# ­­­­­Interpretazione del testo e scelte progettuali

L’obiettivo di questo documento è quello di fornire un quadro generale dell’interpretazione e delle scelte progettuali che sono state prese per il progetto:

1. Le specifiche affermano che “i tag dei nodi servono a caratterizzare in maniera univoca i nodi dell’albero”, ma non viene chiarito se questa proprietà debba essere garantita a run-time (dall’interprete eval) oppure staticamente (da un ipotetico parser). Si assume che sia compito del parser del linguaggio occuparsi di garantire all‘interprete di valutare alberi nei quali non esistono nodi con tag identici, in modo da ottimizzare l’esecuzione dei programmi. Questo controllo può essere fatto staticamente grazie alla struttura degli alberi di espressioni:

|  |
| --- |
| type ide = string;; type exp = … | ETree of tree | …  and tree = Empty | Node of **ide** \* exp \* tree \* tree;; |

Il tag dei Nodi è di tipo ide, ovvero una stringa il cui valore NON dipende dall’ambiente di esecuzione, per questo motivo questo controllo può essere fatto staticamente dal parser del linguaggio (se invece di ide fosse stato exp, allora questo non sarebbe stato possibile).

1. I tipi esprimibili sono stati estesi con gli alberi di espressioni in quanto, come specificato dal testo, le nuove operazioni Select e ApplyOver devono restituire un albero di espressioni opportunamente valutato.
2. Il type checker del linguaggio è stato esteso in modo da poter riconoscere gli alberi di espressioni.
3. E’ stato aggiunto il seguente pattern alle funzioni primitive in modo da eliminare i warning prodotti dal compilatore a causa dei pattern matching non esaustivi: \_ -> **failwith**("Type error - typecheck failed")
4. Di seguito le estensioni fatte alla funzione di valutazione delle espressioni del linguaggio “eval”:
   * 1. **Select(i,et):** affinché eval possa valutare correttamente questa espressione, occorre che “i” rappresenti un identificatore (ide), mentre “et” un albero di espressioni (ETree). Se le premesse sono soddisfatte eval cerca, con una visita anticipata dell’albero (sequenza di visita ricorsiva: nodo, sottoalbero sinistro, sottoalbero destro), il primo nodo con tag uguale a "i". Se un nodo con tag uguale a "i" viene individuato, viene restituito l’albero radicato in esso, altrimenti viene restituito un albero vuoto.
     2. **ApplyOver(tags,f,et):** affinché eval possa valutare correttamente questa espressione occorre che tags rappresenti una lista di identificatori (ide list), “f” una funzione anonima (Fun) e “et” un albero di espressioni (ETree). Se le premesse sono soddisfatte eval applica la funzione f su tutti i nodi dell’albero “et” aventi un tag presente nella list “tags”. In questo caso la scelta fatta nel punto 1 è ininfluente, perché in ogni caso tutto l’albero viene visitato e l’ordine con cui viene fatto non ha importanza.

# Compilazione ed esecuzione

Il file “evalFunEnvFull.ml” contiene il codice del linguaggio visto durante il corso, mentre il file “extended\_evalFunEnvFull.ml” contiene il codice esteso per poter gestire gli alberi di espressioni, più una batteria di test che ha lo scopo di valutare che l’estensione sia stata effettuata correttamente.

|  |  |
| --- | --- |
| Comando shell | Descrizione |
| ocamlbuild extended\_evalFunEnvFull.native | Compila il programma ocaml “extended\_evalFunEnvFull.ml”. Per eseguire il programma occorre eseguire il file generato dalla compilazione: “extended\_evalFunEnvFull.native” |
| ocamlbuild extended\_evalFunEnvFull.native -- | Compila ed esegue il programma ocaml “extended\_evalFunEnvFull.ml”. |