2 Descrizione del progetto

Lo scopo del progetto è implementare un frammento di Rust, un linguaggio di programmazione imperativo, contraddistinto da una gestione efficiente e affidabile della memoria. Le caratteristiche peculiari di Rust sono legate alla gestione delle variabili, e in particolare alla nozione di *ownership*, che consente una gestione sicura della memoria, prevenendo problemi di allocazione o deallocazione che invece affliggono altri linguaggi con la gestione manuale dello heap.

In questa sezione trovate la descrizione del linguaggio, di seguito chiamato TinyRust. Tutti gli esempi di codice TinyRust presentati in seguito sono anche programmi Rust sintatticamente corretti, che possono essere compilati ed eseguiti, a meno nei casi in cui sono stati artificialmente introdotti degli errori per illustrare alcune caratteristiche di Rust. Il progetto chiede di implementare il lexer, il parser e l'interprete small-step di TinyRust.

2.1 Programmi

Un programma in TinyRust ha una dichiarazione di funzioni, che deve comprendere quella della funzione `main`, che è l'entry point del programma. Nel corpo delle funzioni vengono anche definite le variabili. La forma è quindi la seguente:

```
// dichiarazione di funzioni/procedure

fn main() {
  // dichiarazioni di variabili;
  // comandi
}
```

Come sono fatte le dichiarazioni di funzioni e procedure lo vedremo in seguito.

2.2 Dichiarazioni di variabile

I blocchi di dichiarazioni sono porzioni di codice racchiuse tra parentesi graffe: `{` segnala l'inizio del blocco, e `}` la sua fine. I blocchi possono contenere dichiarazioni di variabili, istruzioni ed espressioni, e possono annidare altri blocchi. Inoltre, i blocchi delimitano il contesto di visibilità delle variabili (scope). Il linguaggio ha lo scope statico.

Le variabili vengono dichiarate con la keyword `let`. Ogni variabile è associata ad un valore (tipato) che è allocato in memoria. Per default, le variabili sono *immutabili*, ovvero una volta associate a un valore con il costrutto `let`, non è più possibile modificare tale associazione (vedremo a breve che è anche possibile avere delle variabili mutabili).

Consideriamo ad esempio il seguente programma:

```
fn main() {
  let x = 3;     // variabile immutabile di tipo intero
  let y = x+1;   // idem
  println!("{x}"); // output: 3
  println!("{y}"); // output: 4
}
```

Il compilatore inferisce che il tipo della variabile `x` è intero, e così anche per la variabile `y`. Tra i vari tipi usati da Rust per rappresentare interi di diverse dimensioni e con o senza sengno, in TinyRust consideriamo soltanto il tipo `i32`, ovvero gli interi con segno rappresentati su 32 bit.

Si noti inoltre che la dichiarazione di `y` alla riga 3 *copia* il valore di `x` in quello di `y`, e lo incrementa di 1. Questa osservazione potrà sembrare in questo momento inessenziale, ma verrà ripresa quando più avanti parleremo del concetto di *ownership*. Per il momento, ci interessa solo sapere che se il valore contenuto in una variabile viene "copiato", come in questo caso accade al valore di `x`, allora è possibile continuare ad usare la variabile, come facciamo nel programma di sopra alla riga 4. In seguito vedremo esempi di tipi *non copiabili*, che richiedono un trattamento particolare, chiamato *borrowing*.

2.3 Variabili mutabili e immutabili

Se si tenta di modificare una variabile immutabile, il compilatore di Rust segnala un errore, come nel seguente programma:

```
1 fn main() {
2   let x = 3;   // variabile immutabile di tipo intero
3   let y = x+1;
4   x = x+y;     // errore: x immutabile
5   println!("{x}");
6 }
```

L'errore segnalato dal compilatore è il seguente:

```
cannot assign twice to immutable variable ‘x‘
```

Questo deriva dal fatto che la variabile `x` è dichiarata alla linea 2 come immutabile, mentre l'assegnamento `x = x+y` alla linea 4 cerca di scrivere su `x`.

L'errore nel precedente programma può essere risolto definendo `x` come *mutabile*, aggiungendo la keyword `mut` prima del nome della variabile:

```
1 fn main() {
2   let mut x = 3;   // variabile mutabile di tipo intero
3   let y = x+1;
4   x = x+y;         // ok: x mutabile
5   println!("{x}"); // output: 7
6 }
```

Adesso l'assegnamento alla riga 4 non dà errore, dato che `x` è dichiarata mutabile. A seguito dell'esecuzione dell'assegnamento, `x` viene aggiornata al valore 7, come si può verificare dall'output del comando `println`.

Vediamo adesso un altro esempio, ma usando valori di tipo `String` anziché interi. La differenza cruciale tra i due tipi è che gli interi hanno una dimensione costante, mentre le `String` hanno una dimensione variabile, che può cambiare a tempo di esecuzione. Per questo motivo, Rust memorizza i valori dei due tipi in modi diversi: gli interi (e gli altri tipi scalari, come i booleani, i caratteri, ecc.) sullo *stack*, mentre i valori di tipo `String` sono memorizzati nello *heap*. Questa differenza è importante per capire come vengono copiati o trasferiti i valori di

tipi diversi, argomento che sarà trattato in seguito quando saranno descritti i concetti di *ownership* e di *borrowing*.

Si consideri il seguente programma:

```
1  fn main () {
2    let x = String::from("Ciao");
3    x.push_str(", mondo"); // errore: x non mutabile
4    println!("{x}");
5  }
```

Il compilatore Rust segnala il seguente errore:

```
cannot borrow ‘x‘ as mutable, as it is not declared as mutable
```

Il motivo dell'errore è che alla riga 3 si sta cercando di modificare il valore associato alla variabile `x`, aggiungendo in coda la stringa `", mondo"`. Per risolvere il problema, basta dichiarare `x` come mutabile:

```
1  fn main () {
2    let mut x = String::from("Ciao");
3    x.push_str(", mondo"); // ok: x mutabile
4    println!("{x}");       // output: Ciao, mondo
5  }
```

2.4 Blocchi di dichiarazioni, scope e shadowing

Rust permette di annidare blocchi di dichiarazioni. Nel nostro caso le dichiarazioni saranno di variabile. Quando nel blocco interno viene dichiarata una variabile con lo stesso nome di una variabile nel blocco esterno, nel blocco interno si ridefinisce temporaneamente la variabile. All'uscita dal blocco interno la dichiarazione della variabile mascherata ridiventa attiva. Questo fenomeno nel gergo di Rust, è chiamato *shadowing*. Ad esempio:

```
1  fn main() {
2    let x = 2;     // prima dichiarazione di x
3    let x = x+1;   // seconda dichiarazione di x
4    {
5      let x = x*2;      // terza dichiarazione di x
6      println!("{x}"); // output: 6
7    }
8    println!("{x}");   // output: 3
9  }
```

Nel programma precedente, la variabile `x` è dichiarata tre volte, alle linee 2, 3 e 5. Il blocco interno va dalla linea 3 alla linea 6. Il primo `let` associa a `x` il valore 2, che viene usato in lettura alla linea 3. La prima `x` non esiste più, mascherata dalla nuova dichiarazione alla linea 3. Si noti inoltre che, nella valutazione dell'espressione `x+1` alla riga 3, la variabile `x` (che è quella della riga 2) è associata ad un valore. L'esecuzione della linea 6 farà stampare il valore 6, mentre quello della linea 8 farà stampare il valore 3.

Una variabile dichiarata in un blocco interno non è più visibile nel blocco esterno. Si consideri il seguente esempio:

```
1 fn main() {
2   let x = 3;  // dichiarazione iniziale di x
3   {
4     let y = x+1;   // dichiarazione di y
5   }
6   println!("{y}"); // errore, y non in scope
7 }
```

Questo programma dà un errore di compilazione:

```
cannot find value ‘y‘ in this scope
```

2.5 Funzioni e procedure

Tra le dichiarazioni troviamo quelle di funzioni e procedure. Una funzione restituisce un valore che è l'ultima espressione del corpo della funzione, mentre una procedura non restituisce alcun valore. Sia funzioni che procedure possono avere dei parametri, che devono essere esplicitamente tipati. Nel caso delle funzioni, anche il tipo di ritorno deve essere esplicitamente tipato.

Ad esempio, consideriamo il seguente programma:

```
1  fn foo(x: i32) -> i32 { // dichiarazione di funzione
2    let mut y = 4;   // ambiente locale
3    y = y+x;
4    y+x              // valore di ritorno
5  }
6  fn main() {
7    let x = 3;
8    let y = foo(x);  // chiamata di funzione
9    println!("{y}"); // output: 10
10 }
```

Le righe dalla 1 alla 5 definiscono una funzione di nome `foo` (keyword `fn`) che riceve un intero con il segno (`i32`) e restituisce un intero con il segno (`-> i32`). Alla riga 9 viene stampato il valore 10, che è ottenuto valutando l'espressione alla linea 4 (che determina il valore di ritorno della funzione).

Le procedure, a differenza delle funzioni, non restituiscono alcun valore. Sintatticamente, la differenza tra la dichiarazione di una procedura e di una funzione sta nel tipo di ritorno: nel caso della procedura, questo non è indicato, mentre nel caso della funzione sì.

Ad esempio, consideriamo il seguente programma:

```
fn bar(x: i32) {   // dichiarazione di procedura
  let mut y = 4;
  y = y+x;
  println!("{y}"); // nessun valore di ritorno
}
fn main () {
  let x = 3;
  bar(x);          // chiamata di procedura
}
```

In questo caso l'esecuzione di `bar(x)` farà si che si stampi il valore 7, che è il valore della variabile `y`.

2.6 Blocchi e comandi

I blocchi possono anche essere usati per racchiudere comandi. Questo è comune, ad esempio, per i comandi che devono essere eseguiti nel ramo "then" e nel ramo "else" di un costrutto condizionale, oppure nel corpo di un costrutto iterativo.

Ad esempio, il seguente programma stampa `pari` o `dispari` a seconda del valore assegnato alla variabile `x`:

```
fn main() {
  let x = 3;
  if x % 2 == 0 {
      println!("pari")
  } else {
      println!("dispari")
  }
}
```

2.7 Ownership

Il linguaggio determina la memoria che serve per allocare un valore al momento della dichiarazione della variabile, quindi la quantità di memoria che serve viene stabilita quando si incontra una dichiarazione. Per i tipi di dato base, come ad esempio interi e booleani, la quantità di memoria è stabilita dal tipo di dato medesimo, mentre questo non è in generale vero per dati dinamici, come ad esempio le stringhe.

Il concetto di ownership è centrale in Rust, in quanto permette di garantire la gestione sicura della memoria (diversamente la linguaggi come il C, dove il programmatore ha l'onere di allocare/deallocare la memoria, con la conseguente possibilità di introdurre errori), senza richiedere un *garbage collector* a tempo di esecuzione (come, invece, avviene in Java).

La relazione di *ownership* è definita delle seguenti regole:

- ogni valore ha un *owner*;
- per ogni valore, ci può essere solo un *owner* nello stesso momento;
- quando l'owner non è più in scope, il valore viene eliminato.

Per illustrare la ownership, consideriamo il seguente programma:

```
1  fn main() {
2    let x = String::from("Ciao");
3    let y = x;
4    println!("{x}"); // errore di ownership
5  }
```

Alla riga 2 viene dichiarata una variabile `x`, a cui si associa una stringa e tale variabile diventa *owner* di tale valore. Alla riga 3 viene dichiarata la variabile `y`, a cui si associa il valore di `x` e qui avviene il trasferimento di *ownership* del valore `"Ciao"` da `x` a `y`. Il comando che stampa il valore della variabile `x`, cioè `println!("{x}")`, dà un errore perché la variabile `x` non è più owner del valore, ma con il `let` lo ha trasferito alla variabile `y`.

Anche il passaggio dei parametri ad una funzione o procedura provoca un cambio nella ownership:

```
1  fn fie(s: String) {
2    println!("{s}");
3  }
4  fn main() {
5    let x = String::from("Ciao");
6    fie(x);
7    println!("{x}"); // errore di ownership
8  }
```

La procedura `fie` acquisisce la ownership del valore di tipo `String` associato al parametro `s`. Quando la procedura viene invocata (come alla riga 6), la ownership della `String` è *trasferita* dal chiamante al chiamato, ovvero alla procedura `fie`. Si noti qui la differenza di trattamento tra i parametri di tipi scalari (come ad esempio `i32`), e quelli di tipi dinamici come `String`: con la chiamata di funzione o procedura, i primi vengono *copiati*, mentre i secondi vengono *trasferiti*. Quando la procedura `fie` ha terminato la sua esecuzione, `s` esce dallo scope, e la memoria usata per contenere il valore associato a `s` viene deallocata. Per quest'ultimo motivo, la variabile `x` alla riga 7 non ha più la ownership del valore `"Ciao"`, essendo questo stato deallocato. Quindi, il compilatore segnala il seguente errore alla linea 7:

```
borrow of moved value:  ‘x‘
```

2.8 Borrowing

Il trasferimento della ownership può essere derogato con il *borrowing*, che consente di condividere un valore tra due o più variabili. In sostanza si ha un *aliasing*, che però è noto al momento della scrittura del programma. Per prendere in prestito una variabile, si usa una *reference*: questa si crea con il simbolo `&`. Ad esempio, `&x` è una reference alla variabile `x`. Le references indicate in questo modo sono immutabili: vedremo in seguito come ottenerne anche di mutabili.

Ad esempio, consideriamo il seguente programma:

```
1 fn main() {
2   let x = String::from("Ciao");
3   let y = &x;        // borrow di x a y
4   println!("{y}");   // output: Ciao
5   println!("{x}");   // output: Ciao
6 }
```

La riga 5 stampa `Ciao` dato che la variabile `x` è ancora owner del valore che ha solo prestato alla variabile `y`. Il valore prestato alla variabile `y` viene stampato senza alcun problema anche alla riga 4.

Anche i parametri di una funzione o procedura possono essere "prestati". Consideriamo il seguente programma:

```
1 fn presta(y: &String) {
2   println!("il parametro prestato: {y}");
3 }
4 fn main() {
5   let x = String::from("Ciao");
6   presta(&x); // reference (immutabile) a x
7   println!("il parametro x: {x}");
8 }
```

Qui il parametro attuale `x` è prestato, e infatti è `&x` e quindi `x` non perde l'ownerhip. Dunque, il programma stamperà, alla linea 6, `il parametro x: Ciao`. Avendo dichiarato tra i parametri attuali della procedura `presta` che il tipo della `x` è un borrowing di una `String`, al momento dell'invocazione il parametro attuale deve essere anch'esso prestato.

È importante notare che le reference ottenute nel modo sopra descritto sono sempre immutabili, indipendentemente dal fatto che la variabile originale fosse mutabile o meno. Ad esempio, consideriamo il seguente programma:

```
1 fn main() {
2   let mut x = String::from("Ciao");
3   let y = &x;
4   x.push_str(", mondo"); // errore: y non mutabile
5   println!("{y}");
6 }
```

Poiché le regole per il borrowing impediscono di avere, allo stesso tempo, reference mutabili e immutabili allo stesso valore, il compilatore Rust si lamenta, segnalando il seguente errore:

```
cannot borrow ‘x‘ as mutable because it is also borrowed as
immutable
```

Per ottenere una reference mutabile alla variabile `x` si deve scrivere `&mut x`. Ad esempio, consideriamo il seguente programma:

```
1 fn main() {
2   let mut x = String::from("Ciao");
3   let y = &mut x;   // borrow di x a y (mutabile)
4   y.push_str(", mondo");
5   println!("{y}");  // output: Ciao, mondo
6   println!("{x}");  // output: Ciao, mondo
7 }
```

Il programma dichiara la variabile `x` come mutabile, così come la reference `y`. In questo modo, la modifica del valore associato a `y` (e, quindi, anche a `x`) alla riga 4 ha successo, ed entrambe le `println` stampano lo stesso messaggio.

Si noti invece che il seguente programma, dove abbiamo soltanto rimosso il modificatore `mut` dalla dichiarazione di `x`, produce un errore di compilazione:

```
1 fn main () {
2   let x = String::from("Ciao");
3   let y = &mut x;   // errore: x non mutabile
4   y.push_str(", mondo");
5   println!("{y}");
6   println!("{x}");
7 }
```

Poiché le regole per il borrowing vietano di rendere mutabile una reference ad una variabile che era stata dichiarata come immutabile, il compilatore Rust si lamenta, segnalando il seguente errore:

```
cannot borrow ‘x‘ as mutable, as it is not declared as mutable
```

2.9 Iterazione

Per l'iterazione abbiamo il `loop` ed il `break`. Il comando `loop { c }` itera il comando `c`, a meno che `c` non porti all'esecuzione di un `break`.

Consideriamo il seguente programma:

```
1 fn main () {
2   let mut y = 0;
3   loop {
4     y = 1+y;
5     println!("{y}");
6   }
7 }
```

L'esecuzione di questo programma porta alla stampa della sequenza 1, 2, 3, ..., e non si interrompe mai.

Il comando `break` provoca invece l'interruzione del loop che lo contiene. Consideriamo, ad esempio, il seguente programma:

```
1  fn main () {
2    let mut y = 3;
3    loop {
4      if y==0 { break; }
5      else { println!("{y}"); y = y-1; }
6    }
7    println!("{y}");
8  }
```

In questo caso la linea 7 viene eseguita dopo aver eseguito il `loop` che termina dato che viene eseguito un `break`. L'esecuzione del programma stampa quindi la sequenza 3 2 1 0.

2.10 Loop annidati

Si noti che il `break` interrompe soltanto il `loop` più interno nel quale è incluso. Eventuali `loop` più esterni non sono interrotti. Ad esempio, si consideri il seguente programma:

```
1  fn main () {
2    let mut i = 0;
3    loop {
4      let mut j=0;
5      loop {
6        if j==2 { break; }
7        else { println!("{i},{j}"); j = j+1; }
8      };
9      if i==2 { break; }
10     else {i = i+1; }
11   }
12 }
```

Il programma stampa la sequenza (omettiamo in newline):

```
0,0   0,1   0,2   1,0   1,1   1,2   2,0   2,1   2,2
```

2.11 Blocchi come espressioni (opzionale)

I blocchi possono essere usati come espressioni, ovvero possono restituire un valore. A tal fine, l'ultimo valore calcolato all'interno del blocco, se non terminato da un punto e virgola, viene considerato il valore restituito. Ad esempio:

```
1  fn main() {
2    let x = {
3      let y = 5;
4      y+2          // valore restituito dal blocco
5    };
6    println!("{x}"); // output: 7
7  }
```

2.12 Funzioni annidate (opzionale)

I blocchi possono avere delle dichiarazioni locali. Abbiamo visto per ora come dichiarazioni locali solo variabili. Possiamo estendere consentendo di dichiarare localmente delle funzioni o procedure. Consideriamo il seguente programma:

```
fn main() {
  let x = 4;
  {
    fn interna (y: &i32) {
      println!("{y}");
    }
    interna(&x);
  }
  interna(&x); // errore: not found in this scope
}
```

La procedura interna può essere usata solo nel blocco interno. Valgono le solite regole di visibilità.

Sia nelle funzioni che nelle procedure non ci sono nomi di variabile non locali, a parte altre funzioni o procedure. Per chiarire questa caratteristica consideriamo il seguente pezzo di codice:

```
fn main() {
  let mut y = 4;
  fn scopecheck () {
      y = 6; // errore
  }
  y = 3+y;
}
```

Qui la procedura `scopecheck` dovrebbe aggiornare una variabile non locale, alla quale però non ha accesso e quindi si genera un errore. Tutti i nomi di variabile, a parte quelli di funzioni o procedure, devono essere locali, quindi o essere dei parametri oppure delle dichiarazioni locali.