Progetto Linguaggi & Compilatori
Parte III
Gruppo 2

Andrea Mansi UniUD - 137857
Christian Cagnoni UniUD - 137690

II° Semestre 2020/2021

1 Il Linguaggio

In tutta la relazione ci si riferirà con il nome **MC** al linguaggio ottenuto a partire dalla sintassi concreta di **Chapel**[1]. Si tenga bene presente che MC **non è** Chapel.

Seguirà ora una descrizione del linguaggio MC:

1.1 Sintassi e caratteristiche di base

La sintassi concreta di MC è basata su quella di Chapel, tuttavia, alcune parti potrebbero non combaciare perfettamente e/o essere escluse perchè non richieste dalla consegna (ad esempio, Chapel è un linguaggio ideato per il parallelismo, ma la parte di sintassi concreta relativa a tali costrutti non è stata inclusa in MC). MC quindi prende solo ispirazione da Chapel per la definizione della propria sintassi concreta.

Le principali caratteristiche di MC sono le seguenti:

- Gli **identificatori** delle variabili e delle funzioni (e procedure) devono iniziare con una lettera e possono contenere solo caratteri alfanumerici e "_".
- Il linguaggio è **case sensitive**.
- Il linguaggio supporta l'utilizzo dei **puntatori** (a differenza di Chapel).
- Le **parole riservate** del linguaggio sono: var, void, while, true, True, false, False, then, string, return, real, proc, param, int, in, if, function, for, else, do, continue, char, const, break, bool. Tali parole non possono essere utilizzate come identificatori.
- I **commenti** sono consentiti; quelli singola linea richiedono il carattere //, quelli multi-linea richiedono l'utilizzo di /* e */.
- MC ha **scoping statico**.
- I **tipi sono espliciti**, vanno sempre esplicitati nel codice da parte dell'utente.

1.2 Operatori

Gli operatori aritmetici binari supportati sono:

- + per l'addizione
- - per la sottrazione
- * per la moltiplicazione
- / per la divisione

[1] Il linguaggio di programmazione Chapel è orientato al parallelismo ed è sviluppato da Cray. Supporta un modello di programmazione parallela multithread ad alto livello, supportando astrazioni per il parallelismo dei dati e attività.

- ** per l'elevamento a potenza
- % per il modulo

Gli operatori unari aritmetici:

- + per il segno positivo
- - per il segno negativo

Gli operatori di confronto sono:

- == per il controllo di eguaglianza
- != per il controllo di diseguaglianza
- > e < per il maggiore e minore
- >= e <= per il maggiore-uguale e minore-uguale

Si noti i predicati di uguaglianza e diseguaglianza possano confrontare fra loro tutti i tipi base mentre gli altri predicati sono utilizzabili solo con tipi numerici (Int e Real).

Gli operatori booleani sono:

- && per l'AND
- || per l'OR
- ! per il NOT

Inoltre è presente l'operatore di deferenziazione per i puntatori: $

Post e Pre-incremento/decremento (++ e -- in C) non sono supportati.

Per quanto riguarda le regole di associatività degli operatori:

Operatore	**Associatività**
||	sinistra
&&	sinistra
!	–
assegnamento e confronto	–
+ -	sinistra
* / %	sinistra
**	destra
$	–

Gli operatori sono in ordine: dal meno al più importante in termini di precedenza.

1.3 Controllo di sequenza

In questa sezione vengono illustrate le strutture di controllo supportate da MC; la cui sintassi concreta è ereditata da Chapel.

È presente il classico costrutto di selezione/alternativo **if-then-else**:

- **if** expr **then** statements
- **if** expr **then** statements **else** statements

Si noti come **statements** può essere un singolo statement seguito da ; oppure un insieme di statements inclusi in un blocco; seguono degli esempi esplicativi:

```
1 if true then foo();
2
3 if x > 2 then {
4   var y:int = 10;
5   foo(y);
6   }
7
8 if (z==3) then foo(); else {w=false; foo();}
```

Esiste inoltre una variante ereditata da Chapel, in cui **expr** può essere una dichiarazione di variabile di controllo. Tale dichiarazione restituirà true se valida, causando l'esecuzione del corpo del then. Nel caso in cui tale dichiarazione non sia valida, il risultato sarà false e verrà eseguito il corpo dell'else. In entrambi i casi viene aggiornato l'environment con una nuova entry per la variabile di controllo all'interno del corpo dell'if (nel caso false, la semantica di tale costrutto prevede che la variabile venga inizializzata al suo valore di default).

Esempio:

```
1 if var x:int = y
2 then ... // y e' intero: dichiarazione ok; viene eseguito il then
3 else ... // y non e' intero: dichiarazione non ok; viene eseguito l'else
4
5 if var x:int = "not int" then print(x); // il then non verra' mai eseguito
```

Si noti che in caso di dichiarazioni non valide (riga 6), non verrà generato un errore di tipo perché non comporterebbe errori a run-time; bensì indicano che il flusso d'esecuzione deve procedere con il caso else (se presente).

Quando statements è un blocco, la keyword **then** può essere ommessa.

I costrutti di iterazione supportati da MC sono i seguenti:

- ciclo **for**
- ciclo **while**
- ciclo **do-while**

Il **ciclo for** richiede la seguente struttura:

- **for** index-decl **in** range **do** statements
- **for** range **do** statements

Sono quindi presenti le versioni con e senza dichiarazione di indice; inoltre, similmente all'if-then-else, **statements** può essere un singolo statement o un blocco di statements. Seguono degli esempi:

```
for x in 1..100 do print(x);

for x in 1..100 do {var power:int = 2**x; print(x);}

for 1..10 do print("Spam text!");
```

Quando statements è un blocco, la keyword **do** può essere ommessa.

La variabile di iterazione è di fatto una implicita dichiarazione locale visibile all'interno del corpo del for; tale variabile viene dichiarata come **param**; di conseguenza non è modificabile. Inoltre viene inizializzata con canOverride=false; di conseguenza non sarà possibile sovrascriverla con un altra variabile. Esempio:

```
  for x in 1..100 do {
    writeInt(x);      // ok
    x+=1;             // not ok: x e' param! non modificabile
    var x:int = 43;   // not ok: x gia' dichiarata!
  }
```

Il **ciclo while** richiede la seguente struttura:

- **while** expr **do** statements
- **while** ctrl-decl-stat **do** statements

La versione con il **ctrl-decl-stat** permette di dichiarare una variabile di controllo, in modo analogo al caso dell'if-then-else. **statements** come sempre è un singolo statement o un blocco.

Esempi:

```
while true do {foo(); if (x==true) then break;}

while x>100 do {print(x); x=x+1;}

while var x:int = 0 do {print(x); x=x+1;} /* la dichiarazione della
             variabile di controllo genera un true: si noti come in
             assenza di un break tale ciclo while generi deadlock */
```

Quando statements è un blocco, la keyword **do** può essere ommessa.

Il **ciclo do while** richiede la seguente struttura:

- **do** statements **while** expr ;

Esempi esplicativi:

```
do print(x); while y==true;

do {print(x); x=x*2;} while x<500;
```

Il corpo del do viene valutato prima della guardia, questo comporta che venga eseguito almeno una volta. **statements** come sempre è un singolo statement o un blocco.

1.4 Type-System

Il linguaggio MC supporta i seguenti tipi di base predefiniti:

- **void**
- **integer**
- **real**
- **string**
- **char**
- **boolean**

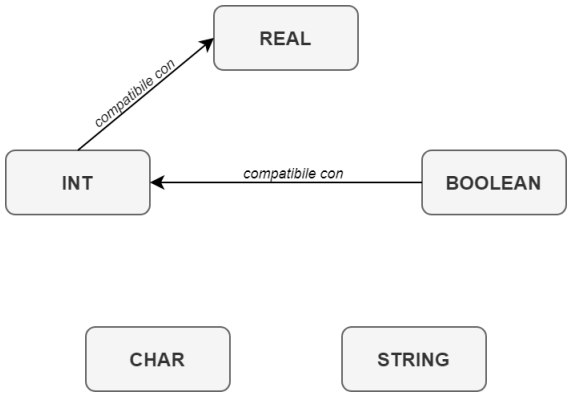
Per quanto riguarda i tipi composti sono disponibili:

- **array** a qualsiasi tipo (base e composto)
- **puntatori** a qualsiasi tipo (base e composto)

MC **non supporta** la definizione di tipi da parte dell'utente.

1.4.1 Compatibilità dei tipi

Segue un diagramma riassuntivo della compatibilità dei tipi primitivi:



Oltre alla ovvia compatibilità tra valori dello stesso tipo; **int** è compatibile con **real**. Questo significa che un valore intero può essere inserito dove è richiesto un valore reale, ma non viceversa! Similmente, **boolean** è compatibile con **int**.

Ad esempio:

```
var x:int;
var y:real;

x = 1;     // ok
x = 1.0;   // not ok: e' richiesto int; real non e' compatibile con int
y = 1;     // ok: 1 e' visto come 1.0
y = 1.0;   // ok
```

```
var z:bool;
x = true; // ok
z = 1;    // not ok: e' richiesto bool; int non e' compatibile con bool
```

Per quanto riguarda la compatibilità tra tipi composti, essa segue la logica dei tipi primitivi. Ad esempio; un array di char sarà compatibile solo con array di char; un array di interi è compatibile con un array di interi o un array di real (un array di interi può essere posizionato dove è richiesto un array di real, ma non vicevarsa) e così via.

1.4.2 Casting esplicito

Oltre alla compatibilità di int con real appena illustrata, che non richiede casting esplicito, in MC è presente l'operatore di casting che ci permette di convertire valori tra diversi tipi.

Le limitazioni sono le seguenti:

- Boolean può essere convertito solo in intero;
- Nessun tipo può essere convertito in boolean;
- Il casting è consentito solo tra tipi di base.

Esempi:

```
var x:int;
var y:real;

x = 1.0:int;     // ok
y = 1:real;      // ok ma non necessario (int e' compatibile con real)

var str:string = (4+10):string  // ok
```

L'operazione di casting genererà (in fase di TAC) la corretta istruzione macchina o chiamata di funzione. Nello specifico, si assume che per le seguenti conversioni sia presente l'opportuna istruzione macchina per la conversione:

- int-to-real;
- boolean-to-int;

Per tutte le altre conversioni supportate, verrà chiamata una opportuna funzione di conversione (che si ipotizza implementata nelle standard-libraries del linguaggio). Esempio:

```
// Input:
var x : string = 5 : string;

// TAC-code:
  param_string 5
  t0 =str fcall convert_from_int_to_string@0,0
  x@1,5 =str t0
```

1.5 Dichiarazioni

La visibilità di una qualsiasi dichiarazione è effettiva dal punto di dichiarazione in poi.

1.5.1 Variabili

Le variabili possono essere dichiarate in più modalità:

- **param**: var. costante a compile-time;
- **const**: var. costante a run-time;
- **var**: variabile regolare;

La sintassi concreta di Chapel prevede anche le variabili di tipo **ref** e **const ref**. Queste sono state rimosse da MC in quanto ridondanti con il comportamento dei puntatori.

Una variabile ref, in Chapel, si comporta come un alias per un altra variabile (l-value). Accedendo a x si altera/utilizza il valore a cui x fa da alias. Questo comportamento è ottenibile in modo analogo in MC con i puntatori (assenti in Chapel). Ad esempio:

```
var x:int = 5;

ref y:int = x;    // (valido in Chapel) y e' un alias per x
var z:int = y*2;  // = 10

var y:int$ = x;   // (valido in MC) y punta a x
var z:int = $y*2; // = 10
```

La sintassi per la dichiarazione è la seguente:

- **modalità** + **identificatori** + **:** + **tipo** + **=** + **exp**

Si noti come **identificatori** può anche essere, oltre che un identificatore unico, una lista di identificatori separati da virgola. Si noti come la parte destra (operatore di assegnamento e espressione) sia opzionale (l'inizializzazione non è obbligatoria).

```
var x:int;
param y:real;
var x:char; // Errore di ridefinizione
var z:int = 3;
const a,b,c:string = "stringa";
```

La ridefinizione nello stesso blocco di una variabile **non è consentito**. Se si entra in un nuovo blocco, la ridefinizione è permessa, ma dopo averla eseguita verrà disabilitata per quel blocco.

1.5.2 Funzioni e procedure

MC permette la definizione di funzioni e procedure, la sintassi è la seguente:

- Procedure: **proc** + **identificatore** + (**params**) + **:** + void + **statements**
- Funzioni: **function** + **identificatore** + (**params**) + **:** + **tipo** + **statements**

statements è un blocco di statements; **tipo** specifica il tipo di ritorno; **params** è opzionale. Seguono ora degli esempi esplicativi:

```
proc foo():void{
  // corpo della proc
}

proc print(str:string):void{
  showString(str);
}

function addInt(x:int,y:int):int{
  return x+y;
}
```

La ridefinizione nello stesso blocco di una funzione e/o procedura **non è consentita**. Se si entra in un nuovo blocco, la ridefinizione è permessa, ma dopo averla eseguita verrà disabilitata per quel blocco. L'**overloading di funzioni** non è consentito in MC; non è quindi possibile, nello stesso blocco, dichiarare due funzioni con stesso identificatore e diverse firme/segnature.

Modalità di passaggio

La modalità di passaggio di default è per valore. MC permette inoltre la modalità di passaggio per valore-risultato. Questa è possibile specificando la keyword **valres** precedentemente all'identificatore. Il modulo di analisi di semantica statica si occuperà del controllo che come parametro venga effettivamente passato un valido l-value. Esempio:

```
  function foo(valres x:int):void{}

  var input:int;

  foo(input); // ok
  foo(1+2);   // not ok
```

1.5.3 Array

MC permette due modalità di definizione di array. Più nello specifico, sono presenti gli array a più dimensioni ed array ad array ("multidimensionali").

Seguono esempi esplicativi di dichiarazione.

```
// -- dichiarazione classica
var x:[1..10]int;     // array a 10 interi, dim:1
var z:[]int;          // array di lunghezza non specificata, dim:1
var w:[1..10,1..10]int; // array dim:2 (matrice 10x10 di int)
var i:[1..5]int = [1,2,3,4,5] // con inizializzazione
```

```
6  var p:[1..5]char$;     // array a puntatori a char
7  var pp:([]char)$;      // puntatore ad array di char
8
9  // -- array di array
10 var u:[1..10][1..10]int;  /* array di lunghezza 10 dove ciascun elemento
11                              e' un array di 10 interi */
12 var k:[1..2][1..3]int = [[1,2,3],[4,5,6]];
```

Alcune osservazioni:

- Si noti come la dimensione sia specificata da una opportuna **rangeexpression**. Tali espressioni $a..b$ possono essere a e b solo di tipo int.
- Nel caso l'inizializzazione sia mancante gli elementi vengono impostati a un valore di default. Se presente (vedi riga 6) deve rispettare la/le lunghezza e la/le dimensione dell'array. Gli elementi vengono inseriti per riga.

Array checked

MC permette la dichiarazione di array la cui indicizzazione viene controllata a run-time. In caso contrario questo controllo non viene effettuato. Gli array di questo tipo possono essere dichiarati mediante la keyword **checked**. Esempio:

```
1  var checked x:[1..10]int;    // array a 10 interi, dim:1
2  var y:[1..10]int;
3  x[20] = 42; // il tac generera' la procedura di controllo
4  y[20] = 42; // nessun controllo
```

In fase di TAC generation viene inserito dell'opportuno codice TAC delegato al controllo degli indici a run-time; codice che genererà errore in caso di indici non validi ("out of bounds").

1.5.4 Puntatori

Per quanto riguarda i puntatori, essi possono puntare a qualsiasi tipo (base e composto). L'operatore di deferenziazione in MC è il dollaro: **$**. MC non possiede il tipo primitivo relativo agli indirizzi, di conseguenza l'operatore di referenziazione (si pensi a &) non va esplicitato; si veda l'esempio sottostante:

```
1 var x:int;       // valore intero
2 var y:int$;      // puntatore ad intero
3 y = x;           /* y ora puntera' a  x (equivalente a y = &x in
4                     linguaggi come C) */
```

MC controlla che qualora come L-value ci sia un puntatore di profondità 1 a un tipo t; a destra dell'assegnamento ci sia effettivamente un tipo t. Se come L-value abbiamo un puntatore di profondità n>1 a un tipo t; a destra è richiesto un puntatore di profondità n-1 a un tipo t.

In dettaglio l'operatore di deferenziazione:

```
1 var x:int$$$;    // puntatore a int di profondita' 3
2
3 ... = $x;        // ritorna un puntatore a int di profondita' 2
4 ... = $$x;       // ritorna un puntatore a int di profondita' 1
5 ... = $$$x;      // ritorna il valore puntato (int)
```

Visto che l'operatore di referenziazione non è esplicito; assegnamenti del tipo:

```
var x:int$;
var y:int$;
x = y;
```

non sono supportati; x richiede un valore intero a destra. L'assegnamento sarebbe andato a buon fine se x fosse stato di tipo int$$.

Seguirà qualche altro esempio esplicativo:

```
var x:int = 5;     // valore intero
var p:int$ = x;    // puntatore a intero; ora p punta a x
var pp:int$$ = p; // ok: puntatore a puntatore a int
var pp:int$$ = x; // not ok!

var y:int = $p;    // ok: deferenziazione $p = 5
y = p;             // not ok!
y = $$pp;          // ok!

var sum:int = $p + y; // ok: equivalente a x+y
```

1.6 Assegnamenti

Oltre all'assegnamento semplice (=) su MC è possibile effettuare gli assegnamenti aumentati. Tali assegnamenti supportano tutte le operazioni aritmetiche presenti in MC. La sintassi è quella usuale utilizzata nella maggioranza dei linguaggi di programmazione. Ad esempio: +=.

2 Implementazione

Per l'implementazione del front-end del compilatore si è partiti con un primo prototipo generato con il tool **BNFC**, successivamente largamente modificato per soddisfare i requisiti richiesti. La grammatica in input è fornita nel file **.cf**.

Per quanto riguarda l'implementazione delle varie fasi; si è deciso di seguire un approccio a **pipeline**, in cui l'output di ciascun modulo viene fornito in input al modulo successivo.

2.1 Lexer

Il lexer prodotto da BNFC ha necessitato di alcune piccole modifiche. In particolare, è stata introdotta una fase di pre-processing dell'input antecedente a quella di lexing. Tale fase ha lo scopo di permettere all'utente di inserire multipli operatori di deferenziazione senza doverli separare da spazi. La fase di preprocessing si occupa dell'inserimento di uno spazio tra ciascuno di tali operatori, altrimenti il lexer produrebbe un errore di sintassi.

```
1  ...
2  var x:$$$int = $$p;        // ok solo con pre-processing
3  var x:$ $ $ int = $ $ p    // ok anche senza pre-processing
4  ...
```

2.2 Abstract Syntax Tree

Nell'implementare il front-end si è deciso di non utilizzare una **Symbol Table**, bensì si è deciso di sfruttare l'esecuzione lazy di Haskell e quindi di mantenere le informazioni necessarie (per i vari passi) direttamente nella sintassi astratta prodotta dal parser.

Per implementare tale approccio è stato quindi necessario modificare l'albero di sintassi astratta prodotto da BNFC introducendo per ciascun costruttore un parametro polimorfo **a**.

Tale parametro polimorfo assumerà un tipo diverso in base alla fase del compilatore attualmente in esecuzione, nel nostro caso:

- Dopo la fase di parsing avrà tipo: **Posn**
- Dopo la fase di analisi di semantica statica avrà tipo: **TCheckResult**
- Dopo la fase di generazione del TAC: **TAC**

dove Posn, TCheckResult e TAC sono:

- **Posn**: posizione del token, data-type fornito dal lexer, contiene informazioni relative alla riga, colonna e pos. assoluta del token/nodo;
- **TCheckResult**: data-type che contiene le informazioni relative al risultato della fase di analisi di semantica statica. Esso può essere di tipo TResult (qualora non siano stati riscontrati errori) o di tipo TError.
- **TAC**: data-type che contiene la lista di istruzioni TAC per quel dato nodo.

Ulteriori dettagli sulla fase di type-checking nel capitolo: Analisi Semantica Statica.

Ulteriori dettagli sulla fase di generazione del codice TAC nel capitolo: Generazione Three Address Code.

2.3 Parser

Per supportare la modifica all'albero di sintassi astratta è stato necessario modificare il parser generato da BNFC. Nello specifico, per ciascun nodo, il parser inserisce nel parametro polimorfo la posizione del token rilevata dal lexer (distinguendo casi terminali e non terminali), producendo appunto un albero in cui il parametro polimorfo è di tipo **Posn**.

2.3.1 Conflitti Il parser contiene 5 confiltti shift-reduce e nessun conflitto reduce-reduce. Tali conflitti possono essere suddivisi in 3 cluster:

- **Else-Statement:** negli stati 130, 132, 174 e 177 relativamente alle regole di produzione del costrutto if-then-else, il conflitto si presenta quando il parser, eseguiti gli statements del blocco then trova un 'else'. Nel caso di else annidati (if-then-else annidati) il parser non è in grado di sapere a chi dei costrutti quell'else appartiene (la grammatica è ambigua). Nel caso della scelta shift (quella adottata) il parser associa l'else all'ultimo costrutto if-then-else.

  Esempio:

  ```
  1  if ... then if ... then ... else ...
  ```

  In questo caso il conflitto si ritrova al punto dell'else; il parser per l'if-then-else interno non è in grado di sapere se applicare la regola che produce un else o che va in empty. In questo caso esegue lo shift, producendo effettivamente (e quindi associando) l'else statements per l'if-then-else più interno.

- **Virgola:** nello stato 285, relativamente alle regole listelementarray (all'interno dell'inizializzzione di un array) si presenta il problema simile all'elsestatement. Anche in questo caso il parser esegue uno shift.

2.4 Analisi semantica statica

Segue ora una descrizione della fase di analisi di semantica statica.

2.4.1 Environment

Per la manipolazione e gestione dell'environment è stato creato un opportuno data-type di nome **Env**, una mappa chiave-valore, dove:

- **chiave**: Stringa
- **valore**: Lista di EnvEntry

Le coppie possibili sono le seguenti:

- **1:** "identificatore variabile" - lista di n EnvEntry di tipo Variable
- **2:** "identificatore funzione" - lista di n EnvEntry di tipo Function
- **3:** "return" - lista vuota
- **4:** "while" - lista vuota

Nel caso 1 e 2; abbiamo la possibilità che esistano più EnvEntry perché potremmo aver ridefinito una variabile/funzione all'interno di un nuovo blocco (dove tale operazione è consentita); di conseguenza, all'interno degli env. di tali blocchi/funzioni le entry potrebbero essere multiple. La più recente è sempre quella in prima posizione, ed è quella a cui bisogna far riferimento.

La ridefinizione (overriding) è gestita con una variabile (per ciascuna entry) di nome **canOverride**. Tale variabile è inizializzata a False, e impostata a True quando si entra in un nuovo environment. Se è true (perchè dentro un nuovo env.), la ridefinizione avrà successo e la nuova entry (che verra poi considerata, essendo in cima alla lista) verrà inizializzata con canOverride=False.

I casi 3 e 4 vengono utilizzati per identificare il fatto di essere o meno all'interno di blocchi di funzioni o blocchi while; per il controllo che gli statements return, continue e break siano posizionati in posizioni valide. Le chiavi return sono presenti in più versioni, ad esempio "return_int" per specificare il tipo consentito nel return.

```
type Env = Map Prelude.String [EnvEntry]
            -- chiave, valore

data EnvEntry
    = Variable {varType::Type, varPosition::LexProgettoPar.Posn,
                varMode::Prelude.String, canOverride::Prelude.Bool,
                size::[Prelude.Integer]}
    | Function {funType::Type, funPosition::LexProgettoPar.Posn,
                funParameters::[Parameter], canOverride::Prelude.Bool}

data Parameter
    = Parameter {paramType::Type, paramPosition::LexProgettoPar.Posn,
                 paramMode::Prelude.String, identifier::Prelude.String}
    deriving(Eq, Ord)

```

Per tali data-types sono state inoltre definite le funzioni Show.

2.4.2 Analisi di semantica statica

La fase di analisi di semantica statica (incluso il type-checking) viene effettuata dal modulo **TypeChecker**. I data-types relativi ai tipi sono definiti all'interno del modulo **Type**; il contenuto è il seguente:

```
1  data BasicType
2    = Type_Integer
3    | Type_Boolean
4    | Type_Char
5    | Type_String
6    | Type_Void
7    | Type_Real
8    deriving (Eq, Ord, Read)
9
10 data Type
11   = B_type {b_type::BasicType}
12   | Array {c_type::Type, dim::Integer}
13   | Pointer {c_type::Type, depth::Integer}
14   deriving (Eq, Ord, Read)
```

Per tali data-types sono state inoltre definite le funzioni Show.

Il modulo fa inoltre uso delle seguenti strutture dati per immagazzinare i risultati della fase di analisi di semantica statica, per un dato nodo.

```
1 data TCheckResult
2 = TResult {environment::Env, t_type::Type,
3            t_position::LexProgettoPar.Posn}
4 | TError {errors::[Prelude.String]}
```

Se si verifica un errore, al nodo verrà associato (nel parametro polimorfo) un oggetto di tipo **TError**; altrimenti un oggetto di tipo **TResult** contenente posizione, tipo dell'operazione e/o nodo (a volte void, a volte un tipo reale) e l'environment.

Gestione errori indotti:
Gli errori vengono propagati dai nodi figli ai nodi padre, tramite l'uso delle funzioni **mergeErrors** e **checkErrors**. Spesso, ai fini di non creare errori insensati, viene controllata la presenza di errori sul figlio prima di generare un ulteriore errore a cascata; segue un esempio:

```
1  var x : int;
2  y = x;
```

Tale codice genera in prima istanza un errore in riga 2 relativo all'inesistenza della variabile y. A quel punto, la parte sinistra dell'assegnamento ritorna un TError, questo causa a cascata la generazione di un errore relativo all'assegnamento (si sta cercando di assegnare un valore di tipo int a una variabile "senza tipo", avendo essa generato un errore). Il secondo errore è stato generato a cascata e non fornisce informazioni utili; infatti, in casi come questo l'errore viene filtrato e ignorato. L'errore che fornisce informazioni realmente utili è quello relativo all'uso di una variabile non dichiarata.

L'output di tale esecuzione è:

```
Variable y undeclared! Position: Row: 2 | Col: 1
```

Senza tali controlli sulla propagazione e generazione degli errori avremmo avuto anche l'errore:

```
Cannot initialize variable y of type void with values of type int!
 Position: Row: 0 | Col: 0
```

che porta solo informazioni ridondanti.

Logica di esecuzione

Segue ora una breve descrizione concettuale del funzionamento di tale modulo. Le funzioni in esso presenti possono essere suddivise in tre categorie:

- funzioni **execute**;
- funzioni **typeCheck**;
- funzioni di supporto;

Le **funzioni execute** ricevono in input un nodo con parametro polimorfo di tipo **Posn** (sostanzialmente ciò che viene restituito dal parser) e l'environment in quel punto del programma; eseguono la chiamata all'opportuna funzione di tipo typeCheck per quel nodo e inoltre esegue le chiamate ricorsive di tipo execute sui nodi figli per continuare la visita dell'albero. Restituiscono il nodo in cui il parametro polimorfo è stato sostituito dal risultato della chiamata checkType (TResult).

Le **funzioni typeCheck** ricevono in input un nodo con parametro Posn e l'environment in quel punto del programma; restituisce in output l'esito della fase di analisi di semantica statica per quel nodo (TResult o TError).

Le funzioni di supporto sono funzioni che svolgono operazioni di base per le fasi di analisi; spesso definite per evitare di replicare codice all'interno del sourcecode.

Debugging Env e TCheckResult

Al fini di rendere più semplice la fase di debugging e controllo del funzionamento della propagazione/aggiornamento dell'environment e dei risultati delle fasi di analisi di semantica statica è stata creata una funzione di printing che semplificasse la visualizzazione dell'albero prodotto dal modulo di analisi di semantica statica. Tale funzione, per ciascun nodo statement visualizza il suo TCheckResult (quindi environment più informazioni relative a posizione e tipo dello statement attuale; oppure l'eventuale errore generato da esso). Tale funzione gestisce anche casi di statements annidati (blocchi, funzioni etc.). Questa funzione è stata di notevole aiuto nelle fasi di sviluppo, segue un esempio:

Input:

```
var x:int = 3;
x = 10;
{ // new block
```

```
  var y:int = x;
  y = "I'll generate error";}
y; // I'm no more visible outside the block! error here
var z:int = x;
z = 100;
```

Esecuzione di **showTypeCheckResult**:

```
[]|int|abs:6,row:1,c:7
[("x",[EnvEntry:[var:int|abs:0,row:1,c:1|mode:"var"|canOverr:False]])]|int|abs:19,row:2,c:5
[("x",[EnvEntry:[var:int|abs:0,row:1,c:1|mode:"var"|canOverr:False]])]|int|abs:31,row:4,c:7
--[("x",[EnvEntry:[var:int|abs:0,row:1,c:1|mode:"var"|canOverr:True]])]|int|abs:31,row:4,c:7
--Errors: ["Incompatible type at abs:40,row:5,c:1 - y requires an int but string is given"]
Errors: ["Unexpected var ident at abs:69,row:7,c:1 - y not defined!"]
[("x",[EnvEntry:[var:int|abs:0,row:1,c:1|mode:"var"|canOverr:False]])]|int|abs:132,row:8,c:7
[("x",[EnvEntry:[var:int|abs:0,row:1,c:1|mode:"var"|canOverr:False]]),("z",[EnvEntry: [var:
    int|abs:126,row:8,c:1|mode:"var"|canOverr:False]])]|int|abs:145,row:9,c:5
```

Ad esempio, in corrispondenza dello statement in riga 2 (x=10) vediamo come il TCheckResult di tale statement sia composto da un environment in cui è apparsa l'entry relativa alla dichiarazione di x in riga precedente. Mentre in riga 4 e 5 vediamo come i due TCheckResult siano stati printati con un indentazione per indicare che appartengono ai due statements all'interno del blocco.

2.4.3 Funzioni predefinite [Environment di partenza]

Per l'implementazione delle funzioni predefinite richieste dalla consegna (writeInt, writeReal, writeChar, writeString, readInt, readReal, readChar, readString) è stato seguito il seguente approccio:

- È stato scritto un file contenente la sintassi concreta per la definizione di tali funzioni/procedure.
- Per tale file in input è stato eseguito il front-end del compilatore.
- Si è estratto l'environment del programma nel punto finale (dopo la dichiarazione di tali funzioni).
- Tale environment è stato impostato come environment di partenza per l'esecuzione del modulo di analisi di semantica statica.

Questo implica che la chiamata a una di tali funzioni avrà successo, nonostante esse non vengano dichiarate nel codice in input (questo perché è come se le loro dichiarazioni fossero sempre presenti in cima a qualsiasi input).

2.4.4 Controllo produzione di sintassi concreta valida

L'output del parser, se linearizzato, dovrebbe produrre sintassi concreta valida. Nello specifico, se richiamiamo il parser su un input, e in seconda battuta lo richiamiamo sulla versione linearizzata dell'output, questo dovrebbe restituire (come linearized tree) la stessa stringa.

Questo controllo è stato implementato, il codice consiste di poche righe:

```
// tree = abstract syntax tree (output del parser sull'input iniziale)
if (printTree tree == printTree (case (p (myLexer (printTree tree))) of
        // Left: il linearized tree, come input ha causato errore.
        // Si restituira' un albero nullo, il match delle stringhe fallira'.
        Left e -> (Abs.StartCode (Pn 0 0 0) (Abs.EmptyStatement (Pn 0 0 0)))
        // Right: caso ok, viene restituito l'albero e si controlla
        //        il match delle stringhe
        Right newTree -> newTree))
then putStrLn "\n Serialization of abstract syntax tree produced legal concrete syntax!"
else putStrLn "\n WARNING: Serialization of abstract syntax tree DID NOT produce legal
  concrete syntax!"
```

2.5 TAC Generation

Avendo seguito l'approccio a pipeline, per la generazione del three address code è stato implementato un ulteriore modulo, chiamato **TacGen.hs**. A tale modulo è associato il modulo **AbsTac.hs**, contenente le strutture dati per la rappresentazione dell'albero e delle singole operazioni richieste dal TAC.

Tale modulo prende in input il risultato del modulo di analisi di semantica statica (un oggetto di tipo **S TCheckResult** - sostanzialmente l'albero generato dal parser dove ciascun parametro polimorfo è stato sostituito da Posn a TCheckResult.

Il modulo segue un approccio molto simile a quello precedente, esegue una visista dell'albero, generando man mano il tac di ciascun nodo, chiamando e propagando tale generazione ai nodi figli.

La fase di TAC generation non viene eseguita nel caso in cui il modulo di semantica statica abbia scaturito errori di compilazione (è possibile in realtà forzare la generazione cambiando un flag nel sorgente).

La struttura dati principale è TAC:

```
data TAC = TAC {code::[TACEntry],funcs::[TACEntry]}
  deriving (Eq, Ord, Show, Read)
```

Le due liste (code e funcs) contengono rispettivamente il codice tac di tutto il codice, escluso il codice relativo alle dichiarazioni di funzioni, contenuto nella lista funcs. Il codice tac è rappresentato sottoforma di **TACEntry**; rappresentanti le singole possibili istruzioni. La divisione tra codice tac relativo a funzioni e il resto del codice è necessario ai fini di gestire il corretto ordine di output di tale codice (prima il tac relativo alle definizioni di funzioni e poi il tac del resto del codice).

Seguono le definizioni delle strutture dati per rappresentare le istruzioni TAC definite all'interno di **AbsTac.hs**

```
data TACEntry
   = TacAssignUnaryOp      {getAddr::Address, unaryOp:TacUnaryOp, first::Address,
                            assignType::Type}
   | TacAssignBinaryOp     {getAddr::Address, binaryOp::TacBinaryOp, first::Address,
                            second ::Address, assignType::Type}
   | TacAssignRelOp        {getAddr::Address, relOp::TacRelOp, first::Address,
                            second::Address, assignType::Type}
   | TacAssignNullOp       {second_is_func::Prelude.Bool, getAddr::Address, first::Address,
                            assignType::Type}
   | TacProcCall           {getAddr::Address}
   | TacFuncCall           {getAddr::Address, first::Address, retType::Type}
   | TacFuncCallLeft       {getAddr::Address}
   | TacParam              {first::Address, paramType::Type}
```

```
   | TacReturnVoid          {}
   | TacReturn              {first:: Address, returnType::Type}
   | TacCastConversion      {getAddr::Address, first:: Address, fromType::Type, toType::Type}
   | TacJump                Label
   | TacLabel               Label
   | TacPointRef            {getAddr::Address,first::Address}
   | TacConditionalJump     {destination::Label, flag::Prelude.Bool, first::Address}
   | TacRelConditionalJump {destination::Label, flag::Prelude.Bool, relOp::TacRelOp,
                            first::Address, second::Address}
   | TacComment             Prelude.String
   | TacError               Prelude.String
   | ExitTac
 deriving (Eq, Ord, Show, Read)
```

Le singole operazioni unarie, binarie e di relazione sono a loro volta definite come segue:

```
data TacUnaryOp = Pos | Neg | Not | Point
 deriving (Eq, Ord, Read)

data TacBinaryOp = IntAdd | RealAdd | IntSub | RealSub | IntMul | RealMul | IntDiv
                 | RealDiv | IntMod | RealMod | IntPow | RealPow | AddrIntAdd
 deriving (Eq, Ord, Read)

data TacRelOp  = EqInt | EqReal | EqString | EqChar | EqBool
               | NeqInt | NeqReal | NeqString | NeqChar | NeqBool
               | GeqInt | GeqReal | GeqString | GeqChar
               | LeqInt | LeqReal | LeqString | LeqChar
               | GtInt | GtReal | GtString | GtChar
               | LtInt | LtReal | LtString | LtChar
               | And | Or
 deriving (Eq, Ord, Read)
```

Per ciascuna istruzione e operazione è stata definita una opportuna istanza di show per permettere il corretto printing del codice TAC generato.

Sono inoltre presenti i costrutti per le etichette e gli indirizzi di valori/variabili:

```
data Label = Label {label_id :: Prelude.String}
 deriving (Eq, Ord, Show, Read)

data Address = AddrString   {content_addr_string :: Prelude.String}
             | AddrInt      {content_addr_int    :: Prelude.Integer}
             | AddrBool     {content_addr_bool   :: Prelude.Bool}
             | AddrReal     {content_addr_real   :: Prelude.Double}
             | AddrChar     {content_addr_char   :: Prelude.Char}
             | AddrAddress  {content_addr_addr   :: Prelude.String}
             | AddrNULL     {}
 deriving (Eq, Ord, Show, Read)
```

Logica di esecuzione

Segue ora una breve descrizione concettuale del funzionamento di tale modulo. Le funzioni in esso presente possono essere suddivise principalmente in funzioni **generate** e funzioni di supporto.

Le funzioni generate visitano l'albero restituito dal modulo di analisi di semantica statica e per ciascun statement/costrutto per il quale sono necessarie istruzioni TAC esegue la loro generazione, eventualmente richiamando e utilizzando il risultato dell'elaborazione dei nodi figli. Queste funzioni sono inoltre incaricate della propagazione di contatori ed etichette che verranno poi utilizzate nelle fasi di generazione tac successive.

2.6 Note sul TAC

Seguiranno ora alcune note relative all'implementazione della fase di TAC generation.

2.6.1 Casting impliciti

Nel caso di casting impliciti dovuti alla compatibilità degli interi con i reali; come da esempio:

```
var x:real = 3;
```

L'approccio seguito è il seguente: non sarà il modulo TAC a occuparsi di rendere esplicito il casting implicito, bensì il modulo di analisi di semantica statica, che convertirà il codice precedente in:

```
var x:real = (3):real;
```

Tale codice produrrà in fase di TAC generation le istruzioni corrette relative al casting necessario. Il modulo TAC non è in grado di sapere se quell'operatore di casting sia stato inserito manualmente dal progammatore o dal typechecker.

2.6.2 Gestione delle variabili di controllo

Come illustrato in precedenza, i costrutti **if-then-else** e **while-do** permettono di dichairare delle variabili di controllo al posto della expression-guard. La guardia varrà quindi True o False a seconda che la dichiarazione sia valida o meno; caratteristica valutabile a tempo di compilazione, dal front-end.

A questo punto, il modulo di generazione del TAC può, analizzando il risultato del modulo precedente per tale dichairazione, operare correttamente; conoscendo a priori il valore della guardia.

Esempi esplicativi:

```
// L'intero if-then-else puo' essere sostituito con le sole istruzioni TAC
// della dichiarazione e inizializzazione di x a 2 e il TAC del blocco
// then.
if var x:int = 2 then ... else ...

// L'intero if-then-else puo' essere sostituito con le sole istruzioni TAC
// della dichiarazione e inizializzazione di x al suo valore di default e
// il TAC del blocco else.
if var x:int = "string" then ... else ...

// la guardia vale true; il tac consistera' nell'inizializzazione di x a 3
// e il corpo del do; la cui istruzione finale e' un jump all'inizio del
// corpo stesso.
while var x:int = 3 do ...

// nessun TAC verra' prodotto!
while var x:int = "string" do ...
```

- Si noti che nel caso di un blocco if-then (senza else) la cui dichiarazione è non valida, la semantica prevede che nessuna istruzione venga eseguita, in quanto andrebbe eseguito il corpo else che però non è presente. (similmente al caso del while-do).

- Si noti come nel caso in riga 14; se all'interno del corpo non è presente un break; il ciclo causerà un deadlock. In tale caso i continue non eseguiranno un jump alla guardia ma direttamente alla label del corpo.

2.6.3 Operatore di referenziazione

Come precedentemente accennato, l'operatore di referenziazione è esplicito. Tuttavia, qualora necessario, esso viene esplicitato in fase di generazione del TAC. Esempio:

```
1 // codice in input:
2 var x : int = 10;
3 var y : int $ = x;
4
5 // TAC generato:
6 x@1,5 =int 10
7 y@2,5 =address &x@1,5
```

3 Compilazione, Testing ed Esecuzione

Compilazione:
Per la compilazione è sufficiente lanciare il comando **make**.

Testing:
Per eseguire una batteria di test è sufficiente eseguire il comando **make demo** oppure **TestProgetto.exe --test**. Questo eseguirà una serie di test costruiti ad-hoc, di complessità crescente, a volte volutamente errati per mostrare il funzionamento del modulo di analisi di semantica statica. I test sono i singoli contenuti dei file .txt contenuti nella cartella tests.

Ulteriori informazioni sulle due modalità di output del tester nelle note a piè pagina.

Aggiunta di nuovi test:
Per aggiungere nuovi test case è sufficiente seguire la seguente procedura:

- Creare un nuovo file, all'interno della cartella tests, di nome n.txt, con n l'intero successivo a quello dell'ultimo test file presente;
- Riportare all'interno del nuovo file il codice in input da testare;
- Modificare la variabile "numberOfTest" con il nuovo valore intero nel file TestProgettoPar.hs;
- Ricompilare ed eseguire.

Sostanzialmente i file dei test devono chiamarsi con un numero intero, in successione a partire da 1: $1, 2, 3....n$; numberOfTest deve essere pari a n.

Esecuzione manuale:
Per richiamare il front-end è sufficiente eseguire **TestProgetto.exe**; inserire l'input ed assicurarsi che a fine linea sia presente un newline; inserire ctrl + z e premere nuovamente invio.

4 Note

4.1 Possibili Migliorie

A seguito dell'implementazione, sono emersi dettagli che hanno portato alla considerazione di alcune possibili migliorie:

- Utilizzare le liste (nel parser) qualora possibile, così da semplificare la struttura dell'albero e delle successive operazioni.
- Eseguire una fase di semplificazione e compressione dell'albero di sintassi astratta prima della fase di analisi di semantica statica (rimuovendo nodi "di passaggio" etc.).

4.2 Modalità del Tester

Il tester può lavorare in due modalità: normale (default) e debug; per cambiare modalità è sufficiente cambiare il valore del flag **includeDebugInfo**. La modalità influisce su cosa viene printato a fine esecuzione; la modalità debug è più esaustiva e permette l'analisi accurata dell'output.

Nella modalità normale viene printato in output:

- codice in input
- linearized tree
- controllo che il linearized tree abbia prodotto sintassi concreta valida
- errori di compilazione
- tac code

Nella modalità debug; oltre a quanto printato dalla modalità normale, vengono inclusi:

- print naive dell'albero di parsing
- print naive dell'albero in output del modulo di semantica statica
- print semplificato del risultato di analisi di semantica statica illustrato in precedenza
- print naive del AbsTac

Esempio di output nella modalità normale (default).

```
C:\Users\Mansitos_Picci\Desktop\workdir>TestProgettoPar.exe
var x:int = 3;
var y:int = 5+2+x;
^Z

[Parsing]

   Parse Successful! :)
```

```
[Input Code]

var x:int = 3;
var y:int = 5+2+x;


[Linearized tree]

var x : int = 3;
var y : int = 5 + 2 + x;

 Serialization of abstract syntax tree produced legal concrete syntax!

[Compiler Errors]

   No compiler errors.

[TAC]

   Tac of pre-defined functions is not included.

----------- Functions declarations -----------


--------------------- Code ---------------------

          x@1,5 =int 3
          t1 =int 5 add_int 2
          t0 =int t1 add_int x@1,5
          y@2,5 =int t0
end:
```