Progetto di Linguaggi e Compilatori 2 parte 3

Gruppo 5

Mirko Franco 90746, Andrea Toneguzzo 90668

Risoluzione dell’esercizio:

Osservazioni:

Partiamo dalla sintassi concreta del linguaggio Oberon, deve inoltre essere possibile dichiarare funzioni procedure e variabili in qualsiasi blocco, non sarà possibile dichiarare tipi di dati particolari, saranno presenti i tipi interi, reali, caratteri e stringhe, saranno presenti i puntatori ed array.
Deve essere possibile passare i parametri con più modalità, in particolare per valore, per riferimento e per valore-risultato.

Senza fare troppi discorsi su come vengono attuati i parametri nel linguaggio originale abbiamo deciso che i parametri dichiarati nella forma

```
VAR [elencoNomiPar] : Type
```

sono passati per *valresult*;
Quelli in forma

```
elencoNomiPar : Type
```

sono passati per *valore*;
E quelli in

```
nome : POINTER TO Type
```

sono parametri per *riferimento*.

In oberon viene data la possibilità di dichiarare dei moduli con all’interno dichiarazioni di variabili, procedure nonchè un corpo che viene eseguito al caricamento del modulo. Il modulo specifica le sue dipendenze con altri moduli e definisce cosa può essere visibile dall’esterno delle sue dichiarazioni.
Per praticità abbiamo deciso di scartare tutta la parte di codice riguardante le importazioni e visibilità del modulo, lasciando solo una dichiarazione del modulo nella forma seguente:

```
MODULE nomemodulo;
  [Dichiarazioni];
BEGIN
  [corpo]
END nomemodulo .
```

Inoltre nelle dichiarazioni abbiamo tolto la possibilità di aggiungere “*” al nome della dichiarazione per renderla visibile all’esterno.

Dentro il corpo dei cicli deve essere possibile utilizzare i costrutti **break** e **continue**, l’equivalente dell’istruzione `EXIT` di oberon coincide con il break, che permette di uscire dal ciclo più interno, mentre introduciamo `CONTINUE` per interrompere l’esecuzine all’interno di un ciclo e saltare direttamente alla valutazione della guardia booleana.

Grammatica del linguaggio:

La sintassi astratta è stata generata tramite il tool bnfc, essa si può trovare nel file allegato

`AbsgOberon.hs`.
Nella cartella bnfc si può trovare il file da cui abbiamo generato i file sorgenti haskell per la generazione della grammatica astratta, relativo prettyprinter e file ausiliari.
Il file `AbsgOberon.hs` è stato a sua volta editato a mano per facilitare alcune operazioni durante il typecheck quindi si sconsiglia di sostituire il nostro file con quello generabile dal tool bnfc.

Type system:

Di seguito riportiamo le formule del type system rispetto la semantica astratta, le formule sono state leggermente semplificate dall'effettiva semantica riportata nel file `AbsgOberon.hs` specialmente per quanto riguarda la nomenclatura, resa più simile alla sintassi concreta del linguaggio e alcuni inteventi nelle liste, o alcuni tipi della semantica astratta, qui collassati nel tipo "superiore" (ad esempio per quanto riguarda i parametri, liste di statements ecc..).

$\frac{(id : A) \in \Gamma}{\Gamma \vdash_L id : A}$ env

$\Gamma \vdash INT : int$

$\Gamma \vdash REAL : real$

$\Gamma \vdash CHAR : char$

$\Gamma \vdash STRING : string$

$\Gamma \vdash TRUE : bool \qquad \Gamma \vdash FALSE : bool$

$\frac{\Gamma \vdash_E e : T \quad T \in \langle bool \rangle}{\Gamma \vdash_E NOT\ e :}$

$\frac{\Gamma \vdash_E e : T \quad T \in \langle int, real \rangle}{\Gamma \vdash_E NEG\ e : T}$

$\frac{\Gamma \vdash_E e : pointer\ to\ T}{\Gamma \vdash_E Dref\ e : T}$

$\frac{\Gamma \vdash_E e : int \quad \Gamma \vdash_E id : pointer\ to\ T}{\Gamma \vdash_E ARR\ id\ e : T}$ es. $i[e]$ $i[e,e]$

$\frac{\Gamma \vdash_E e_1 : T \quad \Gamma \vdash_E e_2 : T \quad T \in \langle int \rangle}{\Gamma \vdash_E e_1\ 'op'\ e_2 : T}$ per $op \in \langle$ 'DIVI' 'MOD' $\rangle$ solo su interi

$\frac{\Gamma \vdash_E e_1 : T_1 \quad \Gamma \vdash_E e_2 : T_2 \quad T_1, T_2 \in \langle int, real \rangle \quad T = max(T_1, T_2)}{\Gamma \vdash_E e_1\ 'op'\ e_2 : T}$ per $op \in \langle * + - \rangle$

$\frac{\Gamma \vdash_E e_1 : T_1 \quad \Gamma \vdash_E e_2 : T_2 \quad T_1, T_2 \in \langle int, real \rangle}{\Gamma \vdash_E e_1 / e_2 : Real}$

$\frac{\Gamma \vdash_E e_1 : bool \quad \Gamma \vdash_E e_2 : bool}{\Gamma \vdash_E e_1\ bop\ e_2 : bool}$ $bop \in \langle$ & (AND), OR $\rangle$

$\frac{\Gamma \vdash_E e_1 : T_1 \quad \Gamma \vdash_E e_2 : T_2 \quad T_1, T_2 \in \langle int, real, char, string \rangle \quad \exists\ max(T_1, T_2)}{\Gamma \vdash_E e_1\ rel\ e_2 : Bool}$ $rel \in \langle <= (Lte),\ >= (Gte),\ > (GT),\ < (LT) \rangle$

$\frac{\Gamma \vdash_E e_1 : T_1 \quad \Gamma \vdash_E e_2 : T_2 \quad T_1, T_2 \in \langle int, real, char, string, bool \rangle}{\Gamma \vdash_E e_1\ req\ e_2 : Bool}$ $req \in \langle =\ (Eq),\ \#\ (Neq) \rangle$

$$\frac{\Gamma \vdash_S S}{\Gamma \vdash_{CS} S}$$

$$\frac{\Gamma \vdash_S S \quad \Gamma \vdash_{CS} CS}{\Gamma \vdash_{CS} S\ CS}$$ sequenza di statem.

$$\frac{\Gamma \vdash_E e : bool \quad \Gamma \vdash_{CS} CS}{\Gamma \vdash_S \text{WHILE } e \text{ DO } CS \text{ END}}$$

$$\frac{\Gamma \vdash_E e : bool \quad \Gamma \vdash_{CS} CS}{\Gamma \vdash_S \text{REPEAT } CS \text{ UNTIL } e}$$

$$\frac{\Gamma \vdash_{CS} CS}{\Gamma \vdash_S \text{LOOP } CS}$$

$$\frac{\Gamma \vdash_E e : T \quad (\forall i\ i \in \langle 1 \dots n \rangle \Gamma \vdash_E e_i : T) \quad (\forall i \in \langle 0..n \rangle\ \Gamma \vdash_{CS} stmts_i)}{\Gamma \vdash_S \text{CASE } e \text{ OF } [e_1 : stmts_1 \dots e_n : stmts_n] \text{ ELSE } stmts_0 \text{ END}}$$

$$\frac{\Gamma \vdash_E e : bool \quad (\forall i \in \langle 1 \dots n \rangle \Gamma \vdash_E e_i : bool) \quad (\forall i \in \langle 0..n \rangle \Gamma \vdash_{CS} stmts_i)}{\Gamma \vdash_S \text{IF } e \text{ THEN } stmts\ [\text{ELSIF } e_1 \text{ THEN } stmts_1 \dots \text{ELSIF } e_n \text{ THEN } stmts_n]\ [\text{ELSE } stmts_0]}$$

$\Gamma \vdash_S$ BREAK $\qquad \Gamma \vdash_S$ CONTINUE (GOON) $\qquad \Gamma \vdash_S$ EXIT

$$\frac{\Gamma \vdash_E e_1 : T_1 \quad \Gamma \vdash_E e_2 : T_2 \quad (T_2 \leq T_1)}{\Gamma \vdash_S e_1 := e_2 : T_1}$$ * inoltre controllo che e_1 sia una lexp.

$\Gamma \vdash_S$ RETURN [e] * controllo statico per assicurarsi tipo corretto di e

Implementazione dei controlli di semantica statica

In questa sezione, descriviamo brevemente come sono stati implementati i controlli di semantica statica. L'idea di base è visitare l'albero di sintassi astratta generato dal parser, con l'ausilio di uno stato in cui vengono memorizzate ed estratte alcune informazioni, che sono:

- Enviroment locale
- Enviroment esterno
- Tipo di ritorno della funzione in che stiamo visitando
- Posizione

Gli environment sono delle liste di coppie chiave-valore, dove la chiave è il nome di un identificatore dichiarato in qualche punto del file sorgente, mentre il valore è la sua descrizione. Ogni chiave dell'Environment è univoca, questo significa che non è corretto dichiarare allo stesso livello una variabile (o costante) e una funzione con lo stesso nome. Quando la processo di visita entra in un nuovo scope, gli Environment vengono fusi insieme, causando possibili effetti di shadowing.

L'elemento posizione viene utilizzato dal sistema di reporting degli errori, quindi il processo di visita ha il compito di tenere questa informazione aggiornata.

Per tenere traccia degli errori, allo stato viene applicata una lista, dove vengono inseriti gli errori semantici. Questa associazione di stato e lista di errori è descritta da una monade, che effettua la concatenazione dei messaggi di errore.

Le funzioni più consistenti di questo stage sono quelle che visitano le espressioni:

- `guessType/2`
- `isConst/1`
- `checkType/3`

La funzione `guessType/2` viene usata per "prevedere" il tipo di una espressione, senza visitare l'intero albero ma solo i nodi superficiali, leggendo l'eventuale operatore della costante, e sfruttando l'environment.
La funzione `isConst/1` viene usata per capire se il valore di una espressione può essere determinato al tempo di compilazione.
La funzione `checkType/3` controlla induttivamente sulla complessità dell'espressione, se il tipo atteso corrisponde, sfruttando le informazioni salvate negli Environment.

Codifica del three address code

Per generare la traduzione del codice in codice intermedio three address code, per prima cosa codifichiamo le istruzioni del three address code in un data-type in haskell in modo da non lavorare direttamente su delle stringhe ma su una struttura dati definita.
Tale struttura può essere trovata nel modulo `Tac` nell'omonimo file sorgente haskell.

Nel modulo vengono definite le istruzioni, divise in:

- operazioni "nullarie", "unarie", "binarie", le quali possono avere diversi tipi di operatori;
- i salti condizionati e incondizionati, nonchè relazionali;
- caricamento dei parametri;
- chiamata di funzioni e procedure;

Le istruzioni possono avere degli operatori aritmetici, logici e relazioni. Vengono definiti gli indirizzi, quali registri temporanei, oppure nomi di variabili, ed è previsto l'assegnamento tramite puntatori e vettori.

Sono definite le etichette per il controllo del flusso.

Vengono definiti i metodi `show` per la stampa del codice relativo alle istruzioni in modo da poter visualizzare l'effettivo three address code.

Generazione del TAC

Purtroppo questa parte non è funzionante, riportiamo un'abbozzo di progettazione del suo funzionamento.
L'idea del modulo riguardante la generazione del codice consiste in una funzione che dato in input l'albero di parsing fornisca la corrispettiva codifica del programma in TAC. Per ottenere questo la funzione deve visitare l'albero e assegnare ad ogni elemento della grammatica astratta, statement, dichiarazioni ed espressioni il relativo tac.
Le informazioni necessarie per procedere nella traduzione sono le seguenti:

- registri temporali attualmente in uso, che non andranno sovrascritti.
- label generate durante la traduzione, le etichette devono essere univoche.
- una symble table contenente i simboli che identificano funzioni, aree di memoria, registri dichiarati, il loro tipo (dimensione in byte).

Osservando inoltre più nello specifico il metodo di traduzione in TAC è importante tener conto per ogni nodo di semantica astratta:

- nel caso di operazioni un possibile registro temporaneo o area di memoria in cui copiare il risultato.
- nel caso del controllo del flusso eventuali etichette alle quali saltare in determinate condizioni, generalmente una etichetta next alla quale saltare, generalmente alla fine dell'esecuzione di una serie di statements oppure uscire da un ciclo o all'invocazione di

un break o continue.

Implementazione

Purtroppo non si è riuscito a sviluppare la parte relativa alla traduzione in TAC. Alleghiamo comunque degli abbozzi di moduli parzialmente impostati, ma non testati a fondo, in particolare i file:
Abs2Tac.hs il quale dovrebbe iniziare la traduzione chiamando la funzione principale, A2TAusiliarie.hs nel quale si trovano alcune funzioni ausiliarie per getire la generazione di etichette e utilizzo dei registri temporanei, Exp2Tac.hs nel quale traduce il codice delle espressioni.

Makefile

Abbiamo fondamentalmente due target, uno per ottenere il programma per eseguire il parsing e visualizza l'albero, l'altro che oltre a fare il parsing esegue i controlli statici e di tipo.

ISTRUZIONI:

Da linea di comando, spostarsi nella directory radice e digitare:

```
$>make parser
```

e

```
$>make checker
```

Per generare rispettivamente i file eseguibili necessari.

Per testare il programma di parsing:

```
$>./TestParser testfile.obe
```

Per testare il programma di checking:

```
$>./TestCheck testfile.obe
```

```
Si consiglia di lanciare:
```

```
make clean
```

per eliminare tutti i file utilizzati nel processo di compilazione.

Nella cartella demo sono presenti alcuni file per testare il programma e osservare i vari comportamenti e segnalazione degli errori rispetto il codice dato in input.