DOCUMENTAZIONE TOY2C

Link Repository: https://gitlab.com/compilatori-a.a.-2020_21/toylang-semancgen-es5_mpp/dandrea-spinelli_es5_scg.git

Gianluca Spinelli matr.: 0522500981

Davide D'Andrea matr.: 0512105536

Analisi Lessicale e Sintattica

Il primo modulo progettuale è stato quello di generare un lexer, usando JFLEX, in grado di trasformare un codice sorgente toy in uno stream di token.

-JFlex generator: C:\JFLEX\bin\jflex -d src srcjflexcup\toy.flex

Successivamente, si è passati alla generazione di un analizzatore sintattico, il quale ha il compito di validare sintatticamente il flusso di token generato dal lexer.

Questo modulo è stato generato attraverso Java CUP. La grammatica S-Attribuita inserita all'interno di CUP non è stata modificata e i conflitti sono stati risolti attraverso l'ausilio delle precedenze.

-CUP generator: java -jar C:\CUP\java-cup-11b.jar -dump -destdir src srcjflexcup/toy.cup 2> dumpfile.txt

SINTAX VISITOR:

È stato realizzato un primo visitor sintattico per la generazione dell'AST stampato in un file XML.

Il file XML è stato generato con l'ausilio delle librerie;

org.w3c.dom.Document;

org.w3c.dom.Element;

javax.xml;

L'albero sintattico generato viene scritto in un file "output.xml".

SEMANTIC VISITOR:

È stato realizzato un secondo visitor atto a effettuare i controlli semantici.

Tipi di base verificati

$\Gamma \vdash null : Null$

$\Gamma \vdash true : Boolean$

$\Gamma \vdash false : Boolean$

$\Gamma \vdash int : Integer$

$\Gamma \vdash float : Float$

$\Gamma \vdash string : String$

REGOLE DI INFERENZA:

Controllo di tipo per l'ID

$$\frac{(x : \tau)}{\in \Gamma \Gamma \vdash x : \tau}$$

Controllo di tipo per l'inizializzazione per l'ID

$$\frac{\Gamma \vdash x : \tau \; AND \; expr : \tau' AND \; \text{controllaCompatTipi}(\tau, \tau') : \tau}{\Gamma \vdash (x := expr) : \tau}$$

Controllo di tipo per l'operazione unaria

$$\frac{\Gamma \vdash arg : \tau_1 \; AND \; getTypeSingle(op, \tau_1) = \tau_1}{\Gamma \vdash (op \; arg) : \tau_1}$$

Controllo di tipo per le operazioni binarie

$$\frac{\Gamma \vdash arg_1 : \tau_1 \; AND \; \Gamma \vdash arg_2 : \tau_2 \; AND \; getTypeOperations(op, \tau_1, \tau_2) = \tau}{\Gamma \vdash (arg_1 \; op \; arg_2) : \tau}$$

Controlli per gli statement

Alcuni attributi sono caratterizzati dal **canNull**, il quale sta a specificare che esso può annullarsi, ergo risulta un attributo opzionale e la sua assenza non implica la scorrettezza della regola semantica.

Con […] intendiamo gli attributi opzionali; con {…} intendiamo una lista.

If statement semantic rule

$$\frac{\Gamma \vdash c\text{: } Boolean\ AND\ \Gamma \vdash if\,\text{Lista}Stat\ AND\ \Gamma \vdash elif List\ \boldsymbol{canNull}\ AND\ \Gamma \vdash \text{E}lse}{\boldsymbol{canNull}\ \ \Gamma \vdash if\ c\ then\ if\,\text{Lista}Stat\ [elif List]\ [\text{Else}]\ \ fi}$$

Elif statement semantic rule

$$\frac{\Gamma \vdash c\text{: } Boolean\ AND\ \Gamma \vdash \text{StatList}}{\Gamma \vdash elif\ c\ then\ \text{StatList}}$$

While statement semantic rule

$$\frac{\Gamma \vdash \text{ListaStat1}\ \ \boldsymbol{canNull}\ AND\ \Gamma \vdash cond\text{: } Boolean\ AND\ \Gamma \vdash \text{ListaStat2}}{\Gamma \vdash while\ [\text{ListaStat1}] \rightarrow cond\ do\ \text{ListaStat2}\ od}$$

Readln statement semantic rule

$$\frac{\Gamma \vdash idList\text{: } \{\tau_1, \dots, \tau_n\}}{\Gamma \vdash readln(idList)}$$

Write statement semantic rule

$$\frac{\Gamma \vdash exprList\text{: } \{\tau_1, \dots, \tau_n\}}{\Gamma \vdash write(exprList)}$$

Assign statement semantic rule

$$\frac{\Gamma \vdash idList\text{: } \{\tau_1, \dots, \tau_n\}\ AND\ \Gamma \vdash exprList\text{: } \{\tau_1, \dots, \tau_n\}}{\Gamma \vdash idList := exprList}$$

Call procedure statement semantic rule

$$\frac{\Gamma \vdash id\text{: } \{\tau_1, \dots, \tau_n\}\ AND\ \Gamma \vdash exprList\text{: } \{\tau_1, \dots, \tau_n\}\ \boldsymbol{canNull}}{\Gamma \vdash id([exprList])}$$

Le liste per espressioni, inizializzazioni, restituzioni, variabili e procedure sono date vere se e solo se i controlli di tipo per gli elementi interni risultano essere corretti.

$getTypeOperations\ (op, \tau_1, \tau_2)$			
op	t1	t2	result
+ - * /	Integer	Integer	Integer
+ - * /	Integer	Float	Float
+ - * /	Float	Integer	Float
+ - * /	Float	Float	Float
+	String	String	String
= < > <= >= <>	Boolean	Boolean	Boolean
= < > <= >= <>	Integer	Integer	Boolean
= < > <= >= <>	Integer	Float	Boolean
= < > <= >= <>	Float	Integer	Boolean
= < > <= >= <>	Float	Float	Boolean
= < > <= >= <>	String	String	Boolean
AND OR	Boolean	Boolean	Boolean

Compatibilità di tipi (per assegnazioni, passaggio di parametri per Callproc() e tipi di ritorno):

controllaCompatibilità (t1, t2)		
t1	**t2**	**result**
Integer	Integer	true
Float	Float	true
Float	Integer	true
Integer	Float	true
String	String	True
Boolean	Boolean	True

Altro riguardo l'analisi semantica

Per la validità del programma, è stato inserito anche un controllo per verificare la presenza della funzione main e che assicuri che sia unica, proprio come ogni altra funzione. Il main si differisce dalle altre funzioni per il fatto che è dichiarabile ovunque nel programma, a prescindere dalle dipendenze delle altre funzioni.
Inoltre, non avendo limitazioni dettate dalla traccia circa il tipo di ritorno del main e dei suoi parametri di ingresso, si è deciso di limitare questi aspetti al singolo tipo void e al non utilizzo dei parametri in ingresso, in modo tale da evitare ambiguità dell'utente con interazioni non effettuabili se non con strumenti diversi rispetto al semplice editing su file di testo, al fine di creare un eseguibile toy funzionante.

Infine, nella tabella dei tipi inerente alle operazioni binarie è stata specificata un'operazione in cui vengono addizionate due variabili di tipo String: con tale operazione si intende la concatenazione tra stringhe, la quale restituisce, ovviamente, una nuova variabile di tipo String.

GENERAZIONE CODICE INTERMEDIO:

Di seguito vengono riportati tutti gli accorgimenti implementativi avuti, al fine tradurre il sorgente toy in codice C (codice intermedio target, stabilito dalla traccia).

- Introdotte le librerie stdio.h, stdlib.h, stdbool.h e string.h per il corretto funzionamento del codice che verrà generato;
- Al fine di implementare i ritorni multipli, si è deciso di utilizzare il concetto di struttura wrapper per racchiudere in un unico record tutti gli attributi di ritorno di una funzione. A tal punto, per ogni funzione presente nel sorgente diversa dal main e con tipo di ritorno diverso da void, viene generata una struttura contenente come parametri tutti i tipi di ritorno dichiarati nel sorgente Toy.
- Il tipo stringa viene implementato tramite un array di chat con dimensione oari a 512;
- Al fine di effettuare una corretta concatenazione di stringhe, in ogni sorgente C generato verrà iniettata una funzione di servizio che, tramite l'uso di malloc e concat (funzione di string.h), permetterà di effettuare tale operazione correttamente;
- I confronti di tipo booleano tra stringe vengono tradotti in C tramite l'ausilio della funzione di libreria strcmp della libreria string.h;
- Il linguaggio C non supporta nativamente i booleani; sono stati introdotti con una libreria, ma il fatto che non vengano supportati nativamente ha creato qualche problema con la funzione scanf: siccome non permette la gestione dei booleani, è stato necessario permettere la generazione di una variabile intera temporanea ogni qualvolta che si utilizzi il booleano in tale funzione. Il valore della temporanea equivarrà al rispettivo valore booleano;