ProgettoLC Parziale Parte 3 Gruppo 3 Relazione

D'Abrosca Gianluca, Lunardi Riccardo, Zuccato Francesco

24 aprile 2023

1 Grammatica

Come riferimenti principali per la sintassi di Pascal sono state utilizzate la wiki di Free Pascal e la documentazione del progetto Lazarus. La grammatica è stata implementata tramite BNFC, consultabile al file `src/modules/grammar.cf`.
Il file Alex non è stato modificato, mentre il parser di Happy è stato ampliato per inizializzare correttamente tutti i parametri definiti nell'albero di sintassi astratta.

1.1 Scelte implementative

Dichiarazioni In questa implementazione, come richiesto dal progetto, si permette dichiarazione e inizializzazione di variabili, costanti, funzioni e procedure prima di ogni blocco.
Per permettere la mutua ricorsione si permette di definire tramite la parola chiave `forward` tutte le funzioni che dovranno essere implementate in seguito, prima però dell'inizio del blocco degli statements: questa modalità permette di eseguire l'analisi statica semantica attraversando l'albero di sintassi astratta una sola volta.

Valore di ritorno In Pascal, le funzioni possono ritornare valori in due modi:

- Assegnando il risultato ad una variabile speciale: result.
- Assegnando il risultato ad una variabile con lo stesso nome della funzione.

È stato deciso di implementare quest'ultima modalità.

Puntatori In Pascal, come consultabile nella documentazione e dagli esempi, il puntatore viene sempre utilizzato insieme a strutture definite dall'utente che nella versione del codice attuale non viene permessa. In quest'implementazione viene data la possibilità di definire puntatori come un altro tipo qualsiasi di variabile, in modo tale che possano essere definiti liberamente, senza dichiarare anche costrutti definibili dall'utente.
Per quanto riguarda la nostra implementazione, un puntatore per essere compatibile con un certo tipo di dato deve esserne uguale. Per esempio, un puntatore ad un array deve avere lo stesso tipo e le stesse identiche dimensioni.

Dichiarazione di array La dichiarazione del range di un array deve essere fatta tramite interi. Questo avviene per evitare che possano essere utilizzare variabili per allocare i range in modo dinamico.
Il range di un array monodimensionale viene così definito: $[index_{sx} \; .. \; index_{dx}]$. Si è assunto che scrivere $[index_{sx} \; .. \; index_{dx}]$ o $[index_{dx} \; .. \; index_{sx}]$ è indifferente. Per quanto riguarda il serializzatore si è scelto che stampi sempre per primo l'intero minore tra i due.

1.2 Parser

Il parser prodotto da BNFC è stato modificato principalmente per inizializzare le diverse strutture dati dell'albero di sintassi astratta, che verrano elaborate e modificate durante l'analisi di sintassi semantica.
In particolare:

- Tramite `tokenLineCol` assegniamo a tutte le strutture la posizione del relativo token nel file parsato;
- Nelle RightExp vengono inizializzati tutti i tipi con un parametro dummy, `T.TBDType`. Il tipo corretto verrà infatti assegnato durante l'analisi semantica qualora necessario.
- Gli enviroment vengono inizializzati come vuoti: anche questi verranno popolati duranti l'analisi di semantica statica.

2 Sintassi astratta

L'albero di sintassi astratta è stato implementato con Haskell nel file `src/modules/AbstractSyntax.hs`.
I parametri delle strutture sono principalmente:

- Intrinseci alla struttura stessa (es. dx o sx per gli assegnamenti)
- Posizione del token nel file parsato
- Tipo dell'espressione
- Enviroment
- Lista degli errori

3 Serializzatore

Per quanto riguarda il serializzatore si è creato il file `src/modules/PrettyPrinter.hs`.
Per creare un'unica funzione che accettasse in input qualsiasi tipo di dato proveniente dalla sintassi astratta si è creata una classe chiamata `PrettyPrinterClass` che, in base alle varie istanze, serializza correttamente l'albero.
Si è ovviamente tenuto in considerazione quando termina una riga per tornare a capo e anche quando inizia un blocco per indentare correttamente le righe al suo interno.

Questo modulo contiene anche del codice utile per poter debuggare la sintassi astratta, in modo tale che questa, una volta serializzata, venga salvata sul file andando a capo in occorrenza delle parentesi graffe, rendendo più leggibile l'output dell'analisi di semantica statica.

4 Type System

Per type system si intende la porzione dedicata alla gestione dei tipi da implementare. Il codice è consultabile al file `src/modules/Types.hs`

4.1 Gerarchia dei tipi

I tipi sono stati implementati secondo una certa gerarchia: è stato utile per gestire alcuni confronti e per le coercion nell'analisi semantica.

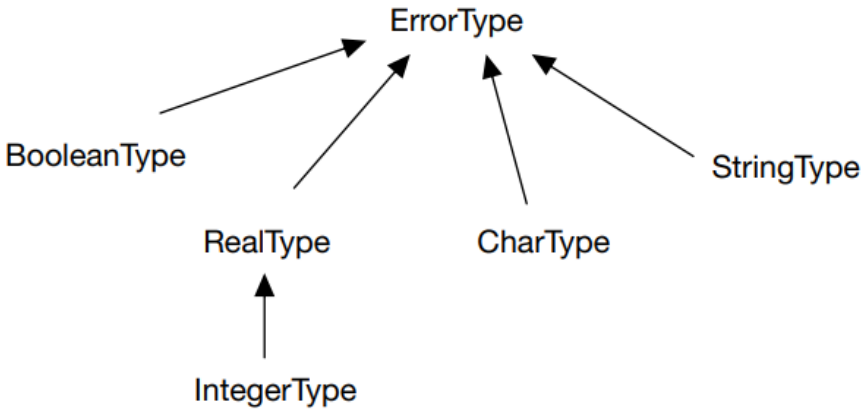

Figura 1: Gerarchia dei tipi

4.2 Implementazione dei tipi

I seguenti tipi sono stati implementati tramite la struttura "data":

- BooleanType
- IntegerType
- RealType
- CharType
- StringType
- ArrayType

- PointerType
- TBDType

Il nome dei tipi è autoesplicativo. Viene precisato che con "Real" si intendono i numeri in virgola mobile e con TBDType un tipo dummy che qualora necessario deve essere sovrascritto con l'analisi di semantica statica.
Per tutti i tipi sono state sovrascritte le istanze di `Show` e `Eq`, in quanto sono state utili a semplificare le operazioni di uguaglianza e di conversione a stringa.
All'interno del file `Types.hs`, sono state implementate anche alcune delle funzioni che gestiscono le relazioni tra i tipi.

5 Type Checker

5.1 Errori

Gli errori sono stati tutti raggruppati nel file `src/modules/ErrorMessage.hs` per una più facile gestione. Ogni funzione accetta come parametri la posizione a cui l'errore si è verificato e opzionalmente altri valori che verranno aggiunto al messaggio.

5.2 Implementazione dell'enviroment

L'enviroment è stato implementato come una mappa tra `String` e `EnvEntry` tramite il modulo `Data.Map`. `String` rappresenta il nome dell'entità nell'enviroment (es. nome della funzione), mentre le `EnvEntry` sono un tipo di dato creato appositamente, che rappresentano appunto un valore della mappa.
Ogni entry dell'enviroment possiede gli stessi 2 parametri:

- `parent_env`, un booleano che permette di capire se è possibile sovrascrivere una determinata entry dell'enviroment o meno. Se False la dichiarazione è stata fatta nello stesso blocco, quindi non è possibile sovrascriverla. Viceversa se True. Viene fatta un eccezione per le ConstEntry e le ForIteratorEntry che non sono sovra scrivibili nei blocchi sottostanti.
- `pos_decl`, una coppia di interi che nel codice del Three Address Code aiuta a distinguere una variabile da una che la ha sovrascritta.

I possibili valori delle entry sono:

- `VarEntry`, che contiene il tipo di una variabile;
- `ConstEntry`, che contiene il tipo della costante;
- `ForIteratorEntry`, che serve a distinguere una qualsiasi variabile da un iterator del for;
- `ProcEntry`, rappresentata dai parametri, espressi tramite una lista di coppie stringa-tipo e da un bool che indica se la procedura è dichiarata in forward;

- `FunEntry`;

FunEntry è rappresentata tramite i suoi parametri, a sua volta formati da una lista di coppie stringa-tipo, il suo tipo di ritorno e da altri tre valori booleani. I valori booleani indicano:

- Se la funzione è stata dichiarata come forward;
- Se il nome della funzione può essere usato come variabile per indicare il valore di ritorno della funzione, nel blocco corrente;
- Se il nome della funzione è già stato usato per indicare il valore di ritorno nel blocco corretto.

Questi ultimi due booleani sono utili durante l'analisi statica: in Pascal, infatti, per restituire un valore all'interno di una funzione, si usa assegnare tale valore ad una variabile con lo stesso nome della funzione.

5.3 Implementazione dell'analisi di semantica statica

Il modulo dell'analisi di semantica statica, consultabile al file `src/modules/StaticSemantic.hs`, è strutturalmente simile a quella del serializzatore. L'implementazione è stata infatti realizzata tramite una classe `StaticSemanticClass` e una funzione `staticsemanticAux`, che è stata definita per ogni struttura dell'albero di sintassi astratta.
In generale, ogni ridefinizione della funzione `staticsemanticAux` va a effettuare i diversi controlli necessari, ritornando alla fine lo stesso tipo di struttura ricevuto in input, ma avendo tutti i parametri in output valutati (es. l'enviroment non sarà più vuoto).
Per tutti i dettagli si consiglia la visione diretta del codice e dei commenti associati.

6 Three-Address Code

6.1 Struttura Dati

La struttura dati principale *State* è composta da :

- `tac :: [ Block ]`
  contiene la lista dei blocchi, dove ogni blocco contiene la sua lista di istruzioni
- `strings :: [ String ]`
  lista di tutte le stringhe presenti nel codice
- `temp_idx, block_idx, str_idx :: [ Int ]`
  variabili utilizzate per la simulazione dello stato
- `string_constants :: [ (String, Address) ]`
  mappa che per ogni costante di tipo stringa ritorna l'address della stringa corrispondente

dove *Block* è definita come :

- `block_name :: String`
  nome univoco che identifica ogni blocco
- `code :: [ Instruction ]`
  sequenza di istruzioni base del tac da eseguire

Gli altri data type di supporto sono:

- `PrimType`
  tipi del tac: int, double, char, bool, address
- `Address`
  tipi di address: int, real, char, bool, program variable, temporary variable
- `Instruction`
  possibile istruzioni del tac: Jump, Binary Assignment, Procedure Call, ecc.
- `UnaryOp, BinaryArithmOp, BinaryRelatOp`
  tipi di operatori: unari (not, -), binari aritmetici (+, -, *, ecc.) o relazionali ($<$, $>$, $\leq$, $=$ ecc.). Si noti come and e or non sono degli operatori di base in quanto sarebbero in constrasto con il three address code.

6.2 Implementazione

L'implementazione del TAC è piuttosto efficiente in quanto:

- **insert dei blocchi** all'interno del tac o è fatto in testa con l'operatore (:) oppure, se fatto in mezzo alla lista tramite (++), comunque il costo non è O(|istruzioni|) ma O(|blocchi|) e ovviamente il numero dei blocchi è minore del numero di istruzioni. Inoltre, se è necessario inserire più blocchi assieme nel mezzo contemporaneamente, la lista viene scandita solo una volta e i blocchi vengono inseriti tutti assieme. Ciò ad esempio avviene per il caso dell'if-then(-else) dove si creano sempre, per semplicità, tre blocchi.
- **insert delle istruzioni** è fatto sempre in testa al blocco desiderato (che va cercato scandendo la lista dall'inizio) e quindi anche in questo caso il costo di aggiungere un'istruzione è O(|blocchi|).
- **reverse** del tac viene fatto solo una volta al termine della generazione del tac, dove viene invertito sia l'ordine dei blocchi sia l'ordine delle istruzioni all'interno di ogni blocco.

In generale il TAC implementato è basato sull'abstract syntax generata dal parser e verificata dalla semantica statica. Quindi il codice del TAC inizia già con l'assunzione di correttezza del codice fornito (altrimenti il modulo di semantica statica lancerebbe almeno un errore e la generazione del tac non verrebbe invocata). Di conseguenza, nel codice del tac, non ci sono molti controlli di verifica di correttezza delle operazioni e dei tipi.
Si sottolinea principalmente l'implementazione di:

- **OR**: generato in modalità shortcut: si valuta prima la right expression sinistra e se è vera non si valuta la parte destra ma si salta direttamente al blocco del codice che segue tramite una $JumpIfTrue$. Se invece è falso allora si procede con la valutazione dell'elemento di destra: se è vero la computazione continuerà arrivando automaticamente al blocco true altrimenti si è ridirezionati al blocco false tramite una $JumpIfFalse$. Entrambi i blocchi poi convergono su un blocco next. L'AND è implementato in modo identico, ma con la differenza che il primo jump è un $JumpIfFalse$ al blocco false.

Anche la generazione del codice dell'if-then-else è molto simile a quella di OR/AND, in emtrambi i casi per semplicità vengono sempre creati tre blocchi nuovi: $then(true)$, $else(false)$, $next$.

7 Guida alla compilazione

Per compilare e testare il progetto è stato creato un `MakeFile`. I comandi principali sono i seguenti:

- `make` o `make ghci` compilerà, se necessario, il BNFC, il parser e i file Haskell. Se l'operazione terminerà con successo, si entrerà direttamente nel `Main.hs` tramite GHCI. A questo punto si possono testare i file .pas della cartella test_files tramite "testami $nome_del_test$" (es. $testami"assignment"$)
- `make ghc` compilerà il `Main.hs`
- `make lexer` compilerà solo il lexer
- `make parser` compilerà solo per parser
- `make clean` eliminerà i file risultanti dalla compilazione nella cartella `bin` e i risultati del serializzatore della cartella `test_files/temp`