Regole semantiche: esercitazione 5 - analisi semantica e generazione del codice C

Abate Francesco, Carmine Ferrara

Analisi Lessicale e Sintattica

Il primo modulo progettuale, è stato quello di generare un modulo lessicale in grado di trasformare un sorgente toy definito sulla base delle specifiche del linguaggio in un token stream. Tale modulo è stato generato tramite il generatore JFLEX, al quale è stata data in input una specifica lessicale dotata di espressioni regolari e stati, formulati sulla base delle specifiche lessicali stabilite dalla traccia.

In seguito si è passati alla generazione di un analizzatore sintattico, il quale ha il compito di validare sintatticamente il flusso di token, precedentemente generato, e di generare un albero sintattico corrispondente al sorgente toy. Questo modulo è stato generato a sua volta tramite l'ausilio del tool Java Cup, a cui è stato dato in pasto un sorgente .cup nel quale è stata definita una grammatica di tipo S-Attribuito (definita sulla base della grammatica dettata dalla traccia). Il modulo parser è stato poi affiancato da un primo visitor, il quale permette di avere una visione XML, dell'AST generato dal parser dopo l'elaborazione sullo stream di token.

Le specifiche JFLEX e CUP, sono state definite sulla base dei vincoli di progettazione definiti dalle specifiche fornite, sono stati presi dovuti accorgimenti soltanto per permettere la generazione corretta del modulo sintattico, altrimenti impedita data la presenza di ambiguità nella grammatica fornita nelle specifiche Toy.

Un riferimento alle specifiche grammaticali e sintattiche viene lasciato a corredo di questo documento.

Analisi Semantica

In questo paragrafo sono riportati tutti i controlli semantici sotto forma di regole di inferenza, implementate nel visitor semantico del progetto.

Tipi di base verificati

$\Gamma \vdash null : Null$
$\Gamma \vdash true : Boolean$
$\Gamma \vdash false : Boolean$
$\Gamma \vdash int : Integer$
$\Gamma \vdash float : Float$
$\Gamma \vdash string : String$

Controllo di tipo per l'ID

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

Controllo di tipo per l'inizializzazione per l'ID

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

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

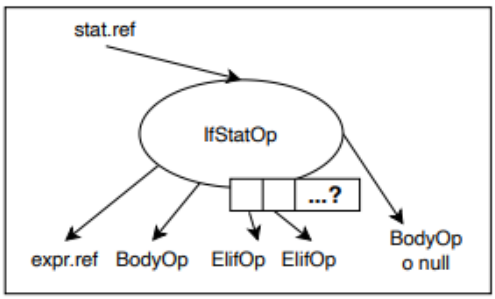
$$\frac{\Gamma \vdash arg_1 : \tau_1 \; AND \; \Gamma \vdash arg_2 : \tau_2 \; AND \; getTypeOperations(op, \tau_1, \tau_2) = \tau}{\Gamma \vdash (arg_1 \; op \; arg_2) : \tau}$$

I controlli per le operazioni booleane e binarie sono include in getTypeOperations, anche se nell'implementazione differiscono da tale funzione. Non sono stati riportati siccome, appunto, sarebbero stati ridondanti; più in generale, le funzioni getType non includono l'operazione durante l'implementazione in quanto la discriminazione iniziale riguardante l'operazione viene effettuata nel metodo di visita dell'espressione.

Controlli per gli statement

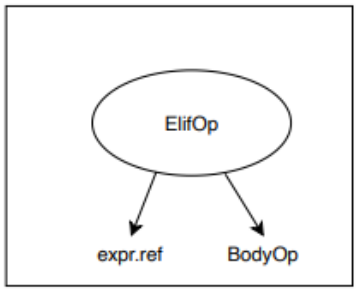
Alcuni attributi sono caratterizzati dal canNull, il quale sta a specificare che esso può annullarsi, ergo risulta un attributo opzionale e la sua assenza non implica la scorrettezza della regola semantica.
Con […] intendiamo gli attributi opzionali; con {…} intendiamo una lista.

If statement semantic rule



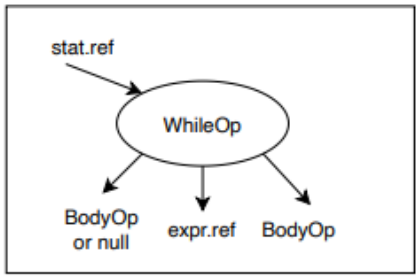
$$\frac{\Gamma \vdash c : Boolean \; AND \; \Gamma \vdash ifBodyStat \; AND \; \Gamma \vdash elifList \; \boldsymbol{canNull} \; AND \; \Gamma \vdash elseBody \; \boldsymbol{canNull}}{\Gamma \vdash if \; c \; then \; ifBodyStat \; [elifList] \; [elseBody] \; fi}$$

Elif statement semantic rule



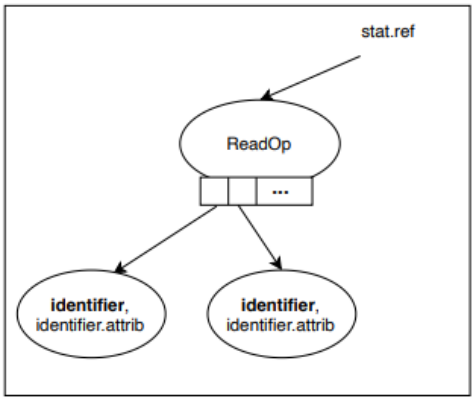
$$\frac{\Gamma \vdash c: Boolean\ AND\ \Gamma \vdash elifBody}{\Gamma \vdash elif\ c\ then\ elifBody}$$

While statement semantic rule



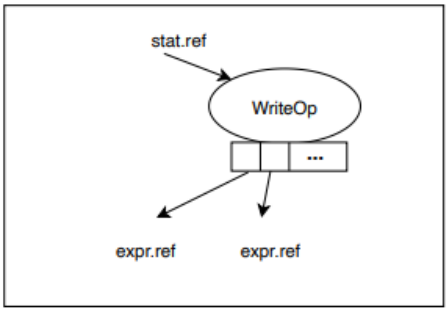
$$\frac{\Gamma \vdash preCondStat\ \boldsymbol{canNull}\ AND\ \Gamma \vdash cond: Boolean\ AND\ \Gamma \vdash whileBodyStat}{\Gamma \vdash while\ [preCondStat] \rightarrow cond\ do\ whileBodyStat\ od}$$

Readln statement semantic rule



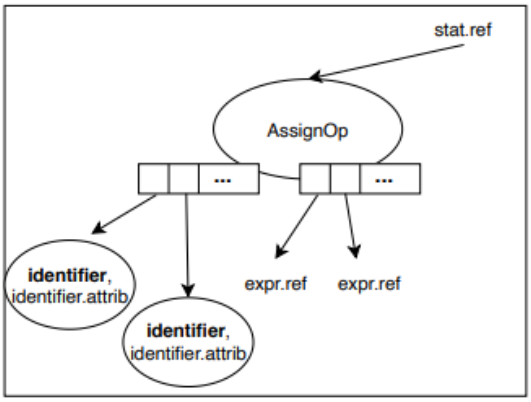
$$\frac{\Gamma \vdash idList: \{\tau_1, \ldots, \tau_n\}}{\Gamma \vdash readln(idList)}$$

Write statement semantic rule



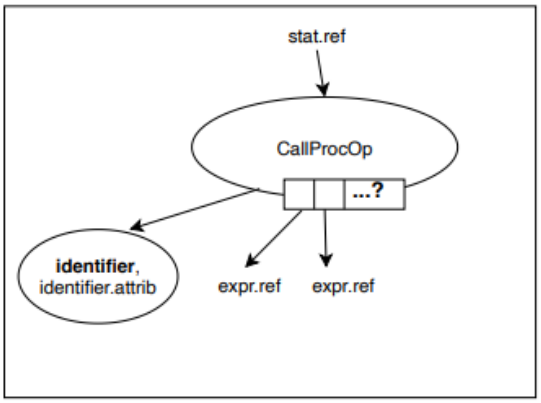
$$\frac{\Gamma \vdash exprList: \{\tau_1, \ldots, \tau_n\}}{\Gamma \vdash write(exprList)}$$

Assign statement semantic rule



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

Call procedure statement semantic rule



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

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.

Tabelle dei tipi

$getTypeOperations(op, \tau_1, \tau_2)$			
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

$getTypeSingle(op, \tau_1)$		
op	t1	result
UMINUS	Integer	Integer
UMINUS	Float	Float
NOT	Boolean	Boolean

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 del 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 indentare correttamente il codice e al fine di non rendere troppo complicata la generazione del codice intermedio rilegata all'indentazione, ad ogni esecuzione del generatore viene eseguito anche un plugin chiamato "Artistic Style" (http://astyle.sourceforge.net/). Tale plugin viene richiamato tramite la funzione exec utilizzando lo stile di Google;

- 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. Per ogni livello di scope viene recuperata la tabella dei simboli associata ad ogni scope, in modo tale da effettuare le corrette deduzioni per ogni traduzione direttamente influenzate dalle informazioni derivanti dall'analisi semantica precedentemente elaborata;

- Il linguaggio C non supporta, nativamente, i tipi booleani, quindi per la traduzione di tale tipo è stato utilizzato il tipo bool offerto dalla libreria stdbool.h. Ci sono volte, però, in cui l'utilizzo del tipo booleano non è ben accetto (ad esempio, la printf non ammette tipi bool in input e non sa come trattarli), quindi viene generata una variabile intermedia intera che assumerà un valore dipendente dalla variabile booleana che si sta trattando (0 se false, altrimenti true);

- Il tipo stringa, invece, viene implementata tramite un array di caratteri di lunghezza prefissata pari attualmente 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;

- Per ogni funzione diversa dal main e con tipo di ritorno diverso da void si specifica un ritorno del tipo struttura definito precedentemente e, all'operazione di restituzione, verrà istanziata una struttura che conterrà tutte le espressioni da restituire definite nel sorgente Toy;

- Se durante un'assegnazione o durante un passaggio di parametri si fa uso di un richiamo a funzione (un esempio palese sono i parametri passati in input alla funzione printf), vengono dati in input tutti i parametri della struttura iterando su di essa;

- 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;

- Gli statement che precedono una condizione in un sorgente toy, vengono inseriti due volte nel sorgente C, prima del blocco while e all'interno, subito al di sotto del blocco operativo, in modo tale da emulare lo stesso comportamento del sorgente in Toy;

- La traduzione in C delle istruzioni Readln e Write, viene effettuata in C tramite l'ausilio delle funzioni printf e scanf della libreria stdio.h, facendo attenzione anche in questo caso al richiamo di funzioni, qualora il risultato della funzione debba essere stampato in output, si fa in modo di accumulare i risultati in una struttura temporanea (dello stesso tipo di ritorno della funzione), che verà scompattata all'interno della funzione printf, in modo tale da mantenere lo stesso ordine di stampa stabilito nel sorgente toy;

- Una volta effettuata la traduzione in codice C, il codice verrà compilato a sua volta (grazie all'ausilio del compilatore CLang) ed eseguito (ciò è stato implementato solo per sistemi operativi con kernel Linux o Windows). Ciò è stato possibile grazie all'introduzione di un piccolo script esterno (compile.sh su linux, vincolato all'utilizzo del terminale XTerm, e Generate.bat su windows, tramite l'ausilio del canonico terminale di sistema), richiamato tramite la funzione exec, il quale permette di eseguire il codice C generato. Per la corretta esecuzione su Linux, quindi, sarebbe necessario scaricare e installare XTerm nel

caso non fosse presente sul dispositivo; per la corretta esecuzione su Windows, invece, è necessario creare un collegamento al prompt dei comandi (cmd) nella cartella testfiles/csources;