A basso livello un qualsiasi elaboratore può eseguire solo le istruzioni implementate dall’architettura su cui si basa, queste istruzioni non sono però pensate per essere particolarmente comprensibili ad una persona ma si basano sui componenti fisici che attueranno poi l’operazione richiesta e che compongono l’unità logica del computer.

Con il tempo sono aumentate dimensioni e , conseguentemente, complessità dei programmi esistenti e questo ha spinto alla creazione di istruzioni più vicine al programmatore per favorire il lavoro di più persone sullo stesso progetto e abbassare il tempo richiesto per capire il funzionamento di un codice già esistente e iniziare a lavorarci.

Nasce la necessità di trasformare istruzioni di alto livello in basso livello che sono come detto le uniche capibili dal computer.

Il progetto di tesi portato riguarda proprio uno di questi programmi, in particolare un interprete scritto in C del linguaggio di programmazione Simpla, che è stato introdotto e discusso durante il corso di tecnologie dei linguaggi artificiali.

Un interprete è un programma che esegue codice sorgente di un certo linguaggio sulla macchina su cui è eseguito.

Vediamo un esempio di programma scritto in linguaggio Simpla, individuiamo 3 sezioni : la prima in alto dove è possibile dichiarare delle variabili che avranno scope globale, la seconda che permette la   
dichiarazione di un numero arbitrario di funzioni e infine la terza che contiene il corpo del Main.

…

L’interprete è un programma avente in input il codice sorgente del linguaggio che deve interpretare (Simpla in questo caso) e che restituisce un output non necessariamente testuale che rappresenta l’attuazione delle istruzioni scritte ad alto livello. In mezzo c’è un lavoro non noto ad ora che esploriamo dividendolo in sezioni, la prima è l’analisi lessicale.

Qui avviene la lettura del testo/codice in ingresso e in base alle regole interne si abbina ad ogni carattere o gruppo di caratteri un token.

Un token può essere visto come una classe, e in questo caso i caratteri sarebbero le istanze. Ovvero a caratteri diversi può corrispondere lo stesso token se sono dello stesso tipo.  
Ad esempio se venissero letti i caratteri “numero” e “resto” a entrambi verrebbe associato il token identificatore, sono entrambe variabili.

È inoltre importante aggiungere che caratteri inutili vengono scartati (fig).

L’implementazione avviene tramite uno strumento chiamato Flex che è un generatore di analizzatori lessicali.

Fornendogli un insieme di coppie expreg : frammento di codice C lui genera un programma C che scannerizza un input e quando trova un match nelle expressioni reg esegue l’azione associata.

Ad esempio la prima riga della slide matcha solo la parola integer e, se la trova, ritorna un token integer.

Ci sono situazioni nelle quali restituire il solo token non è sufficiente per preservare l’intera informazione.  
Un caso è quello visto prima in cui due caratteri o parole diverse tra loro sono collegate allo stesso simbolo, perderei l’informazione di quale parola ho letto, l’istanza usando la comparazione di prima. Per evitarlo viene aggiunto un attributo lessicale che conterrà la parola stessa (vedi slides).

L’analisi sintattica, basandosi sulla grammatica non contestuale del linguaggio controlla la correttezza di ogni frase.

La grammatica è composta da un insieme di regole tramite le quali è possibile derivare ogni frase legale del linguaggio. La forma di queste regole è come quella sulle slides, in questo esempio …

Anche questa sezione è stata generata tramite uno strumento, Bison. Partendo da un insieme di coppie regola grammaticale : frammento codice C genera un analizzatore sintattico.

La funzione dei frammenti di codice è la costruzione di un albero esplorabile che mantenga ordine e contenuto informativo del codice sorgente (spiega figura).

L’analisi semantica ha il compito di calcolare tutte le informazioni che sono necessarie per l’interpretazione ma non sono ricavabili dall’analisi sintattica.

Un esempio è controllare se la chiamata di una certa funzione è preceduta dalla sua dichiarazione, potrebbe essere visto come un compito della sintassi però l’utilizzo di una grammatica non contestuale nella sezione precedente non permette di controllare errori basati sul contesto.

Un altro compito molto importante è il type checking, ad esempio nell’assegnamento è chiaramente voluto che il tipo della variabile assegnata e il tipo del valore assegnato coincidano o comunque siano compatibili. Siccome questo passaggio è eseguito prima dell’interpretazione la sua efficacia dipende dalla staticità del linguaggio. Ovvero se non fosse fortemente tipizzato il tipo delle variabili potrebbe non essere conosciuto fino all’esecuzione e quindi non verificabile in questo stadio del processo.

In ogni caso in Simpla il controllo è possibile ed è stato implementato, per farlo però bisogna avere un modo per tenere traccia di tutte le variabili, funzioni e i loro tipi. La struttura utilizzata per farlo è la symbol table.

In maniera molto semplice consiste in una tabella in cui ogni entrata è una variabile/funzione e i vari campi come vedremo dopo sono utili per informazioni di controllo. Per come vogliamo che funzioni ogni entrata dev’essere univoca, ovvero se esiste già una variabile con identificatore ‘a’ non ne può esistere un'altra. Un modo molto comodo per implementare questo meccanismo sono le hashtable, tabelle molto efficienti in lettura e scrittura e che si basano proprio sull’unicità di una chiave. Purtroppo per come funzionano internamente a meno di essere disposti a sprecare una porzione molto grande di memoria può succedere di incorrere in situazioni chiamate collisioni, SPIEGA SE TEMPO TUTTA STA ROBA.

Un campo interessante della tabella effettivamente impiegata è ambiente. Contiene il nome dello scope al quale la symbol table si riferisce, ne avremo sicuramente uno globale e poi uno per ogni funzione dichiarata all’interno del codice sorgente. In questo modo posso fare un type checking avendo a disposizione informazioni su quali variabili sono disponibili dove.

La fase di esecuzione del codice fa normalmente uso di uno stack, le caratteristiche di questo però sono molto dipendenti da ciò che il linguaggio implementato consente. Ad esempio Simpla che permette la chiamata di funzioni all’interno del corpo di altre funzioni necessita di uno stack dinamico, perché non si può in nessun modo sapere durante la scrittura dell’interprete la dimensione necessaria.

Quando una funzione viene chiamata viene creato un ambiente sulla pila (chiamato RA) nel quale potrà memorizzare i valori delle variabili e le informazioni necessarie per ritornare all’esecuzione del chiamante quando la funzione avrà finito il suo lavoro

A questo si aggiunge uno spazio dedicato alle variabili globali e uno per il main che però può essere visto come un RA dove le informazioni per riprendere il chiamante consistono nella restituzione dell’esecuzione al sistema operativo.

Nella pratica sono stati creati due pile, la prima, stack degli oggetti, contenente semplicemente tutti i valori con il loro tipo che sono in quel momento in utilizzo nel codice. La seconda, stack di attivazione, che tiene traccia di tutti i RA creati, per ognuno sa dove risiede il primo elemento del gruppo sullo stack degli oggetti, sa quanti oggetti appartengono al suo gruppo, e punta alla symbol table di cui è istanza.

Possiamo creare un puntatore anteponendo al nome della variabile uno o più asterischi. Dopo averla dichiarata nel codice ne potremo fare uso in due modi : chiamando la variabile senza asterischi davanti otterremo l’indirizzo a cui questa punta, anteponendoli invece otterremmo il valore che viene puntato.

Il numero di stelle indica il livello di profondità, ovvero il numero di salti per trovare il valore puntato, e non ha limite di profondità.

L’operatore & viene usato per chiedere ad una variabile il suo indirizzo in memoria, non quello della variabile che sta puntando ma il suo attuale.

SPIEGA ESEMPIO.