Progetto LC2, Parte 3
Gruppo 1

Studenti:
Foschiani Luca
Marinato Riccardo
Passalenti Andrea

Assunzioni

Nel seguente progetto abbiamo considerato le scelte di sintassi concreta definite dal linguaggio *Go*. A seguire vengono riassunte una serie di assunzioni.

- Non è possibile dichiarare più di una funzione con lo stesso identificatore;
- non è consientito dichiarare delle variabili con lo stesso identificatore nello stesso blocco, mentre è possibile se avviene su blocchi (e sottoblocchi) diversi;
- è stato usato l'operatore '!' nelle condizioni degli if per veicolare correttamente i salti condizionati;
- nel corpo delle procedure non deve presentare l'istruzione return, sempre richiesta invece nei corpi delle funzioni;
- sono stati implementati solamente i comandi per il controllo della sequenza richiesti nel testo del progetto (*condizionali semplici*, *iterazione indeterminata*);
- le guardie booleane dei controlli di sequenza vengono gestite tramite short-cut mentre le altre espressioni booleani vengono gestite senza short-cut.

Scelte implementative

- Abbiamo generato *lexer* e *parser* tramite il tool BNFC, partendo dalla definizione di una gramatica iniziale;
- la gestione del *Type Checker* e del *Three Address Code* vengono fatte all'interno del parser;
- le funzioni *write* sono trattate come statements che prendendono come argomento una right expression, mentre le funzioni *read* vengono viste come right expressions e hanno una lista di parametri di input vuota.

Grammatica

Di seguito viene riportata la grammatica di partenza.

```
comment "//";
comment "/*" "*/";

entrypoints Start;

rules     Boolean ::= "true" | "false" ;

ExpAnd.         RExp ::= RExp "&&" RExp ;
ExpOr.          RExp ::= RExp "||" RExp ;
ExpNot.         RExp ::= "!" RExp ;
ExpEq.          RExp ::= RExp "==" RExp ;
ExpNeq.         RExp ::= RExp "!=" RExp ;
ExpLt.          RExp ::= RExp "<" RExp ;
ExpLtE.         RExp ::= RExp "<=" RExp ;
ExpGt.          RExp ::= RExp ">" RExp ;
ExpGtE.         RExp ::= RExp ">=" RExp ;
ExpAdd.         RExp ::= RExp "+" RExp ;
ExpSub.         RExp ::= RExp "-" RExp ;
ExpMul.         RExp ::= RExp "*" RExp ;
ExpDiv.         RExp ::= RExp "/" RExp ;
ExpMod.         RExp ::= RExp "%" RExp ;
ExpNeg.         RExp ::= "-" RExp ;
ExpRef.         RExp ::= "&" LExp ;
ExpFuncEmpty.   RExp ::= Id "(" ")" ;
ExpFunc.        RExp ::= Id "(" [RExp] ")" ;
ExpVal.         RExp ::= Val ;
ExpLExp.        RExp ::= LExp ;
ExpPar.         RExp ::= "(" RExp ")";
StRead.         RExp ::= ReadT "(" ")";

Int .           Val ::= Integer ;
Float .         Val ::= Double ;
Char .          Val  ::= Char ;
String .        Val ::= String ;
Bool .          Val  ::= Boolean ;
```

```
rules     ReadT  ::= "readInt" | "readFloat" | "readChar" | "readString" ;

ExpId.    LExp ::= Id ;
ExpArr.   LExp ::= LExp "[" RExp "]" ;
ExpDeref. LExp ::= "*" RExp ;

separator nonempty RExp "," ;

Entry.          Start ::= "package" Id [Decl];

separator Decl "";

DeclVar.         Decl ::= "var"  [Id] Type;
DeclVarInit.     Decl ::= "var"  [Id] "=" [RExp];
DeclVarTypeInit. Decl ::= "var"  [Id] Type "=" [RExp];
DeclVarShort.    ShortVarDecl ::= [Id] ":=" [RExp];

TVoid.           Type ::= "void";
TInt.            Type ::= "int";
TBool.           Type ::= "bool";
TFloat.          Type ::= "float";
TChar.           Type ::= "char";
TString.         Type ::= "string";
TArray.          Type ::=  "[" Integer "]" Type;
TPointer.        Type ::= "*" Type;

token Id (letter | '_')(letter | digit | '_')*;

separator Param ",";

DeclFun.         Decl ::= "func" Id "(" [Param] ")" Type Block;
DeclProc.        Decl ::= "func" Id "(" [Param] ")" "void" Block;

separator nonempty Id ",";

Parameter.        Param ::=  [Id] Type;
ParameterPass.    Param ::=  Pass [Id] Type;
```

```
PassVal.          Pass  ::=  "val";
PassRef.          Pass  ::=  "ref";

separator Stmt "";

BodyBlock. Block ::= "{" [Stmt] "}";

StDecl.           Stmt ::= Decl;
StBlock.          Stmt ::= Block;
StSmpl.           Stmt ::= StmtSmpl;
StIf.             Stmt ::= "if" RExp Block;
StIfElse.         Stmt ::= "if" RExp Block "else" Block;
StWhile.          Stmt ::= "for" RExp Block;
StBreak.          Stmt ::= "break";
StContinue.       Stmt ::= "continue";
StReturn.         Stmt ::= "return" RExp;
StWrite.          Stmt ::= WriteT "(" RExp ")";

rules WriteT ::= "writeInt" | "writeFloat" | "writeChar" | "writeString";

StShortVarDecl.   StmtSmpl ::= ShortVarDecl;
StExp.            StmtSmpl ::= RExp;
StAsgn.           StmtSmpl ::= LExp "=" RExp;
```

Type Checker

Il Type Checker viene gestito interamente all'interno del parser. In particolare per ogni espressione valutata si effettua un controllo diretto attraverso le funzioni *Ok* e *Bad* che, in caso di errore, gestiscono l'errore tramite la stampa di un messaggio a video indicante la linea dell'errore e la causa.
Un errore di lessico viene a distinguersi da uno di tipo direttamente in fase di stampa dello dell'errore, riportando "*Lexical error*" nel primo caso e "*Type error*" nel secondo.
Tutte le funzioni di controllo utilizzate dal type checker sono state definite internamente al parser.
Tra le diverse assunzioni fatte in fase di type checking, abbiamo voluto tenere conto in modo automatico di una conversione dal tipo int al tipo float.

Three Address Code

Il Three Address Code viene generato interamente nel parser, utilizzando un modulo esterno (*TAC.hs*) contenente la struttura dati per gestirne la generazione e le funzioni addette alla stampa. Inoltre ci siamo appoggiati su un ulteriore modulo (*Env.hs*) contenete tutte le funzioni per la gestione dell'environment (inserimento, cancellazione e ricerca di variabili/funzioni). In questo modulo sono stati dichiarati i tipi di dati *ElemVar* e *ElemFun* contenenti rispettivamente tutte le informazioni relative alle variabili ed alle funzioni/procedure dichiarate nell'ambiente. L'ambiente viene definito come una lista di *ElemVar* ed *ElemFun*

Test Case

Sono stati preparati dei test-case significativi che possono essere eseguiti in sequenza attraverso il comando *make demo*, oppure singolarmente attraverso i comandi *make demo1*, *make demo2*, *make demo3*.