Parte 5

Sommario

Introduzione

Con questo progetto si vuole generare il type checker e il TAC generator di un linguaggio simile a Chapel. Partendo dalla struttura creata da BNFC si va a generare la grammatica, particolare attenzione va fatto a questo punto. Per rendere il più possibile flessibile il codice a modifiche, non è stato alterato il codice generato di BNFC, infatti le strutture create sono perfettamente adatte per il type checker e il tac generator. BNFC permette di generare token con la sua posizione relativa e le regex usate per la formulazione dei letterali sono malleabili per poter definire al meglio le varie regole. Inoltre, le regole di associatività sono automaticamente create grazie alla coercion.

L'albero sintattico creato da BNFC, dunque, non viene modificato e viene passato direttamente al type checker che forma un albero totalmente diverso chiamato albero di programmazione. Esso contiene le informazioni aggiuntive che serviranno poi allo stesso type checker e al TAC generator per creare il codice intermedio. Il TAC generator riprocessa l'albero sintattico adoperando anche le informazioni in parallelo dell'albero di programmazione.

L'aggiunta di un nuovo costrutto sintattico va fatta direttamente con BNFC senza dover manipolare a mano il risultato avuto precedentemente. Il nuovo costrutto porterebbe solo all'aggiunta di una nuova regola che poi dovrebbe essere implementata del type checker e nel TAC generator, senza inficiare altre operazioni.

Struttura lessicale

Identificatori

Gli identificatori sono stringhe non quotate che cominciano con una lettera, seguita da ogni combinazione di lettere, numeri e i caratteri "_" e "'". Le parole riservate sono escluse da questo contesto.

Letterali

I letterali per numeri interi, numeri a virgola mobile, caratteri e stringhe usano le convenzioni usuali degli altri linguaggi di programmazione. Le parole riservate per i booleani sono `true` e `false`

```
<Literal>  ::=  <Boolean>
             |  <Char>
             |  <Double>
             |  <Integer>
             |  <String>
```

Parole riservate e simboli

Le parole riservate sono le seguenti

```
false     true      if        then      else      do
while     int       real      char      bool      string
break     continue  ref       var       return    proc
for       in
```

I simboli usati i seguenti

```
{         }         (         )         [         ]
;         :         .         ,         <         >
+         -         *         /         %         !
||        &&        ==        !=        <=        >=
+=        &         ?         ^
```

Commenti

I commenti su singola linea cominciano con `//`, quelli su multipla linea invece sono racchiusi tra `/*` e `*/`

Struttura sintattica

- Un programma è definito come modulo ed è composto da una sequenza di dichiarazioni
- Una dichiarazione può essere di due forme
    - Dichiarazione di variabili. Esse cominciano con la parola chiave "`var`" seguite da una lista di item nella forma `var[<Ident>](:<TypeSpec>)?=<Expr>`. Essi possono essere separati da una virgola e vengono terminati dal punto e virgola. È possibile, inoltre, dichiarare più identificatori separandoli con una virgola. Un esempio è il seguente: `var a,b :int = 10, b = 9;`
      E' possibile anche non inserire la `TypeSpec` in modo tale da ricavare il tipo dalla `<Expr>`.
      Una specializzazione di tipo è composta nella seguente forma:

```
<TypeSpec>        ::=  = <ConstructType>
                    |   [<ArrDecl>] <ConstructType>
<ConstructType>   ::=  <SimpleType>
                    |   * < TypeSpec >
<ArrDecl>         :=   [Integer]
                    |   [Integer..Integer]
<SimpleType>      ::=  bool | char | real | int | string
```

Per esempio `var a[10,20]:int` indica un array di 10 elementi, ognuno composto da un array di 20 interi. Nel caso di inizializzazione di un array `[n]t` dove `n` è il numero di elementi e t è il tipo, dovranno corrispondere `[t1, … tn]` elementi nella espressione a destra.
La dimensione dell'array non è opzionale e va indicata. Nel caso non voglia essere inserita, omettere direttamente la `TypeSpec` che verrà ricavata dall'espressione.
E' possibile dichiarare un array indicando il limite sinistro e il limite destro, normalmente dichiarando `a[2]`, il limite sinistro sarà 0 mentre il destro 1. Indicando invece usando la sintassi `a[3..4]`, allora il limite sinistro sarà 3 mentre il destro 4, come prima il numero degli elementi sarà 2. In questo modo per accedere alla prima posizione dell'array si dovrà indicare con `a[3]` mentre l'ultima posizione con `a[4]`.

Nel caso si voglia dichiarare un puntatore di array di puntatori di array è possibile farlo utilizzando la seguente sintassi come esempio: *[3,4] * [3,4] * string

- Dichiarazione di funzioni o procedure. Esse cominciano con la parola chiave "`proc`" seguite dall'item nella forma `proc <Ident>([<Parameter>]<Block>:<TypeSpec>)?`
  La lista dei parametri `(<Parameter>)` può essere vuota o da uno o più elementi separati da una virgola.
  Inoltre nel caso in cui non venga specificato il tipo (`TypeSpec`) la funzione verrà considerata una procedura, cioè che non ritorna alcun valore.
  Ogni parametro è formato da un `(<Mode>)?<Ident><TypeSpec>` dove le modalità sono

```
<Mode>  ::=  ref
```

  La modalità può essere omessa.
  È possibile chiamare una funzione e non utilizzare il tipo di ritorno. Nel TAC, dunque verrà comunque creato un temporaneo ma che non verrà usato.

- Un blocco ha la forma di

```
<Block>  ::=  {[<BodyStatement>]}
```

  La lista di `<BodyStatement>` viene compresa tra due graffe e ognuno può essere

```
BodyStatement   ::=  <Statement>
                |    <Function> ;
                |    <Declaration>
                |    <Block>
```

Le dichiarazioni, gli statement, le funzioni, le dichiarazioni e altri body possono essere inseriti in un qualsiasi ordine. Lo scoping verrà discusso nelle prossime sezioni.
Le funzioni vengono dichiarate allo stesso modo di come è stato descritto sul modulo con l'aggiunta finale del punto e virgola `;` stessa cosa vale per le dichiarazioni.
Gli statement vengono descritti nel punto successivo.

- Gli statement sono nella forma seguente

```
Statement  :=    <Expr[1]> ;
           |     if ( <Expr> ) then  <Block>
           |     if ( <Expr> ) then  <Block>  else
                 <Block>
           |     do <Block> while ( <Expr> )
           |     for <Expr> in <Expr> do <Block>
           |     while ( <Expr> ) <Block>
           |     return ;
           |     return <Expr> ;
           |     continue ;
           |     break ;
```

- Le espressioni possibili sono le seguenti

```
<Expr>   ::=  <Expr> <Bop> <Expr>
         |    <Expr> ? <Expr> : <Expr>
         |    <Ident>
         |    <Preop> <Expr>
         |    ( <Expr> )
         |    <Literal>
         |    <Ident> ([<Expr>][2])
<Bop>    ::=  = | += | && | || | == | != | < | > |
              <= | >= | + | - | * | / | % | ^
<Preop>  ::=  ! | & | *
```

La precedenza e l'associatività degli operatori sono la seguente:

	operators	precedence	associativity	
	= +=	1	left	
	||	2	left	
	&&	3	left	
	== !=	4	left	
	< > >= <=	5	left	
	+ -	6	left	
	* / %	7	right	
	! & * -	8	left	
	^	9	left	

[1] Chiaramente le espressioni ammesse alla fine saranno solo le adeguato left expression.

[2] Le espressioni vengono separate dalla virgola ","

Note su If Then Else su espressioni

La scelta della sintassi per l'if then else sulle espressioni è stata dettata dalla necessità di non creare conflitti nel parser LALR. Per questo è stata scelta la seguente sintassi (simil c#)

```
expbool ? exp1 : exp2
```

Il significato è il seguente: Se expbool è true allora viene considerata la exp1 altrimenti la exp2.

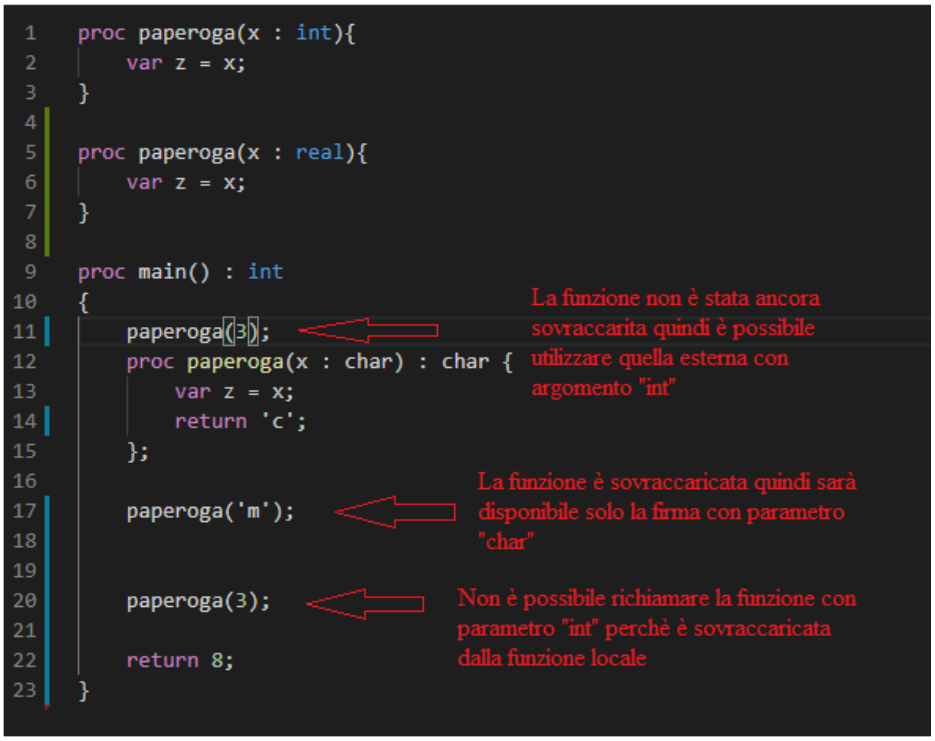
E' stato deciso di non utilizzare la sintassi "`If expbool then exp1 else exp2`" in quanto la grammatica fatta in questo modo avrebbe generato un conflitto reduce/reduce.

Semantica statica

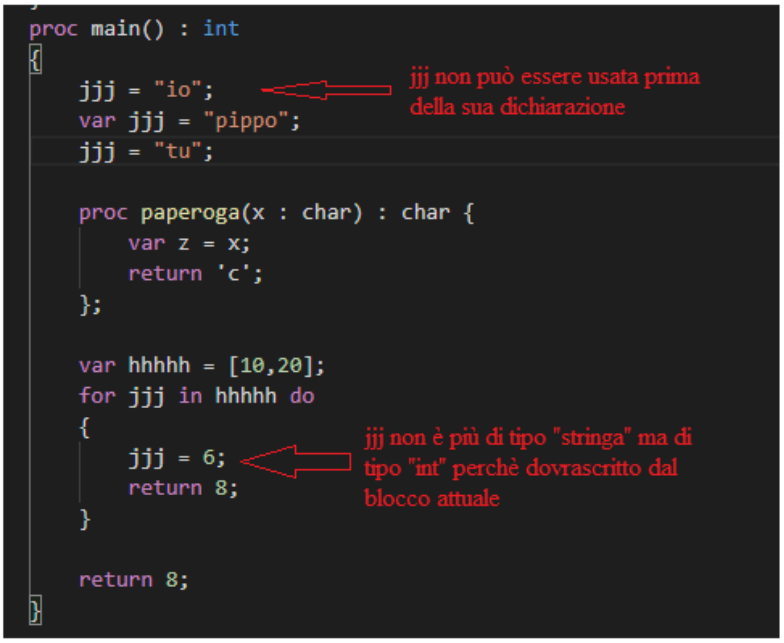
Lo scope è quello canonico, in un blocco non possono essere dichiarati più variabili con lo stesso nome, invece il sovacaricamento delle funzioni è possibile. Quindi più funzioni con lo stesso nome possono essere dichiarate se i tipi della firma sono differenti e il tipo di ritorno è lo stesso. Con stesso tipo di firma si intende stesse modalità di passaggio e tipi riferendosi alle relative posizioni nella lista.

Nel caso in un blocco venga definita una variabile già presente in un blocco esterno, quella locale ha la priorità.

Nel caso di funzioni, quella locale sovrascriverà quella del blocco esterno eliminando anche tutti i relativi overloading. Per esempio, se nel blocco esterno viene dichiarata una procedura "`f`" con due overloading mentre in quella interna solo una, si potrà usare solo la procedura `f` all'interno mentre entrambe quelle esterne saranno nascoste.



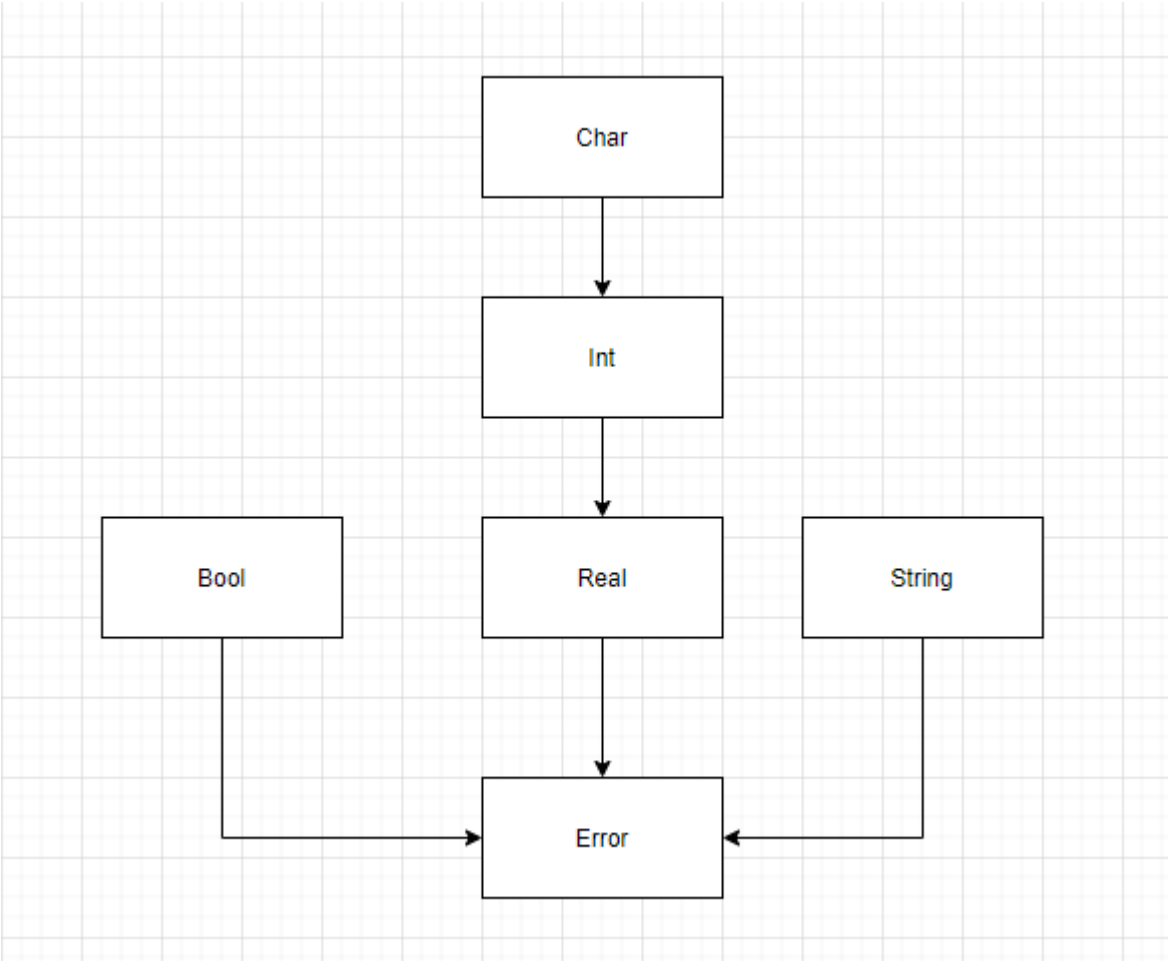
La variabile dichiarata in un blocco può essere usata nel blocco corrente e in quelli annidati dal punto di dichiarazione.



Per poter permettere questo approccio inoltre è stato deciso che in caso di funzioni che abbiamo parametri con modalità differenti di passaggio come per esempio riferimento, anche nella chiamata di funzione si dovrà specificare la modalità riferimento in modo tale da riconoscere la firma adeguata nel TAC.

Tipi

Le relazioni sui tipi semplici vengono mostrate nella figura seguente



Di seguito la descrizione di alcune regole sui tipi

- I tipi predefiniti sono `int`, `real`, `bool`, `string` e `char` che sono i tipi semplici, più il costruttore `[n]t` " array di n valori di tipo `t`" e `*t` "puntatori al tipo `t`". I tipi `bool`, `char`, `int` e `real` sono tipi primitivi e possono essere usati direttamente sul TAC. Nel caso di utilizzo nel TAC del tipo array verrà identificato con address. Questo può capitare quando si utilizza un puntatore di array e si deferenzia per accedere in seguito ad un suo elemento.

```
1
2   proc main() : int
3   {
4       var ar1 = [2,2,2];
5       var pAr1 = &ar1;
6       return (*pAr1)[2];
7
8   }
```

In questo caso nel TAC si avrà come cast il tipo address

- Le stringhe non possono essere modificate in alcun modo. Esse vengono adoperate solo per essere visualizzate in output.
- Il tipo `int` è composto da 4 byte, `real` da 8, `char`, `bool` da 1 e `string` da 8.

- Nel caso di operatori `&` e `*` in sequenza anche inseriti tra parentesi (ad esempio `&(*&(*&*&*&*a))`) verrà eseguito il controllo in cui `&*` o `*&` si elidono. Inoltre, non è possibile eseguire addressing multipli come ad esempio `&&a`, per poter quindi avere un puntatore di un puntatore bisognerà usare una variabile supporto in cui eseguire il primo addressing e poi eseguire il secondo in un assegnamento successivo. La referenziazione multipla (`**a`) è possibile a patto che si abbia un tipo adeguato.
- Il tipo `char` è compatibile con il tipo `int` e il tipo `int` è compatibile con il tipo `real` , quindi il tipo `char` è compatibile con il tipo `real` .
  Se il tipo `t` è compatibile con il tipo `t'` allora per ogni `n`, `[n]t` è compatibile con `[n]t'`. Gli array, dunque, devono avere le stesse dimensioni.
  Questi sono i soli tipi compatibili quindi `[4,6]int` è compatibile con `[4,6]real` mentre `[4,6]real` non è compatibile con `[4,6]int` come `*int` e `*real`
- Per un parametro formale di tipo `t`, abbiamo differenti requisiti in base alla modalità di passaggio
  - Modalità per valore (nessuna modalità specificata). Il tipo del parametro attuale deve essere esattamente uguale a `t`. Questo per poter usare l'overloading delle funzioni.
  - Modalità per riferimento (modalità reference specificata). Il tipo del parametro attuale e il suo passaggio devono essere esattamente uguali a `t`

- Per gli operatori e le varie casistiche di compatibilità la tabella di corrispondenza è la seguente

Operatori/Modalità	Tipi
`+ - * /`	`int × int → int` `float × float → float`
`%`	`int → int`
`|| &&`	`bool × bool → bool`
`!`	`bool → bool`
`-`	`int → int` `float → float`
`== !=`	`t × t → bool with t ∈ {bool, char, int, real, string}`
`< <= > >=`	`int × int → bool` `float × float → bool`

Parametri funzioni	`mod t × mod t → t`
	`with t ∈ {bool, char, int, real, string, *t, [n]t}`
	`with mod ∈ {-empty-, ref}`

Dichiarazione / Assegnamento/ `Return`	`int × int → int`
	`float × float → float`
	`string x string = string`
	`bool x bool = bool`
	`*t1 x *t2 = *t`
	`[n]t1 x [n]t2 = [n]t`
	(i tipi t1 e t2 devono essere compatibili seguendo le precedenti regole)

- Le guardie per le iterazioni “`do while`” “`while`” e i condizionali “`if then`” e “`if then else`” devono essere tutte di tipo `bool`

Altre restrizioni semantiche

- Il “`break`” e il “`continue`” possono essere dichiarati solo nei blocchi di controllo sequenza “`while`” e “`do while`”. In particolare, il `continue` nel “`do while`” comunque controllerà la guardia prima di decidere se continuare l’iterazione o terminarla. Come da consegna, il break e il continue non sono permessi all’interno del ciclo determinato for.
- Nel caso si vogliano definire i limiti degli array il limite destro deve essere maggiore o uguale al limite sinistro, inoltre nel caso in cui ci sia una dichiarazione e il tipo della variabile deve essere ricavato dal contesto, il tipo sarà uguale al primo elemento dell’array, e così via per tutti gli altri elementi.
- Le procedure non è necessario ritornino un valore mentre le funzioni devono ritornarlo sempre.
- Nel caso in cui non si possa ricavare per una dichiarazione il tipo dall’espressione, la variabile verrà considerata come non dichiarata.
- Gli array non possono essere dichiarati vuoti (quindi con 0 elementi)
- È possibile inserire solo una espressione che ritorna un array nell’iteratore del “for” e una variabile nell’item che viene creato a ogni iterazione. Inoltre, l’item creato ha solo scoping all’interno del blocco di iterazione. Può sovrascrivere una variabile esterna già esistente con lo stesso nome.

Type checker

In questa parte verranno descritti i concetti usati per l'implementazione.

Blocchi di programmazione

Il type checker per effettuare i vari controlli e passare una struttura adeguata al TAC generator crea un albero di cui i nodi sono composti da un blocco di programmazione o BP. L'albero viene abbreviato con BPTree.

Un BP viene definito come un blocco con uno scoping locale in cui possono essere dichiarate nuove variabili. In particolare, le nuove variabili già dichiarate nei BP genitori vengono sovrascritte garantendo lo scoping locale.
I blocchi di programmazione vengono definiti da due graffe, una iniziale aperta e una finale chiusa che indicano i limiti del blocco. Uscendo dal blocco le variabili al suo interno non saranno più accessibili.
La struttura del BP contiene quella che viene definita symbol table, una serie di statement utili al controllo semantico statico, il tipo di blocco e gli errori generati al suo interno.
Il BP infine viene inserito all'interno nel nodo dell'albero con un suo identificatore per poterlo localizzare e le posizioni di inizio e fine nel testo dati dai due token indicanti l'apertura e la chiusura.

I blocchi di programmazione definiti sono:

- Blocco principale. Il blocco iniziale che indentifica il programma. Contiene le variabili globali e le funzioni principali
- Blocco funzione. Il blocco identifica le funzioni. Unico possibile figlio del blocco principale ma che può essere anche figlio di tutti gli altri nodi
- Blocco semplice. Un blocco in cui possono essere definite variabili locali o nuovi blocchi. Viene indicato solo dall'apertura e chiusura da una graffa
- Blocco "`if then`". Il blocco che identifica un blocco condizionale "`if then`"
- Blocco "`if then`" e "`else`". Questi blocchi sono fratelli e identificano rispettivamente il blocco "`if then`" e "`else`"
- Blocco "do while". Il blocco che identifica un blocco di sequenza "do while"
- Blocco "while". Il blocco che identifica un blocco di sequenza "while"
- Blocco "for". Il blocco che identifica un blocco di sequenza foreach su variabili di tipo array

Eccetto il blocco principale o funzione tutti gli altri blocchi devono essere discendenti da un blocco funzione.
Inoltre i figli di un nodo sono ordinati per posizione nel caso di blocchi consecutivi. Se un blocco è interno di un altro esso sarà suo figlio.

Tutte queste proprietà saranno utili per poter generare il TAC in modo semplice ed eseguire lo stesso type checking.

Symbol table

La symbol table è una mappa con chiave identificatore (il nome della variabile) e come valore la dichiarazione di funzione o variabile (chiamate entry)

- Variabile. Identificata da una locazione nel codice, il suo tipo e la sua modalità.
- Funzione. Identificata da una lista di coppie e un tipo di ritorno. Le coppie sono composte da una locazione e una lista di variabili. Ogni elemento è un overloading della funzione.

Per avere lo scoping locale di un BP e quindi ricavare la variabile con il tipo corrispondente, vengono unite le mappe dal BP attuale fino al genitore. Una funzione dedicata ricava il path e poi esegue l'unione.

Statement

Negli statement sono inseriti tutti quegli statement di rilevanza utile per eseguire il controllo semantico statico, per esempio i return

Stato

Durante l'esecuzione del typechecker viene passato uno stato contente una tupla di valori

- Lista di entry. In questa lista possono essere inserite eventuali entry che non devono essere inserite nel blocco attuale ma in un blocco successivo.
- Albero di programmazione. L'albero che verrà creato dal type checker e restituito nel passo successo: la generazione del TAC
- Coppia identificativa. Viene passato continuamente il nodo attuale in cui ci troviamo nell'albero, esso viene indicato attraverso l'identificativo, la posizione di apertura blocco e di chiusura blocco

Creazione dell'albero di programmazione

Per come è costituita la grammatica tutti i vari blocchi a parte il principale sono tutti identificati dai token di apertura e chiusura graffa, più precisamente dalla posizione nel testo univoca.
La grammatica descrive precisamente questo blocco attraverso una struttura fissa chiamata `BodyBlock`.
Ogniqualvolta viene processata una struttura di questo tipo allora avviene la costruzione di un blocco che si aggiungerà come figlio al blocco attuale.
Il blocco attuale, quindi, verrà aggiornato con quello appena generato. Alla fine del processamento del nuovo BP, verrà riassegnato il blocco attuale nello stato con quello precedente e si continuerà con le prossime istruzioni.

Un caso speciale avviene con le funzioni: le variabili dichiarate nella firma non devono essere inserite nel blocco attuale ma nel blocco successivo; nello stato c'è una cache in cui è possibile aggiungere le entry che possono recuperate in un successivo momento. Automaticamente alla creazione di un nuovo blocco viene svuotata la cache e i suoi elementi vengono inseriti nel BP appena creato.

Il procedimento di creazione inoltre permette anche di avere un ordine nei blocchi nell'albero che viene usato poi nel TAC per poter trovare il tipo corretto della variabile nel BP.

Infatti, definita la posizione di un token è possibile trovare il BP in cui risiede attraverso la seguente procedura (partendo dal nodo radice):

- Controllando ogni figlio di un nodo, se la posizione di un token non è compresa tra l'inizio e la fine di alcun figlio allora il nodo a cui fare riferimento è quello attuale
- Se esiste un figlio che è sicuramente unico allora si riesegue la procedura su tale nodo.

Tabella di controllo compatibilità

Tutti i controlli tra due tipi vengono effettuati in tale tabella chiamata anche `SupTable`. In essa vengono definiti tutte le possibili combinazioni. In genere il risultato è costituito dal tipo di ritorno e un booleano che indica o meno la compatibilità.
Generalmente per poter continuare la generazione di eventuali successivi controlli, anche nel caso di un errore di compatibilità, può essere ritornato comunque un tipo, altrimenti nel caso che l'errore debba essere bloccante allora può essere ritornato il tipo speciale `Error`.
La tabella inoltre è provvista anche di una modalità di controllo chiamata `SupMode`, questa si è rilevata

necessaria perché due tipi compatibili per una operazione potrebbero non essere compatibili in un'altra. Passando la modalità è possibile indicare eventuali nuove combinazioni.

La tabella è strettamente legata alla tabella di compatibilità degli operatori definita in precedenza.

Check della presenza di RETURN

Il type checking viene effettuato in due "passate": la prima passata svolge la maggior parte del lavoro e produce un primo albero di BP.

La seconda passata serve invece per verificare che le varie funzioni non `void` definite nel codice abbiano almeno un `return` che sia **sicuramente** raggiunto.

Tale funzione prende dunque in input l'albero di BP e restituisce un albero di BP aumentato con eventuali errori riguardanti funzioni con `return` non raggiungibili.

Inizialmente l'algoritmo viene lanciato su tutti i blocchi figli della root (che sappiamo per certo essere composta da funzioni).

L'algoritmo lavora ricorsivamente: verifica dapprima se nel blocco principale è presente almeno un `return`. Se non c'è procede ricorsivamente sui blocchi figli.

L'algoritmo inizia verificando se nel blocco principale della funzione è o meno presente un `return`. Tale informazione è facilmente reperibile in quanto nel BP c'è un campo (chiamato statements[3]) che contiene la lista dei `return` presenti nel blocco (eventuali `return` presenti nei figli del blocco NON compaiono dentro statements).

I) Se è presente almeno un `return` allora la funzione è corretta. Prima di poter concludere l'esecuzione però bisogna rilanciare lo stesso algoritmo su eventuali blocchi funzione figli. (Ovverosia verificare la corretta presenza di `return` anche nelle funzioni definite all'interno della funzione che sto analizzando)

Se non è presente nessun `return` allora si lavora ricorsivamente sui figli, partendo in ordine dal primo, e mi fermo non appena uno dei figli soddisfa la condizione di `return`. Se nessuno dei figli ha almeno un `return` sicuramente raggiungibile allora viene sollevato un errore simile al seguente:

```
"Error in line 14 and column 20: In function "main": there is a possible path in the code with no returns."
```

I figli possono essere di diversi tipi:

If semplici/Cicli While

Questi vengono ignorati (ovverosia quando ne incontro uno passo ad analizzare direttamente il blocco successivo, ignorando i figli). Non ho infatti la certezza che eventuali `return` presenti dentro tali blocchi siano raggiunti (causa guardie sempre false, ad esempio)

[3] Il campo statements viene popolato durante la prima passata del typechecker e contiene i `return` trovati all'interno del blocco, ma non all'interno dei figli. (Quindi ad esempio se il blocco funzione contiene un `if` con dei `return`, questi non risulteranno all'interno del blocco funzione, ma solo all'interno del blocco `if`.

Do While

Sono trattati diversamente dai cicli `while` "classici" in quanto la prima iterazione viene sempre svolta. Se trovo un `return` dentro tale blocco allora la funzione non ha errori, quindi procedo chiamando l'algoritmo sugli eventuali blocchi funzione figli (come nel caso I).

Se non trovo un `return` continuo ricorsivamente sui figli del blocco `Do While`.

Se nei figli risulta almeno un `return` raggiungibile allora il `Do While` è accettata, e dunque l'accettazione risale ricorsivamente, fino ad arrivare al blocco funzione principale.

Se neanche nei figli risultano dei return sicuramente raggiungibili allora procedo andando avanti con il blocco successivo al Do While.

Cicli determinati (For)

Questi blocchi vengono trattati in modo simile ai do While.

Infatti, dato che il linguaggio non accetta dichiarazione di array vuoti (di 0 elementi) si ha la certezza che il codice nel blocco del ciclo for verrà eseguito almeno una volta.

If Then Else

Per rendere valido tale blocco è necessario che compaia almeno un `return` sia nel blocco `then` che nel blocco `else`. (Se succede infatti sono sicuro che almeno un `return` verrà comunque raggiunto). Se tale condizione non si verifica si procede ricorsivamente sui blocchi figli del `then` e dell'`else`, e si procede in maniera simile al `do while`: se non è presente almeno un `return` sia nei figli di `then` che nei figli di else allora procedo andando avanti con il blocco successivo all' `If Then Else`.

Blocchi Semplici

Questo è il caso più semplice: se non trovo `return` continuo ricorsivamente sui figli del blocco, e procedo poi in maniera simile al `Do While`.

Controllo di break e continue

Per poter controllare i `break` e i `continue` viene semplicemente controllato se il BP in cui si trova o un antenato sia di tipo iterazione (`do while`, `while`). Nel caso in cui non sia trovato un blocco iterazione prima di un blocco funzione o del blocco principale allora viene segnalato un errore.

Descrizione Errori TypeChecker

Gli errori nel typechecker sono rappresentati dal tipo di dato ErrorChecker così definito

```
data ErrorChecker = ErrorChecker Loc DefinedError
```

Si noti come ogni errore porta con sé anche una posizione principale, in modo da poter essere il più possibile di aiuto al programmatore per individuare l'errore nel codice.

Il tipo di dato DefinedError (definito in Checker/ErrorPrettyPrinter.hs) dichiara quali sono i possibili errori che il typechecker può sollevare.

Si noti come all'interno del dato DefinedError possano essere presenti ulteriori informazioni di posizione dipendenti dal particolare tipo di errore.

Quando il typechecker incontra un errore va a creare un'entità di tipo ErrorChecker, che poi viene inserita opportunamente nel blocco del BPTree opportuno.

Una volta che il typechecker conclude l'esecuzione vengono estratti tutti gli errori dall'albero tramite la funzione `getTreeErrors` e quindi viene chiamata la funzione `printErrors` per fare il pretty-print degli errori.

Di seguito la descrizione dei possibili errori che possono essere sollevati dal typechecker

ErrorVarNotDeclared

Errore sollevato quando nel codice si utilizza una variabile che non è stata dichiarata

ErrorIncompatibleTypeArrayIndex

Errore sollevato quando si tenta di utilizzare un indice non int su un array.

Esempio

```
var g : [1..3] int = [1,2,3];
g[2.2] = 4;
```

ErrorIncompatibleAssgnOpType

Errore sollevato in fase di assegnamento quando si tenta di assegnare ad una variabile una espressione con tipo non compatibile.

Esempio:

```
var u : int = "ciao";
```

ErrorIncompatibleDeclarationArrayType

Errore sollevato in fase di dichiarazione di array, quando gli elementi dell'array non sono compatibili con il tipo dichiarato.

Esempio

```
var g2 : [1..3] int = [1,2,"3"];
```

ErrorIncompatibleDeclarationType

Errore sollevato in fase di dichiarazione di una variabile, quando l'espressione di inizializzazione non è compatibile con il tipo dichiarato per la variabile.

Esempio

```
var m2 : string = 5;
```

ErrorIncompatibleReturnType

Errore sollevato quando il tipo dell'espressione di un return in una funzione non è compatibile con il tipo di ritorno dichiarato per la funzione.

ErrorIncompatibleUnaryOpType

Errore sollevato quando viene utilizzato un operatore unario con un tipo non compatibile (esempio: NOT <variabile Real>)

ErrorIncompatibleIfTernayOpType

Errore sollevato quando viene utilizzato in un operatore if ternario dei tipi di ritorno non compatibili (esempio: <Bool> && <Real>)

ErrorIncompatibleBinaryOpType

Errore sollevato quando viene utilizzato un operatore binario con dei tipi non compatibili (esempio: <Bool> && <Real>)

ErrorMissingReturn

Errore sollevato quando la funzione non ritorna almeno un return. Questo errore non viene generato nel caso di procedure

ErrorVarAlreadyDeclared

Errore sollevato quando si dichiara più volte una variabile nello stesso blocco, oppure quando si dichiara una variabile nel blocco principale di una funzione, ma tale variabile risulta presente anche nei parametri della funzione stessa.

Questo errore NON viene sollevato quando re-dichiaro una stessa variabile in un sotto-blocco

Esempio corretto

```
var c = 4;
if (c < 5) then {
```

```
    var c = 5;
    return;
 }
```

In questo caso, infatti, la variabile c interna al blocco IF non è la stessa del blocco esterno. Una volta usciti dall'if il valore della variabile c del blocco esterno continua ad essere 4.

ErrorSignatureAlreadyDeclared

Errore sollevato quando si dichiarano più funzioni con lo stesso identico signature.

Esempio dove viene sollevato tale errore

```
proc pluto(x : int) : int{
    return x + 1;
}


proc pluto(x: int) : int {
    return x ;
}
```

Si noti come tale errore non venga sollevato nel caso di overloading di funzioni. Ad esempio:

```
proc pluto(x : int) : real{
    return x + 1.2;
}


proc pluto(x: real) : real {
    return x ;
}
```

ErrorGuardNotBoolean

Errore sollevato quando il typechecker trova che il tipo dell'espressione dentro una guardia (di un if, while) non è un booleano.

Esempio corretto

```
var f : [1..3] int = [1,2,3];
```

ErrorDeclarationCallWithZero

Error sollevato in fase di dichiarazione di array quando si utilizza 0 come bound di un array nel caso di costanti (Non può essere dichiarato un array con 0 elementi)

ErrorDeclarationBoundOnlyConst

Errore sollevato in fase di dichiarazione di array, quando si utilizza una variabile come bound di un array. (Non è possibile utilizzare variabili, ma solo costanti)

ErrorDeclarationBoundNotCorrectType

Errore sollevato in fase di dichiarazione di array, quando si utilizza un valore non intero come bound di array.

Esempio con errore

```
var c = 3;

var f : [1..c] int = [1,2,3];
```

ErrorArrayCallExpression:

Errore sollevato quando effettuo delle operazioni sull'array prima di effettuare l'indexing.

Esempio

```
var g3 : [3] int = [1,2,3];
var d3 = (g3 + 1)[2];
```

ErrorDeclarationBoundArray

Errore sollevato in fase di dichiarazione di array, quando si utilizza un valore non intero come bound di array.

ErrorBoundsArray

Errore sollevato in fase di dichiarazione di array quando il bound destro è minore del bound sinistro.

Esempio con errore

```
var f : [3..1] int = [1,2,3];
```

Esempio corretto

```
var f : [1..3] int = [1,2,3];
```

ErrorWrongDimensionArray

Errore sollevato quando si tenta di accedere ad un array specificando erroneamente gli offset.

Esempio con errore:

```
  var f : [1..3] int = [1,2,3];
  f[4,2] = 2;
// tratto l'array f come array bidimensionale invece è mono-dimensionale
```

ErrorArrayExpressionRequest

Errore sollevato quando si tenta di accedere ad un array utilizzando un indice non valido.

Esempio:

```
var ar1 : [5] int = [1,2,3,4,5];
var d = ar1[[2]];
```

ErrorCantAddressAnExpression

Errore sollevato quando si tenta di fare l'addressing di una espressione (o di una costante).

Esempio:

```
var z = &5;
```

Oppure

```
var z = &(x + 5);
```

ErrorInterruptNotInsideAProcedure

Errore sollevato quando viene trovato un break/continue al di fuori di un ciclo while/do while. Il break può trovarsi solamente all'interno di un blocco while/do while (e naturalmente dentro un blocco interno blocco while/do while)

ErrorCalledProcWithWrongTypeParam

Errore sollevato in fase di chiamata di funzione, quando uno dei parametri passati alla funzione non è compatibile con il tipo dichiarato nella firma.

Nel caso in cui sia stato fatto l'overloading della funzione (e nessuna firma è compatibile con il tipo passato) allora il typechecker considera come funzione chiamata quella con meno errori di tipo.

Esempio:

```
proc fun2(x : int) : real{
    return 1.2;
}

proc fun2(x: int, y : int) : real {
    return 2.4;
}

proc fun2(x: int, y : int, z : int) : real {
    return 3.6;
}

proc fun2(x: int, y : int, z : int, w : real) : real {
    return 4.8;
```

```
}

// main
proc main(kk:real)
{
   fun2(2.3,3);
}
```

In questo caso l'errore restituito con la chiamata `fun2(2.3,3)` è il seguente

```
[…]Parameter in position 1 must be of type Int but was found type Real on
call function fun2.
```

Come se stessi chiamando la fun2 con arietà 2.

ErrorCalledProcWrongArgs

Errore sollevato in fase di chiamata di funzione, quando il numero degli argomenti passati alla funzione differisce da quanto dichiarato nella firma.

Come nell'errore precedente, nel caso in cui sia stato fatto l'overloading della funzione (e nessuna firma è compatibile con il tipo passato) allora il typechecker considera come funzione chiamata quella con meno errori di tipo.

ErrorCalledProcWithVariable

Errore sollevato quando si tenta di chiamare una funzione/procedura utilizzando il nome di una variabile.

Esempio

```
var t = 3;

t(2);
```

ErrorNoPointerAddress

Errore sollevato quando si tenta di effettuare la deferenziazione di un qualcosa che non è un pointer.

Esempio non corretto:

```
var c = 2.2;

var p = *c;
```

Esempio corretto:

```
var k = 2.3;

var pointer_k = &k;

var new_k = *pointer_k;
```

ErrorAssignDecl

Errore sollevato quando effettuo una dichiarazione con inizializzazione utilizzando un operatore diverso dall'=

Esempio

```
var op += 4;
```

ErrorOnlyRightExpression

Errore sollevato quando nel codice viene trovato un elemento che non è una rightExpresison.

Esempio

```
if(a = a)then
{
}
```

ErrorOnlyLeftExpression

Errore sollevato quando nel codice viene trovato un elemento che non è uno statement.

Esempio

```
4;
```

Oppure

```
t + 56;
```

ErrorLeftExpression

Errore sollevato quando tento di effettuare un assegnamento ad un qualcosa che non è una left-expression.

Esempio:

```
var t = 3;
(t + 3) = 6;
```

Dichiarazioni del tipo

```
var t + 3 = 3;
```

Invece non sono permesse dalla grammatica.

ErrorOverloadingIncompatibleReturnType

Errore sollevato quando viene effettuato l'overloading di una funzione, ma i tipi di ritorno sono differenti. È infatti necessario che gli overload abbiano tutti gli stessi tipi di ritorno.

Esempio

```
proc bar(x : int) : real{
   return x + 1.2;
}


proc bar(x: real) : int {
   return 32 ;
}
```

NoDecucibleType

Errore sollevato nel caso in cui il tipo di un elemento nel codice sia già in errore. Viene sempre accompagnato da almeno un altro errore, che specifica qual'è l'errore specifico nell'elemento.

ErrorFunctionWithNotEnoughReturns

Errore creato nella seconda passata di typechecking: segnala che nella funzione esiste una possibile computazione tale che non viene chiamato nessn return. (Vale solo per le funzioni di tipo non Void)

ErrorCantUseExprInARefPassedVar

Errore sollevato quando si tenta di passare come parametro per riferimento una espressione invece che una variabile

Esempio non corretto

```
proc foo(ref x : int){
   if (x < 3) then
   {
      return;
   }
   x = x + 10;
}


proc main()
{
   var b = 5;
   foo(ref (b + 3));
}
```

Esempio corretto

```
proc foo(ref x : int){
```

```
    if (x < 3) then
    {
        return;
    }
    x = x + 10;
}


proc main()
{
    var b = 5;
    b += 3;
    foo(ref b);
}
```

ErrorCyclicDeclaration

Errore sollevato in caso di dichiarazione ciclica di variabile

Esempio

```
var cyc = 4 + cyc;
```

ErrorMissingInitialization

Errore sollevato nel caso in cui manchi l'inizializzazione obbligatoria in fase di dichiarazione di una variabile/array.

ErrorForEachIteratoArray

Errore sollevato nel caso in cui l'iteratore di un for non sia un array

Esempio

```
for a in 4 do { }
```

ErrorForEachItem

Errore sollevato nel caso in cui l'item di un for non sia una variabile

Esempio

```
for 56 in b do { }
```

oppure

```
for a + b in b do { }
```

Tac generation

Il tac generator ritorna una lista di entries che identificano l'istruzione da stampare. Vengono eseguite due passate sull'albero astratto, la prima serve a costruire le entries con i vari tipi, la seconda invece sostituisce le stringhe e genera i vari cast.

Una entry viene identificata da una etichetta e da una tipo di operazione.

- Etichetta. È usata per un salto dovuto a blocchi di iterazione, valutazione lazy dei booleani o identificazioni dei blocchi funzione
- Operazione. Le varie operazioni da poter eseguire sul tac utilizzano due oggetti, quali le etichette descritte in precedenza e le temporanee. Le operazioni sono le seguenti
    - Binaria. Vengono convolte una operazione e tre temporanee
    - Unaria. Viene coinvolta una operazione unaria e due temporanee
    - Nulla. Vengono coinvolte in un assegnamento due temporanee
    - Salto. Viene coinvolta un'etichetta
    - Condizionale vera. Viene coinvolta una temporanea e una etichetta
    - Condizionale falsa. Viene coinvolta una temporanea e una etichetta
    - Relazione. Vengono convolte una operazione, due temporanee e una etichetta
    - Indicizzazione destra. Vengono coinvolte tre temporanee
    - Indicizzazione sinistra. Vengono coinvolte tre temporanee
    - Deferenziazione destra. Vengono coinvolte due temporanee
    - Referenziazione destra. Vengono coinvolte due temporanee
    - Referenziazione sinistra. Vengono coinvolte due temporanee
    - Impostazione parametro. Viene coinvolta una temporanea
    - Chiamata procedura. Viene coinvolta una temporanea e un numero
    - Chiamata funzione. Vengono coinvolte due temporanee e un numero
    - Ritorno. Non viene coinvolto alcun valore
    - Ritorno con valore. Viene coinvolto un temporaneo
    - Nessuna operazione. Non viene coinvolto alcun valore. Usata per stampare una riga vuota o una etichetta senza nessuna istruzione
    - Cast. Vengono coinvolti due temporanee e un'operazione di cast. Usata per indicare un cast di una temporanea
    - Operazione di stringa. Indica una costante stringa statica
- Temporanee. Essa è composta da
    - Una locazione che può essere la posizione della variabile o nel caso di operazioni la locazione dell'operatore.
    - Da una modalità che indica se è identifica una variabile, un temporaneo generato da una operazione o un valore fisso quale una costane
    - Un identificativo che nel caso di variabili è il nome, nei temporanei un numero sequenziale e nei valori fissi la costante
    - Il tipo

Pretty-Printer

La struttura generata viene quindi passata ad una funzione che effettua il pretty-printing.

Alcune note:

Insieme alle variabili viene stampata anche la posizione dove essa viene dichiarata. Ad esempio: x@2,5 sta ad indicare che la variabile x che si sta utilizzando è quella dichiarata in riga 2 colonna 5.

Per i temporanei generati dal TAC viene stampata una posizione preceduta da doppia chiocciola (@@). Tale posizione rappresenta la posizione dell'operatore.

Esempio:

Il seguente codice

```
proc main()
{
    var j = 'c' + 3;
}
```

Genera il seguente TAC

```
                # code of function main
main@1,6:     cast0@@3,13 =_int cast_char_to_int 'c'
              t0@@3,17 =_int cast0@@3,13 plus_int 3
              j@3,9 =_int t0@@3,17
              return
```

In particolare, `t0@@3,17` mi sta indicando che il temporaneo t0 si riferisce all'operatore in linea 3 colonna 17 (ovverosia il '+').

Infatti, viene usato per salvare la somma `'c' + 3`.

Stato

Durante l'esecuzione del tac generation viene passato uno stato contente una tupla di valori

- Tac entries. Una lista di tac entries che verranno poi stampate a schermo
- Progressivo. Un numero che viene incrementato ad ogni creazione di un temporaneo
- L'albero di programmazione. L'albero creato dal type checker
- Etichette di supporto fallthrough. Usate per la valutazione dei booleani
- Etichetta di cache. Viene inserita e recuperata da successive istruzioni. Un tipico esempio e l'etichetta di uscita da un blocco `if then` che deve venire agganciata con le istruzioni successive.
- Tac entries funzione. Vengono inserite le entries delle funzioni che non sono dichiarate all'interno del blocco principale.

Controllo booleani

Il controllo dei booleani viene effettuato attraverso la tecnica fallthrough vista a lezione

Guardie

Per le guardie dei vari controlli quali `if then`, `if then else`, `while` e `while` do viene usata l'etichetta di cache immessa nello stato. Questa etichetta viene inserita alla fine dei vari blocchi. In seguito, ogniqualvolta viene controllato uno statement, viene recuperata l'eventuale etichetta se presente e aggiunta allo statement corrente. Nel caso in cui non sia presente viene inserita l'etichetta vuota senza alcuna operazione.

Nel caso in cui si inserisca un'etichetta dove già presente invece, viene inserita comunque un'istruzione precedente con l'etichetta da inserire con nessuna operazione

Una nota va alla guardia del `do while` che viene processata alla fine dopo aver eseguito almeno un ciclo di iterazione.
Nella creazione del tac della guardia invece viene comunque inserito all'inizio, che però viene saltata per la prima iterazione. Questo è stato necessario per facilitare l'implementazione del tac del `continue` in tale blocco.

Return

Automaticamente per le procedure viene inserito alla fine del blocco un `return`.

Array

La creazione del tac degli array avviene come descritto a lezione, unica nota è che se l'array viene dichiarato partendo da una posizione che non sia lo 0 ma da un certo offset, a ogni indice allora verrà sottratto l'offset

Stringhe

Le stringhe sono implementate similmente al linguaggio Impy. Alle costanti vengono sostituite delle temporanee che poi vengono descritte a fine del tac. La sostituzione avviene nella seconda passata del tac.

Funzione dichiarate all'interno di altri blocchi

Come descritto in precedenza tutti gli statement delle funzioni dichiarate all'interno di altri blocchi vengono inserite nello stato separati da una istruzione senza alcuna operazione. Questo simula quello visto a lezione come stream.

Break e continue

Partendo dalla premessa che ogni blocco di iterazione incomincia con una label identificata con la posizione iniziale dello stesso blocco, il break e il continue saranno semplicemente operazioni di salto verso la posizione del primo blocco `do while` / `while` che viene incontrato.

Generazione Cast & Gestione Stringhe

Il TAC viene generato in due fasi: la prima fase, quella principale, descritta sopra, si occupa di generare quasi tutto, ad eccezione delle istruzioni di cast.

Queste ultime vengono invece generate da una funzione che prende in input la struttura dati contenente TAC costruita nella prima fase. (Tale struttura è una lista)

```
tac = ...
enrichedcasttac = tacCastGenerator (getTac tac)
```

Viene praticamente eseguita una funzione su ogni istruzione TAC tramite una mapM.

```
tacCastGenerator tac = evalState (tacCastGeneratorModified tac)
startCastTacState

tacCastGeneratorModified tac = do
tacs' <- mapM tacCastGenerator' tac
[…]
```

Tale funzione ha essenzialmente il seguente tipo: TACEntry -> [TACEntry]. A partire dall'istruzione TAC in input genera una lista di istruzioni TAC. La lista risultato può avere dimensione 1 o >1 a seconda dei casi:

- La lista ha un solo elemento (significa che non c'è stato bisogno di effettuare nessun cast)
- Una lista con n>1 elementi: i primi n-1 elementi saranno istruzioni di cast, mentre l'ultimo elemento corrisponde all'istruzione TAC originaria (con eventuali nomi di variabili/tipi sostituiti)

Per capire se per una TAC entry c'è bisogno di effettuare dei cast vengono analizzate le Temp (che contengono informazione di tipo). In base ai tipi e al tipo di operazione vengono generati i cast necessari.

Le variabili che vengono introdotte per i cast vengono chiamate cast1, cast2.. per distinguerle più facilmente nel TAC dalle temporanee t0, t1. Il numero incrementale viene gestito tramite una state monad (in modo simile a quanto fatto per le temporanee t0,t1).

```
newcasttemp :: TacCastState String
newcasttemp = do
(k, _w, _t) <- get
put (k + 1, _w, _t)
return $ int2AddrCastTempName k


int2AddrCastTempName k = "cast" ++ show k
```

Tramite questa funzione vengono inoltre gestite le stringhe, aggiungendole alla sezione "static data".

La monade di stato è così definita:

```
type TacCastState = State (Int,Int,[TACEntry])
```

Il primo intero è il progressivo delle variabili `castX`

Il secondo intero è il progressivo delle variabili `ptr$str$X`

L'ultimo elemento è la lista delle TACEntry stringhe da inserire nella sezione "static data".

Codice

Il codice è strutturato in quattro locazioni:

- Una cartella in cui è contenuto il codice del type checker
- Una cartella in cui è contenuto il codice del tac generator
- Una cartella in cui sono inserite le strutture date comuni
- La cartella principale in cui viene generato il codice da BNFC, incluso l'albero di sintassi astratta

Checker

Nella cartella troviamo

- La struttura dati `BPTree` e le sue funzioni ausiliarie
- L'`ErrorPrettyPrinter` che svolge la funzione di stampa degli errori
- La struttura dati della `SupTable`
- La struttura dati della `SymbolTable`
- Il `TypeChecker` che esegue il controllo semantico statico sull'albero di sintassi astratta

ThreeAddressCode

Nella cartella troviamo

- Il `TACStateUtils` che include le funzioni per poter lavorare sullo stato
- Il `TAC` che contiene le strutture dati
- Il `TACPrettyPrinter` che svolge la funzione di stampa del TAC
- Il `TACGEnerator` che crea il tac per l'albero di sintassi astratta

Utils

Nella cartella troviamo

- L'`AbsUtils` che contine funzioni ausiliare per la manipolazione dell'albero di sintassi astratta
- `Type` che rappresenta la struttura dati per i tipi e eventuali funzioni ausiliarie

Cartella principale

Troviamo il codice generato dal BNFC e il file `ChapelParse.hs` che avvia i test.

Tools utilizzati

Per poter sviluppare questa parte di progetto si sono utilizzati i seguenti tool

Di seguito i tool utilizzati:

- Generazione iniziale: BNFC versione 2.8.3
- Lexer: Alex versione 3.2.5
- Parser: Happy versione 1.19.12
- Make: versione 4.3

Il codice finale è compilabile ed è stato testato con GHC 8.8.3.

Come compilare il codice

Per poter compilare i sorgenti eseguire il comando “make”.

Per eseguire la pulizia dei file compilati eseguire il comando “make clean”

Come eseguire il codice e i test

I 26 test creati possono essere eseguiti attraverso il comando “make demo” o singolarmente a uno a uno con il comando “ChapelParse test/test1.txt”.