0.1 Interprete SimplanPlus

L’interprete di **SimpLanPlus** prende in input il codice precedentemente generato dal compilatore e lo esegue un’istruzione alla volta.

0.1.1 Bytecode

Per ogni nodo dell’AST è presente un metodo *String codeGeneration()* che genera il bytecode rispettando la grammatica presente nel file *SVG.g4*. Ogni istruzione è composta da:

- **Label istruzione**: è una stringa obbligatoria che identifica un’istruzone.
- **Primo argomento**: serve per specificare un registro o un’etichetta. È una stringa opzionale perché non tutte le istruzioni prevedono questo argomento (es. POP).
- **Secondo argomento**: serve per specificare un registro o un numero. È una stringa opzionale.
- **Terzo argomento**: serve per specificare un registro, un numero o un’etichetta. È una stringa opzionale.

Per ogni istruzione letta viene creato un oggetto di tipo *Instruction*, definito dalla classe Instruction.java.

0.1.2 CPU e memoria

L’interprete simula il comportamento di una CPU e di una memoria. La CPU ha 9 registri:

- **Stack pointer ($sp)**: punta alla cima dello stack
- **Instruction pointer ($ip)**: indica la prossima istruzione da eseguire
- **Heap pointer ($hp)**: punta alla prima posizione libera dello *heap*
- **Frame pointer ($fp)**: punta all’access link corrente relativo al frame attivo
- **Access link ($al)**: registro utilizzato per attraversare la catena statica degli scope
- **Base stack pointer ($bsp)**: punta alla cella di memoria contenente il vecchio stack pointer, utile nel caso si esca da una funzione prima della fine delle istruzioni tramite un return
- **Accumulatore ($a0)**: registro utilizzato per salvare il valore calcolato da alcune espressioni
- **Registro generico ($t1)**: utilizzabile liberamente all’interno del programma
- **Return address ($ra)**: registro utilizzato per salvare l’indirizzo di ritorno una volta usciti da un frame

La CPU esegue le istruzioni passate al costruttore dall’interprete. Dopodiché, ad ogni iterazione, si accede all’istruzione indicata dal registro *$ip* che poi viene incrementato di uno per passare ad eseguire l’istruzione successiva. L’operazione da eseguire dipende dal campo *code* dell’oggetto *Instruction*.
Per gestire i registri si utilizzano le funzioni *regStore* e *regRead* per aggiornare e leggere il valore contenuto nel registro opportuno.

Per quanto riguarda la *memoria*, questa viene gestita come un array di *Cell*. La memoria è divisa logicamente nello stack (che cresce verso il basso e quindi indici della memoria minori) e lo heap (che parte da indice 0 e cresce verso l'alto e quindi verso indici maggiori).
La classe *Cell* rappresenta una singola cella di memoria. I campi presenti in questa classe sono:

- *Integer val*: rappresenta il valore salvato nella cella di memoria
- *boolean pointed*: indica se la cella della memoria è puntata oppure è vuota e quindi può essere scritta

Sono stati implementati i seguenti metodi per facilitare la gestione delle celle di memoria:

- *void free()*: libera una cella di memoria
- *Integer read()*: ritorna il dato contenuto nella cella di memoria
- *boolean isPointed()*: ritorna un true o false se la cella è in uso oppure no
- *void allocate()*: setta a true il campo *isPointed* e quindi segna come in uso la cella di memoria
- *Integer write(int v)*: aggiorna la cella con *v*

Nella classe *Memory* sono presenti i seguenti metodi per la gestione della memoria:

- *int read(int n)* che ritorna il contenuto della memoria nella cella *n*
- *void write(int add, int val)* che scrive il valore *val* all'indirizzo *add*
- *void free(int add)* che chiama la *free* su quella cella, mettendo il campo *pointed* a false e il campo *val* a null
- *void cleanMemory(int start, int end)* che re-inizializza tutte le celle comprese tra start ed end

0.1.3 Record di attivazione

Lo *stack* in memoria è diviso in record di attivazione (*ra*). Quando si entra in un nuovo blocco, si pusha il nuovo *ra*. All'uscita dal blocco viene fatto una operazione di pop del *ra* attuale. Il record di attivazione è composto come segue:

- **Old frame pointer**: contiene l'indirizzo del frame pointer prima del push del nuovo *ra*
- **Old stack pointer**: contiene l'indirizzo dello stack pointer prima del push del nuovo *ra*
- **Return address**: contiene il valore dell' *$ip* e quindi l'istruzione da eseguire una volta terminata l'esecuzione attuale
- **Access link**: serve per accedere alle variabili dichiarate in scope esterni
- **Spazio di memoria riservato alle variabili**

0.1.4 Come funziona l’interprete?

Il funzionamento dell’interprete possiamo vederlo in *SVM.java*. Il costruttore prende in input il codice passato all’interprete e lo assegna al campo code. La classe SVM ha i seguenti campi privati:

- **Instruction[] code**: andrà a contenere la sequenza ordinata di istruzioni che l’interprete dovrà eseguire
- **Memory memory**: rappresenta la memoria dell’interprete che è logicamente divisa in *heap* e *stack*
- **I registri**: **ip** (inizializzato a 0), **sp** (inizializzato a *memory_size*), **hp** (inizializzato a 0), **fp** inizializzato a *memory_size1*, **ra** (inizializzato a null), **al** (inizializzato a null), **bsp** (inizializzato a *memory_size*)
- **int[] a = new int[10]**: insieme dei registri temporanei

Il metodo *cpu()* della classe SVM esegue le istruzioni all’interno di *code* mediante un ciclo *while(true)* fino a quando lo *heap* supera lo *stack* dando un errore di *Out of memory*. Ad ogni iterazione viene letta l’istruzione da eseguire, vengono salvati gli argomenti che possiete e incrementato l’*ip*. Tramite un grosso switch si gestisce il comportamento in base all’istruzione in esecuzione.