PROGETTO DI LINGUAGGI E COMPILATORI 2

PARTE 3

Gruppo 7.1

Casasola Riccardo mat. 105215

Palù Francesca mat. 100908

Piccolo Cinzia mat. 118471

A. A. 2014/2015

Specifica ed assunzioni fatte

La nostra grammatica si ispira a quella di *Chapel*, di cui è stata utilizzata la stessa sintassi concreta. Viste le documentazioni contrastanti su molti elementi sintattici, in presenza di notazioni diverse, abbiamo scelto caso per caso quella che ci sembrava più adatta. Inoltre, dato che *Chapel* è un linguaggio a oggetti, sono stati considerati solo i suoi costrutti imperativi. Riportiamo di seguito tutte le scelte effettuate:

- oltre agli oggetti, non sono presenti metodi che permettano il parallelismo;
- tra i tipi base di *Chapel* non è presente il tipo `Char`, che quindi è stato aggiunto come da consegna;
- le funzioni standard `write`, `writeln`, `read` e `readln` non sono state prese in considerazione, essendo richiesto nella consegna di implementare altre funzioni per la lettura e la stampa;
- come da consegna non sono stati presi in considerazione i tipi definiti dall'utente, per cui si è scelto di non implementare tuple, record e le operazioni ad essi associate;
- per semplificare, i `range`, utilizzati per gli array e i cicli `for` (ad esempio [1..5]), sono stati vincolati ad essere finiti, ossia non è possibile utilizzare range come [..5] o [5..] che prevedono un numero infinito di elementi. I `domain` per la modifica dinamica dei range non sono stati considerati;
- gli operatori per le operazioni di referenziazione e de-referenziazione non sono presenti nel nostro linguaggio perchè *Chapel* non prevede operatori espliciti. È presente comunque il tipo `Pointer` per poter passare argomenti per riferimento nella chiamata di funzione;
- sono state inserite le label, come da consegna, anche se *Chapel* non ne prevede l'utilizzo. Siccome il cast avviene con la produzione `Id:type`, per evitare conflitti esse vengono richiamate con il costrutto
  `nome_label::` ;
- Al fine di semplificare la grammatica, nel costrutto `if-then-else` è stata resa obbligatoria la presenza del `then`, che in *Chapel* è opzionale;
- Il costrutto `do-while` è stato sostituito con il `repeat-until`;
- *Chapel* prevede che le operazioni booleane, gli `OR` e gli `AND` possano essere effettuate con |,||, & e && . Sono state mantenute solo le opzioni || e &&;
- *Chapel* prevede che sia possibile effettuare assegnamenti sia sui parametri formali delle dichiarazioni di procedura, sia sui parametri attuali delle chiamate di procedura. Queste possibilità non sono state implementate;
- le operazioni i $+=$, $*=$, $-=$ e $/=$ non sono state considerate;

- le istruzioni `break` e `continue` sono prive di label. É stato aggiunto il costrutto `goto` per i salti a label;
- i cast ammessi dall'operatore `cast` sono i seguenti:
  - `Float` → `Int`;
  - `Int` → `Float`;
  - `Int` → `String`;
  - `Char` → `String`;
  - `Float` → `String`;
- l'operatore `+` è utilizzato sia per la concatenazione che per le operazioni aritmetiche. È quindi l'unico operatore aritmetico che accetta oltre ai tipi `Int` e `Float` anche i `Char` e `String`;
- Le operazioni di confronto (`<` , `<=` , `>` , `>=`) sono definite solo sui tipi `Int` e `Float`, mentre le operazioni `!=` e `==` sono definite su tutti i tipi.

Implementazione del parser

È stata realizzata una grammatica attributata, come spiegato a lezione. I datatype utilizzati per gli environment e tutte le funzioni richiamate nel parser sono definiti, in ordine alfabetico, all'interno del file *EnvironmentFunctions.hs*.

Environments

Sono stati utilizzati ambienti in entrata e uscita separati per memorizzare:

- variabili o costanti: `envIn, envOut`;
- funzioni: `envInF, envOutF`;
- etichette: `envInL, envOutL`;

L'attributo dell'environment in entrata memorizza l'ambiente valido per quel nodo (che viene passato dal nodo genitore ai nodi figli), e l'attributo dell'environment in uscita memorizza l'ambiente di uscita del nodo (attributo sintetizzato) . In questo modo è possibile definire delle politiche di scope ogni qual volta si applica un comando o si entra in un blocco.

Ogni ambiente è una lista di oggetti con datatype:

- variabili e costanti: `data Env = Var Id Type Bool Pass`

- `Var`: identifica che l'oggetto in questione è una variabile o una costante;

- `Id`: nome della variabile/costante;

- `Type`: tipo della variabile/costante;

- `Bool`: valore booleano utilizzato per lo scope, serve per abilitare o meno la ridefinizione di una variabile. Se è `False`, la variabile può essere ridefinita nel blocco corrente;

- `Pass`: indica il tipo di passaggio a cui è soggetta la variabile/costante. Di default il passaggio è per valore nelle variabili, per riferimento negli array e per costante nelle costanti.

  Utilizzando la combinazione `Bool Pass` sono state implementate le tipologie di passaggio di parametri richieste nella consegna.

• funzioni: `data EnvF = Fun Id Type [Type]`

- `Fun`: identifica che l'oggetto in questione è una funzione;

- `Id`: nome della funzione;

- `Type`: tipo di ritorno della funzione;

- `[Type]`: lista dei tipi dei parametri formali.

  Si è scelto di non consentire la ridichiarazione di funzioni aventi nomi uguali neanche in blocchi diversi. Le funzioni (procedure) senza tipo di ritorno vengono memorizzate con tipo TypeVoid.

• label: `data EnvL = Label Id Bool`

- `Label`: identifica che l'oggetto in questione è una label;

- `Id`: nome della label;

- `Bool`: valore booleano utilizzato per lo scope.

Gli ambienti delle label e delle variabili vengono inizializzati come liste vuote, mentre l'ambiente delle funzioni è popolato con le funzioni di lettura e scrittura predefinite richieste nella consegna.

La struttura sintattica del nostro linguaggio derivato da *Chapel*

La grammatica del linguaggio da noi implementato è la seguente:

$\langle Prog \rangle \quad ::= \quad \langle ListCmd \rangle$

⟨*Decl*⟩ ::= `var` ⟨*ListId*⟩ `:` ⟨*Type*⟩ `;`
| `var` ⟨*ListId*⟩ `:` ⟨*Type*⟩ `=` ⟨*ExpR*⟩ `;`
| `var` ⟨*ListId*⟩ `:` ⟨*Type*⟩ `=` `[` ⟨*ListExpR*⟩ `]` `;`
| `var` ⟨*ListId*⟩ `:` ⟨*Type*⟩ `=` `[` ⟨*ListExpArrayElem*⟩ `]` `;`
| `const` ⟨*Id*⟩ `:` ⟨*Type*⟩ `=` ⟨*ExpR*⟩ `;`
| `label` ⟨*Id*⟩ `:` `;`
| `proc` ⟨*Id*⟩ `(` ⟨*ListParam*⟩ `)` ⟨*Cmd*⟩
| `proc` ⟨*Id*⟩ `(` ⟨*ListParam*⟩ `)` `:` ⟨*Type*⟩ ⟨*Cmd*⟩

⟨*Param*⟩ ::= ⟨*Id*⟩ `:` ⟨*Type*⟩
| ⟨*Pass*⟩ ⟨*Id*⟩ `:` ⟨*Type*⟩

⟨*Pass*⟩ ::= `val`
| `ref`
| `const`

⟨*ExpR*⟩ ::= ⟨*ExpRange*⟩
| ⟨*ExpR*⟩ + ⟨*ExpR*⟩
| ⟨*ExpR*⟩ − ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `*` ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `/` ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `**` ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `%` ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `:` ⟨*Type*⟩
| `(` ⟨*ExpR*⟩ `)`
| − ⟨*ExpR*⟩
| ⟨*ExpL*⟩
| ⟨*Basic*⟩
| ⟨*ExpR*⟩ == ⟨*ExpR*⟩
| ⟨*ExpR*⟩ < ⟨*ExpR*⟩
| ⟨*ExpR*⟩ <= ⟨*ExpR*⟩
| ⟨*ExpR*⟩ > ⟨*ExpR*⟩
| ⟨*ExpR*⟩ >= ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `!=` ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `&&` ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `^` ⟨*ExpR*⟩
| ⟨*ExpR*⟩ || ⟨*ExpR*⟩
| `!` ⟨*ExpR*⟩
| ⟨*Id*⟩ `(` ⟨*ListExpR*⟩ `)`

⟨*ExpRange*⟩ ::= ⟨*ExpR*⟩ `..` ⟨*ExpR*⟩

⟨*ExpArrayElem*⟩ ::= `[` ⟨*ListExpR*⟩ `]`

⟨*ExpL*⟩ ::= ⟨*Id*⟩
| ⟨*ExpL*⟩ `[` ⟨*ListExpR*⟩ `]`

⟨*Basic*⟩ ::= ⟨*Integer*⟩
| ⟨*Double*⟩
| ⟨*Char*⟩
| ⟨*String*⟩
| ⟨*Boolean*⟩

⟨*Boolean*⟩ ::= `true`
| `false`

⟨*Type*⟩ ::= `int`
| `float`
| `char`
| `string`
| `bool`
| `pointer` ⟨*Type*⟩
| `[` ⟨*ListExpRange*⟩ `]` ⟨*Type*⟩

⟨*Cmd*⟩ ::= ⟨*CommSing*⟩
| `{` ⟨*ListCmd*⟩ `}`

⟨*CommSing*⟩ ::= ⟨*ExpR*⟩ `;`
| ⟨*Id*⟩ `::`
| `if (` ⟨*ExpR*⟩ `) then` ⟨*Cmd*⟩
| `if (` ⟨*ExpR*⟩ `) then` ⟨*Cmd*⟩ `else` ⟨*Cmd*⟩
| `while (` ⟨*ExpR*⟩ `)` ⟨*BlockLoop*⟩
| `repeat` ⟨*Cmd*⟩ `until (` ⟨*ExpR*⟩ `) ;`
| `for` ⟨*Id*⟩ `in` ⟨*ExpRange*⟩ ⟨*ExpFor*⟩ ⟨*BlockLoop*⟩
| `for` ⟨*Id*⟩ `in` ⟨*ExpRange*⟩ ⟨*BlockLoop*⟩
| `select` ⟨*ExpR*⟩ `{` ⟨*ListWhen*⟩ `}`
| `select` ⟨*ExpR*⟩ `{` ⟨*ListWhen*⟩ `otherwise` ⟨*Cmd*⟩ `}`
| ⟨*ExpL*⟩ = ⟨*ExpR*⟩ `;`
| ⟨*Decl*⟩
| `goto` ⟨*Id*⟩ `;`
| `return` ⟨*ExpR*⟩ `;`
| `return ;`
| `break ;`
| `continue ;`
| `try` ⟨*Cmd*⟩ `catch` ⟨*Cmd*⟩

⟨*ExpFor*⟩ ::= `by` ⟨*ExpR*⟩
| `#` ⟨*ExpR*⟩
| `#` ⟨*ExpR*⟩ `by` ⟨*ExpR*⟩

⟨*BlockLoop*⟩ ::= ⟨*Cmd*⟩
| `do` ⟨*CommSing*⟩

⟨*When*⟩ ::= `when` ⟨*ExpR*⟩ ⟨*BlockLoop*⟩

⟨*ListId*⟩ ::= ϵ
| ⟨*Id*⟩
| ⟨*Id*⟩ `,` ⟨*ListId*⟩

⟨*ListCmd*⟩ ::= ϵ
| ⟨*Cmd*⟩ ⟨*ListCmd*⟩

⟨*ListExpR*⟩ ::= ϵ
| ⟨*ExpR*⟩
| ⟨*ExpR*⟩ `,` ⟨*ListExpR*⟩

⟨*ListParam*⟩ ::= ϵ
| ⟨*Param*⟩
| ⟨*Param*⟩ `,` ⟨*ListParam*⟩

⟨*ListWhen*⟩ ::= ϵ
| ⟨*When*⟩ ⟨*ListWhen*⟩

⟨*ListExpRange*⟩ ::= ϵ
| ⟨*ExpRange*⟩
| ⟨*ExpRange*⟩ `,` ⟨*ListExpRange*⟩

⟨*ListExpArrayElem*⟩ ::= ϵ
| ⟨*ExpArrayElem*⟩
| ⟨*ExpArrayElem*⟩ `,` ⟨*ListExpArrayElem*⟩

Type System

Elenco delle regole

Comandi

Un programma è una lista di comandi, quindi una lista di comandi ben formati è un comando ben formato nell'ambiente Γ.

$$(ListCmd)\frac{\Gamma \vdash_{Cmd} Cmd \qquad \Gamma' \vdash_{ListCmd}}{\Gamma \vdash_{ListCmd} Cmd; ListCmd}$$

$$(ListCmd\epsilon)\frac{}{\Gamma \vdash_{ListCmd}}$$

`Command` puó essere un comando singolo o un blocco di comandi. Se l'ambiente soddisfa un programma C e soddisfa un comando D allora l'ambiente soddisfa anche il C con all'interno il comando D. D è un comando con all'interno dichiarazioni ben formate di segnatura $(I : A)$ in Γ.

$$(CmdSing)\frac{\Gamma \vdash D \therefore (I : A) \qquad \Gamma, I : A \vdash C}{\Gamma \vdash D \ in \ C}$$

$$(CmdMult)\frac{\Gamma \vdash D \therefore (I : A) \qquad \Gamma, I : A \vdash C}{\Gamma \vdash \ (\{ \ D \ \}) \ in \ C}$$

Dichiarazioni

Insieme dei tipi base utilizzati.

$$Basic = \{Bool, Int, Float, Char, String\}$$

$[I]$ è una lista di identificatori di tipo A ben formati.

$$(DeclVar)\frac{\Gamma \vdash [I] : A \qquad A \in Basic}{\Gamma \vdash (var \ [I] : A) \therefore ([I] : A)}$$

E è un'espressione di tipo A soddisfatta da Γ.

$$(DeclVarAss)\frac{\Gamma \vdash E : A \qquad A \in Basic}{\Gamma \vdash (var \ [I] : A = E) \therefore ([I] : A)}$$

Gli array monodimensionali vengono definiti come le variabili, con $[E]$ lista di espressioni ed un solo range specificato dal tipo Array. Le espressioni assegnate all'array devono essere tutte del tipo specificato dal tipo Array.

$$(DeclVarAss)\frac{\Gamma \vdash [E] : A \qquad A = B \qquad B \in Basic}{\Gamma \vdash (var \ [I] : Array \ of \ B = [E]) \therefore ([I] : Array \ of \ B)}$$

Gli array multidimensionali sono definiti come le variabili, con $[[E]]$ lista di liste di espressioni, di lunghezza pari alla lista di range. Le espressioni assegnate all'array devono essere tutte del tipo specificato dal tipo Array.

$$(DeclVarAss)\frac{\Gamma \vdash [[E]] : A \qquad A = B \qquad B \in Basic}{\Gamma \vdash (var\ [I] : Array\ of\ B = [[E]]) \therefore ([I] : Array\ of\ B)}$$

Posso definire solo una costante alla volta. All'interno dell'espressione E ci possono essere anche variabili, perché saranno passate per valore, e quindi il loro variare non cambierà il valore della costante.

$$(DeclConst)\frac{\Gamma \vdash E : A \qquad A \in Basic}{\Gamma \vdash (const\ I : A = E) \therefore I : A}$$

Alle label non viene assegnato alcun tipo.

$$(DeclLab)\frac{}{\Gamma \vdash label\ I :}$$

La procedura I non ha tipo di ritorno. Le procedure e le funzioni accettano una lista di parametri $[x : A]$ anche di tipi diversi tra loro. Le procedure possono accettare anche parametri di tipo Array.

$$(DeclProc)\frac{\Gamma \vdash Cmd \qquad \Gamma \vdash [x : A] \qquad A \in Basic \cup \{Array\}}{\Gamma \vdash (proc\ I([x : A])\ Cmd) \therefore (I : proc)}$$

La procedura I ha tipo di ritorno B. Il tipo di ritorno può essere anche un array.

$$(DeclFunc)\frac{\Gamma \vdash [x : A] \qquad \Gamma \vdash B \qquad \Gamma \vdash Cmd \qquad A \in Basic \cup \{Array\} \qquad B \in Basic \cup \{Array\}}{\Gamma \vdash (proc\ I([x : A]) : B\ Cmd) : A \to B \therefore (I : proc))}$$

È previsto il cast implicito per le right-expression nelle dichiarazioni con assegnamento da Int a Float e da Char a String. Quindi se dichiaro una variabile di tipo Float e le assegno un Int non darà errore di tipo, mentre nel caso contrario ci sarà un errore.

$$(ImplicitCast)\frac{\Gamma \vdash E\ : Int}{\Gamma \vdash E\ : Float}$$

$$(ImplicitCast)\frac{\Gamma \vdash E \ : Char}{\Gamma \vdash E \ : String}$$

Right-Expressions

Il simbolo dell'addizione vale anche per la concatenazione, ed anche qui abbiamo il cast implicito.

$$(AddInt)\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int}{\Gamma \vdash (E_1 + E_2) : Int}$$

$$(AddFloat_1)\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : Float \qquad A \in [Int, Float]}{\Gamma \vdash (E_1 + E_2) : Float}$$

$$(AddFloat_2)\frac{\Gamma \vdash E_1 : Float \qquad \Gamma \vdash E_2 : A \qquad A \in [Int, Float]}{\Gamma \vdash (E_1 + E_2) : Float}$$

$$(Concat)\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : A \qquad A \in [Char, String]}{\Gamma \vdash (E_1 + E_2) : String}$$

Le altre operazioni aritmetiche operano il cast implicito da Int a Float.

$$\binom{BinOp}{-, *, /, **}\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int}{\Gamma \vdash (E_1 op E_2) : Int}$$

$$\binom{BinOpFloat_1}{-, *, /, **}\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : Float \qquad A \in [Int, Float]}{\Gamma \vdash (E_1 op E_2) : Float}$$

$$\binom{BinOpFloat_2}{-, *, /, **}\frac{\Gamma \vdash E_1 : Float \qquad \Gamma \vdash E_2 : A \qquad A \in [Int, Float]}{\Gamma \vdash (E_1 op E_2) : Float}$$

L'operazione di modulo può avvenire solo tra interi.

$$Mod\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int}{\Gamma \vdash (E_1 \ \% \ E_2) : Int}$$

Gli operatori di confronto ammettono il confronto tra Int e Float.

$$\binom{Confr}{<, <=, >, >=}\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : B \qquad A, B \in [Int, Float]}{\Gamma \vdash (E_1 confr E_2) : Bool}$$

L'operatore di uguaglianza accetta espressioni dello stesso tipo.

$$(==)\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : A \qquad A \in Basic \cup \{Array\}}{\Gamma \vdash (E_1 == E_2) : Bool}$$

L'operatore di disuguaglianza accetta anche espressioni di tipo diverso tra loro.

$$(!=)\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : B \qquad A, B \in Basic \cup \{Array\}}{\Gamma \vdash (E_1 != E_2) : Bool}$$

$$\binom{BoolOp}{\&\&, ||, \hat{}}\frac{\Gamma \vdash E_1 : Bool \qquad \Gamma \vdash E_2 : Bool}{\Gamma \vdash (E_1 \ BoolOp \ E_2) : Bool}$$

$$\binom{Not}{!}\frac{\Gamma \vdash E : Bool}{\Gamma \vdash ! \ E : Bool}$$

Viene controllato che il numero ed il tipo dei parametri siano quelli definiti dalla procedura, e che il tipo di ritorno della chiamata di procedura sia di tipo Void.

$$(CallProc)\frac{\Gamma \vdash (DeclProc \ I) : A \to Void \qquad \Gamma \vdash [x : A] \qquad A \in Basic \cup \{Array\}}{\Gamma \vdash (I([x : A])) : Void}$$

Viene controllato che il numero ed il tipo dei parametri siano quelli definiti dalla funzione, che il tipo di ritorno della chiamata di funzione sia del tipo dichiarato dalla funzione, e che questo tipo esista sempre.

$$(CallFunc)\frac{\Gamma \vdash (DeclFunc\ I) : A \to B \qquad \Gamma \vdash [x : A] \qquad A, B \in Basic \cup \{Array\}}{\Gamma \vdash (I([x : A])) : B}$$

Left-Expressions

Identificatore semplice. Deve essere presente nello scope del blocco.

$$(ExpLId)\frac{\Gamma \vdash x : A \qquad A \in Basic \cup \{Array\}}{\Gamma \vdash (ExpLId\ x) : A}$$

Quando richiamo un array di elementi di tipo A, le espressioni all'interno delle parentesi quadre devono essere tutti di tipo A.

$$(ExpLArray)\frac{\Gamma \vdash B : Array\ of\ A \qquad \Gamma \vdash [E] : A \qquad A \in Basic}{\Gamma \vdash (ExpLArray\ B[[E]]) : Array\ of\ A \therefore [E] : A}$$

Tipi

$$(Int)\frac{\Gamma \vdash \diamond}{\Gamma \vdash Int}$$

$$(Float)\frac{\Gamma \vdash \diamond}{\Gamma \vdash Float}$$

$$(Char)\frac{\Gamma \vdash \diamond}{\Gamma \vdash Char}$$

$$(String)\frac{\Gamma \vdash \diamond}{\Gamma \vdash String}$$

$$(Bool)\frac{\Gamma \vdash \diamond}{\Gamma \vdash Bool}$$

$$(True)\frac{\Gamma \vdash \diamond}{\Gamma \vdash True : Bool}$$

$$(False)\frac{\Gamma \vdash \diamond}{\Gamma \vdash False : Bool}$$

$$(Pointer)\frac{\Gamma \vdash A \qquad A : Basic}{\Gamma \vdash PointerA}$$

Per la definizione di array non è stata creata alcuna parola riservata, poiché in Chapel non era presente.

$$(Array)\frac{\Gamma \vdash A \qquad A \in Basic}{\Gamma \vdash [List\ of\ Range]A}$$

Il tipo Range è stato creato per i controlli di tipo sugli array.

$$(Range)\frac{\Gamma \vdash A \qquad \Gamma \vdash B \qquad A, B : Int}{\Gamma \vdash A\ ..\ B}$$

Statement

$$(ExpR)\frac{\Gamma \vdash E : A \qquad A \in Basic \cup \{TypeArray\}}{\Gamma \vdash (E;) : A}$$

$$(IfThen)\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash Cmd}{\Gamma \vdash if\ (E)\ then\ Cmd}$$

$$(IfThenElse)\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash Cmd_1 \qquad \Gamma \vdash Cmd_2}{\Gamma \vdash if\ (E)\ then\ Cmd_1\ else\ Cmd_2}$$

$$(While)\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash Cmd}{\Gamma \vdash\ while\ (E)\ Cmd}$$

$$(WhileDo)\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash CmdSing}{\Gamma \vdash while\ (E)\ do\ CmdSing}$$

$$(RepeatUntil)\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash Cmd}{\Gamma \vdash repeat\ Cmd\ until\ E}$$

$$(For)\frac{\Gamma \vdash I : Int \qquad \Gamma \vdash Range : Range \qquad \Gamma \vdash Cmd}{\Gamma \vdash for\ I\ in\ Range\ Cmd}$$

$$(ForDo)\frac{\Gamma \vdash I : Int \qquad \Gamma \vdash Range : Range \qquad \Gamma \vdash CmdSing}{\Gamma \vdash for\ I\ in\ Range\ do\ CmdSing}$$

$$(ForBy\#)\frac{\Gamma \vdash I : Int \qquad \Gamma \vdash Range : Range \qquad \Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int \qquad \Gamma \vdash Cmd}{\Gamma \vdash for\ I\ in\ Range\ \#E_1\ by\ E_2\ Cmd}$$

Per eseguire controlli di tipo su queste espressioni, viene loro assegnato il tipo di E.

$$(by)\frac{\Gamma \vdash E : Int}{\Gamma \vdash by\ E : Int}$$

$$(\#)\frac{\Gamma \vdash E : Int}{\Gamma \vdash \#E : Int}$$

$$(\#by)\frac{\Gamma \vdash E_1 : Int \qquad \Gamma \vdash E_2 : Int}{\Gamma \vdash \#E_1 \ \ by\ E_2 : Int}$$

$$(Select)\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : A \qquad \Gamma \vdash BlockLoop \qquad A \in Basic \cup \{TypeArray\}}{\Gamma \vdash select\ E_1\ \{\ when\ E_2\ BlockLoop\ \}}$$

$$(SelectOtherwise)\frac{\Gamma \vdash E_1 : A \qquad \Gamma \vdash E_2 : A \qquad \Gamma \vdash BlockLoop \qquad \Gamma \vdash Cmd \qquad A \in Basic \cup \{TypeArray\}}{\Gamma \vdash select\ E_1\ \{\ when\ E_2\ BlockLoop\ otherwise\ Cmd\ \}}$$

$$(BlockLoop)\frac{}{\Gamma \vdash Cmd}$$

$$(BlockLoopDo)\frac{\Gamma \vdash CmdSing}{\Gamma \vdash_{do} CmdSing}$$

L'assegnamento segue la regole dell'*Implicit Cast* vista prima.

$$(Assign_1)\frac{\Gamma \vdash ExpL : A \qquad \Gamma \vdash ExpR : A \qquad A \in Basic \cup \{TypeArray\}}{\Gamma \vdash (ExpL = ExpR) : A}$$

$$(Assign_2)\frac{\Gamma \vdash E_1 : Float \qquad \Gamma \vdash E_2 : A \qquad A \in [Int, Float]}{\Gamma \vdash (E_1 = E_2) : Float}$$

$$(Assign_3)\frac{\Gamma \vdash E_1 : String \qquad \Gamma \vdash E_2 : A \qquad A \in [Char, String]}{\Gamma \vdash (E_1 = E_2) : String}$$

$$(GoTo)\frac{\Gamma \vdash label}{\Gamma \vdash goto\ label}$$

Viene controllato se il tipo di E sia uguale al tipo di ritorno di R richiesto dalla funzione.

$$(Return)\frac{\Gamma \vdash E : A \qquad \Gamma \vdash R : B \qquad A, B \in Basic \cup \{TypeArray\}}{\Gamma \vdash (return\ E;) \therefore E : A = B}$$

$$(ReturnVoid)\frac{\Gamma \vdash R : Void}{\Gamma \vdash return;}$$

$$(Break)\frac{}{\Gamma \vdash break;}$$

$$(Continue)\frac{}{\Gamma \vdash continue;}$$

$$(TryCatch)\frac{\Gamma \vdash Cmd_1 \qquad \Gamma \vdash Cmd_2}{\Gamma \vdash try\ Cmd_1\ catch\ Cmd_2}$$

Type checker

Durante la realizzazione del type checker sono stati rispettati tutti i vincoli imposti dal Type System ed inoltre sono stati fatti i seguenti controlli:

- le variabili, le costanti, gli array, le funzioni e le label non possono essere dichiarate più volte nello stesso blocco;
- le costanti non possono essere modificate;

- gli array monodimensionali devono avere un'unica lista di espressioni, mentre quelli multidimensionali devono avere tante liste quante i range presenti nella dichiarazione;
- le funzioni devono avere il return in tutti i rami, o alla fine, in modo da tornare sempre un valore, che deve essere del tipo specificato;
- le chiamate di funzione e di procedura devono avere i parametri del tipo indicato dalla dichiarazione della stessa, senza cast impliciti;
- le istruzioni di `break` e `continue` devono essere sempre all'interno di un loop;
- non è stato implementato il controllo di visibilità della label nel salto `goto label`;

Sono previsti una serie di messaggi d'errore che spiegano i problemi riscontrati durante il parsing, ognuno dei quali indica la riga e la colonna del codice sorgente in cui si è verificato l'errore e ne identifica il tipo.

Attributi utilizzati per il Type checker

- `typ` {`Type`} : memorizza il tipo del nodo;
- `err` {`String`} : memorizza eventuali errori rilevati nelle ExpR;
- `envIn` {`[Env]`} : ambiente delle variabili valido per il nodo corrente;
- `envOut` {`[Env]`} : ambiente di uscita delle variabili del nodo (sintetizzato);
- `envInF` {`[EnvF]`} : ambiente delle funzioni valido per il nodo corrente;
- `envOutF` {`[EnvF]`} : ambiente di uscita delle funzioni del nodo (sintetizzato);
- `typFun` {`Type`} : indica il tipo di ritorno della procedura;
- `paramList` {`[Type]`} : indica la lista dei tipi dei parametri formali di una funzione;
- `isconst` {`Bool`} : indica se la ExpL di un assegnamento è una costante o meno;
- `envInL` {`[EnvL]`} : ambiente delle label valido per il nodo corrente;
- `envOutL` {`[EnvL]`} : ambiente di uscita delle label del nodo (sintetizzato);
- `hasReturn` {`Bool`} : indica se è stato ritornato un dato prima della conclusione della funzione;
- `isInLoop` {`Bool`} : indica se il nodo corrente è situato all'interno di un ciclo (o select).

Three Address Code

Il Tree Address Code è stato definito come una lista di istruzioni, in cui ogni istruzione è definita attraverso un datatype. Il datatype, le funzioni ausiliarie e il pretty printer per il TAC sono contenuti nel file *TreeAddressCode.hs*.

Il datatype utilizzato per codificare le istruzioni è il seguente:

```
data Tac =
```

`AssignOp T T` :
operatore per eseguire gli assegnamenti

| `BinOp Op T T T` :
operatore per eseguire le operazioni binarie

| `Jump T Label` :
operatore per i salti condizionati, in caso la condizione sia `True`

| `JumpNot T Label` :
operatore per i salti condizionati, in caso la condizione sia `False`

| `DeclProc String Id Int` :
operatore per le dichiarazioni di procedure e funzioni

| `Lab Int` :
operatore per inserire le label

| `OnExpJump Label` :
operatore per la gestione delle eccezioni nel `try-catch`

| `ProcCall String T Id [T]` :
operatore per le chiamate di procedure

| `FunCall T String T Id [T]` :
operatore per le chiamate di funzioni

| `Return T` :
operatore per eseguire il `return`

| `UJump Label` :
operatore per i salti incondizionati (`break, continue`)

| `UnOp Op T T` :
operatore per eseguire operazioni unarie

```
type T = String

type Op = String

type Label = Int
```

Attributi TAC:

- `tac` {`[Tac]`} : accumulatore contiene tutto il TAC del nodo corrente;
- `tempLab` {`Int`} : contatore che rappresenta l'indice delle label per il nodo corrente (ereditato);
- `tempLabOut` {`Int`} : contatore che rappresenta l'indice di uscita delle label per il nodo corrente (sintetizzato);
- `tempVar` {`Int`} : contatore che rappresenta l'indice del valore temporaneo per il nodo corrente (ereditato);
- `tempVarOut` {`Int`} : contatore che rappresenta l'indice del valore temporaneo di uscita per il nodo corrente (sintetizzato);
- `addr` {`String`} : contiene l'identificatore delle variabili su cui si opera;
- `addrParamList` {`[String]`} : lista di identificatori di variabili (addr);
- `next` {`Int`} : indica l'istruzione a cui saltare quando si esce da un blocco;
- `beginLoop` {`Int`} : indica la label posta all'inizio del ciclo in cui ci si trova (`while, repeat,for`);

- `endLoop` {`Int`} : indica la label posta alla fine del ciclo in cui ci si trova (`while,repeat,for`);

- `tSelect` {`Int`}: memorizza il temporaneo in cui si salva la guardia principale di un ciclo.

- `endProc` {`Int`}:indica la fine della procedura a cui saltare quando si ottiene un valore di ritorno;

- `isCond` {`Bool`}: utilizzato per effettuare lo short-cut . Indica se l'operazione di `and` o `or` che si vuole valutare è condizione di un ciclo, in quel caso verrà valutata con short-cut, altrimenti no.

Il Three Address Code è stato implementato come mostrato a lezione.

Se è presente una dichiarazione di procedura o di funzione, il TAC la salta, ed esegue le istruzioni successive.

Quando viene chiamata la procedura, effettua una `call nome_proc`, esegue la procedura chiamata, da cui esce quando trova `return;`, altrimenti va avanti fino alla fine.

Nel caso della chiamata di funzione, il valore di ritorno viene salvato in un nuovo temporaneo.

Il salto del `goto label` salta semplicemente alla label chiamata.

Le condizioni `&&` e || vengono valutate con short-cut.

Il costrutto `try-catch` esegue il blocco del `try`, finito il quale esegue l'istruzione `onexceptiongoto label_catch`, o, se non ci sono eccezioni, salta alla fine del `catch`. Segue quindi il blocco del `catch` senza indicazioni di salto.

Il ciclo `for` è stato valutato secondo la sintassi del linguaggio, che prevede di entrare nel ciclo dopo aver verificato se l'indice dato è contenuto nel range indicato. Infatti viene eseguito il TAC dell'indice, nel caso debba essere calcolato, il TAC del range, che è salvato in un temporaneo, viene eseguita l'operazione booleana `t1 in t2`, con `t1` che memorizza l'indice e `t2` che memorizza il range. A questo punto viene valutato il risultato dell'operazione con i soliti salti condizionati, per entrare nel ciclo o saltarlo.

Il TAC del `select` memorizza il temporaneo da confrontare nei `when`, i quali lo ereditano e lo confrontano con la propria espressione. Se è uguale, il salto condizionato porta dentro il corpo del `when`, finito il quale si esce dal `select`, se diverso si salta al `when` successivo.

La dichiarazione nel TAC degli array monodimensionali avviene assegnando ogni elemento della lista alla cella dell'array in progressione. Per gli array multidimensionali questo viene fatto per tutte le liste di elementi, aumentando progressivamente l'indice. In questo modo un array multidimensionale appare come un array monodimensionale dal punto di vista del TAC. Ci è risultato difficile trovare un modo di aumentare e modificare gli indici a seconda della dimensione passata.

File di prova

Sono stati predisposti dei file di prova, per eseguirli va lanciato il comando *make demo**n***, con ***n*** indicante il numero della demo, compreso fra 1 e 4. I file contengono codice corretto, per visualizzare i messaggi di errore si modifichino opportunamente le demo.