengono assunti direttamente dall'applicazione in quanto contenuti nella rappresentazione intermedia. All'interno di questo dominio limitato di possibilità, è sempre prevista un'ulteriore risposta, di default, denominata *"Altro"*, che permette all'utente di fornire una risposta che non corrisponda a nessuna delle alternative predefinite.

Si riporta di seguito, a titolo di esempio, una parte di una possibile rappresentazione intermedia ottenuta dalla prima parte della fase di generazione:

```
(1) input "Domanda 1"
(2) if "Risposta 1" (6)
(3) if "Risposta 2" (8)
(4) if "Risposta 3" (10)
(5) goto (12)
(6) # Codice relativo a Risposta 1
(7) ...
(8) # Codice relativo a Risposta 2
```

```
(9) ...
(10) # Codice relativo a Risposta 3
(11) ...
(12) inputs "Domanda 2"
```

Durante la fase di esecuzione, questa stessa porzione di rappresentazione intermedia, dopo essere stata recuperata dall'eQR code traducendo QRtreebytecode, verrà eseguita dall'interprete o dalla macchina virtuale. La linea (1) scriverà a schermo la stringa "Domanda 1" e, visto che essa contiene un input di tipo indiretto, verrà anche mostrata a schermo un'interfaccia che permetta all'utente di scegliere tra le possibili alternative ("Risposta 1", "Risposta 2", "Risposta 3", "Altro"). Un possibile esempio di questa user interface (UI) è in Figura 5.1, in cui essa è basata su bottoni.

Figura 5.1: Esempio di UI basata su bottoni

Nel caso in cui l'utente clicchi il bottone corrispondente a "Risposta 1", ad esempio, verrà eseguito, all'interno della rappresentazione intermedia, un salto alla linea (6), in cui verrà eseguito il codice relativo alla "Risposta 1", che potrà anche contenere altre domande (esso può essere un sottoalbero di decisione). In modo analogo, nel caso in cui l'utente clicchi il bottone corrispondente a "Risposta 2", il salto verrà eseguito verso la linea (8), a partire dalla quale verrà eseguito il codice corrispondente alla "Risposta 2". Se, invece, l'utente clicca il bottone corrispondente a "Risposta 3", il programma salterà alla linea (10), dove il comportamento sarà analogo ai primi due casi. Infine, nel caso in cui l'utente dovesse scegliere l'alternativa "Altro", verrà eseguita la linea (5) della rappresentazione intermedia, portando il flusso di esecuzione direttamente alla linea (12), cioè alla domanda successiva, che comparirà a schermo. Contrariamente alla prima domanda, questa seconda domanda è un input di tipo diretto, per cui l'utente dovrà inserire esplicitamente un valore, per esempio all'interno di un textbox. Questo valore potrà essere interpretato in modo diverso in base ai comandi successivi: verrà infatti interpretato come stringa nel caso di un'istruzione `if` oppure come numero in caso di un'istruzione `ifc`. Una possibile rappresentazione dell'interfaccia utente di quest'ultimo caso è riportata in Figura 5.2.

Figura 5.2: Esempio di UI per i due tipi di input

5.4.2 Referenze

Un possibile altro vantaggio di usare gli eQR code, ma più in generale i QR code è che, essendo essi stampati su un supporto fisico, possono essere affiancati da informazioni aggiuntive, come testi o immagini, stampate e collocate vicino al QR code stesso. Questo vantaggio può anche essere ulteriormente sfruttato per migliorare l'efficienza della codifica della rappresentazione intermedia in codice binario. Per esempio, nel caso in cui un programma di tipo QRtree generi come output stringhe molto voluminose, che richiederebbero quindi molto spazio interno all'eQR code per essere codificate, esse potrebbero essere estratte al di fuori dell'eQR code ed essere solamente poste vicino ad esso. Lo stesso concetto vale anche per le possibili domande derivanti dai due tipi di input presentati in Sezione 5.4.1. Per questo motivo è stata definita la possibilità di stampare, per le istruzioni di input (`input` ed `inputs`) e di output (`print`, `printex`), le cosiddette *Referenze*. Il termine *Referenza* denota un numero intero che identifica una stringa di testo o un'immagine esterna di supporto stampata nelle vicinanze dell'eQR code. Ad esempio, una risposta molto lunga e complessa può essere sostituita direttamente con la referenza numero 1, che indica di leggere la risposta catalogata come numero 1 tra quelle presenti all'interno di quelle stampate vicino all'eQR code. Un discorso analogo può essere fatto per le immagini, le quali possono anche essere, nella rappresentazione di supporto, integrate con una descrizione testuale.

Tutte le referenze presenti all'interno della rappresentazione intermedia vengono codificate sulla base della notazione esponenziale approfondita in Appendice A, partendo da un numero minimo di 4 bits.

5.4.3 input

Come già riportato in Sezione 5.4.1, l'istruzione `input` serve per richiedere all'utente l'inserimento di un valore di input di tipo indiretto. La struttura dell'istruzione è la seguente:

```
input <constante>
```

Prima della richiesta dell'inserimento, l'istruzione visualizzerà il valore della costante `<constante>` su schermo.

L'istruzione `input` viene codificata con i primi 3 bit pari a `000`, seguiti da un bit che identifica il tipo della costante: se questo bit è settato a 0, `<costante>` sarà una stringa, mentre se esso è settato ad 1, essa sarà una referenza. Segue ulteriormente la codifica della costante, nel rispetto delle regole definite in Sezione 5.2.1 (se è una stringa) o in Sezione 5.4.2 (se è una referenza).

La struttura generale dell'istruzione `input` è riportata in Figura 5.3

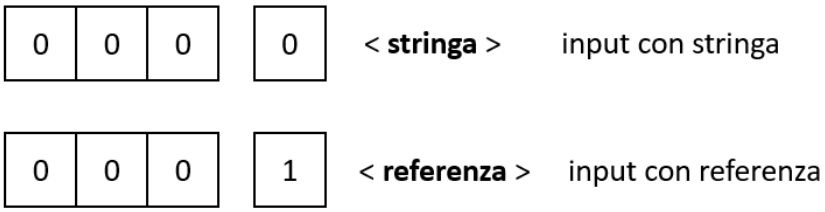


Figura 5.3: Struttura della codifica di input

5.4.4 inputs

Come già riportato in Sezione 5.4.1, l'istruzione `inputs` serve per richiedere all'utente l'inserimento di un valore di input di tipo diretto. La struttura dell'istruzione è la seguente:

```
inputs <constante>
```

Prima della richiesta dell'inserimento, l'istruzione visualizzerà il valore della costante `<constante>` su schermo.

L'istruzione `inputs` viene codificata con i primi 3 bit pari a `001`, seguiti da un bit che identifica il tipo della costante: se questo bit è settato a 0, `<costante>` sarà una stringa, mentre se esso è settato ad 1, essa sarà una referenza. Segue ulteriormente la codifica della costante, nel rispetto delle regole definite in Sezione 5.2.1 (se è una stringa) o in Sezione 5.4.2 (se è una referenza).

La struttura generale dell'istruzione `inputs` è riportata in Figura 5.4

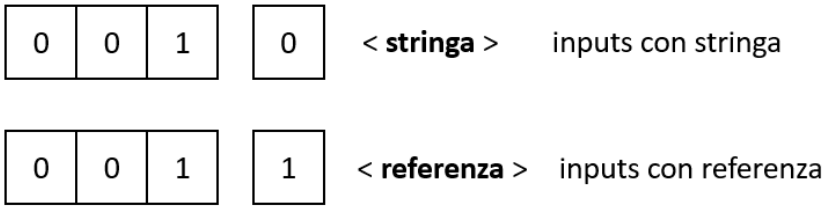


Figura 5.4: Struttura della codifica di inputs

5.4.5 print

L'istruzione `print` serve per stampare una costante sullo schermo. La struttura dell'istruzione è la seguente:

```
print <constante>
```

L'istruzione `print` viene codificata con i primi 3 bit pari a `010`, seguiti da un bit che identifica il tipo della costante: se questo bit è settato a 0, `<costante>` sarà una stringa, mentre se esso è settato ad 1, essa sarà una referenza. Segue ulteriormente la codifica della costante, nel rispetto delle regole definite in Sezione 5.2.1 (se è una stringa) o in Sezione 5.4.2 (se è una referenza).

La struttura generale dell'istruzione `print` è riportata in Figura 5.5

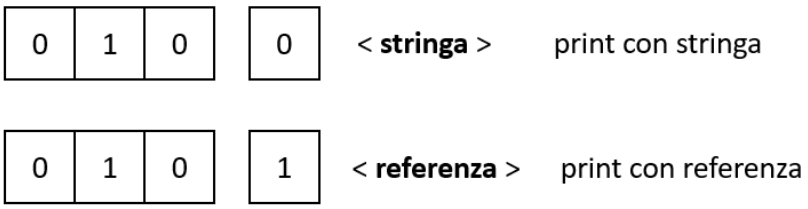


Figura 5.5: Struttura della codifica di print

5.4.6 printex

L'istruzione `printex` serve per stampare una costante sullo schermo e terminare immediatamente l'esecuzione del programma. La struttura dell'istruzione è la seguente:

```
printex <constante>
```

L'istruzione `printex` viene codificata con i primi 3 bit pari a `011`, seguiti da un bit che identifica il tipo della costante: se questo bit è settato a 0, `<costante>` sarà una stringa, mentre se esso è settato ad 1, essa sarà una referenza. Segue ulteriormente la codifica della costante, nel rispetto delle regole definite in Sezione 5.2.1 (se è una stringa) o in Sezione 5.4.2 (se è una referenza).

Un caso particolare di utilizzo di questa funzione può essere quello di usarla per terminare l'esecuzione del programma senza stampare nulla a schermo (il "nulla" stampato a schermo è equivalente alla stampa di una stringa vuota). In questo caso l'istruzione

```
printex ""
```

verrà tradotta in binario come la sequenza di bit `011 0 00 0000011`, dove i primi tre bit rappresentano l'istruzione, il bit `0` indica che la costante `<costante>` è una stringa, la coppia di bit `00` specifica che la stringa è codificata in ASCII-7, mentre la sequenza `0000011` rappresenta il carattere terminatore di stringa end-of-text (ETX), che indica la fine della codifica, da cui verrà ottenuta la stringa vuota.

La struttura generale dell'istruzione `printex` è riportata in Figura 5.6

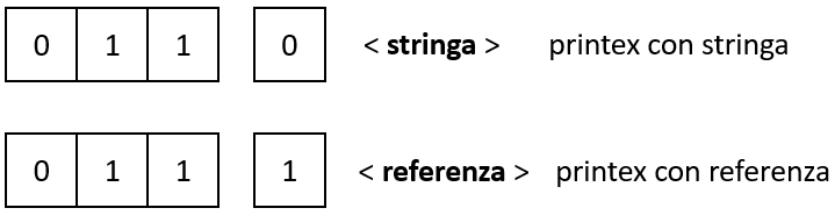


Figura 5.6: Struttura della codifica di printex

5.4.7 goto

L'istruzione `goto` serve per effettuare un salto incondizionato. La struttura dell'istruzione è la seguente:

```
goto <salto_relativo>
```

L'istruzione `goto` viene codificata con i primi 3 bit pari a `100`, seguiti dalla codifica di un numero intero positivo (senza segno) che rappresenta il numero di istruzioni da saltare. È importante porre attenzione al fatto che `<salto_relativo>` permette di fare salti solamente in avanti. Non essendo possibili salti all'indietro,

con QRtree non è possibile la realizzazione di cicli. Questa scelta è dovuta al fatto che per la realizzazione di alberi decisionali i cicli non sono di diretta utilità.

Il valore `<salto_relativo>` è codificato su 4 bit, e con la sequenza 0000 si codifica il salto all'istruzione successiva. Tuttavia, essendo esso un intero senza segno, i 4 bit che rappresentano il salto sono estensibili sfruttando il formato esponenziale definito in Appendice A, lo stesso definito per la codifica delle Referenze.

La struttura generale dell'istruzione `goto` è riportata in Figura 5.7

Figura 5.7: Struttura della codifica di goto

5.4.8 if

L'istruzione `if` serve per effettuare un salto condizionato. La struttura dell'istruzione è la seguente:

```
if <constante> <salto_relativo>
```

L'istruzione `if` viene codificata con i primi 3 bit pari a `101`, seguiti da un bit che identifica il tipo della costante: se questo bit è settato a 0, `<costante>` sarà una stringa, mentre se esso è settato ad 1, essa sarà una referenza. Segue ulteriormente la codifica della costante, nel rispetto delle regole definite in Sezione 5.2.1 (se è una stringa) o in Sezione 5.4.2 (se è una referenza). Vi è, infine, la codifica del numero intero `<salto_relativo>` (equivalente al caso riportato per l'istruzione `goto` in Sezione 5.4.7). Il salto, di un numero di istruzioni pari a `<salto_relativo>`, verrà effettuato se e solo se il valore di `<costante>`, sia essa una stringa o una referenza, è uguale a quello dell'ultimo input inserito dall'utente. Per fare ciò, ogni volta che viene effettuata un'istruzione di input (sia essa di tipo diretto o indiretto), il dato inserito dall'utente viene salvato in una variabile, denominata `tmp_input`, che verrà confrontata con il valore di `<costante>` per decidere se effettuare il salto oppure riprendere l'esecuzione delle istruzioni a partire da quella successiva.

La struttura generale dell'istruzione `if` è riportata in Figura 5.8

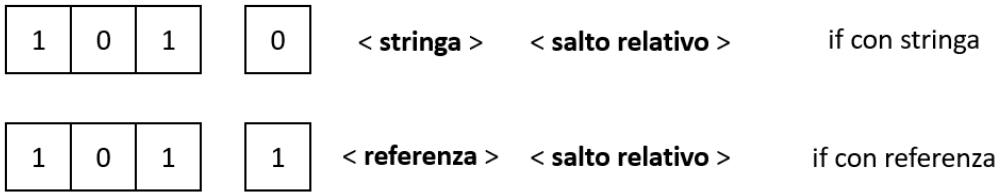


Figura 5.8: Struttura della codifica di if

5.4.9 ifc

L'istruzione `ifc` serve per effettuare un salto condizionato a seguito di un confronto effettuato mediante un operatore relazionale (i.e., `==`, `!=`, `<=`, `>=`, `<`, `>`) tra l'ultimo valore inserito in input dall'utente ed un numero di tipo intero o reale specificato. La struttura dell'istruzione è la seguente:

```
ifc <op_rel> <constante> <salto_relativo>
```

L'istruzione `ifc` viene codificata con i primi 3 bit pari a `110`, seguiti da 3 bit che identificano l'operatore relazionale usato e da un bit che identifica il tipo della costante: se questo bit è settato a 0, `<costante>` sarà un intero, mentre se esso è settato ad 1, essa sarà un numero reale. Segue ulteriormente la codifica della costante, nel rispetto delle regole definite in Sezione 5.2.2 (se è un intero) o in Sezione 5.2.3 (se è un numero reale). Segue infine la codifica di `<salto_relativo>` (equivalente al caso riportato per l'istruzione `goto` in Sezione 5.4.7), che indica il numero di istruzioni da saltare.

La corrispondenza tra i 3 bit relativi alla codifica di `<op_rel>` e l'operatore relazionale da essi codificato è rappresentato in Tabella 5.4.

Tabella 5.4: Corrispondenza tra `<op_rel>` e operatore relazionale codificato.

`<op_rel>`	Operatore relazionale
`000`	`==`
`001`	`!=`
`010`	`<=`
`011`	`>=`
`100`	`<`
`101`	`>`

Le sequenze di bit `110` e `111` non sono attualmente utilizzate e sono lasciate per estensioni future.

Anche in questo caso, per fare il confronto, è necessario avere a disposizione l'ultimo valore di input inserito dall'utente. Analogamente a quanto succede per l'istruzione `if`, questo valore viene salvato in `tmp_input`, che verrà confrontata con il valore di `<costante>` per decidere se effettuare il salto oppure riprendere l'esecuzione delle istruzioni a partire da quella successiva.

Tutte le volte che il confronto tra `tmp_input` e `<costante>` (basato sull'operatore relazionale specificato in `op_rel`) restituisce un risultato *true*, il salto viene eseguito; invece, l'esecuzione viene normalmente ripresa a partire dall'istruzione successiva nel caso in cui questo confronto ritorni un risultato *false*.

La struttura generale dell'istruzione `ifc` è riportata in Figura 5.8

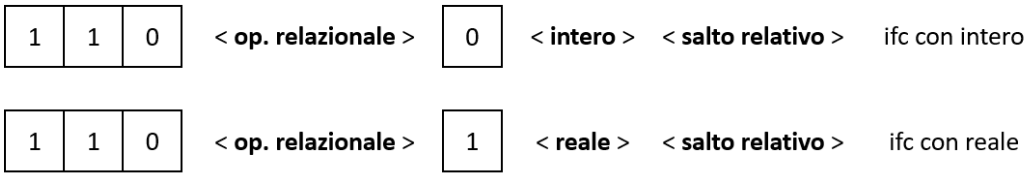


Figura 5.9: Struttura della codifica di ifc

Capitolo 6

QRprog

6.1 Introduzione

Il secondo dialetto che è stato definito e sviluppato durante il lavoro di questa tesi è stato denominato *QRprog* e, rispetto a *QRtree*, è un dialetto molto più generale, in grado di tradurre programmi *general purpose*. Questo tipo di programmi non ha una struttura fissa ed inoltre presenta un insieme di possibili istruzioni pressoché infinito per cui, seppur presenti enormi vantaggi dal punto di vista della vastità delle operazioni che possono essere eseguite (e, conseguentemente, dei suoi contesti applicativi), presenta anche grandi svantaggi di standardizzazione rispetto a QRtree, soprattutto se si parla del linguaggio di alto livello che deve essere tradotto. Per questo motivo, quando nella Sezione 7.2 verranno mostrati alcuni esempi del possibile utilizzo di QRprog, questi esempi conterranno un linguaggio di alto livello C-like, ma molto generale. In aggiunta, il suo difetto principale consiste nella maggior occupazione di memoria rispetto a dialetti più specifici, come *QRtree*. Il lavoro di questa tesi si concentra, per questo dialetto, sulla definizione della rappresentazione intermedia usata per la generazione del QRbytecode, detta *QRprogAssembly* e della corrispondente rappresentazione binaria, detta *QRprogbytecode*.

In particolare, la Sezione 6.3, si occuperà della definizione dell'instruction set di questa rappresentazione intermedia, riportando tutte le istruzioni definite ed i loro principali contesti applicativi, con alcuni esempi.

6.2 Specifiche

Come già dichiarato precedentemente all'interno di questo capitolo, definire un insieme di regole equivalenti a quanto è stato fatto per le specifiche di *QRtree* risulta decisamente più complesso, in quanto la natura general purpose del linguaggio di alto livello non permette di limitare più di tanto ciò che si deve o non si deve poter fare usando quel linguaggio. Tuttavia, sfruttando le potenzialità di *QRscript*

introdotte all'interno di *QRscript header* (Sezione 4.2.1), è possibile definire, per ogni dialetto di *QRscript*, diverse versioni, con caratteristiche diverse tra di loro. Il lavoro svolto durante questa tesi ha dunque definito la versione 1 di questo dialetto, la quale, seppur sia il più generale possibile, deve aderire alle specifiche riportate in questa Sezione.

In particolare, le sottosezioni successive si occuperanno di riportare i dettagli principali delle specifiche di *QRprog*.

La Sezione 6.2.1 definirà le modalità di codifica dei tipi di dato supportati dal dialetto, mentre la Sezione 6.2.2 approfondirà la parte iniziale che compone i dati che verranno inseriti all'interno dell'eQR code, cioè il cosiddetto QRprog header. La Sezione 6.3, invece, si occuperà della definizione formale dei componenti principali che compongono QRprog, che possono essere riassunti in due punti fondamentali:

1. Una rappresentazione intermedia, denominata *QRprogAssembly*, che permette di rappresentare in modo efficiente e più compatto possibile il programma general purpose di alto livello.

2. Una rappresentazione binaria, caso particolare di QRbytecode, che per questo dialetto è stata denominata *QRprogbytecode*, che permette di compattare ulteriormente la rappresentazione QRprogAssembly e che verrà usata come interfaccia per la generazione e per la lettura dell'eQR code.

6.2.1 Tipi di dato QRprog

Equivalentemente a QRtree, anche i formati costanti che possono essere utilizzati in QRprog sono stringhe, numeri interi e numeri reali.

La codifica di tutti i tipi costanti è introdotta da una sequenza di 4 bit `0000`, seguita dall'identificatore del tipo da codificare (seguendo la specifica riportata in Tabella 6.1) e dalla codifica del valore della costante in modo conforme a quanto riportato successivamente.

Stringhe

La codifica delle stringhe è gestita in maniera pressoché equivalente a QRtree, ovvero permettendo il doppio tipo di codifica `ASCII-7` [23] e `UTF-8` [22] in base al tipo di caratteri da cui è formata la stringa. In questo modo, se la stringa contiene solo caratteri codificabili su 7 bit, è possibile risparmiare, rispetto ad una codifica valida in maniera generale (UTF-8 per qualsiasi stringa), un bit per ogni carattere. Inoltre, mentre in questa versione del dialetto non sono ancora state definite le operazioni per l'implementazione dei dizionari, ciò può essere facilmente integrato a partire dalle specifiche di *QRtree*, presentate in Sezione 5.2 ed in Sezione 5.3.

Tipo	Codice binario
`Int`	000
`Float 16`	001
`Float 32`	010
`Float 64`	011
`String ASCII-7`	100
`String UTF-8`	101
`String DICT`	110

Tabella 6.1: Tipi di dato di *QRprog*

Interi

La codifica dei numeri interi è stata radicalmente cambiata rispetto a QRtree. Infatti, è stato deciso di rimuovere la limitazione di avere solamente due lunghezze possibili (16 e 32 bit), per passare ad una codifica completamente basata sulla notazione esponenziale descritta in Appendice A.

Reali: FP16, FP32 e FP64

La codifica dei numeri reali è stata invece solamente ritoccata rispetto a *QRtree* aggiungendo la codifica a doppia precisione (FP64), che prevede la codifica del numero su 64 bit sempre secondo lo standard IEEE Standard for Floating-Point Arithmetic (IEEE 754) [26], che prevede 1 bit per il segno, 11 bit per l'esponente e 52 bit per la mantissa. La doppia precisione offre una maggiore precisione rispetto alla singola precisione, consentendo la rappresentazione di numeri con una precisione approssimativa di 15-16 cifre decimali.

6.2.2 QRprog header

Così come accade per *QRtree*, anche nel caso di *QRprog* il *QRprogbytecode* è introdotto dal QRscript header, che dà la possibilità di sfruttare tutte le potenzialità introdotte in Sezione 5.3.

In seguito al *QRscript header* si trova invece il *QRprog header*, specifico per questo dialetto. Mentre il *QRtree header* poteva contenere un numero elevato di comandi volti a sfruttare la particolare natura di un programma ad albero decisionale, il *QRprog header* contiene, alla versione 1 del dialetto, un solo dato, denominato

`fileId`, che rappresenta, all'interno di un insieme di programmi diversi, ma collegati fra di loro, l'indice dell'ordine in cui il file corrispondente verrà chiamato durante l'interpretazione dell'insieme.

Ciò è necessario in quanto, per la natura general purpose di *QRprog*, è possibile che un programma ad alto livello, contenuto all'interno di un file, debba, durante la sua esecuzione, chiamare una funzione definita all'interno di un altro file.

Per questo motivo, *QRprog* fornisce un metodo per supportare l'esecuzione multifile, che può essere considerato il duale rispetto alla *continuation*: mentre in quest'ultima un file troppo grosso viene gestito suddividendolo in più eQR code, usando la funzionalità di import di *QRprog*, si fa in modo che più funzioni, suddivise in file diversi, possano interagire fra di loro.

Per fare ciò, è stato necessario fare in modo che il linguaggio ad alto livello supportasse un costrutto che permettesse di importare funzioni da altri file. È stato scelto, a questo fine, di sfruttare il costrutto derivato da Javascript, del tipo:

```
import {fun1 as f1, ...} from "file_name"
```

Questo costrutto viene interpretato da *QRprog* mappando l'indice del file `file_name` ad una lista contenente tutti i nomi delle funzioni a cui fare riferimento, così che esse possano essere raggiungibili da qualsiasi file che le ha importate.

Durante l'esecuzione di una sessione di *QRprog*, a tutti i file che vengono incontrati viene assegnato un indice intero senza segno e progressivo (a partire da 0), corrispondente al `fileId` citato in precedenza. All'interno del *QRprog header* esso verrà inserito tramite la sua rappresentazione binaria, derivata dalla notazione esponenziale definita in Appendice A.

Durante il processo di interpretazione, la chiamata ad una funzione esterna al file attualmente in esecuzione può presentare due comportamenti diversi: se l'eQR code contenente la funzione è già stato scansionato con successo, e l'applicazione è in possesso della rappresentazione intermedia risultante da tale lettura, il flusso di esecuzione si sposterà semplicemente verso il nuovo file, per poi ritornare a quello originale alla fine dell'esecuzione della funzione. Se, al contrario, l'eQR code contenente la funzione da invocare non è ancora stato scansionato, il flusso di esecuzione di *QRprog* verrà interrotto, richiedendo all'utente di scansionare l'eQR code. Una volta che tale scansione sarà stata effettuata, il flusso riprenderà esattamente da dove si era interrotto, con un comportamento equivalente al caso precedente.

6.3 Dialetto

Così come accade per *QRtree*, dopo tutta la parte di header, il QRbytecode continua con la parte denominata *Code*, contenente la traduzione del programma di alto livello da eseguire.

Per quanto riguarda *QRprog*, il programma scritto in linguaggio di alto livello viene tradotto nella rappresentazione intermedia, denominata *QRprogAssembly*, composta da 12 istruzioni fondamentali, che permettono di eseguire molte operazioni, fra le quali input, output, assegnazioni, dichiarazioni, loop, chiamate e ritorni da funzioni. Ciascuna di queste istruzioni è identificata da un codice di quattro bit; la corrispondenza tra le istruzioni definite ed i loro codici è riportata in Tabella 6.2. Come per *QRtree*, anche questo instruction set è estensibile secondo la notazione esponenziale riportata in Appendice A. Essa, inoltre, permette di ottenere la rappresentazione binaria (*QRprogbytecode*) da inserire all'interno dell'eQR code. È importante notare, per comprendere meglio le successive sezioni, come tutte le variabili presenti all'interno del programma di alto livello vengano mappate, all'interno della rappresentazione intermedia, in registri Assembly-Like, denominati attraverso la struttura *R{register_id}*, dove *register_id* è un valore intero incrementale usato per distinguere diversi registri tra di loro.

Nome dell'istruzione	**Codice binario**
OPR	0000
OPV	0001
JMP	0010
JMPL	0011
JMPF	0100
JMPR	0101
RET	0110
IN	0111
OUT	1000
EXIT	1001
PUSH	1010
POP	1011

Tabella 6.2: Instruction set di *QRprogAssembly*

6.3.1 Registri

Come anche riportato in precedenza, *QRprog* sfrutta, per la corretta gestione delle variabili interne al programma di alto livello, dei registri definiti con una

struttura molto simile a quella di Assembly (`R{register_id}`). Così come succede per le variabili, anche i registri usati da *QRprog* sono di due tipi: normali e vettoriali. Un registro vettoriale è un registro particolare, con la struttura `R{register_id}[register/constant]`, che rappresenta una variabile di tipo vettoriale in cui `[register/constant]` rappresenta l'indice del vettore. Oltre alle variabili di tipo vettoriale, questo tipo di registri è anche utilizzato, in *QRprog*, per gestire le variabili matriciali, di qualsiasi dimensione; invece di usare registri di tipo matriciale, è infatti stato deciso di indicizzarle singolarmente utilizzando l'algoritmo denominato *row major*. Questo algoritmo permette di calcolare l'indice (in una rappresentazione ad array) corrispondente ad un elemento qualsiasi di una matrice multidimensionale, navigando all'interno di essa riga per riga.

In generale, per una matrice n-dimensionale, come la matrice di interi `int a[X_0][X_1]...[X_n]`, un suo elemento generico (come `a[x_0][x_1]...[x_n]`), viene indicizzato dall'indice `i`, di valore:

$$i = x\_n + X\_n * (x\_n - 1 + X\_n - 1 * (... + X\_1 * x\_0))$$

Ad esempio, l'elemento a[2][2] della matrice definita come a[3][4] (cioè di dimensione 3x4), sarà indicizzato come a[10], in quanto `i = 2 + 4*2`.

Un registro viene codificato attraverso il suo indice (intero senza segno), secondo la notazione esponenziale definita in Appendice A. Un registro vettoriale, invece, può essere indicizzato in due modi: tramite un altro registro oppure tramite un valore letterale. Nel primo caso, il registro vettoriale viene codificato dalla concatenazione della codifica dei due registri, mentre nel secondo caso esso viene codificato dalla concatenazione della codifica del registro e da quella della costante, come specificato in Sezione 6.2.1.

6.3.2 Label

Diversamente da quanto accade in *QRtree*, in *QRprog* è necessario implementare salti sia in avanti che indietro, così da permettere il supporto di costrutti di tipo loop. Una label rappresenta quindi il punto di codice a cui è necessario saltare nel caso in cui si stia effettuando un salto. Tuttavia, QRprog supporta la compilazione di programmi interni ad un eQR code che fanno riferimento ad altri programmi interni ad altri eQR code, senza il bisogno di concatenarli. Per questo, sono necessari due tipi di label: una label cosiddetta *interna*, che sarà responsabile di effettuare tutti i salti interni allo stesso eQR code, ed una cosiddetta *esterna*, che sarà responsabile dei salti esterni all'eQR code di cui essa fa parte. La rappresentazione intermedia, per risultare più leggibile, conterrà queste label con il loro nome; prima di essere codificate, esse saranno trasformate seguendo i seguenti passi: una label interna sarà riconosciuta tramite un nome identificativo autoincrementale all'interno del suo file, mentre una label esterna sarà riconosciuta da una stringa del tipo:

```
<uint, uint>
```

dove i due numeri interi rappresentano il file in cui saltare fra quelli disponibili ed il punto a cui saltare in quel file.

Ma mentre per una label esterna la codifica è semplice (essa viene codificata dalla codifica dei due numeri interi senza segno che la rappresentano secondo il formato esponenziale definito in Appendice A), lo stesso discorso non vale per una label interna. Infatti, una label interna, in fase di codifica viene dapprima trasformata in un numero intero, stavolta con segno, che rappresenta la quantità relativa del salto da effettuare, e solo successivamente questo numero viene codificato seguendo quanto riportato in Appendice A per i numeri con segno. Infine, al fine di distinguere i due tipi di label in fase di decodifica, la codifica di una label interna è introdotta da un bit settato a `0`, mentre la codifica di una label esterna è introdotta da un bit settato ad `1`.

6.3.3 OPR

L'istruzione `OPR` è l'istruzione che permette di dichiarare, inizializzare ed assegnare valori ai registri di tipo `non vettoriale`.

La struttura dell'istruzione è la seguente:

```
OPR <reg_dest>, <exp>[<type>]
```

L'istruzione `OPR` viene codificata con i primi 4 bit pari a `0000`, seguiti dalla codifica del registro destinazione e dalla codifica della catena di operazioni che compongono l'espressione `<exp>`.

L'operazione svolta da questa istruzione in fase di interpretazione è abbastanza semplice: il valore ottenuto dalla valutazione dell'espressione `<exp>` viene inserito all'interno del registro destinazione (`<reg_dest>`). Tuttavia, è opportuno notare come la complessità di questa istruzione derivi dal fatto che il parametro `<exp>` può rappresentare valori molto diversi tra di loro, a partire da semplici valori letterali (come nel caso di un'inizializzazione del tipo `int a = 0`) fino ad arrivare a complesse espressioni matematiche composte da più operazioni tra loro concatenate (ed espresse come riportato in Appendice B). In quest'ultimo caso, in fase di interpretazione dell'istruzione, l'interprete si occuperà dapprima della valutazione dell'espressione e solo successivamente il valore finale sarà inserito all'interno del registro.

Il parametro `<type>` è un parametro opzionale utilizzato solamente in caso di traduzione di un'operazione di inizializzazione di un registro. In questo caso, l'unica azione aggiuntiva svolta dall'interprete sarà che il registro, oltre ad ottenere il suo valore, verrà anche etichettato con il suo tipo. Tipici esempi di utilizzo dell'istruzione `OPR` sono rappresentati dalle inizializzazioni di variabili non vettoriali e da assegnazioni ad esse, come riportato nella seguente porzione di codice.

```
int a;                          OPR R0, 0
int b, c;                       OPR R1, 0
                                OPR R2, 0
float d;                        OPR R3, 0.0f16
b = 4;                          OPR R1, 4
c = 5;                          OPR R2, 5
a = (a + b) * 3;                OPR R0, R0 R1 PLUS 3 STAR
```

Listing 6.1: Esempio di utilizzo istruzione OPR

6.3.4 OPV

L'istruzione `OPV` è l'istruzione che permette di dichiarare, inizializzare ed assegnare valori ai registri di tipo `vettoriale`. La struttura dell'istruzione è la seguente:

`OPV <reg_dest>, <exp>[<type>]`

L'istruzione `OPV` viene codificata con i primi 4 bit pari a `0001`, seguiti dalla codifica del registro destinazione e dalla codifica della catena di operazioni che compongono l'espressione `<exp>`.

In fase di interpretazione delle istruzioni, il comportamento di questa istruzione è analogo a quello dell'istruzione `OPR`, con la sola differenza che `OPV` lavora con registri vettoriali. Tipici esempi di utilizzo dell'istruzione `OPV` sono rappresentati dalle inizializzazioni di variabili vettoriali e da assegnazioni ad esse, come riportato nella seguente porzione di codice.

```
int m[3][5];                OPV R0[15], 0
m[2][1] = 4;                OPV R0[11], 4
m[1][3] = 6;                OPV R0[8], 6
```

Listing 6.2: Esempio di utilizzo istruzione OPV

6.3.5 JMP

L'istruzione `JMP` è l'istruzione che permette di effettuare salti *incondizionati* verso una parte di codice introdotta da una *label*. La struttura dell'istruzione è la seguente:

`JMP <label>`

L'istruzione `JMP` viene codificata con i primi 4 bit pari a `0010`, seguiti dalla codifica della `<label>`.

In fase di interpretazione delle istruzioni, la `JMP` permette di interrompere il flusso corrente di esecuzione e di farlo riprendere in una posizione diversa, sia essa precedente o successiva rispetto a quella corrente.

Questa istruzione è l'istruzione di salto più generale, in quanto esegue il salto in modo incondizionato. Sono state definite, però, tre varianti di questa istruzione per la gestione di casi più particolari rispetto a quelli gestiti dalla `JMP`, riportate di seguito. Tipici esempi di utilizzo dell'istruzione `JMP` sono rappresentati dai salti incondizionati eseguiti dai costrutti `if`, `if-else`, `for` e `while`: in questi casi, insieme all'istruzione `JMP`, per gestire correttamente i vari flussi possibili dei costrutti, si trova anche l'istruzione `JMPL`. Per questo motivo, la porzione di codice contenente gli esempi di utilizzo di questa istruzione verrà mostrata nella Sottosezione seguente, corrispondente alla `JMPL`, in modo da fornire una panoramica più ampia della traduzione dei costrutti.

6.3.6 JMPL

L'istruzione `JMPL` è l'istruzione che permette di effettuare salti *condizionati* e dipendenti dal valore dell'ultima valutazione delle espressioni effettuate dalle istruzioni di `OPR` o `OPV`. La struttura dell'istruzione è la seguente:

```
JMPL <label>
```

L'istruzione `JMPL` viene codificata con i primi 4 bit pari a `0011`, seguiti dalla codifica della `<label>`.

In fase di interpretazione, questa istruzione controlla semplicemente il valore contenuto all'interno di una variabile, denominata `last_evaluation` e, se essa contiene un qualsiasi valore che può essere associato a `True` (cioè un valore diverso da 0), eseguirà l'istruzione `JMP`; in caso contrario, l'istruzione non avrà alcun effetto ed il flusso di esecuzione ripartirà dall'istruzione successiva. Come già riportato in precedenza, tipici esempi di utilizzo dell'istruzione `JMPL` sono rappresentati dai salti condizionati eseguiti dai costrutti `if`, `if-else`, `for` e `while`, riportati, insieme ai salti di tipo `JMP`, nella porzione di codice successiva.

```
while(i < 10) {                         L1: OPR R3, R0 10 GE
    a = a + 1;                              JMPL L2
    i = i + 1;                              OPR R2, R2 1 PLUS
}                                           OPR R0, R0 1 PLUS
                                            JMP L1

for(i = 0; i < 10; i = i+1) {           L2: OPR R0, 0
    a = a + 2;                          L3: OPR R4, R0 10 GE
}                                           JMPL L4
                                            OPR R2, R2 2 PLUS
                                            OPR R0, R0 1 PLUS
                                            JMP L3

if (j == 0) {                           L4: OPR R5, R1 0 NEQ
    a = a + 3;                              JMPL L5
} else {                                    OPR R2, R2 3 PLUS
```

```
17      a = a + 4;                                  JMP L6
18  }                                           L5: OPR R2, R2 4 PLUS
```

Listing 6.3: Esempio di utilizzo istruzioni JMP e JMPL

6.3.7 JMPF

L'istruzione `JMPF` è l'istruzione che permette di effettuare salti *incondizionati* verso una *label*, che però, a differenza di quanto accade nella `JMP`, rappresenta il punto di ingresso al codice di una funzione. La struttura dell'istruzione è la seguente:

`JMPF <label>`

L'istruzione `JMPF` viene codificata con i primi 4 bit pari a `0100`, seguiti dalla codifica della `<label>`.

In fase di interpretazione, questa istruzione aggiorna il contesto del flusso di esecuzione, indicando che le istruzioni che verranno eseguite successivamente appartengono ad un'altra funzione rispetto a quelle eseguite fino a quel momento. Ciò fa si che sia possibile avere registri con lo stesso nome all'interno di funzioni diverse, permettendo così di avere una codifica più leggera in fase di passaggio al *QRbytecode*. L'istruzione `JMPF` viene utilizzata per tradurre le chiamate a funzioni che vengono incontrate all'interno del programma ad alto livello, come riportato nella seguente porzione di codice.

```
1 [...]
2
3 function_call();              JMPF function_call
4
5 [...]
```

Listing 6.4: Esempio di utilizzo istruzione JMPF

6.3.8 JMPR

L'istruzione `JMPR` è l'istruzione che permette di effettuare salti *condizionati* e dipendenti dal valore del registro fornito come parametro. La struttura dell'istruzione è la seguente:

`JMPR <reg>, <label>`

L'istruzione `JMPR` viene codificata con i primi 4 bit pari a `0101`, seguiti dalla codifica del registro `<reg>` e da quella della `<label>`.

In fase di interpretazione, questa istruzione controlla semplicemente il valore contenuto all'interno del registro `<reg>` e, se esso contiene un qualsiasi valore che può essere associato a `True`, eseguirà l'istruzione `JMP`; in caso contrario, l'istruzione

non avrà alcun effetto ed il flusso di esecuzione ripartirà dall'istruzione successiva. Il tipico esempio di utilizzo dell'istruzione `JMPF` è rappresentato dalle condizioni dei costrutti formati da un solo registro, come riportato nella seguente porzione di codice.

```
while(a) {                  [...]
    [...]                   JMPR R0, L2
}                           [...]

if(a) {                     [...]
    [...]                   JMPR R0, L3
}                           [...]
```

Listing 6.5: Esempio di utilizzo istruzione JMPR

6.3.9 RET

L'istruzione `RET` è l'istruzione che permette di ritornare da una funzione precedentemente chiamata attraverso l'istruzione di `JUMPF`. La struttura dell'istruzione è la seguente:

```
RET
```

L'istruzione `RET` viene codificata utilizzando solamente i 4 bit, posti a `0110`, corrispondenti al codice dell'istruzione.

In fase di interpretazione, questa istruzione permette di ritornare al precedente contesto di esecuzione, eseguendo quella che è un'operazione inversa a quella eseguita dalla `JMPF`. È importante notare come questa istruzione non venga usata per segnalare la terminazione della funzione principale del programma. Per fare ciò, viene usata l'istruzione `EXIT` successiva.

L'istruzione `RET` è l'ultima istruzione di tutte le funzioni che non siano quella principale, identificata come `main`. Un semplice esempio del suo utilizzo può essere trovato nella seguente porzione di codice.

```
fn function_call() {
    [...]                   [...]
                            RET
}
```

Listing 6.6: Esempio di utilizzo istruzione RET

6.3.10 EXIT

L'istruzione `EXIT` è l'istruzione che rappresenta la fine della funzione principale del programma. La struttura dell'istruzione è la seguente:

```
EXIT
```

L'istruzione `EXIT` viene codificata utilizzando solamente i 4 bit, posti a `1001`, corrispondenti al codice dell'istruzione.

In fase di interpretazione, questa istruzione fa sì che il contesto di esecuzione venga completamente svuotato, terminando quindi il programma.

L'istruzione `EXIT` è l'ultima istruzione della funzione `main`. Un semplice esempio del suo utilizzo può essere trovato nella seguente porzione di codice.

```
fn main() {
    [...]                       [...]
                                EXIT
}
```

Listing 6.7: Esempio di utilizzo istruzione EXIT

6.3.11 IN

L'istruzione `IN` è l'istruzione che permette di richiedere all'utente l'inserimento di un valore in input. La struttura dell'istruzione è la seguente:

`IN <reg>`

L'istruzione `IN` viene codificata con i primi 4 bit pari a `0111`, seguiti da un bit che identifica il tipo del registro `<reg>` e dalla codifica del registro `<reg>`. Se il bit rappresentante il tipo del registro è settato a 0, `<reg>` sarà un registro normale, mentre se esso è settato ad 1, `<reg>` sarà un registro vettoriale.

In fase di interpretazione, questa istruzione può, prima di richiedere l'inserimento dell'input all'utente, stampare a video una stringa predefinita dallo sviluppatore all'interno dell'interprete. In seguito all'input del valore da parte dell'utente, quel valore sarà inserito all'interno del registro `<reg>` e le opportune operazioni di conversione saranno effettuate in base al tipo di registro in cui il dato inserito deve essere memorizzato.

Il linguaggio ad alto livello definito per l'implementazione di `QRprog` supporta un'operazione di input chiamata `input`: pertanto, il tipico utilizzo dell'istruzione `IN` è riportato nella seguente porzione di codice.

```
[...]

int a;                      OPR R0, 0
input a;                    IN R0

[...]
```

Listing 6.8: Esempio di utilizzo istruzione IN

6.3.12 OUT

L'istruzione `OUT` è l'istruzione che permette di stampare valori di output. La struttura dell'istruzione è la seguente:

`OUT [<val>/<reg>/<reg_list>] [<stringa_di_formato>]`

Come si può notare dalla struttura peculiare di questa istruzione rispetto alla sua controparte di input, l'istruzione di `OUT` supporta due tipi diversi di output, quello formattato e quello non formattato. Un output è non formattato quando il valore da stampare a schermo è semplicemente un valore letterario oppure un valore contenuto all'interno di un registro (sia esso semplice o vettoriale). In questo caso la struttura dell'istruzione sarà la seguente:

`OUT <val>/<reg>`

In fase di interpretazione, l'esecuzione di questa istruzione visualizzerà semplicemente il valore letterale `<val>` oppure il valore contenuto all'interno del registro `<reg>`.

Quando si usa un output formattato, invece, oltre a fornire i registri di cui si vogliono visualizzare i valori, deve essere fornita anche una stringa contenente una frase in cui alcune parti sono mancanti e rimpiazzate da sottostringhe del tipo `%[dsf]`, alle quali, al momento dell'esecuzione dell'istruzione, verranno sostituiti i valori contenuti nei registri nello stesso ordine in cui essi sono stati forniti. In questo caso la struttura dell'istruzione sarà la seguente:

`OUT <reg_list> <stringa_di_formato>`

In fase di interpretazione, l'esecuzione di questa istruzione visualizzerà la stringa `<stringa_di_formato>`, all'interno della quale, nelle posizioni in cui sono state inserite le sottostringhe introdotte in precedenza, saranno stati inseriti i valori dei registri contenuti in `<reg_list>`.

La seguente porzione di codice mostra un esempio di utilizzo dell'istruzione `OUT`, sia nel caso di output formattato sia in quello di output non formattato.

```
1  int a, b, c;                          OPR R0, 0
2                                        OPR R1, 0
3                                        OPR R2, 0
4  print "START";                        OUT "START"
5  a = 4;                                OPR R0, 4
6  b = 5;                                OPR R1, 5
7  c = a + b;                            OPR R2, R0 R1 PLUS
8
9  printf "%d + %d = %d", a, b, c;       OUT R0, R1, R2 "%d + %d = %d"
10 print "END";                          OUT "END"
```

Listing 6.9: Esempio di utilizzo istruzione OUT

6.3.13 PUSH

Come già introdotto in precedenza parlando dell'istruzione `JMPF`, *QRprog* permette anche di avere funzioni diverse da quella principale a cui è possibile saltare durante l'esecuzione del programma. Tuttavia, una delle operazioni più richieste quando si tratta della chiamata di funzioni è quella del passaggio di argomenti a questa nuova funzione.

Questa operazione viene gestita dall'istruzione `PUSH`, che permette di passare i valori contenuti all'interno di una serie di registri alla nuova funzione chiamata dalla `JMPF`. La struttura dell'istruzione è la seguente:

```
PUSH <[reg / literal]_list>
```

L'istruzione `PUSH` viene codificata con i primi 4 bit pari a `1010`, seguiti dalla codifica della lista di argomenti che compongono `<[reg / literal]_list>`.

In fase di interpretazione, questa istruzione inserisce i valori contenuti nei registri della `<[reg / literal]_list>` all'interno di uno stack mantenuto dall'interprete, che verrà in seguito usato, all'interno della funzione chiamata, attraverso l'istruzione di `POP`, per fare in modo che essa possa accedere ai valori passato come parametri all' istruzione `JMPF`. Un tipico esempio di utilizzo di questa istruzione è riportato di seguito.

```
fn main() {                                 OPR R0, 0
    int a, b, c;                            OPR R1, 0
                                            OPR R2, 0
    [...]                                   [...]
                                            PUSH R1, R2
    a = mult(b, c);                         JMPF mult
    [...]                                   [...]
}

fn mult(int b, int c) {
                                            [...]
    int product;                            OPR R2, 0
    product = b * c;                        OPR R2, R0 R1 STAR
                                            PUSH R2
    return product;                         RET
}
```

Listing 6.10: Esempio di utilizzo istruzione PUSH

6.3.14 POP

L'istruzione di `POP` è l'istruzione che permette di accedere ai valori contenuti all'interno dello *stack* usato dall'istruzione `PUSH`. La struttura dell'istruzione è la seguente:

```
POP <[reg / literal]_list>
```

L'istruzione `POP` viene codificata con i primi 4 bit pari a `1011`, seguiti dalla codifica della lista di argomenti che compongono `<[reg / literal]_list>`.

Tipicamente, questa istruzione è la prima che viene chiamata all'interno di una funzione chiamata dalla `JMPF` che riceve dei valori come argomenti. In fase di interpretazione, essa permette di inserire i valori ottenuti dallo *stack* (cioè quelli inseriti dalla `PUSH`) all'interno dei registri contenuti all'interno di `<reg_list>`. Un tipico esempio di utilizzo di questa istruzione è riportato di seguito.

```
1  fn main() {
2      [...]                                [...]
3      a = mult(b, c);                      JMPF mult
4                                           POP R0
5      [...]                                [...]
6  }
7
8  fn mult(int b, int c) {                  POP R0, R1
9      [...]                                [...]
10 }
```

Listing 6.11: Esempio di utilizzo istruzione POP

Capitolo 7

Implementazione

Questa sezione si occuperà di presentare le parti principali del software realizzato a supporto del lavoro di questa tesi. Usando gli strumenti presentati nel Capitolo 3 e sfruttando le capacità del linguaggio Python, sono state implementate le varie parti di QRscript presentate in Figura 4.1. In particolare, facendo riferimento alla Figura 4.1, sono stati implementati, per il lavoro di questa tesi, i punti 2 e 5 per quanto riguarda il dialetto *QRtree* ed i punti 1 e 6 per quanto riguarda il dialetto *QRprog*. I restanti punti intermedi del processo (esclusi i punti 3 e 4 che saranno presentati alla fine di questa Sezione), sono stati svolti all'interno della tesi di Matteo Rosani. A causa del livello di complessità dato soprattutto dalle varie parti di header presenti all'interno della specifica di QRscript, il codice implementato durante lo svolgimento di questa tesi presenta alcune limitazioni rispetto a quanto descritto teoricamente. In particolare, per quanto concerne *QRtree*, il software è stato implementato considerando un header contenente tutte le specifiche discusse in Sezione 4.2.1 disattivate e con il *QRtree header* disattivato.

7.1 QRtree

Il lavoro di questa tesi, per quanto concerne questo particolare dialetto, si è concentrato sulla traduzione da *QRtreeAssembly* a *QRtreebytecode* e viceversa, implementando così i punti 2 e 5 del processo di funzionamento di *QRscript*.

Tuttavia, al fine di dare una panoramica generale su tutto il lavoro, è opportuno discutere brevemente dell'intero progetto. Seguendo le varie fasi definite in Figura 4.1, il progetto è stato diviso dapprima in due macro-fasi, chiamate `encode` e `decode`, corrispondenti alle fasi di *generazione* ed *esecuzione* di *QRscript*; queste due macro-fasi sono state, a loro volta, divise in 3 sotto-fasi, in tutto e per tutto corrispondenti alle 6 frecce presenti in figura. In particolare, la fase di `encode` è formata da una prima sotto-fase, chiamata `HighLevelToIntermediate`, il cui obiettivo è la traduzione da linguaggio ad alto livello a *QRtreebytecode*, seguita da una

seconda sotto-fase, detta `IntermediateToeQRbytecode` (descritta dettagliatamente in Sezione 7.1.1) e conclusa con la creazione dell'eQR code basato sul *QRtreebytecode* ottenuto al punto precedente (`eQRbytecodeToeQRcode`). La fase di `decode` è, invece, formata da una prima sotto-fase, chiamata `eQRcodeToeQRbytecode`, che legge l'eQR code e riottiene il corrispondente *QRtreebytecode*, seguita da una seconda sotto-fase, detta `eQRbytecodeToIntermediate` (descritta dettagliatamente in Sezione 7.1.2) e conclusa con un software simile ad una macchina virtuale che permette l'esecuzione della rappresentazione intermedia ottenuta al punto precedente (`IntermediateToHTML`). In questo caso, è stato scelto di eseguire la rappresentazione intermedia tramite la creazione e l'esecuzione di una pagina `HTML`, ritenuta lo strumento più adatto per questo particolare caso di applicazione. L'esecuzione, infatti, prevede un'interazione con l'utente basata su messaggi che vengono stampati a schermo e input inseribili tramite tastiera oppure tramite bottoni di scelta.

7.1.1 IntermediateToeQRbytecode: da QRtreeAssembly a QRtreebytecode

Il modulo `IntermediateToeQRbytecode`, che ha il compito di tradurre *QRtreeAssembly* in *QRtreebytecode*, è composto da tre file principali: un file di gestione generale, che si occupa semplicemente di coordinare il flusso di esecuzione del modulo, uno Scanner ed un Parser, che sfruttano le potenzialità descritte in Sezione 3.2.

Lo Scanner definisce i token necessari per il riconoscimento di tutti i simboli da cui può essere composta la rappresentazione intermedia. In particolare, per quanto riguarda *QRtree*, essi sono:

- i nomi delle istruzioni dell'instruction set di *QRtree*, cioè tutte quelle istruzioni definite nel dettaglio in Sezione 5.4
- gli operatori relazionali descritti in Sezione 5.4.9
- i tipi di dato supportati da *QRtree*, cioè numeri interi, numeri reali e stringhe
- le parentesi tonde (aperte e chiuse), utili nella struttura della rappresentazione intermedia per definire i punti di salto in caso di istruzioni quali `goto`, `if` o `ifc`.

Oltre a questi token, specifici per il linguaggio da processare, lo Scanner ne definisce alcuni, standard, che solitamente si trovano in tutti i file di questo tipo ed aiutano una più rapida e corretta valutazione di tutti i simboli. Infatti, esso specifica il comportamento di *skip* o *ignore* (che fa sì che lo Scanner non reagisca in alcun modo alla lettura di un particolare simbolo), per alcuni caratteri particolari, molto comuni ma poco utili al fine dell'analisi della sintassi, quali gli spazi, i newline, e il carattere di end-of-file (eof). Infine, come da best practices, esso definisce un token

di errore, detto *error*, che permette di individuare qualsiasi simbolo sconosciuto (cioè non rientrante fra nessun degli altri definiti dallo Scanner) e di segnalarlo come estraneo al particolare linguaggio.

Tutti i token definiti all'interno dello Scanner sono infine inseriti all'interno di un vettore, detto *tokens*, fondamentale in quanto esso sarà, attraverso il file di gestione, passato al Parser che lo userà per il controllo della sintassi della rappresentazione intermedia.

Il Parser, invece, definisce al suo interno principalmente due elementi: tutti i dati e le funzioni di supporto che gli serviranno per effettuare la traduzione, e le regole grammaticali del dialetto *QRtree*. Sfruttando questi dati ed ottenendo come parametri di input il nome del file su cui verrà trascritta la traduzione ed i risultati della computazione dello Scanner, il Parser riporterà, all'interno del file di output, il risultato della traduzione, espresso, in questo caso, in binario.

In particolare, fra i dati di supporto usati dal Parser, troviamo:

- la definizione del carattere *end-of-text (EOT)* per entrambe le codifiche possibili delle stringhe: `ASCII-7 (0000011)` e `UTF-8 (00000011)`
- la funzione `stringEncoding(...)` che permette la codifica delle stringhe incontrate all'interno della rappresentazione intermedia nel formato che a loro più si addice
- le funzioni, riportate in Appendice A, per la codifica delle referenze e dei valori dei salti
- la funzione `twos_complement_binary(...)` che permette la codifica i numeri interi secondo il formato *CA2*

Le regole grammaticali, invece, descrivono la corretta sintassi supportata da *QRtree*. In particolare:

- ogni programma QRtree è definito come una lista di operazioni
- un'operazione è definita come segue:

  `( <n_linea> ) <istruzione>`

  dove `n_linea` è il numero della linea in cui è contenuta l'istruzione, usato come riferimento per l'esecuzione dei salti e `<istruzione>` è una delle istruzioni definite in Sezione 5.4
- ogni istruzione è definita come descritto in Sezione 5.4

Ad ogni regola grammaticale di definizione delle istruzioni è associata un'azione particolare. Questa azione dapprima raccoglie tutti i componenti facenti parte dell'istruzione e, successivamente, costruisce e stampa sul file di output la rappresentazione della traduzione usando le corrispondenze fondamentali definite in Sezione 5.4 e le funzioni di supporto elencate in precedenza.

Un esempio di regola grammaticale con relativa azione associata, è presentata nella porzione di codice successiva.

```
def p_input(self,p):
        '''
        input : INPUT constant
        '''

        if isinstance(p[2], int):
            p[0] = '0001' + self.referenceEncoding(p[2])
        else:
            p[0] = '0000' + self.stringEncoding(p[2])
        self.output.write(p[0])
```

Listing 7.1: Esempio di regola grammaticale per le istruzioni

Come si può notare dalla porzione di codice precedente, l'azione riconosce se la costante `<constant>` è una referenza (numero intero) oppure no e, in base al risultato di questo riconoscimento, chiama la funzione corrispondente per la sua corretta codifica. La codifica finale è quindi rappresentata dalla concatenazione delle codifiche fondamentali con le codifiche ottenute dalle funzioni. Questa logica vale per tutte le istruzioni, anche per quelle più complesse, come quelle di `if` e `ifc`.

7.1.2 eQRbytecodeToIntermediate: da QRtreebytecode a QRTreeAssembly

Il modulo `eQRbytecodeToIntermediate`, che ha il compito di tradurre *QRtreebytecode* in *QRtreeAssembly*, è composto da tre file principali: un file di gestione generale, che si occupa semplicemente di coordinare il flusso di esecuzione del modulo, uno Scanner ed un Parser, che sfruttano le potenzialità descritte in Sezione 3.2.

Lo Scanner definisce i token necessari per il riconoscimento di tutti i simboli da cui può essere composta la rappresentazione binaria. In questo caso, lavorando con una rappresentazione puramente binaria, il set di token principale è particolarmente semplice, ed è formato solamente dal carattere `0` e dal carattere `1`.

Tuttavia, mentre la costruzione della lista di token principali risulta ovvia, questo tipo di rappresentazione presenta una difficoltà decisamente rilevante dal punto di vista dell'analisi semantica: infatti, mentre nel caso precedente il Parser aveva la possibilità di lavorare con molti token diversi (rendendo il riconoscimento della semantica del linguaggio più immediata), in questo caso il Parser è costretto a lavorare con due soli token, i quali però possono produrre un insieme abbastanza

ampio di sintassi diverse (aumentando la probabilità ed il numero di ambiguità del linguaggio). Per questo motivo, all'interno dello Scanner, sono stati definiti degli *stati*. Uno stato si riferisce alla condizione corrente del processo di compilazione e, all'interno dello Scanner, è possibile definire dei token in modo tale che essi siano validi sono nel caso in cui il processo di compilazione si trovi in un determinato stato. Quando quindi il Parser arriva a gestire una regola che può essere ambigua, esso può istruire lo Scanner di passargli, da quel momento fino al prossimo cambio di stato, solamente i token che fanno parte dello stato corrente.

Nel caso particolare di questo modulo, questa funzionalità è stata fondamentale per:

- capire, in base ai bit precedenti la stringa, se essa fosse codificata in `ASCII-7` o `UTF-8` e, quindi, quanti bit leggere per ottenere ogni carattere. A questo scopo, è stato definito un token `BYTE` sia nello stato `ascii-7` (formato da 7 bit) che nello stato `utf-8` (formato da 8 bit)

- capire, in base ai bit precedenti ad un numero, se esso fosse codificato su 16 o 32 bit e, conseguentemente, quanti bit leggere per ottenerlo. A questo scopo, è stato definito un token `NUMBER` sia nello stato `n16` (formato da 16 bit) che nello stato `n32` (formato da 32 bit)

- capire, in base ai bit precedenti ad una referenza o ad un salto, in quanti bit essi fossero codificati. A questo scopo, è stato definito uno stato `ref`, all'interno del quale gli unici token validi sono quelli che identificano le referenze codificate su 4, 8, 16 o 32 bit, che studia i bit in base alla definizione della notazione esponenziale in Appendice A e recupera il valore della referenza/salto

Il discorso fatto nella Sezione precedente riguardante i token standard definiti all'interno degli Scanner è ancora valido, ma con una precisazione: questi token speciali devono essere riconoscibili da qualunque stato; per questo motivo essi sono definiti all'interno di uno stato particolare, detto `any`, che fa sì che quel particolare token a cui si riferisce possa essere riconosciuto come valido da tutti gli stati.

Come nel caso precedente, il Parser definisce al suo interno principalmente due elementi: tutti i dati e le funzioni di supporto che gli serviranno per effettuare la traduzione, e le regole grammaticali derivanti dalla traduzione del dialetto *QRtree* (queste regole sono praticamente l'inverso delle regole descritte nella Sezione precedente). Sfruttando questi dati ed ottenendo come parametri di input il nome del file su cui verrà trascritta la traduzione ed i risultati della computazione dello Scanner, il Parser riporterà, all'interno del file di output, il risultato della traduzione, espresso, in questo caso, in *QRtreeAssembly*.

In particolare, fra i dati di supporto usati dal Parser, troviamo:

- un contatore, chiamato `curline`, che permette di ricavare l'indice che, in *QRtree* è posto di fianco ad ogni istruzione e ne denota la riga all'interno della rappresentazione intermedia

- la funzione `binStrToStrAscii(...)` che permette la decodifica delle stringhe `ASCII-7`
- la funzione `binStrToStrUtf(...)` che permette la decodifica delle stringhe `UTF-8`
- le funzioni, riportate in Appendice A, per la decodifica delle referenze e dei valori dei salti
- la funzione `from_twos_complement_binary(...)` che permette la decodifica dei numeri interi secondo il formato *CA2*

Le regole grammaticali, invece, descrivono la corretta sintassi che dovrebbe avere una rappresentazione binaria ottenuta dalla traduzione di un programma di tipo *QRtree*. In particolare:

- ogni programma è definito come una lista di istruzioni
- ogni istruzione è definita attraverso il suo codice di 3 bit, come specificato in Sezione 5.4, seguito dai suoi vari componenti
- ogni componente è riconosciuto usando una logica inversa rispetto a quella definita per la sua codifica. Ad esempio, nell'istruzione `print`, un bit settato a `0`, successivo al codice `010` (rappresentante l'istruzione), denota che il componente successivo è da intendersi come stringa, ed il bit ancora successivo identifica la codifica usata per la traduzione della stringa.
- ottenute tutte queste informazioni, sfruttando le funzioni di supporto e gli stati descritti in precedenza, le azioni associate a ciascuna regola grammaticale ricostruiscono l'istruzione, riportandola alla stessa struttura che aveva prima della traduzione in binario

Un esempio di regola grammaticale con relativa azione associata, corrispondente alla stessa istruzione riportata nell'esempio 7.1, è presentata nella porzione di codice successiva.

```
1  def p_input(self,p):
2      '''
3      input : ZERO ZERO ZERO ZERO constant
4            | ZERO ZERO ZERO ONE number
5      '''
6
7      if(p[4] == '0'):
8          if(p[5][0] == "00"):
9              self.output.write("(" + str(self.curline) + ") input " +
       '"' + self.binStrToStrAscii(p[5][1]) + '"' + '\n')
10         else:
```

```
11          self.output.write("(" + str(self.curline) + ") input " + '"' + self.binStrToStrUtf(p[5][1]) + '"' + '\n')
12      else:
13          self.output.write("(" + str(self.curline) + ") input " + str(self.binRefToIntRef(p[5])) + '\n')
14      self.curline += 1
```

Listing 7.2: Esempio di regola grammaticale per le istruzioni

Come si può notare dalla porzione di codice precedente, l'azione è in grado di riconoscere, attraverso i token letti ed i dati che vengono ritornati per ogni componente, la natura di questi ultimi (siano essi stringhe, referenze e affini o numeri non rappresentanti referenze) e, per ciascuno di essi, chiama la funzione corrispondente per la sua corretta decodifica. La decodifica finale è quindi rappresentata dalla concatenazione delle decodifiche fondamentali con le decodifiche ottenute dalle funzioni. Questa logica vale per tutte le istruzioni, anche per quelle più complesse, come quelle di `if` e `ifc`.

7.2 QRprog

Invece, per quanto riguarda il dialetto *QRprog*, il lavoro svolto per questa tesi si è concentrato sulla traduzione da linguaggio ad alto livello a *QRprogAssembly* e sulla successiva interpretazione di quest'ultimo dopo la lettura dell'eQR code. Sono stati quindi implementati i punti 1 e 6 del processo di funzionamento di *QRscript*. La panoramica del progetto è quasi totalmente equivalente a quella fatta per *QRtree*. L'unica differenza risiede nella scelta di come viene effettuata l'esecuzione della rappresentazione intermedia; mentre per *QRtree* la scelta è ricaduta su una pagina `HTML`, nel caso di *QRprog* è stato scelto di sfruttare un `interprete`, che permette l'esecuzione diretta del codice corrispondente alla rappresentazione intermedia. Le sezioni successive si occuperanno di esporre in maniera dettagliata le due fasi implementate: in particolare, la Sezione 7.2.1 tratterà la parte di traduzione da linguaggio ad alto livello a *QRprogAssembly*, mentre la Sezione 7.2.2 si concentrerà sul processo di interpretazione delle istruzioni di *QRprogAssembly*, che ha la finalità di ottenere, per la rappresentazione intermedia fornita, un comportamento equivalente a quello che si otterrebbe eseguendo il programma ad alto livello in un compilatore/interprete specializzato.

7.2.1 QRprog: da linguaggio di alto livello a QRprogAssembly

Il modulo `HighLevelToIntermediate` ha il compito di tradurre il programma scritto nel linguaggio ad alto livello in *QRprogAssembly*. Prima di introdurre il modulo ed i suoi dettagli implementativi, è necessario però fare una considerazione: per

linguaggio ad alto livello si intende, in generale, un qualsiasi linguaggio di programmazione di livello più alto rispetto all'Assembly. In questa categoria ricadono un numero enorme di linguaggi di programmazione, ognuno più o meno diverso dagli altri; risulta quindi evidente che definire un traduttore "universale" sia una soluzione inattuabile dal punto di vista pratico. Al fine di implementare un esempio delle funzionalità che *QRprog* vuole offrire, è stato quindi definito un semplice linguaggio ad alto livello, con una struttura molto simile al C, ma con un range di istruzioni più limitato. Il linguaggio ad alto livello definito durante il lavoro svolto per questa tesi, infatti, supporta:

- i tipi definiti in Sezione 6.2.1
- dichiarazione di ed assegnazioni a variabili
- i principali operatori (i.e., >, <, >=, <=, =, +, -, *, /, AND, OR, NOT, ^ e %)
- le istruzioni di input (tramite `input`) e di output (tramite `print` e `printf`)
- i costrutti `if`, `if-else`, `for` e `while`
- l'introduzione di funzioni (tramite `fn`) ed il ritorno da funzioni (tramite `return`)
- un'operazione di import di funzioni da altri file, simile a quella di Javascript, tramite le keyword `import`, `from` e `as`

Dal punto di vista implementativo, la composizione di questo modulo è simile a quella vista per i moduli precedenti di *QRtree*: un file di gestione generale si occupa di coordinare il flusso del modulo, mentre uno Scanner ed un Parser si occupano materialmente della traduzione.

I token definiti dallo Scanner corrispondono, con la sola aggiunta della punteggiatura, alla lista di elementi supportati dal linguaggio appena riportata.

Il Parser, invece, definisce al suo interno le regole grammaticali per la traduzione da linguaggio ad alto livello a rappresentazione intermedia. A questo scopo, esso definisce alcuni dati e funzioni di supporto, tra i quali si trovano:

- alcune funzioni di incremento di contatori che hanno lo scopo di definire gli indici incrementali da assegnare a *registri* e *label*
- le funzioni per la corretta gestione degli import di funzioni da altri file
- le funzioni per l'indicizzazione delle matrici con la tecnica *row major*

Le regole grammaticali, invece, descrivono la corretta sintassi supportata da QRprog. In particolare:

- ogni programma QRprog è definito come una lista di import seguita da una lista di funzioni
- la lista di import è una sequenza, formata da 0 o più elementi, di operazioni di `import`, con una struttura molto simile a quella di Javascript
- ogni funzione appartenente alla lista ha un insieme di argomenti di input ed un body formato da una lista di operazioni
- ogni operazione è una combinazione degli elementi supportati dal linguaggio definiti in precedenza

Ad ogni regola grammaticale di definizione delle operazioni è associata un'azione particolare. Questa azione dapprima raccoglie tutti i componenti facenti parte dell'istruzione e, successivamente, costruisce e stampa sul file di output la rappresentazione della traduzione usando le corrispondenze definite in Sezione 6.3 e le funzioni di supporto elencate in precedenza.

7.2.2 QRprog: interpretazione di QRprogAssembly

Il modulo `eQRbytecodeInterpreter`, che ha il compito di interpretare la rappresentazione intermedia *QRprogAssembly*, è anch'esso composto da tre file: uno Scanner ed un Parser, i quali, equivalentemente ai loro corrispettivi file per gli altri moduli, sfruttano le potenzialità descritte in Sezione 3.2 ed un terzo file che rappresenta il vero e proprio interprete. Lo Scanner definisce tutti i token necessari per il riconoscimento della rappresentazione intermedia denominata *QRprogAssembly*, ovvero: le istruzioni appartenenti all'instruction set di *QRprog*, definite in Sezione 6.3, gli operatori e le costanti (tipi e valori) supportati, introdotti in Sezione 6.2.1, i registri, le label ed i simboli di punteggiatura. Il Parser, invece, è in questo caso leggermente diverso dagli altri Parser introdotti in precedenza: esso, infatti, seppur definisca, come tutti gli altri, le regole grammaticali per il riconoscimento semantico del dialetto *QRprog*, produce un output molto diverso: invece che riportare su un file una traduzione da un linguaggio ad un altro, questo Parser si limita a costruire una mappa di informazioni, che verrà in seguito usata dall'interprete per rendere più agevole il suo compito di esecuzione del codice. Questa mappa generata ha una struttura particolare, definita qui di seguito:

- il primo elemento della mappa, con chiave `labels`, è a sua volta una mappa, contenente le corrispondenze fra le label incontrate all'interno della rappresentazione intermedia e l'indice della loro prima istruzione all'interno del secondo elemento della mappa
- il secondo elemento, con chiave `instructions`, è infatti un array all'interno del quale vengono immagazzinate tutte le istruzioni che dovranno essere

eseguite dall'interprete. Queste istruzioni vengono memorizzate tramite una particolare rappresentazione in forma testuale, contenente tutte le informazioni necessarie per la loro esecuzione. Nella maggior parte dei casi, questa trasformazione non ha effetti molto evidenti sulla struttura dell'istruzione, ed il passaggio interno al Parser è eseguito puramente al fine di controllare la rappresentazione intermedia dal punto di vista semantico. Per esempio, l'istruzione *QRprogAssembly*

OPR R1, R0 1 MINUS

corrispondente ad un'assegnazione al registro `R1`, del valore `R0 - 1`, sarà rappresentata, all'interno di questo elemento, come:

OPR R1, R0 1 Operand.MINUS

Tornando alle regole grammaticali, esse sono, in questo caso, piuttosto semplici, e si limitano a definire un programma di tipo *QRprog* come una lista di istruzioni appartenenti all'instruction set del dialetto, come definite in Sezione 6.3. Le azioni associate a queste regole grammaticali si occupano invece di costruire la rappresentazione testuale che verrà inserita all'interno della mappa di output.

Il file contenente il vero e proprio interprete, quindi, non deve fare che chiamare l'esecuzione dello Scanner e del Parser, in modo da ottenere la mappa corrispondente alla rappresentazione intermedia da eseguire ed iniziarne l'esecuzione dalla prima istruzione. Per questo scopo, l'interprete utilizza due classi: la classe principale, chiamata `Interpreter`, contiene tutti i metodi necessari per l'esecuzione delle istruzioni: per esempio, per l'istruzione `OPR`, questa classe definisce la funzione `execute_opr` come segue:

```
1 def execute_opr(self, args):
2     self.last_evaluation = self.evaluateExpression(args[1].copy())
3     self.setRegisterValue(args[0], self.last_evaluation)
```

Listing 7.3: Funzione per l'esecuzione dell'istruzione `OPR`

dove la funzione `evaluateExpression(...)` valuta l'espressione contenuta all'interno della rappresentazione testuale, sostituendo eventuali registri con i loro valori ed eseguendo tutte le operazioni, mentre la funzione `setRegisterValue(...)` assegna al registro destinazione il valore ottenuto dalla funzione precedente.

La seconda classe, interna ad `Interpreter`, denominata `ExecutionContext`, viene usata per rappresentare il contesto di esecuzione corrente, che include il file corrente, il numero di istruzione corrente ed i registri correnti. Questo contesto è necessario per la gestione dell'esecuzione di diverse funzioni, indipendenti tra di loro, appartenenti allo stesso o a diversi file.

L'interpretazione inizia eseguendo il metodo `execute(...)` dell'interprete, che inizia l'esecuzione delle istruzioni nel contesto corrente. L'interprete cicla attraverso le istruzioni del file corrente, richiamando il metodo appropriato per ciascuna istruzione. Durante l'esecuzione, il contesto corrente può cambiare a causa di salti o chiamate di funzione, e l'interprete gestisce il contesto corrente in modo appropriato.

7.3 QRScript: da QRbytecode a eQR code e viceversa

La parte che accomuna i due lavori presentati in questo capitolo è la parte di codifica e di decodifica del QR code, fondamentale per il funzionamento di *QRtree* e *QRprog*.

La parte di codifica consiste principalmente nella preparazione e nell'inserimento dei dati all'interno del QR code. In questa fase risulta delicata la decisione relativa alla modalità di input con cui i dati verranno inseriti all'interno del codice. Come già specificato in Sezione 2.2.1, lo standard che definisce i QR code supporta quattro tipi di codifica (numerica, alfanumerica, binaria e kanji). Durante la fase di implementazione di *QRscript* è stato valutato l'uso di due di queste: quella numerica e quella binaria, entrambe con i propri vantaggi e svantaggi.

Infatti, mentre le codifiche di tipo kanji ed alfanumerica si sono rilevate da subito non particolarmente adatte per le funzionalità richieste da questo progetto, la scelta fra le altre due si è rivelata tutt'altro che semplice.

Partendo dal fatto che un eQR code incapsula al suo interno dei dati derivanti da un file di tipo binario (contenente *QRbytecode*) potrebbe sembrare ovvio che la codifica da usare sia quella binaria. Tuttavia, mentre è vero che la codifica binaria si addice particolarmente al tipo di dati di *QRbytecode*, è altrettanto importante notare come, rispetto alla codifica numerica, sia meno efficiente in termini di spazio di archiviazione. Fatta questa considerazione, allora, potrebbe venire in mente che la codifica numerica sia quella più adatta, in quanto, come discusso molte volte all'interno di questa tesi, uno dei principali obiettivi di *QRscript* è quello di rendere la codifica interna ai QR code il più compatta possibile. Tuttavia, anche in questo caso, ci trova a dover affrontare un problema non indifferente: per utilizzare la codifica di tipo numerico, è necessario che tutti i dati, prima di essere inseriti all'interno del QR code, siano convertiti in numeri decimali. Ma se lo stream di dati da inserire nell'eQR code inizia con una sequenza di uno o più bit settati a 0 (caso tutt'altro che raro, come si può dedurre dalla descrizione di *QRscript header* in Sezione 4.2.1) è possibile, se non probabile, che questa sequenza venga interpretata in modo non corretto oppure proprio rimossa, stravolgendo il significato del *QRbytecode*, che non potrà quindi essere recuperato in modo corretto.

A seguito di tutte queste considerazioni, quindi, è stato deciso di lasciare, come aderente alle specifiche di *QRscript*, entrambi i tipi di codifica, con la scelta ricadente sullo sviluppatore dell'applicazione. A questo proposito, è consigliato usare la codifica numerica per tutti quei *QRbytecode* che iniziano sicuramente con un bit a `1`, ed usare la codifica binaria per tutti gli altri.

Nel caso specifico del lavoro svolto in questa tesi, essendo necessario rimanere il più generali possibile, è stato deciso di utilizzare la codifica di tipo binario.

Il modulo denominato *eQRbytecodeToeQRcode* contiene al suo interno la funzione che permette di creare l'eQR code partendo dal nome del file binario in cui è contenuto il *QRbytecode*.

Questa funzione svolge, principalmente, tre compiti:

1. legge il *QRbytecode* contenuto all'interno del file il cui nome è passato come parametro e costruisce i dati da inserire all'interno dell'eQR code (come già accennato nella parte di limitazioni del software, la parte di header viene inserita "artificialmente" in quanto non implementata)

2. usa il metodo `bytes(...)` di Python per convertire i dati appena ottenuti in uno stream di byte

3. sfrutta le potenzialità della libreria `qrcode` per costruire l'eQR code e lo salva in un file *png*

La parte di decodifica, invece, consiste principalmente nella lettura e nella pulizia dei dati contenuti all'interno dell'eQR code.

Il modulo denominato *eQRcodeToeQRbytecode* contiene al suo interno la funzione che permette di leggere l'eQR code e ricreare un file, contenente il *QRbytecode*, equivalente a quello usato per la generazione del codice.

Questa funzione svolge, principalmente, tre compiti:

1. sfrutta le librerie `opencv2` e `pyzbar` per leggere l'*eQR code*, ottenendo come risultato uno stream di byte equivalente a quello ottenuto al punto 2 della codifica

2. trasforma, usando il metodo `_format_bytes`, lo stream di byte in una stringa binaria e la ripulisce da tutti quei dati aggiunti al punto 1 della codifica (padding ed header)

3. salva la nuova stringa binaria in un file con estensione *bin*.

La parte più interessante di questo processo è sicuramente la funzione `_format_bytes`. Questa funzione, necessaria solamente in caso di utilizzo della codifica di tipo binario, è stata sviluppata per far fronte ad un problema molto particolare: la codifica binaria dei QR code utilizza il formato di carattere ISO/IEC 8859-1[27], noto anche come Latin-1. Questo formato di carattere è un set di

caratteri di codifica a 8 bit, per cui le rappresentazioni binarie dei suoi caratteri possono arrivare fino al numero 255. Tuttavia, durante la codifica e l'inserimento dei dati all'interno del QR code, per rappresentare tutti i caratteri con codice superiore a 127, esso ha bisogno di due byte (uno di controllo + uno di valore) e *QRbytecode* è uno stream di bit che potremmo definire "pseudo-casuali", per cui non è possibile garantire che, nella costruzione dello stream di byte da inserire all'interno dell'eQR code, non siano presenti byte non correttamente riconoscibili da `ISO/IEC 8859-1`. Per questo motivo prima dell'utilizzo di questa funzione, in fase di decodifica, è stato molto comune trovare sequenze di byte più lunghe di quanto ci si aspettasse ed anche leggermente diverse. Ciò accadeva in quanto, quei byte con codice superiore a 127, venivano divisi in due byte con una struttura molto simile ai caratteri con codice superiore al 127 utilizzati in `UFT-8`. Si è rivelato quindi necessario definire una funzione che fosse in grado di riconoscere questi byte divisi e, partendo dal risultato della lettura dell'eQR code, ricostruire i dati in modo esatto.

La funzione `_format_bytes`, che si occupa proprio di questo, è riportata nella sezione di codice successiva.

```
1  def _format_bytes(data_in):
2    i = 0
3    while i < len(data_in):
4        byte = data_in[i]
5        i += 1
6        if byte >= 194:
7            next_byte = data_in[i]
8            i += 1
9            yield int((byte & 0b00011111) << 6 | (next_byte & 0
     b00111111))
10       else:
11           yield int(byte)
```

Listing 7.4: metodo `_format_bytes`

Il valore `194` è usato come base di confronto in quanto esso è il minor valore del primo byte nel caso in cui quello di partenza sia stato diviso in due. I byte che non rientrano in questa categoria vengono restituiti immutati. Quelli che invece entrano nel caso di controllo hanno bisogno di essere trasformati, insieme al byte che li segue. Le maschere e le operazioni bit a bit effettuate in riga (9) permettono di ricostruire il byte originario partendo dai due in cui è stato diviso.

Capitolo 8

Risultati

Al fine di verificare il corretto funzionamento dei due dialetti di *QRscript* presentati all'interno di questa tesi, è stato deciso di sviluppare alcuni *use-cases*, semplici, ma altrettanto rappresentativi, che potessero presentare in maniera chiara e concisa le principali funzionalità di *QRscript*. All'interno di questo capitolo verranno mostrati tutti i passi del processo di funzionamento di *QRscript* contenuti in Figura 4.1 e, per ognuna di queste fasi, verrà mostrato l'output corrispondente.

In particolare, per la cosiddetta fase di generazione, verranno mostrati:

- Il programma eseguibile scritto in linguaggio di alto livello, che rappresenta il punto di partenza delle funzionalità di *QRscript*
- La rappresentazione intermedia corrispondente, conforme alle specifiche del dialetto adottato nel caso specifico
- Il *QRbytecode*, cioè la rappresentazione binaria corrispondente alla rappresentazione intermedia ottenuta al punto precedente
- L'eQR code corrispondente al *QRbytecode*

Per la fase di esecuzione, invece, verrà mostrato solamente il risultato finale dell'interpretazione della rappresentazione intermedia, in quanto i risultati delle altre sottofasi sono equivalenti a quelli delle sottofasi corrispondenti nella fase di generazione.

Gli esempi che verranno trattati all'interno di questo capitolo sono due: uno relativo a *QRtree* ed uno relativo a *QRprog*. Inoltre, in seguito ai due esempi, verrà trattato un terzo esempio, sempre relativo a *QRprog*, che esplorerà le capacità più avanzate della prima versione di questo dialetto. Al fine di rendere la trattazione compatta, per questo esempio verrà riportato come immagine solamente il risultato finale dell'esecuzione di *QRscript*, mentre tutti gli altri aspetti saranno trattati in formato testuale.

8.1 QRtree

L'esempio scelto per il dialetto *QRtree* è un caso applicativo che è stato discusso diverse volte all'interno di questa tesi: quello di un macchinario guasto al quale occorre eseguire una diagnostica. L'idea che sta dietro al programma di alto livello è quindi quella di condurre l'utente attraverso le varie possibilità di malfunzionamento ed indicare infine una possibile soluzione al problema. Per brevità e semplicità, il programma di alto livello qui presentato contiene al suo interno la soluzione ad un numero molto limitato di possibili problemi. Tuttavia, sfruttando le potenzialità di *QRscript* descritte nel Capitolo 5, è possibile estendere questa soluzione ad un numero più elevato di opzioni.

Il programma di alto livello usato per questo esempio è il seguente:

```
input "Quale codice di errore viene mostrato dal macchinario?"
if "A12":
    inputs "Qual è la temperatura interna del macchinario in
            gradi centigradi?"
    ifc <= 15:
        print "Pressione troppo bassa. Rifornire il gas."
        exit
    else ifc >= 100:
        print "Temperatura troppo elevata."
        print "Verificare il funzionamento delle ventole."
        exit
    else:
        print "La temperatura è normale. Chiamare un tecnico."
        exit
else if "C15":
    print "La batteria è danneggiata. Sostituirla."
    exit
else if "F28":
    print "Il macchinario è a corto di olio. Rifornire."
    exit
else:
    print "Chiamare un tecnico."
    exit
```

Dato quindi un macchinario guasto che mostra un codice di errore, il programma chiede, per prima cosa, quale sia questo codice; l'utente, a questo punto, si troverà di fronte a 4 possibili scelte, rappresentate da altrettanti bottoni: `A12`, `C15`, `F28` e `Altro/Other`. Nel caso in cui l'utente selezioni il codice di errore `A12`, il programma intuirà immediatamente che il problema è dovuto ad una temperatura anomala del macchinario e quindi chiederà all'utente quale essa sia. A questo punto

l'utente dovrà inserire la temperatura (espressa in gradi centigradi) all'interno del box che gli verrà presentato: se il valore inserito dall'utente è inferiore a 15 gradi, il programma noterà che questo è un valore troppo basso, il che può essere dovuto ad una pressione del gas troppo bassa. Sarà quindi sufficiente rifornire il macchinario di gas; se invece il valore inserito dall'utente risulta maggiore di 100 gradi, il programma consiglierà di controllare le ventole di raffreddamento del macchinario, in quanto questo rappresenta un valore troppo alto. Se invece la temperatura si trova nella finestra di funzionamento del macchinario (in questo caso $15 < x < 100$), il programma noterà che la temperatura ha un valore perfettamente normale e consiglierà quindi di contattare un tecnico per valutare al meglio le motivazioni dietro a questo codice di errore. Se invece l'utente ha selezionato, come risposta alla prima domanda, il codice `C15`, il programma consiglierà di cambiare la batteria, in quanto questo codice di errore si presenta in caso di un suo danneggiamento. Nel caso in cui, invece, il codice di errore scelto dall'utente sia `F28`, il programma suggerirà un rifornimento di olio, di cui il macchinario è rimasto a corto. Infine, se l'utente preme il pulsante `Altro/Other`, il programma, osservando che il particolare codice di errore non rientra fra le sue possibili alternative, terminerà la sua esecuzione senza essere in grado di fornire un particolare consiglio se non quello di contattare un tecnico.

La rappresentazione intermedia generata seguendo le specifiche riportate nel Capitolo 5 a partire dal programma di alto livello riportato in precedenza è il seguente:

```
(0) input "Quale codice di errore viene mostrato dal macchinario?"
(1) if "A12" (6)
(2) if "C15" (15)
(3) if "F28" (16)
(4) printex "Chiamare un tecnico."
(5) goto (17)
(6) inputs "Qual è la temperatura interna del macchinario in gradi
          centigradi?"
(7) ifc <= 15 (10)
(8) ifc >= 100 (12)
(9) printex "La temperatura è normale. Chiamare un tecnico."
(10) printex "Pressione troppo bassa. Rifornire il gas."
(11) goto (14)
(12) print "Temperatura troppo elevata."
(13) printex "Verificare il funzionamento delle ventole."
(14) goto (17)
(15) printex "La batteria è danneggiata. Sostituirla."
(16) printex "Il macchinario è a corto di olio. Rifornire."
```

Alla rappresentazione intermedia appena riportata, viene poi applicata un'ulteriore traduzione per ottenere la rappresentazione binaria *QRtreebytecode*. La rappresentazione binaria ottenuta per questo esempio è la seguente:

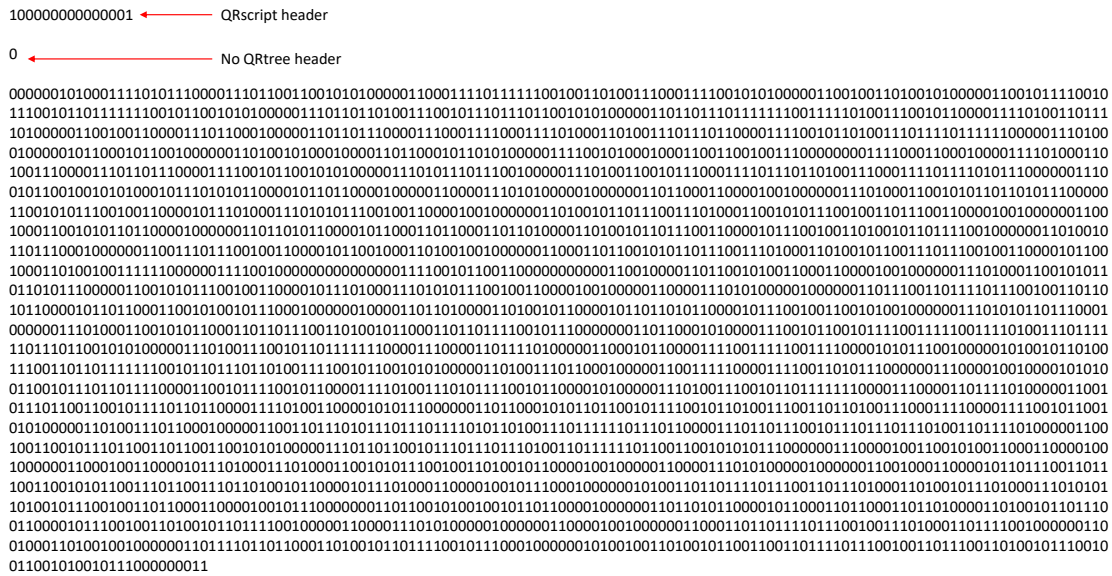


Figura 8.1: Rappresentazione binaria dell'esempio relativo a QRtree

Analizzando l'immagine precedente, è possibile fare diverse considerazioni: per prima cosa, si può notare che il *QRtreebytecode* inizia con una serie ben precisa di bit. Il primo bit settato a `1` rappresenta il fatto che non è stato necessario aggiungere ulteriore padding alla rappresentazione già esistente, in quanto, con l'aggiunta di questo bit, essa aveva già una lunghezza multipla di 8. Dopo questo bit, all'interno del *QRscript header*, si possono inoltre notare:

- La *continuation* disabilitata tramite un bit settato a `0`
- La *sicurezza* disabilitata tramite la sequenza di bit `0000`, che rappresenta il profilo di sicurezza nullo
- L'uso dell'*URL* disabilitato tramite un bit settato a `0`
- *QRtree* è settato come dialetto usato dall'eQR code, indicato dal suo codice `0000`
- La versione di *QRtree* usata è la 1 (codice `0001`)

Finita la sezione di *QRscript header*, troviamo un bit settato a zero, che indica che il *QRtree header* non è stato utilizzato. Dal bit successivo fino alla fine, invece, si trova la parte di *Code* di *QRtreebytecode*, contenente la traduzione delle istruzioni componenti il programma da eseguire. Il *QRtreebytecode* generato in questo esempio ha una lunghezza di 3208 bit, pari a circa il 13.6% della capacità massima di un QR code.

Infine, il QRtreebytecode viene inserito all'interno dell'eQR code, generando il codice qui riportato:

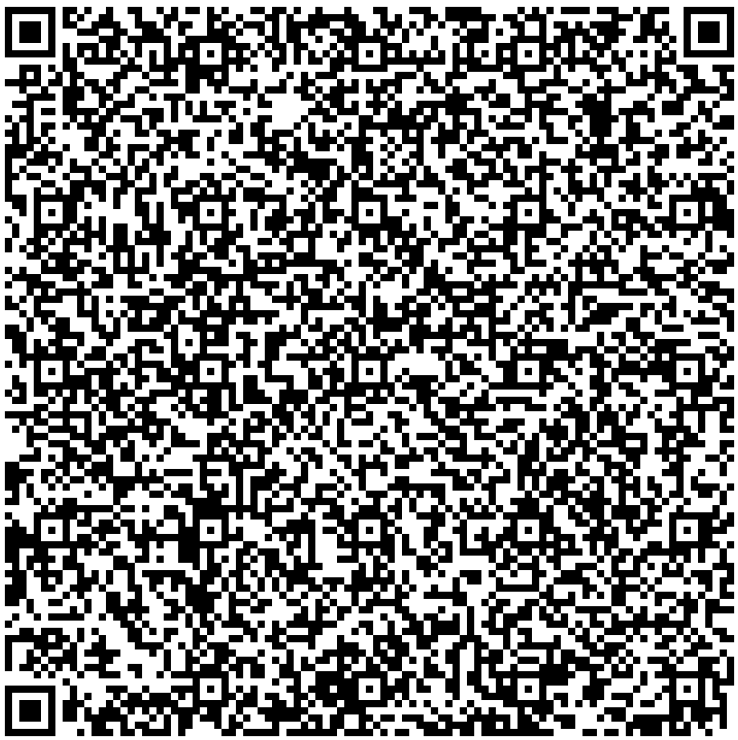


Figura 8.2: eQR code dell'esempio relativo a QRtree

L'eQR code generato dall'esecuzione di *QRscript*, usando le specifiche riportate in questo esempio, è un QR code versione 21. Un codice di questa versione ha una grandezza di 101x101 moduli, corrispondenti a 10201 bit di capacità di memorizzazione. L'uso di una versione così grande rispetto alla lunghezza del *QRtreebytecode* è tutt'altro che casuale: come riportato diverse volte all'interno di questa tesi, infatti, uno dei principali obiettivi di *QRscript*, insieme alla compressione dei dati da inserire all'interno degli eQR code, è quello di preservare la leggibilità di questi ultimi attraverso l'utilizzo dei vari livelli di correzione degli errori. Nella maggior

parte dei possibili casi applicativi di *QRscript*, infatti, gli eQR code generati sarebbero soggetti a molti eventi che potrebbero danneggiarli: per questo, per generare il QR code precedente, è stato utilizzato un livello di correzione degli errori H (il più alto possibile). Stando a [28], un QR code versione 21 che adotta un tale livello di correzione degli errori, riduce il suo spazio di memorizzazione dei dati di input, nel caso di input binari, a 403 byte, ovvero 3224 bit, dimensione che rende questa versione la più appropriata per contenere il *QRtreebytecode* generato. Tutto il resto dello spazio di memorizzazione interno al QR code sarà invece occupato dai bit di correzione degli errori.

La UI derivante dalla scannerizzazione dell'eQR code precedente, nel caso della scelta del bottone `A12`, è riportata in Figura 8.3:

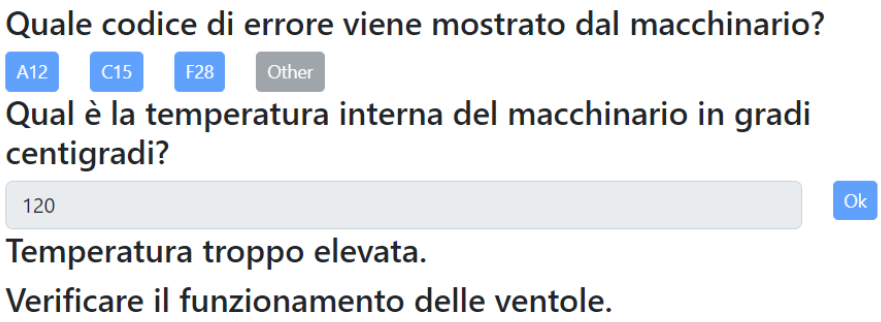


Figura 8.3: Esempio di UI di QRtree

8.2 QRprog

L'esempio scelto per il dialetto *QRprog* è invece un programma general purpose per il calcolo del fattoriale di un numero inserito come input dall'utente. È stato scelto questo esempio in quanto, nella generalità di *QRprog*, esso rappresenta un programma dalla logica molto semplice, ma con un'esecuzione molto particolare; com'è noto, infatti, il classico modo per calcolare il fattoriale di un numero intero è attraverso l'uso della ricorsione, che rende il codice da scrivere piuttosto breve e semplice ma, in un contesto come quello di *QRprog*, in cui la fase di esecuzione prevede anche l'interpretazione della rappresentazione intermedia generata, può rivelarsi anche molto interessante. Il programma ad alto livello utilizzato per questo esempio è il seguente:

```
fn main()
    int in, result;
    input in;
    result = factorial(in);
```

```
        printf "The factorial of %d is %d", in, result;


    fn factorial(int a)
        int b,c,n;

        if(a == 0 | a == 1)
            return 1;

        else
            b = a - 1;
            c = factorial(b);
            n = a * c;
            return n;
```

Il programma, per prima cosa, richiede in input un valore intero all'utente (tramite `input`), valore del quale dovrà essere calcolato il fattoriale. Ricevuto questo valore, esso viene passato alla funzione ricorsiva `factorial` che, alla fine della sua esecuzione, restituirà il risultato finale alla funzione principale.

La rappresentazione intermedia generata seguendo le specifiche riportate nel Capitolo 6 a partire dal programma ad alto livello riportato in precedenza è il seguente:

```
main:
        OPR R0, 0
        OPR R1, 0
        IN R0
        PUSH R0
        JMPF factorial
        POP R1
        OUT R0, R1 "The factorial of %d is %d"
        EXIT
factorial:
        POP R0
        OPR R1, 0
        OPR R2, 0
        OPR R3, 0
        OPR R4, R0 0 NEQ R0 1 NEQ AND
        JMPL L1
        PUSH 1
```

```
        JMP L2
L1:
        OPR R1, R0 1 MINUS
        PUSH R1
        JMPF factorial
        POP R2
        OPR R3, R0 R2 STAR
        PUSH R3
L2:
        RET
```

È importante notare che, per una questione di chiarezza, durante la fase di generazione, il *QRprogbytecode* generato conterrà al suo interno le label delle funzioni corrispondenti all'interno del file del programma ad alto livello. Tuttavia, per una questione di spazio (i nomi sarebbero dovuti essere salvati ogni volta come stringhe, di fatto sprecando tanto spazio di memorizzazione), è stato deciso di non riportare il nome delle varie label all'interno dell'eQR code. Per questo motivo, il *QRprogbytecode* generato durante la fase di esecuzione di *QRprog* potrebbe essere leggermente diverso da quello appena riportato. In particolare: la label della funzione main viene totalmente rimossa ed il flusso di esecuzione della rappresentazione intermedia parte dalla prima istruzione riportata, mentre tutte le altre label sono riportate con un nome del tipo `label_counter`, dove `counter` è un contatore che inizia da 0. Nell'esempio appena riportato, quindi, il *QRprogbytecode* generato durante la fase di esecuzione non avrà la label `main`, mentre `factorial` sarà riportato come `label_0` e così via.

Così come accade in *QRtree*, alla rappresentazione intermedia appena riportata viene applicata un'ulteriore traduzione, al file di ottenere la rappresentazione binaria *QRprogbytecode*. La rappresentazione binaria ottenuta per questo esempio è la seguente:

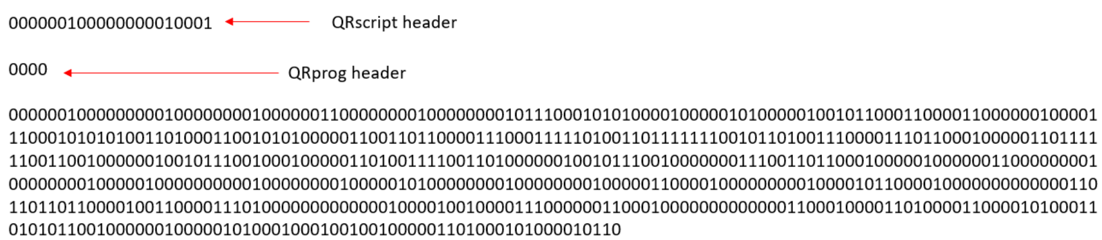


Figura 8.4: Rappresentazione binaria dell'esempio relativo a QRprog

Analizzando l'immagine precedente, è possibile fare diverse considerazioni: il primo bit del *QRprogbytecode* è settato a `0`, il che rappresenta il fatto che i bit

successivi, fino ad arrivare al primo `1` sono bit di padding, necessari per far sì che la rappresentazione binaria abbia una lunghezza multipla di 8. In particolare, si può notare come, in questo caso, i bit di padding siano 7. A seguito di questi bit, si trova il *QRscript header*, con tutte le sue parti:

- La *continuation*, che è disabilitata attraverso un bit settato a 0
- La *sicurezza*, anch'essa disabilitata tramite la sequenza di bit `0000`, che rappresenta il profilo di sicurezza nullo
- L'uso dell'*URL* disabilitato tramite un bit settato a `0`
- *QRprog* è settato come dialetto usato dall'eQR code, ed è indicato attraverso il suo codice `0001`
- La versione di *QRprog* usata è la 1 (codice `0001`)

Finita la sezione dedicata al *QRscript header*, troviamo quella dedicata al *QRprog header*, che contiene il `fileId`, necessario per riconoscere il file a cui il codice corrispondente apparteneva nel caso in cui debba essere eseguito il codice di più file in contemporanea (e.g., il codice di un file chiama il codice di un altro file). I bit seguenti il *QRprog header*, fino alla fine del codice binario, rappresentano la parte di *Code* di *QRprogbytecode*, contenente la traduzione delle istruzioni componenti il programma da eseguire. Il *QRprogbytecode* generato in questo esempio ha una lunghezza di 648 bit, pari a circa il 2.7% della capacità massima di un QR code. Infine, il *QRprogbytecode* viene inserito all'interno dell'eQR code, generando il codice riportato di seguito:

Figura 8.5: eQR code dell'esempio relativo a QRprog

L'eQR code generato dall'esecuzione di *QRscript*, usando le specifiche riportate in questo esempio, è un QR code versione 8. Un codice di questa versione ha una grandezza di 49x49 moduli, corrispondenti a 2401 bit di capacità di memorizzazione. L'uso di una versione così grande rispetto alla lunghezza del *QRprogbytecode* è tutt'altro che casuale: anche in questo caso, infatti, l'eQR code è stato generato utilizzando il livello di correzione degli errori `H`, ovvero il più alto. Ciò comporta che il QR code abbia una maggiore resistenza ai danni, ma anche che il suo spazio di memorizzazione dedicabile ai dati diminuisce. Stando a [28], un QR code versione 8 che adotta un tale livello di correzione degli errori, riduce il suo spazio di memorizzazione dei dati di input, nel caso di input binari, a 84 byte, ovvero 672 bit, dimensione che rende questa versione la più appropriata per contenere il *QRprogbytecode* generato. Tutto il resto dello spazio di memorizzazione interno al QR code sarà invece occupato dai bit di correzione degli errori.

Quando l'eQR code precedente verrà scansionato, verranno eseguite dapprima tutte le traduzioni inverse al fine di ritornare ad avere la rappresentazione intermedia *QRprogAssembly*, che verrà infine interpretata, generando il seguente output:

```
1 Insert the input value: 6
2 The factorial of 6 is 720
```

Listing 8.1: Output dell'interprete per l'esempio di QRprog

Capitolo 9

Conclusioni

Il lavoro di questa tesi si è concentrato sulla definizione di un nuovo ed innovativo modo per gestire l'inserimento di dati all'interno di un QR code, denominato *QRscript* e dei suoi primi 2 dialetti; il primo, *QRtree*, si basa su uno pseudo-linguaggio ad alto livello definito specificatamente per il suo caso applicativo e si presta particolarmente bene alla traduzione ad eQR code di programmi ad albero decisionale, cioè quei programmi formati da blocchi if-else che, a seguito di alcune domande e relative decisioni, forniscono una soluzione basata sulle risposte fornite. *QRtree* eccelle particolarmente nella compattezza della sua rappresentazione binaria, caratteristica che gli permette di essere molto adatto al contesto applicativo dei QR code. Il limite principale di questo dialetto risiede nel fatto che non tutti i possibili programmi sono riducibili ad alberi decisionali, per cui il suo range di supporto è piuttosto limitato. Tuttavia, è necessario notare che questa limitazione è anche il principale motivo per il quale il dialetto risulta molto efficace nel suo contesto di applicazione.

Per quanto riguarda il secondo dialetto, *QRprog*, è invece possibile fare un discorso quasi opposto: esso è stato infatti pensato come primo dialetto che potesse supportare un contesto general purpose. Ciò vuol dire che, potenzialmente, esso potrà, in futuro, essere esteso sempre di più in modo tale che sia possibile gestire un numero molto grande di programmi di diversi tipi. Tuttavia, seppur molto interessante, questa soluzione si scontra con due problemi molto grandi: il primo risiede nella complessità, che aumenta sempre di più ogni volta che viene aggiunta una nuova possibile funzionalità; il secondo è invece più di tipo concettuale. *QRscript* infatti, grazie alle potenzialità fornite dal suo header, può supportare un numero di diversi dialetti pressoché infinito: risulta quindi decisamente vantaggioso, sia dal punto di vista della complessità sia dal punto di vista della compattezza della rappresentazione binaria, sviluppare diversi dialetti specializzati, più simili a *QRtree*, rispetto ad un unico grande dialetto.

I possibili lavori futuri che si basano su quanto inizialmente definito su questa

tesi potrebbero concentrarsi dapprima su uno studio approfondito di altri contesti applicativi dei QR code, per capire dove essi saranno maggiormente diffusi o potrebbero essere utilizzati con buoni risultati in futuro, per poi passare alla definizione di pseudo-linguaggi ad alto livello specializzati (oppure alla limitazione delle funzionalità supportate di linguaggi ad alto livello già esistenti) e delle relative rappresentazioni intermedie e binarie, così da definire nuovi e potenti dialetti che possano essere sfruttati per una gamma vastissima di operazioni.

Bibliografia

[1] 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.

[2] Cata Teuta, S. Patel Payal, Arti Ramesh, and Toru Sakaguchi. QR Code: A New Opportunity for Effective Mobile Marketing. *Journal of Mobile Technologies, Knowledge and Society*, 2013:ID748267, 2013.

[3] Scanova. How to use qr codes for effective marketing. *Scanova*, 2019.

[4] Sim Liew Fong, David Wui Yung Chin, Rabab Alyaham Abbas, Arshad Jamal, and Falah Y. H. Ahmed. Smart City Bus Application With QR Code: A Review. In *IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS 2019)*, pages 34–39, 2019.

[5] Wenying Wen, Yunpeng Jian, Yuming Fang, Yushu Zhang, and Baolin Qiu. Authenticable medical image-sharing scheme based on embedded small shadow QR code and blockchain framework. *Multimedia Systems*, 29:831–845, 2023.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Yuhong Dong, Zetian Fu, Stevan Stankovski, Siyu Wang, and Xinxing Li. Nutritional Quality and Safety Traceability System for China's Leafy Vegetable Supply Chain Based on Fault Tree Analysis and QR Code. *IEEE Access*, 8:161261–161275, 2020.

[19] Zhengxin Fu, Yuqiao Cheng, Sijia Liu, and Bin Yu. A new two-level information protection scheme based on visual cryptography and QR code with multiple decryptions. *Measurement*, 141:267–276, 2019.

[20] Yang-Wai Chow, Willy Susilo, Jianfeng Wang, Richard Buckland, Joonsang Baek, Jongkil Kim, and Nan Li. Utilizing QR codes to verify the visual fidelity of image datasets for machine learning. *Journal of Network and Computer Applications*, 173:102834, 2021.

[21] Information technology – Automatic identification and data capture techniques – QR Code bar code symbology specification. *ISO/IEC 18004:2015 [Electronic resource]*, 2015.

[22] Yergeau, F. UTF-8, a transformation format of ISO 10646. *RFC 3629*, 11 2003.

[23] Information technology — ISO 7-bit coded character set for information interchange. *ISO/IEC 646:1991*, 1991.

[24] YAML Organization. Yaml ain't markup language (yaml) version 1.2, 2009.

[25] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 5th edition, 2014. Section 2.4.2: Two's complement representation of numbers.

[26] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019*, 2019.

[27] International Organization for Standardization. Iso/iec 8859-1, 1987. ISO standard 8859-1:1987.

[28] QR Code official website from Denso Wave. https://www.qrcode.com/en/about/version.html.

[29] Internet Engineering Task Force. Rfc 8259 - the javascript object notation (json) data interchange format, 2017.

[30] World Wide Web Consortium. Extensible markup language (xml) 1.0, 2008.

Appendice A

Notazione esponenziale

La notazione esponenziale usata per la codifica di varie componenti (rappresentate da numeri interi) presenti all'interno dei dialetti di *QRscript* ha come base un'idea abbastanza semplice: partendo da un numero prefissato di bit (ad esempio, 4), ogni volta che l'intero senza segno non è rappresentabile su quel numero di bit, questo numero viene raddoppiato ed i primi `n` bit (dove `n` è il numero di bit precedente, non ancora raddoppiato), vengono tutti settati ad `1`. Quindi, ad esempio, i numeri compresi fra 0 e 14 verranno rappresentati su 4 bit (e.g., $8_{10} = 1000_{\text{EC}}$); i numeri compresi tra 15 e 29 saranno codificati su 8 bit, con i primi 4 bit settati a `1111` (e.g., $25_{10} = 11111010_{\text{EC}}$); i numeri compresi tra 30 and 299 saranno codificati su 16 bit, con i primi 8 bit settati a `11111111` (e.g., $100_{10} = 1111111101000110_{\text{EC}}$) e così via.

Questo tipo di codifica assicura la compattezza e l'estensibilità della rappresentazione: infatti, il numero di bit usato per la codifica è direttamente proporzionale alla grandezza del numero che viene rappresentato e permette di codificare interi arbitrariamente grandi, senza alcun limite superiore.

Questa codifica esponenziale, in *QRscript*, può essere utilizzata in due casi diversi: per gli interi senza segno e per quelli con segno. Mentre nel primo caso l'algoritmo di conversione viene seguito alla lettera, nel secondo caso esso viene integrato da un ulteriore bit, che viene preposto al risultato dell'algoritmo: questo bit settato a 0 rappresenta un intero con segno positivo, mentre se esso è settato ad 1 l'intero con segno è negativo.

A.1 Algoritmo di conversione

L'algoritmo usato per la codifica in notazione esponenziale svolge due compiti principali: calcola il numero di bit settati ad 1 necessari per ricadere all'interno del gruppo di codifica a cui appartiene il numero da rappresentare (e.g., 0-14, 15-29, 30-299, ...) ed infine codifica la parte del numero ancora non rappresentata (che

in termini numerici è la differenza tra il numero passato come input ed il primo numero del suo gruppo) in un numero di bit pari al numero di bit a 1 trovati al passaggio precedente.

Le due funzioni che si occupano di queste operazioni sono riportate nella porzione seguente di codice. A titolo di esempio, il valore base della lunghezza della rappresentazione binaria all'interno di queste funzioni è settato a 4 in quanto esso è il valore minimo usato per la codifica delle *References* introdotte in Sezione 5.4.2. Questa scelta è stata fatta in quanto questo è il campo applicativo in cui queste funzioni sono state più utilizzate. Tuttavia, se si ritenesse necessario applicare questo tipo di estensione con una diversa lunghezza minima, è sufficiente sostituire i 4 presenti all'interno delle funzioni con il valore scelto.

```
1  def exp_notation_ones_value(ones: int) -> int:
2    if ones == 0
3      return 0
4    if ones == 4
5      return 2**ones - 1
6    return exp_notation_ones_value(ones//2) + 2**(ones//2) - 1
7
8  def exp_notation_encoding(value: int) -> str:
9    length = 4
10
11   while True:
12     ones = 0 if length == 4 else length//2
13     max_value = 2**(length - ones) - 1
14     cur_value = value - exp_notation_ones_value(ones)
15     if cur_value < max_value:
16       return "1" * ones + format(cur_value, f'0{4 if length == 4
     else length // 2}b')
17     length = length * 2
```

Listing A.1: Funzioni per la codifica in notazione esponenziale

La funzione `exp_notation_encoding(...)` è la funzione che viene chiamata quando è necessario eseguire la codifica di un numero intero senza segno. Questo numero viene passato come argomento alla funzione tramite `value`. La variabile `length` rappresenta la lunghezza della rappresentazione e viene inizializzata al suo valore minimo. In seguito a ciò inizia un loop che terminerà solamente quando la rappresentazione binaria finale del numero sarà pronta. Ad ogni iterazione, `ones` viene settato al numero di bit pari ad 1 necessari alla codifica in caso essa venga completata all'interno dell'iterazione (alla prima iterazione vale 0 e nelle successive è pari alla metà del valore di `length`). Successivamente vengono calcolati due valori fondamentali: il primo, `max_value`, rappresenta l'intero più grande rappresentabile usando i soli bit non riservati ai bit settati ad 1; il secondo, `cur_value`, rappresenta il numero che è necessario rappresentare in quei medesimi bit.

Esso è calcolato come la differenza tra il valore passato in input ed il valore che viene rappresentato dai bit settati ad 1. Quest'ultimo valore è il risultato della funzione `exp_notation_ones_value(...)` che, prendendo come input il numero di bit settati ad 1 necessari per ogni iterazione, restituisce il valore rappresentato da quei bit. Per esempio, nel caso in cui `ones` sia pari a 4, il risultato di questa funzione è 15; se invece `ones` è pari ad 8 (i.e. `1111 1111`), il valore rappresentato da questi 8 bit sarà 30 (15 + 15). Come si può notare, questo valore rappresenta anche il primo valore facente parte del gruppo di interi che può essere rappresentato all'iterazione successiva.

Una volta calcolati questi due valori, se `cur_value` è minore di `max_value`, significa che è possibile terminare la rappresentazione con i bit forniti dall'iterazione corrente, per cui la funzione ritorna una stringa contenente tanti 1 quanto è il valore di `ones`, seguito da una codifica binaria del valore contenuto in `cur_value` in un numero appropriato di bit basato sull'iterazione corrente. Altrimenti, il valore di `length` viene raddoppiato e si procede ad un'ulteriore iterazione dell'algoritmo.

L'algoritmo usato per la decodifica, invece, è riportato di seguito:

```
1 def exp_notation_decoding(value: str) -> int:
2     if len(value) > 4 and not value.startswith("1" * (len(value) //
      2)):
3         raise Exception("Wrong format of exponential uint")
4     if len(value) == 4:
5         return int(value, 2)
6     return exp_notation_ones_value(len(value) // 2) + int(value[len
     (value) // 2:], 2)
```

Listing A.2: Funzione `exp_notation_decoding(...)`

Questa funzione permette, data come input la stringa che indica la rappresentazione binaria di un intero senza segno, di ottenere l'intero, distinguendo fra i tre possibili casi: se la lunghezza della rappresentazione (`len(value)`) è maggiore della più piccola possibile, ma il numero di bit settati ad 1 al suo inizio non è coerente, viene visualizzato un messaggio di errore; se invece la rappresentazione è lunga quanto la minore lunghezza possibile, il risultato atteso è ottenuto semplicemente dalla decodifica classica da binario ad intero; infine, se la rappresentazione ha una lunghezza maggiore (e la sua struttura è valida), `exp_notation_decoding(...)` la divide in due parti (la parte introduttiva con tutti i bit settati a 1 e la parte restante) e:

- fa uso della funzione `exp_notation_ones_value(...)` introdotta in precedenza per ottenere il valore rappresentato dalla prima
- decodifica la seconda usando la classica decodifica da binario ad intero
- somma i due risultati parziali per ottenere quello finale

Analogamente alle funzioni precedenti, è possibile generalizzare anche la funzione `exp_notation_decoding(...)` a gruppi di bit di lunghezza minima variabile, sostituendo i valori 4 con il valore della lunghezza minima desiderata.

Appendice B

Notazione polacca inversa

La notazione polacca inversa, conosciuta anche con l'acronimo *RPN* (reverse polish notation) è una sintassi alternativa per le formule matematiche. Con la RPN è possibile effettuare qualsiasi tipo di operazione, con il grosso vantaggio di poter eliminare tutti i problemi dovuti alle precedenze tra operatori e anche quelle dovute alle parentesi.

In questo tipo di notazione, detta anche *postfissa*, appaiono prima gli operandi e, solo successivamente, gli operatori. Per esempio, mentre la notazione infissa (quella più comunemente usata) denota un'addizione tra due numeri `x` e `y` come `x + y`, la RPN la rappresenta come `x y +`

L'idea di base dietro alla valutazione di espressioni scritte in notazione polacca inversa è quella di avere uno stack in cui gli operandi si accumulano: nell'esempio presentato in precedenza, lo stack riceve dapprima l'operando `x`, poi l'operando `y`. Un operatore, invece, preleva dallo stack tutti gli operandi di cui ha bisogno, esegue l'operazione corrispondente al suo simbolo e reinserisce il risultato all'interno dello stack. Essendo questa un'operazione che coinvolge uno stack, gli operandi prelevati sono sempre da considerarsi dall'ultimo al primo. Nell'esempio precedente ciò non avrebbe alcun effetto, data la natura commutativa dell'addizione; tuttavia, nel caso di un divisione (`x y /`), al momento dell'esecuzione dell'operando `/`, gli operandi vengono estratti dallo stack in questo ordine: prima la `y` e poi la `x`. Diventa a questo punto immediato comprendere come sia indispensabile, per la corretta esecuzione dell'operazione di divisione, considerare i due operandi come estratti in ordine inverso. Se l'espressione completa è scritta correttamente, alla fine di tutte le operazioni, lo stack avrà un solo elemento al suo interno. Questo elemento è il risultato finale dell'espressione.

B.1 Algoritmo di conversione

Data un'espressione scritta con la notazione infissa, essa può essere trasformata in notazione polacca inversa attraverso il seguente algoritmo.

- `input` = espressione in notazione infissa.
- `output` = espressione in notazione polacca inversa.
- `stack` = stack in cui verranno inseriti gli operatori.
- Si identificano i vari componenti (operandi, operatori e parentesi) all'interno dell'espressione `input`, procedendo da sinistra a destra.
- Ogni operando che viene incontrato viene aggiunto alla fine della stringa `output`.
- Ogni volta che viene incontrato un operatore, invece, è necessario fare alcune considerazioni: se l'operatore in cima a `stack` ha una precedenza minore rispetto a quello letto, oppure si tratta di una parentesi aperta, esso viene inserito in cima allo stack. In alternativa, se l'operatore in cima allo stack ha precedenza maggiore o uguale all'operatore corrente, esso viene estratto dallo stack e concatenato alla fine della stringa `output`. Questo controllo deve essere effettuato ciclicamente fino a quando non sia incontrato un operatore che rientri nel caso precedente. Solo a questo punto l'operatore corrente può essere inserito in cima a `stack`.
- Le parentesi di apertura devono sempre essere inserite in cima allo stack.
- Ogni volta che viene incontrata una parentesi di chiusura, è necessario estrarre da `stack` ed inserire alla fine della stringa `output` tutti gli operatori fino ad incontrare una parentesi di apertura oppure fino ad avere `stack` vuoto (in qual caso è necessario riportare un errore di parentesizzazione dell'espressione originale). Tutte le parentesi di apertura estratte da `stack`, così come tutte le parentesi di chiusura incontrate, devono quindi essere scartate.
- Dopo aver letto l'ultimo carattere della stringa `input`, `stack` viene svuotato ed ogni operatore viene inserito alla fine della stringa `output`.

È importante notare come questo algoritmo sia valido solamente nel caso in cui tutti gli operatori coinvolti nell'espressione siano associativi a sinistra.

Appendice C

Definizione di dizionari globali e specifici

Un dizionario di tipo *globale* o di tipo *specifico* può contenere una o più lingue. Ogni lingua deve avere un suo identificatore di 2 o più caratteri (ad esempio, "it" per l'italiano, "en" per l'inglese, ...) ed una lista di parole. L'ordinamento con cui le parole sono inserite all'interno del dizionario è particolarmente importante, in quanto l'indice della parola all'interno della lista sarà l'identificatore che permetterà di fare riferimento alla parola stessa.

Per definire questi tipi di dizionari, è stato deciso di sfruttare il formato `YAML`. `YAML` [24] (YAML Ain't Markup Language) è un linguaggio di serializzazione dei dati (molto simile a `JSON` [29] ed `XML` [30]). Tuttavia, rispetto ai suoi concorrenti presenta un vantaggio particolarmente importante per quanto riguarda l'aspetto applicativo del lavoro di questa tesi: è molto semplice sia da leggere che da scrivere, grazie alla sua sintassi basata semplicemente su spazi e linee vuote, a differenza degli altri due linguaggi, che fanno pesante uso di altri simboli all'interno delle loro sintassi.

Lo schema per la definizione di un dizionario di tipo globale o specifico 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"
```

Un esempio molto semplice di un possibile dizionario, secondo la sintassi appena descritta, è riportato di seguito:

```
- language: en
  words:
    "Yes"
    "No"
    "What"
    "Which"
- language: it
  words:
    "Si"
    "No"
    "Quale"
    "Cosa"
```