ESAME DI STATO A.S. 2016/2017

OB-LA-DI

CARLO BLASI

I.T.I.S. “BENEDETTO CASTELLI”
CLASSE 5°AI INFORMATICA

Indice

Premessa 2

Motivazioni 2

Introduzione 3

Cos'è un compilatore 3

Funzioni di un compilatore 3

Componenti di un compilatore 3

Implementazione 5

Grammatica 5

Linguaggio di implementazione 7

Linguaggio implementato 9

Analisi lessicale 12

Analisi sintattica 15

Analisi semantica 18

Generazione del codice 19

Utilizzo 26

Conclusione 28

Sitografia e bibliografia 29

Premessa

Questo progetto è servito a due scopi: imparare qualcosa sulla progettazione di un linguaggio e sull'implementazione di un compiler, e avere la possibilità di costruire un importante sistema software che sta alla base di tutto ciò che è possibile fare con un computer.

Realizzare un compiler è un lavoro intenso in ordine di ore di lavoro e risorse siccome comprende le problematiche relative alla progettazione e all'implementazione di un linguaggio di programmazione. Usare un linguaggio esistente non avrebbe semplificato il problema in quanto i linguaggi che usiamo tutti i giorni sono troppo complessi per essere implementati da una sola persona nel corso di un anno scolastico. Il linguaggio che è stato implementato in questo progetto è quindi ispirato a quelli più diffusi ma ridotto all'essenziale.

Motivazioni

Le motivazioni che mi hanno spinto a scegliere un progetto come questo come tesina sono la mia curiosità di sapere come si muovono gli ingranaggi che si trovano dietro a ciò che uso tutti i giorni, e la soddisfazione che sapevo avrei ottenuto dal creare un linguaggio di programmazione tutto mio.

Introduzione

Cos'è un compilatore

Tutti i programmi vengono scritti in linguaggi di programmazione che rendono il lavoro più semplice per i programmatori, ma non per i computer, che invece interpretano sequenze di particolari istruzioni. I testi dei programmi devono quindi essere tradotti in una sequenza di istruzioni che il computer può capire ed eseguire. Questa traduzione può essere automatizzata, quindi può essere essa stessa un programma. Il programma traduttore prende il nome di compilatore (o compiler, i due nomi verranno usati arbitrariamente), e il testo da tradurre viene chiamato file sorgente (o meglio, codice sorgente).

Funzioni di un compilatore

Un compiler riceve come input il file di testo che contiene le istruzioni scritte in un determinato linguaggio, si assicura che non ci siano errori lessicali e sintattici (nel caso in cui ci siano l'esecuzione si ferma e vengono mostrati gli errori rilevati) e genera come output un file contenente le istruzioni eseguibili dal computer che si intende prendere in considerazione.

Componenti di un compilatore

Un compiler è composto da diverse parti ognuna avente un compito preciso necessario affinché le parti successive possano funzionare correttamente.

Il ***lexer*** è la parte che si occupa di spezzare il codice sorgente in piccoli pezzi detti token. Ogni token è un unità elementare del linguaggio, per esempio una keyword, il nome di una variabile o una costante.

Il ***parser*** è la parte che si occupa di fornire una rappresentazione sintattica del programma, spesso (e nel caso di questo progetto) sotto forma di un albero. Le funzioni principali dell'albero sintattico (syntax tree) sono mostrare la precedenza delle operazioni e rilevare gli errori sintattici.

Il ***code generator*** è la parte che si occupa di generare il codice assembly per la macchina su cui il compilatore è in esecuzione a partire dall'albero sintattico.

Implementazione

Grammatica

Una grammatica precisa e corretta permette di semplificare il processo di implementazione, soprattutto del parser, in quanto questo rispecchia quasi completamente la sintassi del linguaggio. La grammatica viene definita attraverso un altro linguaggio, o notazione: l'EBNF, estensione del BNF.

Verso la fine degli anni '50, vennero realizzati i primi linguaggi di programmazione di alto livello, tra questi i più famosi sono il FORTRAN e l'ALGOL. John Backus era alla guida dello sviluppo del FORTRAN ed ha inventato una notazione semplice, precisa e abbastanza versatile da poter descrivere la sintassi di qualsiasi linguaggio di programmazione. Usando questa notazione un compilatore può determinare la correttezza sintattica di un programma. Peter Naur, editor del report dell'ALGOL, popolarizzò questa notazione usandola per descrivere la sintassi completa del linguaggio. In loro onore questa notazione viene chiamata Backus-Naur Form (BNF). In questo progetto viene usata la versione estesa di questa notazione, chiamata Extended Backus-Naur Form (EBNF), che aggiunge regole per definire la ripetizione e l'opzionalità, risultando in definizioni più compatte.

La sintassi viene definita da simboli terminali e nonterminali, dove i nonterminali devono essere sostituiti secondo le regole sintattiche definite dall'EBNF fino a costituire uno o più simboli terminali. Le strutture di controllo permesse sono:

- la sequenza: i simboli appaiono da sinistra a destra, il loro ordine è importante;
- la scelta: i simboli alternativi sono separati da una barra verticale `|`, solo un simbolo può essere scelto, il loro ordine non è importante;
- l'opzione: il simbolo opzionale deve essere tra le parentesi quadre `[` e `]`, il simbolo può non essere preso o essere preso solo una volta;
- il raggruppamento: più simboli, di solito opzionali, possono essere raggruppati tra le parentesi tonde `(` e `)`;

• la ripetizione: il simbolo ripetibile deve essere tra le parentesi graffe `{` e `}`, il simbolo può essere ripetuto zero o più volte.

Una regola sintattica solitamente prende la forma di un'equazione con a sinistra dell'uguale un simbolo nonterminale e a destra dell'uguale i simboli con cui va sostituito. Questo esempio può descrivere ogni numero intero positivo o negativo:

```
<cifra> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<numero> = [ + | − ] <cifra> { <cifra> }
```

Parafrasando queste regole grammaticali in italiano:

• Una cifra è definita come uno dei dieci caratteri da "0" a "9".

• Un numero è definito come la sequenza di tre simboli: un segno opzionale (se è presente deve essere uno delle alternative "+" o "–"), seguito da una cifra, seguita da una ripetizione di zero o più cifre, dove ogni cifra è scelta indipendentemente dalla lista di alternative della regola sintattica della cifra.

Si può notare come con un numero finito di equazioni possiamo definire tutti i numeri interi possibili. Il mezzo per ottenere ciò è la ricorsione che permette la sostituzione arbitraria del nonterminale `<cifra>`.

La notazione EBNF genera linguaggi senza contesto, ovvero linguaggi in cui la sostituzione del simbolo a sinistra dell'uguale con ciò che viene derivato dall'espressione a destra dell'uguale è sempre permessa, indipendentemente dal contesto in cui il simbolo si trova.

Infine si può distinguere la grammatica in questo progetto come LL(1) (left leftmost) perché scorre la frase da sinistra a destra e le possibili sostituzioni vengono scelte da sinistra a destra. Il numero all'interno delle parentesi indica il numero di simboli lookahead, ovvero simboli da leggere in anticipo necessari per decidere quale produzione grammaticale usare. Le grammatiche LL(1) sono di grande interesse pratico perché parser per questo tipo di grammatica sono facili da implementare ed efficienti. Molti linguaggi di programmazione sono progettati per essere LL(1) per questo motivo.

Linguaggio di implementazione

Scala

Il nome Scala sta per "Scalable Language", ovvero linguaggio scalabile, questo perché Scala può essere usato sia per piccoli progetti che per sistemi complessi come quelli di Twitter, Linkedin o Intel.

La sintassi è concisa e poco verbosa, permette quindi di scrivere programmi più complessi in meno righe di codice. Infatti molti elementi sono facoltativi, ad esempio il punto e virgola alla fine di una riga, il punto prima di chiamare una funzione, le parentesi di una funzione (se la funzione non richiede parametri), il return per ritornare il valore di una funzione. Scala è un linguaggio a oggetti puro, ovvero qualsiasi cosa è un oggetto, non esistono primitivi, e qualsiasi operazione è in realtà un metodo, incluse le operazioni matematiche. Ad esempio, l'operazione `a + 1` viene tradotta dal compilatore di Scala in `a.+(1)`.

Scala viene eseguito sulla JVM (Java Virtual Machine) ed è completamente interoperabile con Java, questo significa che i programmi in Scala possono essere eseguiti su tutti i dispositivi su cui è installato Java (la maggior parte) e che la grandissima selezione di librerie Java può essere utilizzata nei progetti in Scala.

Programmazione funzionale

La parte funzionale di Scala traspare grazie al grande numero di caratteristiche che sono solitamente offerte dai linguaggi funzionali:

- nessuna distinzione tra dichiarazione ed espressione, ogni riga di codice quindi restituisce un valore;
- inferenza del tipo dei dati;
- funzioni anonime e closures (funzioni come variabili);
- variabili e oggetti immutabili;
- lazy evaluation (valutazione "pigra"), ovvero la strategia di valutazione che rimanda il più possibile la computazione di una espressione, finché il suo valore non

sia richiesto, portando a vantaggi teorici come l'aumento di prestazioni e la possibilità di creare strutture dati infinite;

• funzioni di ordine superiore, funzioni che possono prendere altre funzioni come parametri e/o restituire funzioni come risultato;

• currying, ovvero la tecnica che trasforma la valutazione di una funzione che prende multipli parametri nella valutazione di una sequenza di funzioni che prendono un solo parametro ciascuno;

• tail recursion, ricorsione a coda, per cui non bisogna aspettare che tutte le chiamate ricorsive vengano fatte (occupando memoria) prima di calcolare il risultato della funzione;

• tuple, elenchi finiti ed ordinati di valori;

Particolarmente importanti sono le funzioni di libreria utilizzabili per iterare all'interno di collezioni ed eseguire operazioni sugli elementi che le compongono, che rendono inutili le strutture cicliche dei linguaggi imperativi. In Scala queste funzioni sono il folding, il reducing e il mapping.

Il folding è un'operazione che consiste nel far 'avvolgere' gli elementi di una collezione da una funzione ed accumularne il valore di questo avvolgimento, ma il valore iniziale viene passato come argomento.

Il reducing è un'operazione simile al folding, ma vengono considerati a coppie gli elementi della collezione e la funzione passata come argomento viene applicata a due a due.

Il mapping è un'operazione che applica una funzione ad ogni elemento della collezione e restituisce una collezione coi risultati.

Linguaggio implementato

OB-LA-DI

OB-LA-DI (pronunciato *obladì*, palese tributo alla canzone “Ob-La-Di Ob-La-Da” dei Beatles) è un linguaggio imperativo general-purpose molto primitivo, esso infatti utilizza solo dati di tipo intero ed eventualmente string literals (stringhe costanti di caratteri) per l’interazione con gli utenti.

La sintassi riprende dal linguaggio BASIC, uno dei linguaggi più famosi, la regola sintattica per cui ogni dichiarazione deve iniziare con una delle keyword del linguaggio, mentre riprende il resto dal linguaggio C, ad esempio il punto e virgola alla fine di ogni dichiarazione.

Le variabili vengono dichiarate con la keyword `var` seguita dalla lista di variabili da dichiarare.

```
var a, b, c;
```

L’assegnamento inizia con la keyword `let` seguita da un’espressione matematica. La precedenza nelle espressioni matematiche è implicita in base all’operazione (moltiplicazione e divisione più importanti di addizione e sottrazione) ma è buona pratica forzarla con l’uso delle parentesi, che sono invece obbligatorie per l’operazione di negazione. Se il risultato di un’espressione matematica supera $2^{63} - 1$ degli errori sorgeranno durante l’esecuzione del programma.

```
let a = 1;
let b = 2;
let c = a + b;
let a = c * (-1);
```

Le condizioni possono essere verificate con le strutture `if` e `if-else`. I cicli si ottengono con la struttura `while`. Diversamente dal C non sono necessarie le parentesi intorno alle condizioni. A destra e a sinistra dell’operatore logico può trovarsi un’espressione matematica.

L'input e l'output si ottengono grazie alle funzioni `print`, `println`, e `input`. `print` (stampa) e `println` (stampa e aggiungi nuova linea) vengono seguite dagli argomenti da stampare (variabili, espressioni matematiche o stringhe), separati da una virgola.

`input` viene seguito dal nome della variabile il cui valore vogliamo venga inserito dall'utente. Vengono accettati solo numeri interi mentre i numeri con la virgola o altri simboli vengono troncati a partire dal carattere invalido. Qualsiasi altro valore porterà o ad un errore durante l'esecuzione del programma o ad un risultato inaspettato.

È possibile commentare il codice includendolo tra due '#', non esiste un'abbreviazione per i commenti su una sola riga.

La grammatica del linguaggio, espressa in EBNF, è la seguente:

```
<statements> ::= { <statement> }
<statement> ::= "var" <ident> { “,” <ident> } “;”
              | "let" <ident> “=“ <expression> “;”
              | "if" <expression> <relop> <expression> “{”
                      <statements>
                 “}”
                 [ "else" "}"
                      <statements>
                 “}" ]
              | "while" <expression> <relop> <expression> “{”
                      <statements>
                 “}”
              | "print" ( <expression> | <string> )
                       { "," ( <expression> | <string> ) } “;”
              | "input" <ident> “;”

<expression> ::= [ + | − ] <term> { ( + | − ) <term> }
<term> ::= <factor> { ( ∗ | / ) <factor> }
<factor> ::= <ident> | <constant> | "(" <expression> ")"
<ident> ::= <letter> { <letter> | <digit> }
<string> ::= "”" { <letter> | <digit> } "”"
```

```
<letter> ::= a | .. | Z
<constant> ::= <digit> { <digit> }
<digit> ::= 0 | .. | 9
<relop> ::= <[ = | > ] | >[=] | =
```

Per quanto riguarda l'editor con cui scrivere programmi in OB-LA-DI, invece di creare uno strumento stand-alone, ho scritto il file per il syntax highlighting del linguaggio (un file XML che definisce le espressioni regolari che determinano quali parti del linguaggio vanno colorate in modo diverso) per il famoso editor di testo Sublime Text, così da ottenere il vantaggio di poter usare lo stesso software per tutti i linguaggi di programmazione che ho usato per il progetto e non dover cambiare tra un programma e l'altro.

Analisi lessicale

L'analisi lessicale è il processo che si occupa di produrre una sequenza di token dato un codice sorgente. Ogni token è un unità elementare del linguaggio (detta anche lessema), per esempio una keyword, il nome di una variabile o una costante. Il processo viene anche detto ***tokenization*** (tokenizzazione) e la parte di programma che si occupa dell'analisi lessicale si chiama analizzatore lessicale o ***lexer***. Questa è la prima fase di tutti i moderni compilatori e avviene solitamente in un solo passaggio. Di seguito un esempio di tokenizzazione. Questa riga di codice viene spezzata nelle sue unità più piccole ed esse vengono categorizzate:

```
let b = a + 1;
```

Token	Tipo
`let`	`keyword`
`b`	`identifier`
`=`	`equal`
`a`	`identifier`
`+`	`operation`
`1`	`constant`
`;`	`linebreak`

In questo progetto il lexer è una funzione completamente imperativa che, dato come parametro il percorso del file sorgente, restituisce un vettore di oggetti Token. Al suo interno la funzione ricava un vettore di caratteri dal file sorgente grazie alle funzioni di libreria. Attraverso un semplice ciclo, che va avanti fino a che non finiscono i caratteri da leggere, la funzione controlla ogni carattere e in base al tipo (lettera, cifra o simbolo) controlla o meno i caratteri successivi e infine aggiunge al vettore da restituire il token trovato, attribuendogli un valore e un tipo.

Gli spazi nel codice sorgente vengono spesso utilizzati per separare token diversi come variabili e keyword, mentre ad esempio non sono necessari per separare

operazioni matematiche e costanti, quindi sono importanti in un lexer nonostante non costituiscano un tipo di token a sé.

Questa fase solitamente riconosce gli errori grammaticali e i caratteri invalidi, in questo progetto invece gli errori individuati sono commenti lasciati aperti e stringhe non chiuse. Ciò non significa che gli errori grammaticali passano inosservati, infatti una keyword scritta male non verrà riconosciuta durante la fase di parsing e la compilazione verrà terminata.

L'analisi lessicale costituisce la prima fase di un compiler e il suo output (la sequenza di token) costituisce l'input dell'analisi sintattica.

Il lexer in questo progetto è un metodo all'interno della classe Lexer. Questo è il codice essenziale con la spiegazione nei commenti.

```
def lex(): ArrayBuffer[Token] = {

    var token = ""
    var ch = ' '
    var i = 0

    while (i < source.length - 1) {

        ch = source(i) // prendo il carattere successivo

        if (ch == '#') {

            // consuma i caratteri fino al prossimo "#"
            // se il file termina prima, lancia un'eccezione
            // altrimenti incrementa l'indice
        }
        else if (ch == '"') {

            // consuma i caratteri fino al prossimo '"'
            // se il file termina prima, lancia un'eccezione
            // altrimenti incrementa l'indice
        }
        else if (ch.isLetter) {

            // vengono aggiunti i caratteri successivi fino a quando non
            // si incontra uno spazio o un carattere invalido
            // controllo se la sequenza di caratteri trovata corrisponde
            // ad una keyword in tal caso aggiungo il token come "keyword"
            // altrimenti aggiungo il token come "ident" (identifier)
            // incrementa l'indice
        }
        else if (ch.isDigit) {

            // vengono aggiunte tutte le cifre successive
```

```
            // aggiungo token "constant"
            // incrementa l'indice
        }
        else if (ch == '{' || ch == '}') {

            // aggiungo token "paren"
            // incrementa l'indice
        }
        else if (ch == '(' || ch == ')') {

            // aggiungo token "arithparen" (parentesi matematica)
            // incrementa l'indice
        }
        else if (ch == '=') {

            // aggiungo token "eq"
            // incrementa l'indice
        }
        else if (ch == '+' || ch == '-' || ch == '/' || ch == '*') {

            // aggiungo token "op"
            // incrementa l'indice
        }
        else if (ch == '<' || ch == '>') {

            // controllo se il carattere successivo è "="
            // nel caso di "<" controllo se quello successivo è ">"
            // aggiungo token "relop" (operatore di relazione)
            // incrementa l'indice
        }
        else if (ch == ';') {

            // aggiungo token "linebreak"
            // incrementa l'indice
        }
        else if (ch == ',') {

            // aggiungo token "comma"
            // incrementa l'indice
        }
        else  {
            // incrementa l'indice
        }
    }

    // aggiungo token "EOF" (fine del file)
    // restituisco il vettore di token
}
```

Analisi sintattica

L'analisi sintattica è il processo che si occupa di fornire una rappresentazione sintattica del programma, spesso (e nel caso di questo progetto) sotto forma di un albero. Le funzioni principali dell'albero sintattico (syntax tree) sono di mostrare la precedenza delle operazioni e di rilevare gli errori sintattici. La parte di programma che si occupa dell'analisi sintattica si chiama ***parser***.

Un parser può essere implementato attraverso diverse tecniche, in questo progetto la tecnica utilizzata si chiama "a discesa ricorsiva" (recursive descent). Il metodo è molto semplice e consiste nel creare una funzione di parsing per ogni simbolo nonterminale della grammatica del linguaggio e darle lo stesso nome del nonterminale, ogni occorrenza di un nonterminale nella sintassi si traduce nella chiamata della funzione corrispondente. In questo modo il parser generato rispecchia molto da vicino la grammatica del linguaggio. Prendiamo per esempio le produzioni grammaticali necessarie per definire un'espressione matematica:

```
<expression> ::= [ + | - ] <term> { ( + | - ) <term> }
<term> ::= <factor> { ( * | / ) <factor> }
<factor> ::= <ident> | <constant> | "(" <expression> ")"
```

Queste produzioni si traducono in tre funzioni, chiamate appunto "expression", "term" e "factor" (per la dimostrazione non è necessario conoscere le produzioni "ident" e "constant", la prima si riferisce al nome di una variabile mentre la seconda si riferisce ad una costante numerica). Le parti di codice dedicate all'aggiunta dei nodi al parse tree che si sta formando è stata eliminata per facilitare la lettura ed è stata sostituita con dei commenti. L'oggetto Token ha sia un attributo tipo che un attributo valore, per ovvi motivi (diverse operazioni matematiche ad esempio), e così è anche l'oggetto Nodo che va a costruire l'albero sintattico.

```
def expression(): Unit = {
    // aggiunto nodo "expr" all'albero
```

```
    if (nextToken.tokenValue == "-" || nextToken.tokenValue == "+") {
        if (nextToken.tokenValue == "-")
            // aggiunto nodo “neg” (operazione di negazione) al nodo “expr”
            nextToken = getNextToken()
    }
    term()

    while (nextToken.tokenValue == "+" || nextToken.tokenValue == "-") {

        // aggiunto nodo “op” (operazione) al nodo “expr” con il valore + o -
        nextToken = getNextToken()
        term()
    }
}


def term(): Unit = {

    // aggiunto nodo “term” al nodo “expr”
    factor()
    nextToken = getNextToken()

    while (nextToken.tokenValue == "*" || nextToken.tokenValue == "/") {

        // aggiunto nodo “op” (operazione) al nodo “term” con il valore * o /
        nextToken = getNextToken()
        factor()
        nextToken = getNextToken()
    }
}


def factor(): Unit = {

    // aggiunto nodo “factor” al nodo “term”

    if (nextToken.tokenType == "ident") {
        ident()
    }
    else if (nextToken.tokenType == "constant") {
        constant()
    }
    else {

        if (nextToken.tokenType == "arithparen") {

            nextToken = getNextToken()
            expression()

            if (nextToken.tokenType != "arithparen") {
                // lancia un errore perché la parentesi non è stata chiusa
            }
        }
```

```
            else {
                // lancia un errore perché la parentesi non è stata aperta
            }
        }
    }
```

Si può notare come il codice scritto rispecchi la grammatica: la scelta espressa dalla barra verticale si traduce direttamente in condizioni che controllano quale delle opzioni è da scegliere, le parentesi quadre invece si traducono in condizioni dove però non è strettamente necessario scegliere. La ripetizione indicata dalle parentesi graffe si traduce in un ciclo while. Queste funzioni si chiamano a vicenda e sono ricorsive (da qui il nome "a discesa ricorsiva").

Il parser ha anche il compito di identificare i problemi sintattici, ovvero determinare quali dichiarazioni o espressioni non sono conformi alla grammatica del linguaggio e, in tali casi, mostrare un messaggio di errore che permetta al programmatore di risolverli. Gli errori più comuni sono l'assenza del punto e virgola alla fine di una dichiarazione e la mancata chiusura delle parentesi in un espressione matematica.

Analisi semantica

L'analisi semantica è il processo, svolto di solito dopo il parsing, che riempie la tabella dei simboli (***symbol table***) e che controlla che le variabili siano dichiarate e inizializzate prima del loro utilizzo nel resto del codice.

La tabella dei simboli è una struttura che si occupa di memorizzare i dati rilevanti di tutte le variabili che vengono usate all'interno del codice sorgente. In questo progetto la symbol table è un vettore di Entry.

Un oggetto Entry ha una valore, che può essere il nome della variabile o la stringa costante, un tipo che servirà in futuro per implementare più tipi di variabile, una specie, "costante" o "string literal", lo scope di visibilità della variabile e una variabile booleana che indica se la variabile è inizializzata o no.

Il code generator controlla la tabella dei simboli ogni volta che incontra una variabile per assicurarsi che essa sia stata dichiarata e inizializzata e per recuperare il suo indirizzo per produrre il giusto codice assembly.

La tabella dei simboli viene riempita traversando l'albero sintattico. Una variabile viene inserita nella tabella dei simboli quando si trova all'interno di un nodo di tipo "vardec" (dichiarazione di variabili) e sono all'inizio non inizializzate. Vengono considerate inizializzate quando si trovano all'interno di un nodo di tipo "=" (assegnamento).

Inoltre l'analisi semantica si assicura che le variabili siano usate all'interno del proprio blocco di visibilità, in cui sono state dichiarate. Una variabile dichiarata all'interno di un ciclo while, ad esempio, non può più essere usata terminato il ciclo e deve essere dichiarata di nuovo, andando a costituire una nuova Entry (con un blocco di visibilità diverso) nella tabella dei simboli.

Generazione del codice

Il ***code generator*** è la parte che si occupa di generare il codice assembly per la macchina su cui il compiler è in esecuzione a partire dall'albero sintattico.

Questa fase, nei compiler dei linguaggi più diffusi, è preceduta da una fase di ottimizzazione. Molteplici codici intermedi vengono di solito generati e passati a successive ottimizzazioni per ottenere un codice eseguibile il più efficiente possibile. Nella generazione del codice viene scelta una traduzione al codice assembly della macchina per cui si intende compilare per ogni tipo di nodo. Un nodo di tipo "op" (operazione) genera un set di istruzioni macchina, un nodo di tipo "constant" ne genera un altro, e così via.

Assembly

Il linguaggio assembly è appena al di sopra del linguaggio macchina in quanto a facilità di lettura e scrittura da parte di un programmatore. Data la sua "vicinanza" alla macchina, assembly è diverso per ogni architettura o famiglia di processori. Inoltre l'assembly per macchine a 32 bit è diverso da quello per macchine a 64 bit in quanto queste ultime usano registri e indirizzi che sono il doppio più grandi (64 bit invece di 32). Inoltre esistono due tipi di sintassi: AT&T e Intel. Io ho scelto di usare la seconda per la sua semplicità e minore verbosità. In questo progetto le funzioni native del linguaggio, come la stampa e l'input da tastiera, sono state scritte interamente in assembly e non si affidano a librerie esterne.

Stack Machine

Il codice assembly generato in questo progetto prende in considerazione una stack machine, ovvero un computer che utilizza una pila (lo stack, che è anche il nome della parte di memoria in questione) per memorizzare e manipolare valori. Una stack machine pura ha operazioni per inserire e recuperare valori sulla pila e operazioni matematiche che prendono in considerazione l'elemento o gli elementi in cima alla

pila. Le operazioni più importanti per lavorare sullo stack sono il `push` per aggiungere un valore in cima alla pila, e il `pop` per recuperare il valore in cima alla pila. La mia macchina target (computer Macintosh con processore Intel a 64 bit) non è una stack machine pura e non ha operazioni matematiche che agiscono direttamente sullo stack quindi lo stack viene utilizzato per memorizzare le variabili locali e per memorizzare temporaneamente i valori che verranno spostati nei registri prima che delle operazioni matematiche vengano eseguite su di essi. Il risultato di un'operazione viene poi sempre posto in cima allo stack per essere utilizzato successivamente.

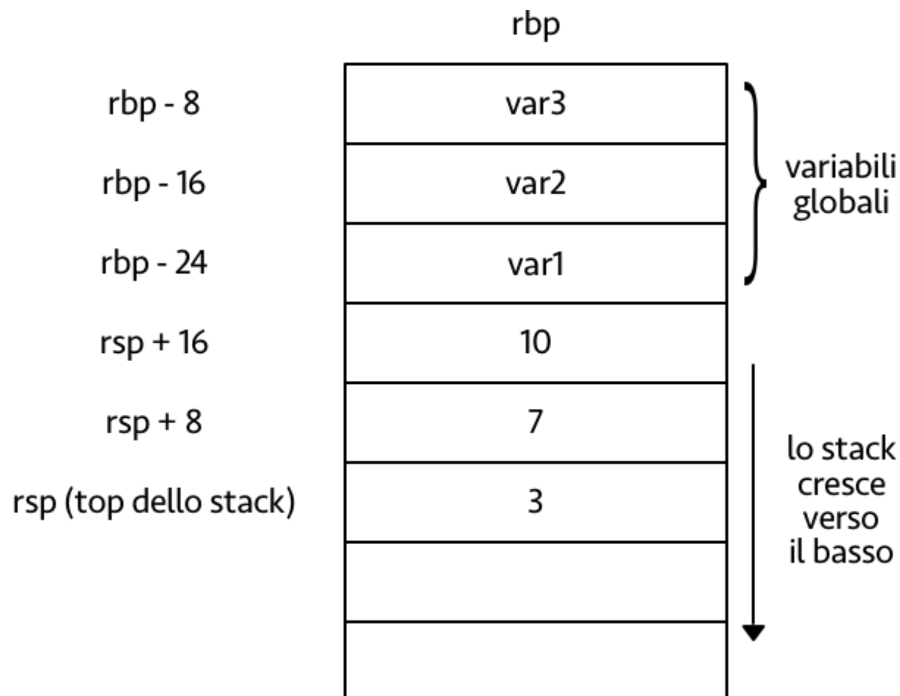
La generazione del codice per una stack machine è più semplice in quanto lo stack è una struttura molto semplice e non pone dei limiti sulla quantità di variabili da memorizzare (un limite fisico esiste, ma non è rilevante in questo caso perché praticamente irraggiungibile), mentre utilizzando i registri il programmatore deve tenere conto del numero limitato di registri e tenere quindi in memoria solo quanto necessario includendo quindi problemi sull'efficienza che nel caso di un compilatore educativo non sono necessari. L'unico svantaggio di lavorare sullo stack sono le minori prestazioni, in quanto le operazioni sui registri sono molto più veloci di quelle sulla memoria.

Allocazione della memoria

Lo stack può essere riferito attraverso due registri che contengono gli indirizzi di memoria rispettivamente della cima dello stack e della base dello stack. Lo base dello stack (registro rbp, ***base pointer***) all'inizio dell'esecuzione del programma punta allo stesso indirizzo della cima dello stack (registro rsp, ***stack pointer***).

Lo stack cresce verso il basso, ovvero ogni volta che si aggiunge un valore l'indirizzo indicato dal registro rsp viene diminuito del numero di byte del valore aggiunto. Al contrario, recuperare un valore dallo stack incrementa l'indirizzo indicato da rsp. Come fare per avere un'area di memoria statica in questa struttura dinamica? ***rbp*** e ***rsp*** devono puntare allo stesso indirizzo, quindi sottraendo al registro rsp il numero di byte che vogliamo riservare per le variabili del programma si crea un'area di memoria statica rispetto a rbp perché alle variabili si può accedere con rbp meno

l'offset di byte del valore da recuperare. Lo stack può continuare a crescere verso il basso rispetto a rsp.



Le stringhe costanti non vengono allocate sullo stack, vengono invece memorizzate nel segmento data, una porzione di memoria statica per le variabili globali (già inizializzate) allocata dal sistema operativo per ogni programma in esecuzione.

Implementazione

Prima di generare il codice, nel file di output vengono scritte le istruzioni per l'allocazione dello spazio sullo stack necessario alle variabili locali e vengono inizializzate le stringhe costanti.

L'albero sintattico, prima di passare al code generator, viene astratto, ovvero viene semplificata la sua rappresentazione, riducendone i nodi e rendendolo più generale. Numerosi nodi, ad esempio, dopo il parsing hanno un solo nodo collegato ad essi, quindi eliminando questi nodi inutili facilitiamo il traversamento dell'albero e conserviamo solo i nodi che interessano al code generator.

```
Value: STATEMENTS
    Value: VARDEC
        Value: IDENT
            Value: a, type: ident
    Value: VARASSIGN
        Value: IDENT
            Value: a, type: ident
        Value: =, type: eq
            Value: EXPR
                Value: TERM
                    Value: FACTOR
                        Value: CONST
                            Value: 0, type: constant
```

```
Value: STATEMENTS
    Value: VARDEC
        Value: a, type: ident
    Value: =, type: eq
        Value: a, type: ident
        Value: 0, type: constant
```

Lo stesso albero sintattico, non astratto (a sinistra) e astratto (a destra).

Il codice viene generato a partire dai nodi terminali, quindi il traversamento dell'albero viene fatto in profondità, per poi risalire alla radice. La sequenza di istruzioni corrispondente ad ogni nodo viene scritta sul file di output Le variabili vengono sostituite dai loro indirizzi, li stessi memorizzati nella tabella dei simboli.

Di seguito il codice del code generator la cui parte in assembly è stata sostituita da commenti che spiegano approssimativamente le istruzioni generate. A causa di qualche incongruenza nella definizione degli attributi dell'oggetto di tipo Nodo, la maggior parte delle volte il dato rilevante al code generator è il valore del nodo e non il tipo, ma è solo in effetti è solo il nome dell'attributo a cambiare, non la vera natura del Nodo. A questa funzione vengono passati, oltre all'albero sintattico astratto e alla tabella dei simboli, due contatori (omessi dal codice qui sotto) che tengono traccia dei blocchi di visibilità e servono per recuperare dalla tabella dei simboli le variabili corrette per il blocco di visibilità per cui si stanno generando le istruzioni assembly.

```
def generate(AST: Node, symTable: ArrayBuffer[Entry]): Unit = {
    if (AST.nodeType == "eq") {
        // genera il codice per l'espressione matematica (chiamata ricorsiva)
        // recupera l'indirizzo in memoria della variabile a cui è stato
        // assegnato il valore dell'espressione matematica, quindi sposta il
        // valore che si trova in cima allo stack (il risultato
        // dell'espressione) a questo indirizzo
    }
    else if (AST.nodeType == "ident") {
        // recupera l'indirizzo in memoria di questa variabile
```

```
    // metti il valore di questa variabile in cima allo stack
}
else if (AST.nodeType == "constant") {
    // metti il valore di questa costante in cima allo stack
}
else if (AST.nodeType == "strliteral") {

    // questo tipo di nodo corrisponde ad una stringa
    // metti l'indirizzo di questa stringa in cima allo stack
}
else if (AST.nodeValue == "+") {

    // genera il codice dei due fattori da sommare (chiamata ricorsiva)
    // sposta il valore del primo fattore in un registro
    // sposta il valore del secondo fattore in un altro registro
    // somma i valori dei due registri e metti il risultato in cima allo
    // stack
}
else if (AST.nodeValue == "-") {

    // genera il codice dei due fattori da sottrarre (chiamata ricorsiva)
    // sposta il valore del primo fattore in un registro
    // sposta il valore del secondo fattore in un altro registro
    // sottrai il valore del secondo registro al valore del primo registro
    // metti il risultato in cima allo stack
}
else if (AST.nodeValue == "/") {

    // genera il codice dei due fattori da dividere (chiamata ricorsiva)
    // sposta il valore del primo fattore in un registro
    // sposta il valore del secondo fattore in un altro registro
    // dividi il valore del primo registro per il valore del secondo
    // registro e metti il risultato in cima allo stack
}
else if (AST.nodeValue == "*") {

    // genera il codice dei due fattori da moltiplicare (chiamata
    // ricorsiva)
    // sposta il valore del primo fattore in un registro
    // sposta il valore del secondo fattore in un altro registro
    // moltiplica i valori dei due registri e metti il risultato in cima
    // allo stack
}
else if (AST.nodeValue == "NEG") {

    // genera il codice del valore da negare (chiamata ricorsiva)
    // recupera il valore (che si trova in cima allo stack)
    // negalo e rimettilo in cima allo stack
}
else if (AST.nodeValue == "CONDBR") {

    // genera il codice dei valori da comparare (chiamata ricorsiva)
    // quindi in base all'operatore relazionale (<, >, <>, =, <=, >=)
    // genera il codice per continuare l'esecuzione quando la condizione
```

```
        // non è soddisfatta
        // quindi genera il codice del blocco di quando la condizione è
        // soddisfatta (chiamata ricorsiva)
        // quindi, se c'è un blocco 'else', genera il codice per continuare
        // l'esecuzione dopo l'else (per non eseguire il blocco 'else' nel
        // caso in cui la condizione dell'if sia stata soddisfatta)
        // quindi genera il codice del blocco 'else' (chiamata ricorsiva)
    }
    else if (AST.nodeType == "relop") {

        // genera il codice del primo valore da comparare (chiamata ricorsiva)
        // sposta il primo valore, che si trova in cima allo stack, in un
        // registro
        // genera il codice del secondo valore da comparare (chiamata
        // ricorsiva)
        // sposta il secondo valore, che si trova in cima allo stack, in un
        // altro registro
        // genera l'istruzione di comparazione tra i due registri
    }
    else if (AST.nodeValue == "IFBLOCK") {
        // genera il codice del blocco 'if' (chiamata ricorsiva)
    }
    else if (AST.nodeValue == "ELSEBLOCK") {
        // genera il codice del blocco 'else' (chiamata ricorsiva)
    }
    else if (AST.nodeValue == "PRINT") {

        // genera il codice dei parametri da stampare (chiamata ricorsiva)
        // i parametri da stampare si trovano quindi nello stack
        // recupera ogni parametro sullo stack e stampalo chiamando le due
        // funzioni di stampa (numeri e stringhe utilizzano due funzioni
        // diverse)
    }
    else if (AST.nodeValue == "PRINTLN") {

        // come per il nodo di tipo 'PRINT', ma stampa un '\n' dopo gli altri
        // parametri
    }
    else if (AST.nodeValue == "LOOPST") {

        // genera il codice dei valori da comparare (chiamata ricorsiva)
        // quindi in base all'operatore relazionale (<, >, <>, =, <=, >=)
        // genera il codice per continuare l'esecuzione quando la condizione
        // non è soddisfatta
        // quindi genera il codice del blocco di quando la condizione è
        // soddisfatta (chiamata ricorsiva)
        // e genera l'istruzione per tornare indietro nel codice e controllare
        // ancora la condizione (creando di fatto un ciclo)
    }
    else if (AST.nodeValue == "WHILEBLOCK") {
        // genera il codice del blocco 'while' (chiamata ricorsiva)
    }
    else if (AST.nodeValue == "VARDEC") {
```

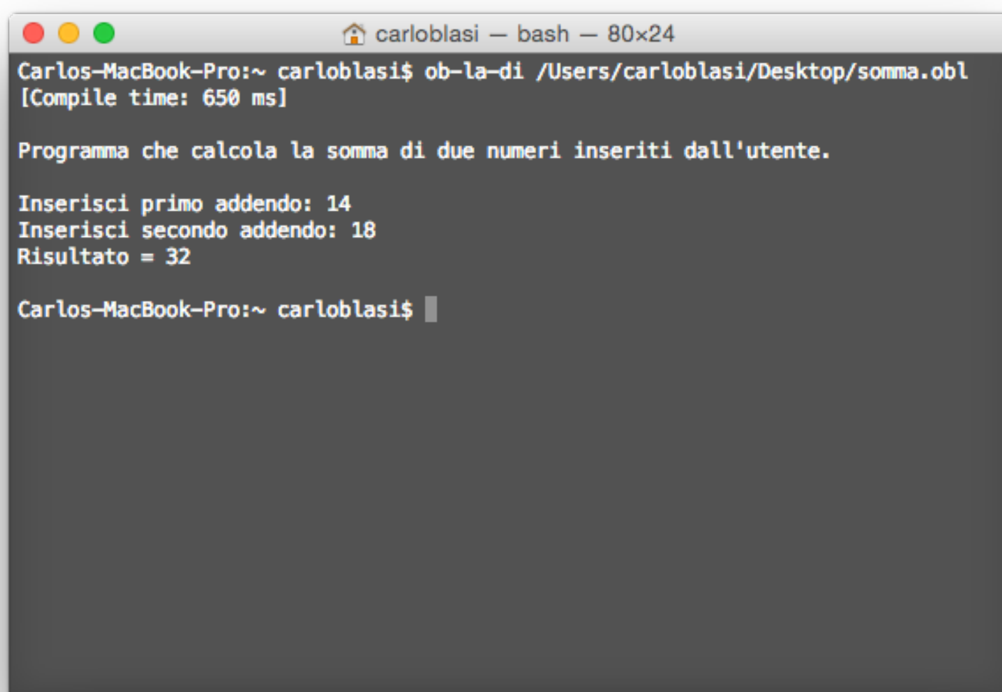
```
        // caso aggiunto per evitare che i nodi collegati ad esso vengano
        // presi in considerazione dal code generator, come succederebbe se
        // questo caso fosse lasciato all'ultimo 'else'
        // è un eccezione perché contiene nodi 'ident' che sono considerati
        // importanti per il code generator nonostante nessuna istruzione deve
        // essere generata da questi nodi
    }
    else if (AST.nodeValue == "INPUT") {

        // chiama la funzione per l'input da tastiera
        // metti il valore inserito dall'utente in cima allo stack
        // recupera l'indirizzo della variabile a cui deve essere assegnato
        // questo valore
        // quindi sposta il valore che si trova in cima allo stack a questo
        // indirizzo
    }
    else {

        // se il tipo di nodo non è rilevante, genera il codice dei nodi
        //collegati ad esso (chiamata ricorsiva)
    }
}
```

Utilizzo

Il progetto è disponibile per il download all'indirizzo https://github.com/carloblasi/OB-LA-DI. Nella cartella `release` si trovano il file .jar, lo script per l'esecuzione e il file per il syntax highlighting di Sublime Text. Per installare il compilatore bisogna spostare il file .jar e lo script nella cartella `/usr/local/bin` raggiungibile dal Finder usando lo shortcut Command + Maiusc + G e scrivendo tale percorso. Infine l'assembler NASM è necessario per terminare la compilazione, esso può essere installato con Xcode, Homebrew o MacPorts. A questo punto si può usare il compilatore invocandolo da terminale con il comando `ob-la-di` e passandogli il percorso del file da compilare. Se non viene passato nessun parametro o il file non esiste o non è della giusta estensione (.obl) viene mostrato un errore. In caso di compilazione riuscita viene mostrato il tempo impiegato dal compilatore e il programma viene eseguito. Il file eseguibile generato viene salvato all'interno della cartella dell'utente.



Compilazione ed esecuzione di un programma scritto il OB-LA-DI.

Compatibilità

Questo compilatore è pensato per essere eseguito su sistemi Mac OS X con processore Intel 64 bit, approssimativamente tutti i computer Apple realizzati dal 2007 ad oggi. Nonostante i modelli siano numerosi e diversi l'architettura resta la stessa e, di conseguenza, anche il codice assembly da generare è lo stesso.

La compatibilità con Windows o Linux non è stata aggiunta per semplificare il progetto ma, grazie alla modularità del progetto, sarebbe sufficiente modificare solo il generatore del codice in modo che produca il codice assembly adeguato.

Testing

Il progetto è stato testato con la riuscita compilazione ed esecuzione di un programma (il famoso "Fizz Buzz") sui seguenti dispositivi:

- MacBook Pro fine 2011, Mac OS X Yosemite
- MacBook Air inizio 2014, Mac OS X Yosemite
- MacBook Air metà 2013, macOS Sierra
- iMac fine 2012, Mac OS X El Capitan
- MacBook Pro fine 2010, Mac OS X Yosemite

Si ringraziano per la disponibilità Alekos Filini, Aurelio Pennacchio e Andrea Cassamali.

Conclusione

Costruire un compiler mi ha posto davanti a problemi (spesso grossi) che ho avuto la fortuna di superare grazie alle infinite risorse trovate su internet. Inoltre è stata un'esperienza che si è protratta (anche se irregolarmente a causa degli studi) per diversi mesi e mi ha permesso di assumere nuove interessanti conoscenze su quello che succede dietro le quinte di ogni programma e ho imparato ad utilizzare un nuovo linguaggio che usa un paradigma diverso da quello imparato tra i banchi e che vorrei davvero continuare ad usare, magari anche in ambito lavorativo.

Sono soddisfatto del risultato a cui sono arrivato, nonostante avrei voluto aggiungere molte funzionalità che avrebbero reso il progetto più completo e professionale.

Sitografia e bibliografia

- Wikipedia (www.wikipedia.com)
- Stackoverflow (www.stackoverflow.com)
- NASM Assembly Language Tutorials (www.asmtutor.com)
- NASM Tutorial (http://cs.lmu.edu/~ray/notes/nasmtutorial)
- Introduction to X86-64 Assembly for Compiler Writers (https://www3.nd.edu/~dthain/courses/cse40243/fall2015/intel-intro.html)
- Diversi siti di corsi universitari di scienze informatiche
- “Let’s build a compiler!” (newsletter) di Jack Crenshaw, 1988
- “Compiler Construction” di Niklaus Wirth, 1996
- “Scala Cookbook: Recipes for Object-Oriented and Functional Programming” di Alvin Alexander, 2013