PROGETTO DI
LINGUAGGI E COMPILATORI 2

Gruppo 3 - Parte 3

Studente:	Matricola:
Elia Calligaris	114039
Luca Geatti	113964
Federico Igne	114081
Veronica Minati	109908

mail: calligaris.elia@spes.uniud.com
mail: geatti.luca@spes.uniud.com
mail: ignefederico@gmail.com
mail: veronicaminati33@gmail.com

Anno Accademico 2015-2016

Indice

1 Introduzione

La seguente relazione è stata redatta contestualmente allo svolgimento della terza parte del progetto di Linguaggi e Compilatori 2 durante l'anno accademico 2015/2016. Per lo svolgimento del progetto e la stesura di tale documento sono state seguite le specifiche fornite dal Prof. Marco Comini; per ulteriori informazioni si veda il testo integrale della consegna.

Tutti i file sorgenti necessari verranno forniti assieme a questa relazione, con relativi *Makefile* per la compilazione.

2 Makefiles e funzioni di test

I sorgenti sono compilabili, tramite il `Makefile` fornito, con il comando `make`. Vengono inoltre fornite diverse demo con l'obiettivo di mostrare alcune features principali del progetto.

Vengono rese disponibili le seguenti demo, accessibili attraverso il comando `make <nome demo>`:

1. *demoSimple* e *demoFunTest*: semplici esempi di codice;
2. *demoSpigot*: esempio di test (leggermente modificato) trovato sulla pagina di Zeno;
3. *demoDet*: esempio di test (leggermente modificato) trovato sulla pagina di Zeno;
4. *demoQuickSort*: esempio di test (leggermente modificato) trovato sulla pagina di Zeno.

Inoltre i riferimenti ad alcuni *test case* sono dati nel corpo della relazione.

Inoltre nei vari esempi si è cercato di utilizzare l'intera gamma di costrutti messi a disposizione dal linguaggio, oltre a mettere in luce alcuni aspetti della sintassi come le modalità di passaggio dei parametri, l'utilizzo dei puntatori, e la definizione di funzioni a diversi livelli di visibilità.

3 Specifiche

Specifiche del linguaggio

- Possibilità di dichiarare funzioni e/o variabili in ogni blocco
  - le funzioni hanno visibilità indipendente dalla loro posizione all'interno del blocco, rendendo possibile la (mutua) ricorsione;
  - le variabili vengono inserite nelle scope secondo l'ordine di dichiarazione
- Sono state incluse le funzioni predefinite read/write richieste.
- Sono previste le modalità di passaggio richieste (value, reference, value-result, constant), con relativi controlli di semantica statica.
- E' previsto un costrutto di try-catch, la cui sintassi concreta è `try [Statements] catch [Statements] end try`.
- Sono previsti i seguenti comandi per il controllo di sequenza:
  - Condizionali: `if`
  - Iterazione indeterminata: `while`, `repeat-until`
  - Salto: `goto(label)`
  - Iterazione determinata: `for`
- Per semplicità non sono previsti tipi definiti dall'utente.

Specifiche del type system

- Il type system implementato non prevede cast espliciti o impliciti.
- Il tipo del valore assegnato ad una variabile deve corrispondere al tipo della stessa.
- Il tipo ritornato da una funzione deve corrispondere a quello dichiarato nella segnatura.
- Ogni possibile branch all'interno di una funzione deve contenere un return (eccetto in presenza di istruzioni di salto che interrompono il normale flusso di esecuzione).
- `Break` (`exit`) e `continue` devono venir utilizzati soltanto all'interno di un loop (determinato o indeterminato).
- Numero, tipo e modalità dei parametri attuali deve corrispondere a quanto dichiarato nella segnatura di funzioni/procedure.
- L'indice di iterazione del costrutto di iterazione determinata (`for`) non può essere modificato esplicitamente all'interno di loop.
- Viene proibita la modifica delle costanti.
- Viene controllata la presenza delle variabili utilizzate nello scope corrente.
- La procedura `program` deve essere dichiarata (al più) una volta sola all'interno di ogni programma.

Specifiche del TAC generato

- Le guardie booleane prevedono short-cut con la tecnica *fall-through*. Per semplificare il codice si è scelto di non avere le short-cut in altri contesti.
- Si è ritenuto naturale valutare espressioni ed argomenti da sinistra a destra.
- Sono stati predisposti comandi di try-catch per produrre blocchi di codice protetto.
- Per motivi implementativi gli operandi delle *right expressions* sono salvati in variabili fresh, senza preoccuparsi dell'ottimizzazione e dello spazio occupato.

Soluzione realizzata

- Lexer e parser sono implementati nei files `Lexer.x` e `Parser.y`.
- La sintassi astratta è fornita nel file `AbstractSyntaxTree.hs`.
- Il pretty printer è implementato nel file `PrettyPrinter.hs`.
- Il type checker è implementato nel file `TypeChecker.hs`

La generazione del Three Address Code (TAC) è stata impostata nel seguente modo:

- Viene fornita una sintassi astratta per il TAC nel file `TacAST.hs`
- Nel file `TacGen.hs` viene fornito il codice che produce un albero di sintassi astratta per il TAC a partire da uno per il linguaggio
- Il file `TacPrettyPrinter.hs` fornisce il pretty printing del TAC.

4 Parser

4.1 Conflitti Shift Reduce

Il parser presenta

- 0 conflitti Reduce/Reduce
- 5 conflitti Shift/Reduce

Per analizzare uno ad uno questi ultimi conflitti si è generato il file "outp.txt" tramite la direttiva *happy –info=outp.txt Parser.y*.

1. Il primo conflitto si trova nello stato 18 dell'automa del parser. Gli *item* di questo stato sono
   - Decl → 'procedure' Id . FormalIn ListDecl ListStmt 'end' 'procedure'
   - Decl → 'procedure' Id . ListDecl ListStmt 'end' 'procedure'

   ovvero: il parser ha riconosciuto fino a *'procedure' Id* ed ora sta leggendo il terminale '('. In tal caso può fare due operazioni:
   - Reduce 39: sceglie di ridurre con la regola $Decl \rightarrow \epsilon$; in questo caso il parser vede una dichiarazione di procedura senza parametri in cui la prima statement ha una left-expression tra parentesi tonde.
   - Shift: il parser ha riconosciuto il nome di una procedura ed avanza riconoscendo i suoi parametri

   In caso di conflitto Shift/Reduce, Happy sceglie sempre di eseguire lo Shift. Un codice del genere è quindi vietato:

   ```
   procedure withoutParam
       (x) := 1
   end procedure
   ```

   Si vuole dichiarare una procedura *withoutParam* senza parametri, ma il parser riconosce la left-expression della prima statement come l'insieme dei parametri della procedura. Dato che non c'è nessuna direttiva per specificare il tipo del parametro x, questo risulterà in un errore. Si può verificare questo comportamento eseguendo "make shift1".

2. Il secondo conflitto si trova nello stato 30 dell'automa. Gli item importanti al fine di capire il conflitto sono:

   - SimpleStmt → Id . '(' sep0(RExpr,',') ')'
   - SimpleStmt → Id .

   Il parser ha riconosciuto fino ad *Id* ed ora sta leggendo una '('. Le operazioni possibili sono:

   - Reduce 79: sceglie di applicare la regola $SimpleStmt \rightarrow Id$. In altre parole, il parser riconosce la chiamata di una procedura che non ha argomenti e le parentesi le associa ad una left-expression appartenente alla statement successiva.
   - Shift: la parantesi viene vista come l'inizio delle dichiarazioni dei parametri della procedura.

   Come prima, l'azione Shift è quella che viene in realtà eseguita. Questo comportamento non crea grandi limitazioni; un caso in cui il programmatore potrebbe cadere in errore è il seguente:

```
program
    var a:int := 1
    withoutArgs
    (a) := b
end program

procedure withoutArgs
    x:=2
end procedure
```

   Nella quarta riga il parser riconosce (a) come parametro per la procedura *withoutArgs*. In questo modo però la statement successiva incomincia con ':=', causando un errore di parsing. Si noti che questo, assieme al precedente, sono i comportamenti assunti dal compilatore ufficiale di Zeno.

3. Il terzo conflitto si trova nello stato 33. Gli item presenti sono:

   - LExpr → LExpr1 .
   - LExpr2 → LExpr1 . '[' sep1(RExpr,',') ']'

   Il parser ha riconosciuto una $LExpr1$ ed ora legge una '['. Le operazioni possibili sono:

   - Reduce 31: si esegue la regola $LExpr \rightarrow LExpr1$. Il parser riconosce una left-expression e vede la '[' come l'inizio della dichiarazione di un array.
   - Shift: il parser continua leggere la '[' e riconosce un accesso agli array del tipo $a[i]$.

   Viene eseguita l'azione Shift; tale comportamento è quello corretto ai fini di un parsing consistente con la sintassi del linguaggio.

4. Il quarto conflitto si trova nello stato 60. Gli item presenti sono:

- RExpr → Id . '(' sep0(RExpr,',') ')'
- LExpr2 → Id .

Questo è l'equivalente al secondo conflitto, specializzato per una right-expression. Non crea grandi limitazioni per il programmatore; un esempio in cui si potrebbe incontrare questo errore è il seguente:

```
program
    var a:int := 1
    var b:int := withoutArgs
    (a) := b
end program

procedure withoutArgs:int
    x:=2
    return x
end procedure
```

Il comportamento del parser è analogo a quello del caso 2 e lo si può testare con il comando "make shift4".

5. Il quinto conflitto si trova nello stato 169. Gli item presenti sono:

   - RExpr → LExpr .
   - LExpr1 → '(' LExpr . ')'

   Il parser ha riconosciuto una left-expression ed ora sta leggendo ')'. Le operazioni possibili sono:

   - Reduce 28: si riduce usando la regola $RExpr \rightarrow LExpr$. Il parser riconosce la left-expression come una right-expression e dopo vede l'intera espressione ($RExpr$) come una left-expression, riducendola in seguito ad una right-expression.
   - Shift: legge prima la ')' e poi riduce l'intera espression ($LExpr$) ad una right-expression.

   Le due operazioni sono equivalenti, infatti entrambe portano ad una right-expression.

5 Type System

Definiamo ora il *type system* per il linguaggio implementato dal nostro parser. Uno degli aspetti su cui si è posta maggiore attenzione è quello di definire un type system il più possibile vicino a quello del linguaggio Zeno, integrando dove necessario le ulteriori specifiche presenti nella consegna del progetto. Nella seguente definizione si è scelto di operate alcune semplificazioni sintattiche per rendere più fluida la lettura della stessa.

Di seguito indicheremo con σ, σ', σ'' gli environment contenenti i tipi di variabili, costanti, label e funzioni. Indicheremo con $\sigma[\tau/x]$ l'environment σ ampliato con la dichiarazione di x di tipo τ.

5.1 Types

Basic types

$$\frac{}{\vdash_T \tau}$$

Per un qualsiasi $\tau \in \{\texttt{int}, \texttt{boolean}, \texttt{real}, \texttt{char}, \texttt{string}\}$

Composite types

$$\frac{\vdash_T \tau \qquad n_1, \ldots, n_k \in \mathbb{N}}{\vdash_T \texttt{array[}n_1\texttt{,} \ldots \texttt{,}n_k\texttt{]of}\ \tau} \qquad \text{(Array)}$$

$$\frac{\vdash_T \tau}{\vdash_T \tau\texttt{@}} \qquad \text{(Pointer)}$$

5.2 Declarations

Variable declaration

$$\frac{x \notin \sigma \qquad \vdash_T \tau}{\langle \sigma, \texttt{const}\ x\texttt{:}\tau \rangle \to \sigma[\tau/x]}$$

$$\frac{x \notin \sigma \qquad \vdash_T \tau \qquad \sigma \vdash_{RE} r : \tau}{\langle \sigma, \texttt{const}\ x\texttt{:}\tau\ \texttt{:=}\ r \rangle \to \sigma[\tau/x]}$$

$$\frac{x_1, \ldots, x_n \notin \sigma \qquad \vdash_T \tau}{\langle \sigma, \texttt{var}\ x_1\texttt{,} \ldots \texttt{,}x_n\texttt{:}\tau \rangle \to \sigma[\tau/x_1, \ldots, \tau/x_n]}$$

$$\frac{x_1, \ldots, x_n \notin \sigma \qquad \vdash_T \tau \qquad \sigma \vdash_{RE} r : \tau}{\langle \sigma, \texttt{var}\ x_1\texttt{,} \ldots \texttt{,}x_n\texttt{:}\tau\ \texttt{:=}\ r \rangle \to \sigma[\tau/x_1, \ldots, \tau/x_n]}$$

Label declaration

$$\frac{l \notin \sigma}{\langle \sigma, \texttt{label}\ l\texttt{:} \rangle \to \sigma[\ l\ ]}$$

Function declaration

$$\frac{\begin{array}{c}\vdash_T \tau \qquad \vdash_T \tau_1 \qquad \ldots \qquad \vdash_T \tau_n \qquad m_1, \ldots, m_n \in \{\texttt{val}, \texttt{var}, \texttt{valres}, \texttt{const}\} \\ \langle \sigma\left[(\tau_1 \times \cdots \times \tau_n) \to \tau/f, \tau_1/x_1, \ldots, \tau_n/x_n\right], ds \rangle \to \sigma' \qquad \sigma' \vdash_S ss\end{array}}{\langle \sigma, \texttt{function}\ f\texttt{(}m_1\ x_1\texttt{:}\tau_1\texttt{,} \ldots \texttt{,}m_n\ x_n\texttt{:}\tau_n\texttt{):}\tau\ ds\ ss\ \texttt{end function} \rangle \to \sigma\left[(\tau_1 \times \cdots \times \tau_n) \to \tau/f\right]}$$

Declaration list Le liste di dichiarazioni sono trattate in modo particolare per permettere di avere la mutua ricorsione fra funzioni, ma evitare cicli nell'inizializzazione delle variabili; in questo senso vengono prima inserite nell'*environment* le segnature delle funzioni, e di conseguenza si procede ad analizzare le singole dichiarazioni in ordine di apparizione.

$$\frac{\langle \sigma, d_1 \rangle \to \sigma' \qquad \langle \sigma', d_2 \rangle \to \sigma''}{\langle \sigma, d_1\ d_2 \rangle \to \sigma''}$$

dove in σ sono presenti le segnature di tutte le funzioni visibili nell'attuale scope.

5.3 Right and Left expressions

Constants

$$\frac{}{\sigma \vdash_{RE} const(\tau) : \tau}$$

dove

- $\tau \in \{\texttt{int}, \texttt{boolean}, \texttt{real}, \texttt{char}, \texttt{string}\}$
- $const(\tau)$ è una costante di tipo τ.

Arrays

$$\frac{\vdash_T \tau \qquad \forall\, 1 \le i \le n \quad \sigma \vdash_{RE} r_i : \tau}{\sigma \vdash_{RE} \texttt{[}\, r_1 \texttt{,} \ldots \texttt{,} r_n \,\texttt{]} : \texttt{array[}n\texttt{]}\ \texttt{of}\ \tau}$$

$$\frac{\sigma \vdash_{LE} a : \texttt{array[}n\texttt{]}\ \texttt{of}\ \tau \qquad \sigma \vdash_{RE} i : \texttt{int}}{\sigma \vdash_{LE} a\texttt{[}\, i \,\texttt{]} : \tau}$$

Arithmetic operators

$$\frac{\sigma \vdash_{RE} r_1 : \tau \qquad \sigma \vdash_{RE} r_2 : \tau}{\sigma \vdash_{RE} r_1\ op\ r_2 : \tau}$$

dove

- $\tau \in \{\texttt{int}, \texttt{real}\}$;
- $op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{mod}, \texttt{div}\}$.

$$\frac{\sigma \vdash_{RE} r : \tau \qquad \tau \in \{\texttt{int}, \texttt{real}\}}{\sigma \vdash_{RE} \texttt{-}e : \tau}$$

Relation operators

$$\frac{\sigma \vdash_{RE} r_1 : \tau \qquad \sigma \vdash_{RE} r_2 : \tau}{\sigma \vdash_{RE} r_1 \, op \, r_2 : \text{boolean}}$$

dove

- $\tau \in \{\texttt{int}, \texttt{boolean}, \texttt{real}, \texttt{char}, \texttt{string}\}$;
- $op \in \{\texttt{<}, \texttt{>}, \texttt{>=}, \texttt{<=}, \texttt{=}, \sim\texttt{=}\}$.

Logic operators

$$\frac{\sigma \vdash_{RE} r_1 : \texttt{boolean} \qquad \sigma \vdash_{RE} r_2 : \texttt{boolean} \qquad op \in \{\texttt{and}, \texttt{or}\}}{\sigma \vdash_{RE} r_1 \, op \, r_2 : \texttt{boolean}}$$

$$\frac{\sigma \vdash_{RE} r : \texttt{boolean}}{\sigma \vdash_{RE} \texttt{not}\, e : \texttt{boolean}}$$

Left expressions

$$\frac{}{\sigma, (v : \tau) \vdash_{LE} v : \tau}$$

$$\frac{\sigma \vdash_{LE} e : \tau}{\sigma \vdash_{RE} e : \tau}$$

$$\frac{\sigma \vdash_{LE} e : \tau \qquad \tau \in \{\texttt{int}, \texttt{real}\} \qquad pre \in \{\texttt{inc}, \texttt{decr}\}}{\sigma \vdash_{LE} pre \, e : \tau}$$

Pointers

$$\frac{\sigma \vdash_{LE} e : \tau}{\sigma \vdash_{RE} \texttt{\&}e : \tau\texttt{@}}$$

$$\frac{\sigma \vdash_{RE} e : \tau\texttt{@}}{\sigma \vdash_{LE} \texttt{@}e : \tau}$$

Function call

$$\frac{\sigma \vdash_{LE} f : (\tau_1 \times \cdots \times \tau_n) \to \tau \qquad \forall\, 1 \le i \le n \quad \sigma \vdash_{RE} x_i : \tau_i}{\sigma \vdash_{RE} f\texttt{(}x_1\texttt{,} \ldots \texttt{,}x_n\texttt{)} : \tau}$$

5.4 Statements

Assignment

$$\frac{\sigma \vdash_{LE} l : \tau \qquad \sigma \vdash_{RE} r : \tau}{\sigma \vdash_{S} l := r}$$

Procedure call

$$\frac{\sigma \vdash_{LE} p : (\tau_1 \times \cdots \times \tau_n) \to () \qquad \forall\, 1 \le i \le n \quad \sigma \vdash_{RE} x_i : \tau_i}{\sigma \vdash_S p(x_1, \ldots, x_n)}$$

Exit and Continue

$$\frac{}{\sigma \vdash_S s}$$

$$\frac{\sigma \vdash_{RE} b : \texttt{boolean}}{\sigma \vdash_S s \,\texttt{on}\, b}$$

con $s \in \{\texttt{exit}, \texttt{continue}\}$

Return

$$\frac{\sigma \vdash_{RE} r : \tau}{\sigma \vdash_S \texttt{return}\, r}$$

Assert

$$\frac{\sigma \vdash_{RE} e : \texttt{boolean}}{\sigma \vdash_S \text{assert}\ e}$$

While

$$\frac{\sigma \vdash_{RE} b : \texttt{boolean} \qquad \langle \sigma, ds \rangle \to \sigma' \qquad \sigma' \vdash_S ss}{\sigma \vdash_S \texttt{while}\, b\, ds\, ss\, \texttt{end while}}$$

$$\frac{\langle \sigma, ds \rangle \to \sigma' \qquad \sigma' \vdash_S ss \qquad \sigma \vdash_{RE} b : \texttt{boolean}}{\sigma \vdash_S \texttt{repeat}\, ds\, ss\, \texttt{until}\, b}$$

Selection

$$\frac{\sigma \vdash_{RE} e : \texttt{boolean} \qquad \sigma \vdash_S ss \qquad \{\sigma \vdash_{RE} e' : \texttt{boolean} \qquad \sigma \vdash_S ss'\} \qquad (\sigma \vdash_S ss'')}{\sigma \vdash_S \texttt{if}\, e\, \texttt{then}\, ss\, \{\, \texttt{elsif}\, e'\, \texttt{then}\, ss'\,\}\, (\texttt{else}\, ss'')}$$

con $\{\, p \,\}$ indicante 0 o più occorrenze del pattern p, e $(\, p \,)$ indicante un'occorrenza opzionale.

For

$$\frac{\sigma \vdash_{LE} i : \texttt{int} \qquad \sigma \vdash_{LE} begin : \texttt{int} \qquad \sigma \vdash_{LE} end : \texttt{int} \qquad \langle \sigma, ds \rangle \to \sigma' \qquad \sigma' \vdash_S ss}{\sigma \vdash_S \texttt{for}\, [\texttt{decreasing}]\ i\ \texttt{:=}\ begin \ldots end\ \texttt{do}\ ds\ ss\ \texttt{end for}}$$

Label

$$\frac{\sigma \vdash_S s}{\sigma, l \vdash_S l\texttt{:}s}$$

$$\frac{}{\sigma, l \vdash_S \texttt{goto}\, l}$$

Statement list

$$\frac{\sigma \vdash_S s_1 \qquad \sigma \vdash_S s_2}{\sigma \vdash_S s_1\ s_2}$$