Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Il linguaggio Janus concorrente: definizione, compilazione e sperimentazione

Relatore:
Chiar.mo Prof.
Ivan Lanese

Presentata da:
Andrea Sammarchi

II sessione - secondo appello
Anno Accademico 2018/2019

Indice

Capitolo 1

Introduzione

Alla base di questo progetto vi è lo studio di *Janus*, un linguaggio di programmazione reversibile sviluppato da Christopher Lutz e Howard Derby alla Caltech nel 1982 [1]. Janus è un linguaggio che supporta una computazione deterministica in avanti e indietro, in questa relazione ne verrà esaminato il comportamento partendo da semplici istruzioni fino a costrutti più complessi. La tesi che mi è stata proposta dal professore Ivan Lanese ha suscitato subito il mio interesse per poter approfondire lo studio dei linguaggi di programmazione, interpreti e compilatori. Il progetto richiedeva lo sviluppo di un compilatore per Janus, l'aggiunta di un meccanismo di programmazione concorrente e scambio di messaggi, infine una parte di sperimentazione volta a provare la correttezza del linguaggio lavorando con alcuni algoritmi concorrenti.

La tesi è articolata da un capilo di **background** dove verrà presentata la logica e la sintassi di *Janus*. Inoltre verrà presentato **ANTLR** nella sua ultima versione: ANTLR4 [2], un generatore di parser, il quale una volta presa una grammatica in input ci fornisce tutti gli strumenti utili per lo sviluppo del compilatore. Il terzo capitolo si occupa di riassumere come è stato svolto il lavoro per la scrittura del compilatore, dalla stesura della grammatica in forma EBNF (Extended Backus-Naur Form), l'implementazione delle regole di parsing, la generazione del codice in uscita ed in particolare sullo sviluppo del costrutto *fork and join* e del meccanismo di **message passing**, per dotare *Janus* di un meccanismo di programmazione concorrente. Infine, prima del capitolo conclusivo, c'è una parte di sperimentazione dove verranno illustrtati gli esperimenti fatti ed i motivi dei loro successi o fallimenti.

Capitolo 2

Background

2.1
Cos è Janus?

Janus è un linguaggio di programmazione reversibile [3], questo significa che è in grado di esegure un programma in due direzioni, in avanti ed indietro. *Janus* è quindi capace di supportare computazioni che eseguono istruzioni seguendo il normale flusso del programma, oppure di eseguire lo stesso programma in maniera inversa. Mostriamo un esempio. Supponiamo di scrivere un programma che presi due valori li incrementi e poi ne faccia la somma. Il programma in questione avrà un corpo del genere:

```
a += 1
b += 1
a += b
```

Dove **a** e **b** sono le variabili prese in input e l'operatore += memorizza in **a** la somma tra **a** e **b**. Questo breve frammento di codice, se scritto in *Janus* potrà essere eseguito in due modi; in avanti, dove seguiamo sequenzialmente le istruioni scritte, ed indietro, dove la computazione parte dal basso ed avrà quindi un esecuzione del genere:

```
a -= b
b -= 1
a -= 1
```

Si noti che non solo l'esecuzione è invertita ma anche le istruzioni stesse. Dove nella computazione in avanti si ha l'istruzione di incremento per **a**, nella computazione inversa si ha il decremento. Questo significa che *Janus* supporta una computazione deterministica biettiva, è stato il primo linguaggio di programmazione di questa categoria.

Janus applica un approccio imperativo, la gestione della memoria è globale e non locale, per effetto di questo le procedure non producono valori di ritorno, ma prendendo valori per riferimento si possono comunque produrre risultati.

2.2

Sintassi di Janus

Un programma *Janus* è definito da un certo numero di procedure, queste vengono dichiarate utilizzando la parola chiave ***procedure***, seguita dal nome ed una lista di parametri. I parametri presi in input, come anticipato, sono passati per riferimento, il che significa che la procedura mantiene aggiornati i valori delle variabili. Le procedure vengono chiamate utilizzando le parole chiave riservate

call, per la chiamata alla procedura in avanti, e ***uncall*** per la chiamata all'indietro, seguito dal nome della procedura e gli eventuali parametri. Similmente al linguaggio *C*, e molti altri linguaggi, *Janus* include la procedura *main*, che ad oggi non è previsto che prenda in input alcun parametro.

2.2.1

Operatori di assegnamento composti

La versione di *Janus* a cui mi sono ispirato, disponibile in [4], utilizza una sinstassi similare al linguaggio *C*, gli operatori di assegnamento composti permettono di modificare il valore di una variabile con una sola operazione, come nel modo seguente:

```
a += 1
a -= b
```

La prima istruzione equivale a scrivere:

```
a = a + 1
```

Mentre la seconda quivale a:

```
a = a - b
```

Non è possibile invece utilizzare la stessa variabile sia come l-valore sia come r-valore, questo perchè l'istruzione risultante non sarebbe sempre biettiva. L'istruzione *a -= a* da sempre valore *0* ed è un istruzione non invertibile.

2.2.2

Data type

I tipi di dato supportato da questa versione di *Janus*, sono gli interi e array, questi possono essere dichiarati, globalmente, nel modo seguente:

```
int var
int vett[n]
```

Gli array possono anche essere dichiarati ed inizializzati seguendo la notazione del linguaggio *C*:

```
int y[3] = {1,2,3}
int z[3] = {4,5,6}
```

All'interno delle procedure è anche possibile dichiarare le variabili in modo locale, ma queste devono essere deallocate esplicitamente quando non più utilizzate. Ad una mancata o errata deallocazione di variabili locali, il compilatore genera un errore, mandando in fallimento la compilazione. L'utilizzo di variabili locali è possibile utilizzando le parole chiave **local** e **delocal** in questo modo:

```
local int var
delocal [assertion]
```

dove i vincoli di asserzione devono essere rispettati in fase di deallocazione nella modalità **forward**, ed in fase di allocazione nella **reverse**. Un asserzione significa che deve essere soddisfatta un'uguaglianza, altrimenti il programma termina generando un errore. Ad esempio, un asserzione per deallocare la variabile *var* può essere scritta in questo modo:

```
delocal int var = 0
```

dove la variabile *var* deve essere uguale a zero per soddisfare l'asserzione e far proseguire l'esecuzione del programma.

2.2.3

Swap operator

L'operatore di *swap* permette lo scambio del contenuto di due varibiali utilizzando una singola istruzione come la seguente:

```
varA <=> varB
```

la cui espressione viene interpretata come:

```
varA ^= varB
varB ^= varA
varA ^= varB
```

dove si utilizza l'operatore **XOR**, **OR** esclusivo bit a bit.

2.2.4

Assegnamento

In *Janus* l'assegnamento è consentito solamente se la variabile su cui si vuole scrivere ha valore zero. Un assegnamento del tipo:

```
varA = 5
```

non è detto che vada a buon fine se la variabile viene in qualche modo utilizzata precedentemente. Il compilatore, per verificare che si possa assegnare un valore ad una variabile, inserisce un istruzione di asserzione prima dell'assegnamento. L'istruzione precedente viene quindi tradotta in:

```
assert(varA == 0)
varA = 5
```

2.2.5

If then else

Abbiamo a disposizione due costrutti fondamentali in *Janus*, il primo è il condizionale *if then else*, come per il *C* ed altri linguaggi, esso ha bisogno di una condizione da verificare prima di entrare nel corpo, in più il costrutto si conclude con un asserzione, la quale diverrà la guardia nel caso della **reverse**. La sintassi è la seguente:

```
if [Cond] then
  [Body]
else
  [Body]
fi [Assert]
```

2.2.6

Loop

Se abbiamo bisogno di un ciclo, Janus mette a disposizione il costrutto *loop*, simile ad una combinazione tra il *while-do* e il *do-while*, con l'aggiunta di un asserzione prima dell'iterazione. Il corpo del *do* viene eseguito prima dell'ingresso nel ciclo *while* e come ultimo blocco da iterare. Per maggiore chiarezza viene mostrato il costrutto *loop* di *Janus* tradotto come espressione del *while* in pseudocodice

```
from [Cond0]
do
  [Stmt1]
loop
  [Stmt2]
until [Cond1]
```

```
assert(Cond0)
  [Stmt1]
while(!(Cond1))
  [Stmt2]
  assert(!(Cond0))
  [Stmt1]
```

dove notiamo che la *from condition*, quì indicata come *Cond0*, viene valutata prima del ciclo *while*, e la *until condition* diviene la guardia negata per essere il valore di soglia. Nel caso della chiamata alla *reverse* le condizioni vengono invertite e quindi anche le asserzioni, proprio per questo bisogna prestare particolare attenzione nel scegliere le guardie, sopratutto se queste utilizzano valori locali che dovranno essere deallocati in maniera corretta pensando ad entrambe le direzioni.

2.2.7

Esempio: Successione di Fibonacci

Si vuole mostrare un primo esempio di programma biettivo scritto in *Janus*, ovvero la successione di Fibonacci, cioè una successione di numeri interi in cui ciascun numero è la somma dei due precedenti, eccetto i primi due che sono per definizione *f(0) = 1* e *f(1) = 1*.

```
procedure fib(int x1, int x2, int n)
    if n = 0 then
        x1 += 1
        x2 += 1
    else
        n -= 1
        call fib(x1, x2, n)
        x1 += x2
        x1 <=> x2
    fi x1 = x2
```

le cui corrispondenti funzioni di forward e reverse in pseudocodice, sono le seguenti:

```
void fib_forward(x1,x2,n)
    if (n == 0)
        x1 += 1
        x2 += 1
        assert(x1 == x2)
    else
        n -= 1
        fib_forward(x1,x2,n)
        x1 += x2
        x1 <=> x2
        assert(!(x1 == x2))
void fib_reverse(x1,x2,n)
    if (x1 == x2)
        x2 -= 1
        x1 -= 1
        assert(n == 0)
    else
        x1 <=> x2
        x1 -= x2
        fib_reverse(x1,x2,n)
        n += 1
        assert(!(n == 0))
```

Notiamo come la **forward** rispetti l'esecuzione del codice *Janus* scritto, mentre la **reverse** si comporta in modo opposto. In questo esempio bisogna prestare attenzione ai valori che passiamo alla procedura, nel caso della forward bisogna passare un valore n diverso da zero, mentre nel caso della chiamata alla reverse passiamo un valore n uguale a zero, il quale sarà incrementato nel ramo *else*. Anche i valore di *x1* e *x2* dovranno essere passati diversamente a seconda del caso, in particolare nel caso della *reverse* dovranno essere inizializzati diversi da zero ma anche diversi tra loro. I loro valori devono essere ad un passo di distanza della successione di Fibonacci, ad esempio possiamo prendere F_n uguale a 55, ed F_{n-1} uguale a 34.

2.3

ANTLR

ANTLR [2], acronimo di **ANother Tool for Language Recognition**, è un generatore di parser. I parser top-down, come quello utilizzato da **ANTLR**, costruiscono un albero di derivazione a partire dal simbolo iniziale, usando informazioni della stringa di ingresso per decidere quale produzione applicare.

ANTLR può generare il *lexer (lexical analyser)*, il *parser* e l'*AST (Abstract Sintax Tree)* da una grammatica in formato simile al **EBNF** (**Extended Backus-Naur Form**).

Al momento, **ANTLR** supporta come target i seguenti linguaggi: **C++**, **Java**, **Python** e **C#**. Per questo progetto si è utilizzato come target **Java**.

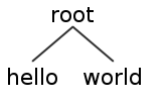
Una volta scritta la grammatica basterà darla in input al generatore di parser, **ANTLR** sarà in grado di creare le classi necessarie per scrivere le regole del linguaggio da noi creato.

Scrivendo una semplice grammatica come questa:

```
root : 'hello' ID;
ID   : [a-z]+ ;
```

dove *root* è la radice principale, *'hello'* la stringa che deve essere presa in input, ed *ID*, che risulta essere un token simbolico, può essere una qualsiasi stringa alfabetica. **ANTLR** è in grado così di fornirci gli strumenti per scrivere regole e comportamento del nostro linguaggio. Gli strumenti prevedono classi di funzioni utili per la visita dell'albero sintattico, l'ingresso e l'uscita da una regola, o di comportamenti generali su tutto il programma.

L'albero sintattico dell'esempio risulta quindi in questo modo:



dove la stringa passata in input è la classica *"hello world"*.

2.3.1

Come lavora ANTLR

La strategia principale di **ANTLR** si chiama LL(*) ed è un estensione naturale di LL(k). I parser LL(k) prendono decisioni, distiguendo tra le produzioni alternative, utilizzando al massimo k simboli di lookahead. Al contrario i parser LL(*) possono guardare arbitrariamente molto più avanti e possono prendere

decisioni di analisi riconoscendo se i token seguenti appartengono ad un normale linguaggio.

2.3.2

Parser LL(*)

Una grammatica determina esattamente quello che definisce un dato token e quali sequenze di token sono considerate valide. I parser riconoscono sequenze di token, i parser generati da **ANTLR** utilizzano una grammatica definita in questo modo:

G = (N, T, P, S, Π**,** η **)**

- **N** è un insieme finito di simboli non terminali
- **T** è un insieme finito di simboli terminali
- **P** è un insieme finito di produzioni
- **S** $\in$ **N** è il simbolo iniziale
- Π è un insieme di predicati semantici
- η è un insieme finito di azioni

I parser esistenti per grammatiche regolari LL sono lineari ma spesso poco pratici, perché non possono analizzare flussi infiniti come i protocolli socket e gli interpreti interattivi. Nel primo di due passaggi, questi parser devono leggere l'input da destra a sinistra.

ANTLR propone, invece, una strategia da un passaggio più semplice da sinistra a destra chiamata LL(*) che innesta DFA sui parser LL. Un DFA lookahead corrisponde alla normale porzione associata a uno specifico non terminale e ha uno stato di accettazione per ogni partizione. Ad un certo punto, i parser LL(*) espandono la produzione se la partizione corrisponde all'input rimanente. Di conseguenza, i parser LL(*) sono $O(n^2)$, ma in pratica vengono esaminati tipicamente uno o due token. Come con le precedenti strategie di analisi, esiste un parser LL(*) per ogni grammatica LL regolare. A differenza dei precedenti, i parser LL(*) possono prendere come input una grammatica LL regolare predefinita; gestiscono i predicati inserendo ponti speciali nel DFA lookahead.

Per completezza si inserisce una definizione formale di DFA lookahead:

I DFA Lookahead sono DFA aumentati con predicati. Formalmente data una grammatica **G = (N, T, P, S,** Π**,** η **)**, il corrispondente DFA **M = (S, Q,** Σ**,** Δ**, D, F)** dove :

- **S** è lo stato del sistema ereditato dal parser circostante
- **Q** è un insieme finito di stati
- Σ è l'alfabeto

- Δ è la mappatura della funzione di transizione Q $\times\Sigma \rightarrow$ Q
- **D** $\in$ **Q** è lo stato iniziale
- **F** = **f1, f2, ... , fn** è un insieme finito di stati finali

2.3.3

Il lexer

I linguaggi di programmazione sono fatti di parole-chiave e costrutti definiti in modo preciso, scopo principale del processo di compilazione è quello di tradurre le istruzioni ad alto livello del linguaggio di programmazione nelle istruzioni di basso livello del linguaggio macchina o di una macchina virtuale che rappresenta l'architettura per l'esecuzione.

Un programma sorgente viene scritto mediante un editor che può produrre un file fatto di istruzioni e costrutti consentiti dal linguaggio di programmazione in uso. Il testo reale effettivo del file viene scritto usando caratteri di un particolare insieme o sottoinsieme di un set, pertanto un sorgente può essere considerato come un flusso di caratteri terminanti con un marcatore di fine file EOF (End Of File).

Un file sorgente viene inviato come flusso ad un lexer carattere per carattere da una qualche interfaccia di input. Il lavoro del lexer è impacchettare il flusso di caratteri (altrimenti) insignificante in gruppi che, elaborati dal parser, acquistano significato. Ogni carattere o gruppo di caratteri raccolto in questo modo viene detto token. I token sono componenti del linguaggio di programmazione in questione come parole chiave, identificatori, simboli ed operatori. Il lexer rimuove commenti e spaziatura dal programma, ed ogni altro contenuto che non ha un valore semantico per l'interpretazione del programma. Il lexer converte il flusso di caratteri in un flusso di token che hanno un significato individuale stabilito dalle regole del lexer stesso.

In questo progetto, come anticipato, il lexer viene fornito ad **ANTLR** e può essere dichiarato ed utilizzato in questo modo in **Java**:

```
janusLexer lexer = new janusLexer(<InputStream>);
```

dove la classe **janusLexer**, generata da **ANTLR**, prende in input tutto il flusso dati, ovvero il codice sorgente. Vengono poi generati i tokens utilizzando la classe **CommonTokenStream** nel modo seguente:

```
CommonTokenStream tokens = new CommonTokenStream(lexer);
```

Il flusso di token generato dal lexer viene ricevuto in ingresso dal parser:

```
janusParser parser = new janusParser(tokens);
```

utilizzando la classe generata **janusParser**. Un lexer di solito genera errori riguardanti le sequenze dei caratteri che non riesce a mettere in corrispondenza con gli specifici tipi di token definiti dalle sue regole.

2.3.4

Il parser

Un lexer raggruppa sequenze di caratteri che riconosce nel flusso dotate di un valore semantico individuale. Esso non considera il loro valore semantico nel contesto dell'intero programma poiché questo è dovere del parser. I linguaggi sono descritti attraverso grammatiche. Come anticipato, una grammatica determina esattamente quello che definisce un dato token e quali sequenze di token sono considerate valide. Il parser organizza i token che riceve in sequenze ammesse definite della grammatica del linguaggio. Se il linguaggio viene usato esattamente come definito nella grammatica, il parser sarà in grado di riconoscere i pattern che costituiscono specifiche strutture del discorso e raggrupparle insieme opportunamente. Se il parser incontra una sequenza di token che non corrisponde a nessuna delle sequenze consentite, solleverà un errore ed in alcuni casi proverà a recuperarlo facendo alcune assunzioni sulla natura di tale errore e sulle informazioni trovate durante il parsing.

Il parser controlla che il token sia conforme alla sintassi del linguaggio definita dalla grammatica. Di solito il parser converte le sequenze di token per le quali è stato costruito facendole corrispondere ad un altra forma come un albero sintattico astratto (AST). Un AST è più facile da tradurre in un linguaggio obiettivo poiché contiene implicitamente informazioni aggiuntive per la natura della sua struttura. Indubbiamente, creare un AST è la parte più importante nel processo di traduzione del linguaggio.

Il parser genera una o più tavole dei simboli che contengono informazioni riguardanti i token incontrati, ad esempio se il token è il nome di una procedura o meno oppure se aveva qualche valore specifico. Le tavole dei simboli sono impiegate nella generazione del codice oggetto e nel controllo sui tipi, ad esempio, in modo che un intero non venga assegnato ad una stringa ecc. ANTLR usa le tavole dei simboli per velocizzare il processo di trovare la corrispondenza (match) dei token per il fatto che un intero viene mappato con un particolare token, invece di cercare la corrispondenza con la stringa della sua rappresentazione testuale, il che è molto più veloce. Alla fine l'AST viene tradotto in un formato eseguibile, potrebbe essere anche realizzato il linking di qualche libreria, ma non essendo argomento di questa trattazione, non verrà ulteriormente trattato nel seguito.

Un parser di solito genera errori che attengono alle sequenze di token che non riesce a far corrispondere alle specifiche strutture sintattiche consentite, come stabilito dalla grammatica.

I lexer e i parser sono entrambi riconoscitori: i lexer riconoscono sequenze di caratteri, i parser riconoscono sequenze di token. Un lexer o un parser converte un flusso di elementi (siano essi caratteri o token) traducendoli in altri flussi di elementi, come i token, che rappresentano strutture più ampie o gruppi di elementi o nodi in un AST.

ANTLR permette la definizione di regole che il lexer dovrebbe seguire per tokenizzare un flusso di caratteri e regole che il parser deve usare per interpretare il flusso di token. ANTLR è in grado di generare un lexer ed un parser che si

può usare per interpretare programmi scritti in un linguaggio e tradurli in altri linguaggi e altri AST. Il progetto di ANTLR consente molta estensioni ed ha molte applicazioni nel campo dei linguaggi [2].

2.4

Concorrenza e message passing in Janus

2.4.1
Introduzione

L'approccio di Janus alla reversibilità consiste nel limitare i costrutti del linguaggio classico per renderli biettivi. L'approccio *causal-consistent* [5] alla reversibilità, consiste nel consentire di annullare qualsiasi azione, a condizione che le sue eventuali conseguenze siano annullate in anticipo. Ciò è ragionevole nei sistemi concorrenti e distribuiti, in cui molte istruzioni non possono svolgersi contemporaneamente, quindi non è possibile trovare un'ultima istruzione unica. Questo approccio è stato applicato in vari calcoli di processi, e con linguaggi funzionali concorrenti. In tutti i casi, questo approccio si è basato sul tenere traccia della storia e delle informazioni causali, per consentire la reversibilità.

Questo lavoro mira a integrare i due approcci: vogliamo usare l'approccio di *Janus*, limitando quindi il linguaggio per evitare di perdere informazioni durante il calcolo, ma applicandolo a sistemi concorrenti, mirando quindi alla reversibilità *causal-consistent*. Per fare ciò estenderemo lo *Janus* sequenziale con costrutti per la creazione di **thread**, e per la comunicazione via **message passing**.

2.4.2

Aggiunta della concorrenza

Per aggiungere un meccanismo di concorrenza, prima di tutto, abbiamo bisogno dei **threads**:

Un buon metodo, per creare e distruggere i threads in modo simmetrico è utilizzare un *fork and join*. Possiamo usare qualcosa come:

fork P_1 **and** P_2 **join**

Il quale esegue P_1 e P_2 in parallelo. Si considera di avere uno stato globale unico per evitare problemi legati all'allocazione e alla deallocazione dello stato.

Oltre a threads, abbiamo bisogno di un meccanismo di scambio di messaggi, uno sincrono ed uno asincrono. Se uno è asincrono, una possibilità è quella di utilizzare delle porte, ovvero code di messaggi. In questo caso la cosa più semplice è avere un set di porte che viene definito staticamente. La funzione di invio può essere:

```
send(x,p)
```

che accoda il valore della variabile x alla porta p. Dopo l'invio il valore di x dovrà essere uguale a zero.

Per quanto riguarda la ricezione si può utilizzare la forma

```
receive(y,p)
```

che consente di rimuovere un valore dalla coda indicizzata dalla porta p per poi memorizzarlo nella variabile y. Il valore della variabile y, prima della ricezione dovrà essere uguale a zero. La funzione di *receive* è bloccante nel caso la coda sia vuota.

Le procedure di *send* e *receive* possono anche essere viste come uno scambio tra la variabili, ed il valore è la coda stessa. Se si utilizza la modalità sincrona, l'invio e la ricezione potrebbero interagire direttamente senza code.

Capitolo 3

Il compilatore

L'obiettivo del progetto proposto era quello di creare uno strumento che preso un file di testo scritto in linguaggio *Janus*, potesse leggero ed eseguire il programma corrispondente in modo corretto. Al progetto originale *Janus* doveva essere aggiunto un meccanismo di **message passing** ed un sistema con l'utilizzo di threads per fornire una **programmazione concorrente**.

In questa sezione si vuole presentare lo sviluppo del compilatore per Janus, in particolare la scrittura della grammatica, i linguaggi utilizzati e le scelte implementative.

3.1 Estensione del linguaggio: le struct

Durante lo sviluppo del compilatore si è resa necessaria l'aggiunta di un tipo di dato composto, le struct. Più avanti verrà chiarito il modivo di questa implementazione.

Come sappiamo, una struttura dati è una dichiarazione di tipo di dati composti che definisce un elenco di variabili, dove l'accesso alle diverse variabili è consentito tramite un singolo puntatore o dal nome dichiarato dalla struttura. Viene quindi aggiunto in *Janus* il tipo di dato **struct**, il quale come per definizione può contenere al suo interno diverse variabili. La grammatica della **struct** è la seguente:

```
struct : 'struct' typeName paramDeclare 'end' ;
```

dove le parole riservate *struct* ed *end* sono i delimitatori rispettivamente di inizio e fine, **typeName** è il nome della struttura scelto dall'utente e **paramDeclare** è una lista di parametri che possa essere riconosciuta da *Janus*.

Una struttura dati in *Janus* può essere dichiarata in questo modo:

```
struct data
    int a
    int b
    ...
    int y[n]
end
```

La struttura dati può quindi essere istanziata in questo modo:

```
struct data mystruct
```

Le **struct**, in questa implementazione del compilatore per *Janus* [6], sono dichiarabili solo in modo globlale.

3.2

Grammatica

ANTLR genera analizzatori lessicali. La prima cosa da fare è stata la definizione della grammatica, ricordando che una grammatica è composta da un insieme finito di simboli non terminali, un insieme finito di simboli terminali, un insieme finito di produzioni e un elemento iniziale non terminale, seguendo le definizioni del generatore di parser ho potuto definere una grammatica per *Janus*. Come citato nella sezione di ***Background***, *Janus* è costituito da *procedure*, la grammatica doveva quindi saper risconoscere una sequenza di procedure, definita nel modo seguente:

```
grammar janus;

program    : procedures? mainFun;
procedures : procedure | procedure procedures;
```

dove **grammar** è la parola chiave per definire la grammatica, seguita dal nome. Subito sotto due simboli non terminali; **program**, la cui produzione indica che il programma può essere composto da una serie di procedure (il carattere *'?'* in ANTLR indica che il simbolo può anche non essere presente) ed una funzione main obbligatoria. Il non terminale **procedures** indica la presenza di una o più procedure.

La funzione di *main*, non è stata inserita tra le **procedure** perchè si comporta in modo diverso rispetto alle funzioni *Janus*, essa ad esempio non prende parametri in input e solo in questa possono essere dichiarate variabili globali. Un altro motivo per cui è stata dichiarata separatamente è proprio per la biettività di *Janus*. Per tutte le procedure dovranno esserci le corrispondenti funzioni di *forward* e *reverse*, tranne ovviamente per il *main* che deve restare l'unica funzione principale.

Altri estratti della grammatica verranno presentati in questo capilo, mentre quella completa è disponibile su github [6] a nome "janus.g4".

Di seguito un estratto riguardante i costrutti e le operazioni principali.

```
loopConstructor : 'from' condition doExp? loopExp?
                    'until' condition
                ;

doExp : 'do' block
      ;

loopExp : 'loop' block
        ;

ifConstructor : ifExpression fiExpression
              | ifExpression elseExpression fiExpression
```

```
                ;

ifExpression : 'if' '('? condition ')'? 'then' block
             ;

elseExpression : 'else' block
               ;

fiExpression : 'fi' condition
             ;


localParamDeclare : local type variableName array?
                    (opcondition value)?
                  ;

local : 'local'
      | 'delocal'
      ;

condition : '('? value opcondition value ')'?
          | condition logicalExpression condition
          ;
```

Il non terminale **block**, la cui produzione non è presente, può essere una qualsiasi produzione dei costrutti o delle operazioni presenti nella grammatica concesse all'interno di un blocco istruzioni.

3.3

Generazione file

Una volta scritta la grammatica è stato possibile darla in input ad ANTLR. Il generatore di parser è stato così in grado di generare gli oggetti necessari per la stesura delle regole del linguaggio. Come detto in precedenza ANTLR genera il parser, lexer e l'albero sintattico astratto, oltre a questo crea altre classi utili per la scrittura delle regole del linguaggio. Viene generata una classe per ogni simbolo non terminale della grammatica, con all'interno metodi che possono essere richiamati in fase di parsing. Ad esempio, per quanto riguarda l'estratto della grammatica Janus, verrà creato il file *janusParser$ProceduresContex.class*, dove al suo interno troveremo i seguenti metodi:

- *enterRule()*
- *exitRule()*
- *getRuleIndex()*

In *enterRule* indichiamo cosa fare quando il parser, scorrendo l'albero sintattico, incontra un token, mentre in *exitRule* indichiamo cosa fare quando il parser esce dal nodo. La funzione *getRuleIndex* ritorna un indentificatore univoco della regola.

3.4

Come lavora il compilatore Janus

Come tutti i programmi *Java*, il compilatore *Janus* ha un file principale con la funzione *main*. Per prima cosa l'interprete verifica che il file in input esista ed abbia estensione **.jan**. Una volta verificato l'input viene creata la classe di generazione del codice. Poichè, come anticipato, *Janus* ha una sintassi molto simile al *C/C++*, si è voluto seguire l'approccio della versione più recente di *Janus* [4], e generare file in linguaggio *C++*. Di questo si occupa proprio una classe chiamata **genereteCode.java**, la quale si dedica a creare il file sorgente *C++* e a scrivere su di esso a seconda dei metodi che vengono richiamati. Fondamentalmente la classe genereteCode ha almeno un metodo per ogni regola che viene richiamata. Ognuno di questi metodi, dato il valore preso in input, lo legge e ne produce il corrispondente codice *C++*.

Le prime istruzioni da inserire nel sorgente *C++* sono le inclusioni. Gli **include** fondamentali che inserisce il programma sono:

```
#include <stdio.h>
#include <assert.h>
#include <math.h>
```

altre inclusioni, vengono aggiunte nel caso il parser incontri token riguardanti il *message passing* oppure il costrutto *fork and join* che vedremo in dettaglio più avanti.

A questo punto il programma inizializza il **lexer**, prendendo come input lo stream del sorgente che gli viene passato. Da questo vengono create delle strutture dati, ovvero i **token**. I **token**, come abbiamo visto nella sezione di **Background**, saranno presi in input dal **parser**, naturalmente dopo che quest'ultimo è stato dichiarato.

Quando si incontra il token su *procedure*, la classe che richiamiamo *enterProcedureContext()* passa come input alla generateCode il nome della funzione, letto dal **parser**, così da poter scrivere il codice corrispondente:

```
void ProcedureName_forward();
void ProcedureName_reverse();
```

per l'intestazione della funzione, e:

```
void ProcedureName_forward(){
void ProcedureName_reverse(){
```

Sarà poi il metodo *exitProcedureContext()* a chiudere la funzione. Ovviamente se la funzione prende argomenti saranno richiamate altre classi, e chiamati altri metodi per dichiarare i parametri. Ricordando che i valori sono passati per riferimento, da una dichiarazione *Janus*, di questo tipo:

```
procedure fibonacci(int x1, int x2, int n)
```

si ottiene il codice *C/C++* generato:

```
void fibonacci_forward(int &x1, int &x2, int &n);
```

3.5

Parsing tree

L'albero sintattico generato da ANTLR viene visitato una prima volta per la dichiarazione delle funzioni. In questa prima passata si va alla ricerca delle sole procedure all'interno del sorgente *Janus*. Proprio perchè è necessaria prima la sola dichiarazione delle funzioni, nella visita vengono analizzati solo i nomi delle funzioni ed i relativi argomenti se presenti. Ovviamente questa visita non passa per la funzione di *main*, poichè essa non ha bisogno del prototipo.

Le visite prinpicali sull'albero, e forse un po' più complesse, sono quelle che richiamano le classi **jWriterF.java** e **jWriterB.java**, rispettivamente per la funzione di *forward* e quella di *reverse*. La classe per generare la funzione di *forward* in realtà è abbastanza semplice. Come anticipato, ogni volta che si incontra un **token**, la classe generalmente richiama due metodi, uno di entrata ed uno di uscita. Una volta analizzate le stringhe in input ed applicate le regole

necessarie alla conversione, queste vengono usate per la generazione del codice. Ad esempio se abbiamo un istruzione di incremento come:

```
a += b
```

la sua conversione sarà praticamente 1:1, dove si rende necessaria solo l'aggiunta del separatore di istruzioni ";" . Mentre una dichiarazione come:

```
local int a
    ...
    ...
delocal int a = b
```

viene convertita in:

```
int a = 0;
    ...
    ...
assert(a == b);
```

La classe *jWriterF.java* si occupa inoltre di tenere il conto delle procedure, dei costrutti e dei tabulati di indentazione, per una lettura più agevole del codice in uscita. Un costrutto in cui vale la pena soffermarsi è il *loop*. Come si è visto nel capitolo di Background, il *loop* ha la seguente sintassi:

```
from [Cond0]
do
  [Stmt1]
loop
  [Stmt2]
until [Cond1]
```

la cui traduzione in pseudocodice è:

```
assert(Cond0)
  [Stmt1]
while(!(Cond1))
  [Stmt2]
  assert(!(Cond0))
  [Stmt1]
```

dove si vede che la **Cond0** ed il primo **statement** vengono ripetuti due volte. Il metodo che si occupa delle regole del *loop*, in caso vi sia l'espressione *do*, non salva le istruzioni in una qualche struttura o array per poi rigenerare il codice, perchè non sappiamo in anticipo quante istruzioni, o magari quanti costrutti annidati abbia lo **statement 1**, ma fa una secoda visita del sottoalbero radicato della *do expression* richiamando la propria classe ricorsivamente. Mentre la **Cond1**, essendo un istruzione condizionale, viene salvata in un'unica stringa e convertita in fase di generazione del codice.

La classe che si occupa di scrivere le funzioni di *reverse* si deve comportare

in modo inverso rispetto alla *forward*. Nonostante ANTLR metta a disposizione la possibilità di una post-visita, ovvero una visita dell'albero partendo dalle foglie e risalendo alla radice, si è scelto di utilizzare un meccanismo di ricorsione per ogni regola. Si è scelto di scrivere la ricorsione perchè altrimenti si sarebbe ottenuto un programma completamente al contrario, mentre alcuni nodi dell'albero non devono essere invertiti. Ad esempio le condizioni del costrutto *if then else* devono essere invertite, ma non le parole chiave del costrutto stesso. Un codice *Janus* che utilizza un *if then else* potrebbe essere il seguente:

```
if a = 0 then
     ...
     ...
else
     ...
     ...
fi a = b
```

Il codice in uscita, nel caso della *forward*, risulta in questo modo:

```
if (a == 0) {
     ...
     ...
     assert (a == b)
}else{
     ...
     ...
     assert (!(a == b))
 }
```

Mentre nel caso della *reverse* risulta:

```
if (a == b) {
     ...
     ...
     assert (a == 0)
}else{
     ...
     ...
     assert (!(a == 0))
 }
```

Dove si nota che la codizione di *fi* diventa un asserzione nell'*if* e la sua negata nell'*else* nel caso della forward. Nel caso della *reverse* vengono prima invertite le codizioni tra l'*if* e il *fi*, poi utilizzate come condizioni per gli assert.

Quando si incontrano più procedure queste non devono essere invertite nella sequenza di inizializzazione, ma solo le loro istruzioni. Quello di cui in pratica si occupa la classe *jWriterB.java* è di leggere il token, se questo è una procedura si deve generare il codice per dichiararla e gli eventuali parametri, dopo di che

viene fatta una visita nel solo sottoalbero della procedura in questione, passando come argomenti prima il figlio destro e poi quello sinistro, così ricorsivamente fino a tornare alla radice.

Il codice in uscita della dichiarazione di una variabile locale, visto in precedenza, nel caso della *reverse*, risulta quindi essere:

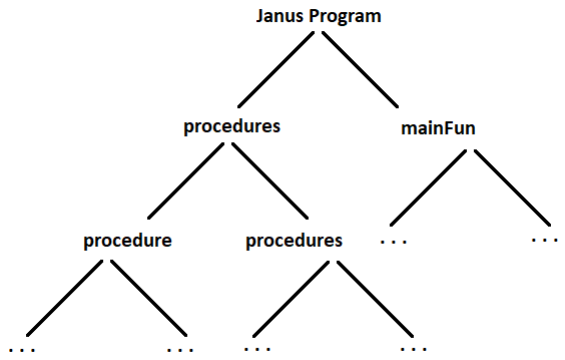
```
int a = b;
    ...
    ...
assert(a == 0);
```

mentre quello del *loop*, per completezza, nel caso della *reverse* risulta essere:

```
assert(Cond1)
  [Stmt1]
while(!(Cond0))
  [Stmt2]
  assert(!(Cond1))
  [Stmt1]
```

A questo punto il programma ha quasi terminato, manca soltanto la funzione *main*, la quale per scelta progettuale è separata dalle altre procedure, quindi avrà una procedura di visita tutta sua nel solo suo albero radicato, chiamando la classe *jWriterF.java*. Chiaramente la procedura principale ha comportamenti diversi dalle altre, ad esempio abbiamo visto che in *Janus* è possibile la dichiarazione di variabili globali esclusivamente nel *main*, per questo la classe *jWriterF.java* si comporterà di conseguenza.

Un estratto dell'albero sintatto di Janus è il seguente:



dove le prime visite dell'albero per il prototipo delle funzioni e per le dichiarazioni vere e proprie partono dal token **procedures**, mentre l'ultima ovviamente inizia dal token **mainFun**.

Capitolo 4

Implementazione del meccanismo di concorrenza

4.1
Fork and join

Per l'aggiunta del costrutto *fork and join*, prima di tutto si è dovuto aggiungere tale meccanismo alla grammatica. Quello che tale costrutto deve fare è eseguire due blocchi di istruzioni in concorrenza.

La produzione, risulta essere:

```
forkandjoin : 'fork' (typeName structName)?
              block 'and' block 'join';
```

dove tra gli apici singoli troviamo le parole chiave, i non terminali ***typeName structName*** sono rispettivamente il tipo ed il nome della struttura dati che possiamo passare come argomento. Infine ***block***, che può essere una sequenza di istruzioni, oppure un costrutto, in generale qualsiasi cosa concessa da *Janus* che può stare all'interno di una **procedura**, anche ovviamente una nidificazione di *fork and join*.

Una volta definita la grammatica è necessario definire le regole del costrutto in questione. Diversamente dall'implementazione dei costrutti visti fino ad ora, il *fork and join* richiede dei meccanismi da aggiungere, ed essendo questo progetto un programma che genera codice *C++*, questi meccanismi devono essere aggiunti nel codice in uscita.

Per l'utilizzo di threads si è usata la libreria ***pthread*** di *C/C++*, questa sarà inclusa in fase di generazione codice se durante la procedura di parsing viene incontrato una *fork and join*. Oltre a questo, come richiesto dalla libreria, viene aggiunta la dichiarazione dei threads:

```
pthread p1, p2;
```

Vengono definiti due threads per ogni costrutto *fork and join* che viene visitato nell'albero sintattico. Il nome che viene assegnato al thread è automatico, inzia con la lettera p a cui segue un intero sequenziale.

In C/C++ i threads devono essere definiti come puntatori a funzioni, i quali quando chiamati dallo scheduler mandano in esecuzione la funzione puntata. Ora con questo e seguendo i criteri di *Janus*, vanno definite due funzioni per ogni thread, quindi quattro funzioni per ogni **fork and join**. Il risultato dei prototipi delle funzioni generati da una procedura chiamata *test()*, con all'interno il costrutto per la concorrenza sarà in questo modo:

```
void test_forward();
void test_reverse();
void *program_1_forward(void *arg);
void *program_2_forward(void *arg);
```

```
void *program_1_reverse(void *arg);
void *program_2_reverse(void *arg);
```

dove come visto precedentemente, **test** è il nome della procedura ed ogni thread ha il suo prototipo per eseguire la *forward* e la *reverse*. I threads vengono numerati sequenzialmente, in caso di *fork and join* nidificate i threads vengono numerati a partire da quelli più esterni, come mostrato quì sotto:

```
fork      ---> program_1
   fork   ---> program_3
     ...
   and    ---> program_4
     ...
   join
and       ---> program_2
   fork   ---> program_5
     ...
   and    ---> program_6
     ...
   join
join
```

Dopo la dichiarazione dei prototipi vengono dichiarate le funzioni con i relativi blocchi di istruzioni. Questo viene fatto facendo una visita in più nel sottoalbero del nodo costituito dalla *fork and join*. La prima visita che viene fatta sull'**AST** si occupa della dichiarazione dei threads e di tutte le istruzioni al suo interno. Una seconda visita viene fatta per la chiamata alla *pthread_create*, dove il thread viene solo creato e quindi in questa visita si salta tutto il sottoalbero del costrutto, poichè le istruzioni erano già state prelevate e perchè tali istruzioni non devono essere presenti all'interno della **procedura** che inizializza il costrutto *fork and join*.

Per rendere il concetto più chiaro viene mostrato il primo programma che ha utilizzato il costrutto *fork and join* con il relativo codice *C++* in uscita. Il programma è il seguente, dove viene omessa la funzione di *main*:

```
1   procedure first_fork()
2       fork
3           local int a
4           a += 1
5           print a
6           a -= 1
7           delocal int a = 0
8       and
9           local int b
10          b += 2
11          print b
```

```
12                  b -= 2
13                  delocal int b = 0
14              join
```

Come è facile vedere, il programma non fa altro che dichiarare due variabili, incrementarle per poi stamparne il valore e deallocarle dopo aver decrementato del valore precedente. In questo caso il programma funziona allo stesso modo in entrambe le direzioni, dove nel caso della *forward* segue semplicemente la sequenza delle istruzioni, nel caso della *reverse* la deallocazione diventa un istruzione di allocazione uguale a zero e l'istruzione *a -= 1* diventa *a += 1*, vice versa per l'istruzione a riga 4.

Il codice uscente delle sole funzioni dei threads, è quindi questo:

```
1  void *program_1_forward(void *arg){
2          int a = 0;
3          a += 1;
4          printf("%d\n",a);
5          a -= 1;
6          assert(a == 0);
7  }
8  void *program_2_forward(void *arg){
9          int b = 0;
10         b += 2;
11         printf("%d\n",b);
12         b -= 2;
13         assert(b == 0);
14 }
15 void *program_1_reverse(void *arg){
16         int a = 0;
17         a += 1;
18         printf("%d\n",a);
19         a -= 1;
20         assert(a == 0);
21 }
22 void *program_2_reverse(void *arg){
23         int b = 0;
24         b += 2;
25         printf("%d\n",b);
26         b -= 2;
27         assert(b == 0);
28 }
```

La funzioni di *forward* e *reverse* della procedura di esempio **first_fork**, devono solo dichiarare i threads e passare alla **pthread_create** i parametri corretti. Quindi per l'esempio precedente il codice in uscita della sola *forward* è il seguente:

```
1  void test_forward(){
```

```
2           pthread_create(&p1,NULL,program_1_forward,(void *)NULL);
3           pthread_create(&p2,NULL,program_2_forward,(void *)NULL);
4   }
```

La funzione di *reverse* non viene mostrata perchè assolutamente identica a quella di *forward*, non è necessario invertire le due istruzioni di creazione dei thread. La **pthread_create** per passare parametri alla sua funzione di riferimento usa un puntatore. Proprio per questo motivo si è cercato un modo per passare comodamente valori ai threads, poichè non sapendo in anticipo quanti paramatri saranno usati all'interno della procedura. Inizialmente si era pensato di fare una copia di tutte le variabili all'interno della funzione ed inserirle in una struttura dati, ed infine a seconda di come viene gestita la memoria, aggiornare i valori alla terminazione dei thread. Una soluzione del genere risultava complicata, in più dovevano essere considerati anche i parametri passati alla procedura, oltre quelli dichiarati al suo interno, e tra questi ultimi solo quelli dichiarati prima della *fork and join*. Così si è deciso di dare a *Janus* la possibilità di dichiare strutture dati, e all'utente di decidere a quali variabili i threads possano accedere e a quali no.

Come abbiamo visto nella grammatica del *fork and join* è possibile passare una struttura dati come argomento al costrutto concorrente. Il tipo ed il nome della struttura vanno messi dopo la parola chiave **fork**. Ad esempio se vogliamo passare la struttura **mystruct** a due threads la scriviamo in questo modo:

```
fork data mystruct
     ...
     ...
and
     ...
     ...
join
```

cosi che entrambi i thread possano accedere alle variabili della struttura. Nel codice in uscita, come ci si aspetta verrà passato ai threads un puntatore alla struttura dati e quindi la **pthread_create** risulta essere modificata in fase di generazione del codice in questo modo:

```
pthread_create(&p1,NULL,program_1_forward,(void *)mystruct);
```

4.2

Gestione della memoria

In questa versione di *Janus* ci sono due modi per utilizzare la memoria condivisa. Il primo è quello basato su **shared memory**, cioè quello che sostanzialmente fa il *C++*, ovvero passare la struttura per riferimento. In questo caso abbiamo la struttura dati condivisa fra i threads ma non abbiamo un meccanismo per la mutua esclusione quando, per esempio si vuole accedere alla variabile, a meno delle funzioni di *send* e *receive* citate precedentemente. Il secondo metodo,

quello di default è proprio il servizio di **message passing**, dove in generale non prevede la memoria condivisa. Per implementare questo metodo bisognava fare in modo che la struttura dati passata alla *fork and join* venisse copiata in locale dal thread. Per realizzare questo meccanismo si sono impostate delle regole che permettono di inizializzare la struttura ovunque sia necessaria e copiare al suo interno i valori impostati nella **procedura**. Ad esempio se all'interno di una procedura dichiaro la banale struttura seguente:

```
struct data
    int a
end
```

e supponiamo di passare tale struttura ad una *fork and join*. In primo luogo nel codice in uscita la struttura deve essere dichiarata prima delle funzioni, sia dei threads sia delle procedure, questo per lo scope di *C++*. La struttura viene inizializzata sia all'interno dei threads sia all'interno della **procedura** stessa, e da quest'ultima, tramite la **pthread_create**, passiamo il puntatore come argomento della funzione. Una volta che le procedure dei threads ricevono questo puntatore, tramite la funzione **memcpy** del linguaggio *C*, fanno una copia dell'intera struttura in locale e faranno riferimento sempre a quella.

La scelta del tipo di memoria, è un opzione del programma, viene impostato con un flag in fase di parsing dall'utente, questo procedimento verrà illustrato più avanti.

4.3

Il message passing

Il servizio di **message passing** richiesto per *Janus* deve fornire le funzioni di *send* e *receive*, in modo sincrono ed asincrono. Le funzioni richieste devono scrivere (o leggere) un valore da un determinata porta. La porta non è altro che una coda con le classiche operazione di **enqueue** e **dequeue**. Le porte possono essere dichiarate in modo globale nella procedura *main* utilizzando la parola riservata '*port*' seguita dal nome in questo modo:

```
port p
```

Le porte possono essere dichiarate all'interno delle procedure in modo locale utilizzando le parole riservate *local* e *delocal*, come abbiamo visto per altri tipi di dato le porte locali vengono allocate e deallocate in questo modo:

```
local port p
    ...
    ...
    ...
delocal port p
```

La coda e le sue operazione sono gestite separatamente, si è creato un *header C++* che viene incluso nel sorgente in uscita quando necessario. Questo header

chiama un sorgente *C++* di nome **queue.cpp**, il quale si occupa di tutta la gestione del meccanismo di **message passing**. Le operazioni all'interno del file sono le seguenti:

- ConstructQueue
- DestructQueue
- Enqueue
- Dequeue
- isEmpty
- asend
- arcv
- ssend
- srcv

La **ConstructQueue** viene chiamata per inizializzare la struttura dati che rappresenta la coda. Tale struttura è formata dai campi **size**, **limit** e due puntatori, uno alla testa (**head**) e uno alla coda (**tail**). Il campo **size** tiene traccia del numero di elementi presenti in coda mentre il campo **limit** è limitatore di dimensione massima che può avere la coda, il quale se non impostato diversamente ha di default **limit = 65000**, ovvero può contenere al massimo 65000 elementi, arrivati a quel valore non vengono più inseriti elementi in coda ed il programma genera un messaggio di saturazione. I puntatori **head** e **tail**, sono puntatori a strutture dati di nome **Node**, che contengono due campi; il primo è il valore intero che si vuole memorizzare, difatti si tratta di code di interi. Il secondo è un puntatore al precedente. La funzione **ConstructQueue** ritorna un puntatore alla coda.

L'operazione **DestructQueue** non ritorna alcun valore, prende in input un puntatore di tipo coda, ne cancella tutti gli elementi sequenzialmente, ed infine libera la memoria occupata dal puntatore.

La procedura di **Enqueue** prende in input due parametri, un puntatore alla coda ed un puntatore di tipo **Node**, il quale dovrà essere aggiunto alla coda. La procedura ritorna **FALSE** se gli argomenti non sono corretti o se si è superato il valore **limit**, altrimenti si verifica se la coda è vuota o sono presenti altri elementi. A seconda di questi ultimi due casi, si procede nell'aggiungere il nodo alla struttura dati ed aggiornare opportunamente tutti i puntatori.

La procedura di **Dequeue** copia il nodo che trova in prima posizione, il quale verrà usato come valore di ritorno. Se la coda è vuota torna un valore nullo, altrimenti cancella il nodo e aggiorna i relativi puntatori.

La funzione **isEmpty** verifica se la coda non contiene elementi, in questo caso ritorna il valore **TRUE**, altrimenti se presente almeno un elemento ritorna **FALSE**.

Le funzioni **asend** e **arcv** vengono utilizzate per il meccanismo asincrono, mentre le funzioni **ssend** e **srcv** per quello sincrono.

Per scrivere e leggere la coda abbiamo bisogno di una sezione critica dove il thread può operare sulla struttura dati evitando di scrivere su un valore in cui sta operando un altro thread. Bisogna assicurasi anche di evitare il più possibile situazioni di **deadlook** e **starvation**, per questo si è scelto di usare i semafori come meccanismo di sincronizzazione.

Il primo meccanismo che si vuole mostrare è quello asincrono, vengono utilizzati trè semafori per ogni coda che viene inizializzata, ovvero ogni volta che viene dichiarata una porta in *Janus*. I semafori utilizzati sono i seguenti:

- mutex
- read
- write

tutti seguiti dal nome della porta per essere differenziati. Il semaforo **mutex** è un semaforo binario, viene inizializzato con valore uno, poichè solo un processo per volta può avere la mutua esclusione. Il semaforo **read** viene inizializzato con valore **limit** ed il semaforo binario **write** viene inizializzato con valore zero.

Per incrementare e decrementare il valore del semaforo si utilizzano le funzioni **sem_wait()** e **sem_post()** della libreria **semaphore** del linguaggio *C* di cui si vuole ricordare brevemente il comportamento. La **sem_wait()** decrementa (**locks**) il semaforo preso in input. Se il valore del semaforo è maggiore di zero allora decrementa di uno e la funzione ritorna immediatamente. Se il valore corrente del semaforo è zero il chiamante si blocca fino a quado non è possibile eseguire il decremento, ovvero fino a quando il valore non supera lo zero. La **sem_post()** incrementa (**unlocks**) il semaforo passato in input. Se il valore corrente del semaforo diventa maggiore di zero, un processo o thread, che è bloccato nella chiamata alla **sem_wait()**, viene svegliato per procedere.

La funzione di invio asincrono **asend**, il cui prototipo è il seguente:

```
void asend(int data, Queue *pQ,sem_t *mutex,
           sem_t *write, sem_t *read);
```

prende come argomenti il valore che si deve inserire in coda, il puntatore alla coda di riferimento e tutti e trè i semafori. Il corpo della funzione è il seguente:

```
sem_wait(read);
sem_wait(mutex);
    // CRITICAL SECTION
    // call enqueue function
    // END CRITICAL SECTION
sem_post(mutex);
sem_post(write);
```

dove viene prima decrementato il valore del semaforo **read** e poi quello **mutex**, quindi per quello che sappiamo dal linguaggio *C* se il valore del semaforo rimane maggiore-uguale a zero il processo non viene bloccato. Una volta entrati nella **critical section** viene chiamata la funzione **Enqueue** con i giusti argomenti, infine viene rilascita la mutua esclusione ed incrementato il valore del semaforo **write**.

La funzione di ricezione asincrono **arcv** ritorna il valore in testa alla coda e, come si vede dal prototipo, prende gli stessi parametri della **asend** a meno del valore **data**.

```
int arcv(Queue *pQ, sem_t *mutex,
            sem_t *write, sem_t *read);
```

La procedura di ricezione dichiara la variabile di ritorno su cui copiare il valore preso dalla testa della coda, una volta prelevato il valore con la **Dequeue**, libera la memoria occupata dal puntatore del nodo passato in ingresso. La sezione critica nel caso di ricezione viene gestita in questo modo:

```
sem_wait(write);
sem_wait(mutex);
// CRITICAL SECTION
// call dequeue function
// END CRITICAL SECTION
sem_post(mutex);
sem_post(read);
```

dove l'ingresso alla **critical section** viene gestito dai due semafori binari.

Per quanto riguarda la gestione del meccanismo di **message passing** sincrono, come anticipato si utilizzano le funzioni **ssend** e **srcv**. Per il sincronismo serve un meccanismo che blocchi il processo *sender* dopo aver inviato il messaggio, e venga risvegliato dopo che il processo in ricezione ha tolto il nodo dalla coda. Per questo si è scelto di utilizzare un ulteriore semaforo solo nel caso in cui si utilizzi il meccanismo sincrono. Questo semaforo chiamato **sync** è inizializzato a zero e viene passato come argomento aggiuntivo alle funzioni **ssend** e **srcv**. I prototipi molto simili ai precedenti, risultano i seguenti:

```
void ssend(int data, Queue *pQ,sem_t *mutex,
              sem_t *write, sem_t *read, sem_t *sync);

int srcv(Queue *pQ, sem_t *mutex, sem_t *write,
              sem_t *read, sem_t *sync);
```

Anche le funzioni risultano molto simili tra loro, infatti la funzione **ssend** rispetto alla **asend**, alla fine delle operazioni di **Enqueue** e dopo aver incrementato i semafori **mutex** e **write** aggiunge l'istruzione

```
sem_wait(sync);
```

e questo vuol dire che il processo si mette in attesa, perchè dato il semaforo **sync** inizializzato a zero la **sem_wait** blocca il processo. Come per l'invio anche la ricezione si comporta allo stesso modo.

La funzione **srcv**, una volta rilasciata la mutua esclusione, incrementa il semaforo **sync** per comunicare al mittente che il messaggio è stato ricevuto.

4.4

Possibili scenari di sincronizzazione

Si vogliono mostrare alcuni possibili scenari nel meccanismo di **message passing** partendo dalla comunicazione asincrona.

Supponiamo due processi **A** e **B**, dove **A** vuole inviare un valore a **B** tramite una qualche porta. I valori dei semafori all'avvio del programma sono:

- mutex = 1
- read = limit
- write = 0

il programma potrebbe comportarsi in questo modo:

- **A** parte per primo ed esegue la **wait**, prima su **read** poi su **mutex**
- **B** si mette in attesa sulla prima **wait**, trova il valore di **write** uguale a 0
- **A** lascia la sezione critica, ed esegue la **post** prima su **mutex** poi su **write**
- **B** può procedere perchè prima trova **write** ad 1 e successivamente **mutex** ad 1
- **B** esce dalla sezione critica, incrementa **mutex** e **read**

Si vuole mostrare ora un altro scenario dove si intrecciano di più le istruzioni:

- **A** arriva per primo ed esegue la **wait** su **read**
- **B** arriva subito dopo e prova ad eseguire la **wait** su **write**, ma si blocca trovandolo a 0
- **A** prosegue sino alla fine, dove l'ultima istruzione è la **wait** su **write**
- **B** prosegue anche lui come precedentemente

Avendo n processi che inviano ed n processi che ricevono, il semaforo binario **mutex** permette l'accesso alla critical section un task per volta, questo soprattutto per l'ordine in cui i semafori vengono incrementati e decrementati, una **send** non fa mai una **wait** sul semaforo **write**, come una **receive** non fa una **post** sulla **write**. I semafori sono alternati proprio per fare in modo che solo

un task per volta entri nella sezione critica, e che una funzione non modifichi il valore di un semaforo che non sia di sua competenza.

Con n processi sender ed m processi receiver, se non si presta attenzione al codice *Janus* potrebbe esserci **starvation**, ovvero alcuni processi potrebbero non aver mai accesso alla risorsa interessata, perchè magari già prelevata da un altro task. Per questo ci vengono in aiuto le **porte**, con la possibilità di dichiararne più di una, si possono inviare direttamente i pacchetti al processo interessato. In alternativa si usa il meccanismo di message passing sincrono, di cui vediamo alcuni scenari possibili.

Per avere un servizio sincrono, come visto in precedenza si utilizzano le funzione **ssend** e **srcv**. Con tale meccanismo si rende necessaria l'aggiunta di un semaforo di nome **sync** per inviare un segnare di conferma ricezione. I semafori inizialmente avranno questa configurazione:

- mutex = 1
- read = limit
- write = 0
- sync = 0

il semaforo sync inizializzato a zero non permette al primo processo che vuole fare una **wait** di andare oltre, questo perchè il primo processo sarà proprio colui che invia. Come prima vediamo uno scenario dove un task **sender**, identificato come **A**, vuole inviare un valore ad un altro task (**receviver**) chiamato **B**.

- **A** arriva prima ed esegue entrambe le **wait** su **read** e **mutex**
- **B** si blocca sul semaforo **write** con una **wait**
- **A** nella sezione critica, mette il valore interessato nella coda e fa una **post** prima su **mutex** poi su **write**
- **A** fa una **wait** su **sync**, trovandolo a zero si blocca
- **B** procede decrementando **write** e **mutex** ed entrando in sezione crititca
- **B** rilascia **mutex** ed incrementa **read**
- **B** fa un **post** su **sync** e termina
- **A** può eseguire la **wait** su **sync** e termina

In questo esempio si nota l'importanza del semaforo binario **sync** inizializzato a zero, con un valore maggiore il processo **A** avrebbe terminato la sua esecuzione senza aspettare la conferma di ricezione da parte di **B**.

Si può vedere che con più threads l'accesso alla coda rimane esclusivo, e che

la terminazione dei tasks sia gestita correttamente dal semaforo **sync**, si ricorda che i semafori usati dalla libreria **semaphore.h** utilizzano la modalità FIFO. Per la mutua esclusione vale lo stesso motivo del meccanismo di message passing asincrono, dove il semaforo **mutex** è fondamentale per l'ingresso nella **critical section**. Per la terminazione corretta dei threads il semaforo binario **sync** blocca qualsiasi processo se la coda è vuota e viene incrementato solamente quando un elemento è stato tolto dalla coda, ovvero ricevuto dal task interessato. Supponiamo di avere n processi che generano ed inviano valori, chiamati $\mathbf{A}_1$,...,$\mathbf{A}_n$. Inoltre supponiamo di avere m processi riceventi chiamati $\mathbf{B}_1$,..,$\mathbf{B}_m$, ed un'unica porta p su cui mandare i dati.

- uno qualsiasi degli $\mathbf{A}_i$ esegue una **wait** su **read** e **mutex**
- i processi $\mathbf{B}_1$,...,$\mathbf{B}_m$ si bloccano sulla **wait** del **write**, e si mettono nella coda dei semafori
- i processi $\mathbf{A}_1$,...,$\mathbf{A}_{i-1}$,...,$\mathbf{A}_{i+1}$,...,$\mathbf{A}_n$ sono in coda su **mutex** se il valore **limit** è maggiore di **read**, altrimenti sono nella coda **read**.
- $\mathbf{A}_i$, una volta uscito dalla **critical section**, rilascia **mutex** ed incrementa **write**
- $\mathbf{A}_i$ si blocca sul **sync**
- ora viene svegliato il primo della coda tra i $\mathbf{B}_1$,...,$\mathbf{B}_m$ oppure uno tra $\mathbf{A}_1$,...,$\mathbf{A}_{i-1}$,...,$\mathbf{A}_{i+1}$,...,$\mathbf{A}_n$
- se entra uno tra i $\mathbf{B}_1$,...,$\mathbf{B}_m$, una volta letto l'elemento dalla coda, incrementa **sync** ed A_i può terminare
- se entra uno tra gli $\mathbf{A}_1$,...,$\mathbf{A}_{i-1}$,...,$\mathbf{A}_{i+1}$,...,$\mathbf{A}_n$, si comporta come $\mathbf{A}_i$ e si mette nella coda del semaforo **sync**

i tasks, uno per volta concludono le loro operazioni anche nel caso in cui tutti i processi **sender** arrivassero per primi. In questo ultimo caso i processi **receiver**, prima di terminare, incrementano **sync** e così facendo man mano sbloccano i **sender** che possono dunque terminare. Mentre se arrivassero per primi, tutti i processi **receiver** sarebbero in coda nella **write** in attesa di incremento da parte dei tasks che inviano.

4.5

La vista da Janus

Dal punto di vista del codice *Janus* tutto risulta più semplice. Come visto precedentemente una qualsiasi funzione di **message passing** non deve prendere in input la coda ed i semafori, ma solamente la variabile con valore da inviare e la porta su cui scrivere. Si vuole mostrare un esempio di codice **Janus** ed il risultato della compilazione in *C++*, per vedere come si comporta il meccanismo e la sua traduzione.

```
1  procedure mp()
2      local port p
3
4      fork
5          local int a
6          a += 1
7          print a
8          ssend(a,p)
9          delocal int a = 0
10     and
11         local int b
12         srcv(b,p)
13         print b
14         delocal int b = 1
15     join
16
17    delocal port p
18 procedure main()
19     call mp()
```

Questo semplice programma, esegue la procedura **mp()**, la quale dichiara una porta locale *p* ed inizializza un costrutto *fork and join*. Il primo thread dichiara ed incrementa di uno la variabile *a*, dopo averla stampata utilizza la funzione **ssend** per scrivere *a* sulla porta *p*. La variabile *a* infine viene deallocata se soddisfa la condizione di asserzione **a == 0**, dove si vede facilmente che viene soddisfatta, poichè qualsiasi funzione di **send** azzera la variabile che viene copiata nella coda dopo aver terminato. Il secondo thread dichiara la varibile *b* su cui scrivere e si mette in ricezione sulla porta *p*. Una volta letto il valore lo stampa e viene deallocato se uguale ad uno.

Il codice *C* in uscita viene mostrato in sezioni, la prima riguarda le dichiarazioni globali, le quali rispetto al codice *Janus* sono le seguenti:

```
1   Queue *pQ_p;
2
3   sem_t mutex_p;
4   sem_t write_p;
5   sem_t read_p;
6   sem_t sync_p;
7
8   pthread_t p1,p2;
9
10  void mp_forward();
11  void mp_reverse();
12  void *program_1_forward(void *arg);
13  void *program_2_forward(void *arg);
14  void *program_1_reverse(void *arg);
```

```
15    void *program_2_reverse(void *arg);
```

Come ci si aspettava dalla dichiarazione di una porta in *Janus*, viene dichiarato un puntatore e quattro semafori. Il puntatore di tipo **Queue** è alla coda. Dopo i semafori vengono dichiarati i threads che come abbiamo visto nella sezione della *fork and join* sono due per ogni costrutto. Dopo di che si passa ai prototipi delle funzioni, come sappiamo per *Janus* abbiamo necessità della **forward** e della **reverse** per ogni processo. I threads sono ovviamente puntatori a funzioni, i quali prendono in input un puntatore a void e nel caso verrà utilizzato per passare una struttura dati.

I thread, in questo caso molto semplici, sono fatti in questo modo:

```
17    void *program_1_forward(void *arg){
18            int a = 0;
19            a += 1;
20            printf("%d\n",a);
21            ssend(a,pQ_p,&mutex_p,&write_p,&read_p,&sync_p);
22            a = 0;
23            assert(a == 0);
24    }
25    void *program_2_forward(void *arg){
26            int b = 0;
27            assert(b == 0);
28            b = srcv(pQ_p,&mutex_p,&write_p,&read_p,&sync_p);
29            printf("%d\n",b);
30            assert(b == 1);
31    }
32    void *program_1_reverse(void *arg){
33            int a = 0;
34            assert(a == 0);
35            a = srcv(pQ_p,&mutex_p,&write_p,&read_p,&sync_p);
36            printf("%d\n",a);
37            a -= 1;
38            assert(a == 0);
39    }
40    void *program_2_reverse(void *arg){
41            int b = 1;
42            printf("%d\n",b);
43            ssend(b,pQ_p,&mutex_p,&write_p,&read_p,&sync_p);
44            b = 0;
45            assert(b == 0);
46    }
```

Le funzioni di **forward** e **reverse** si comportano come ci si aspettava in entrambi i threads. Infatti, in entrambe le direzioni i parametri passati alla **ssend** e alla **srcv** sono coerenti con il codice *Janus* scritto in precedenza.

I semafori ed i threads vengono inizializzati dalla funzione **mp()** in entrambe le direzioni:

```
48   void mp_forward(){
49            sem_init (&mutex_p,0,1);
50            sem_init (&write_p,0,0);
51            sem_init (&read_p,0,limit);
52            sem_init (&sync_p,0,0);
53            pQ_p = ConstructQueue(limit);
54
55            pthread_create(&p1,NULL,program_1_forward,(void *)NULL);
56            pthread_create(&p2,NULL,program_2_forward,(void *)NULL);
57   }
58
59   void mp_reverse(){
60            sem_init (&mutex_p,0,1);
61            sem_init (&write_p,0,0);
62            sem_init (&read_p,0,limit);
63            sem_init (&sync_p,0,0);
64            pQ_p = ConstructQueue(limit);
65
66            pthread_create(&p1,NULL,program_1_reverse,(void *)NULL);
67            pthread_create(&p2,NULL,program_2_reverse,(void *)NULL);
68   }
69
70   assert(isEmpty(pQ_p))
71
72   int main(){
73         mp_forward();
74   }
```

Le funzioni sono praticamente identiche, a parte l'argomento della **pthread_create()** che chiama la corrispondente funzione di **forward** o **reverse**. I semafori sono inizializzati con i valori visti in precedenza, e la coda viene inizializzata con valore di **limit**. Infine la funzione **main** chiama la **mp_forward()** grazie alla parola riservata **call**.

Il motivo per cui la **mp_forward()** e la **mp _reverse()** sono molto simili è perchè le istruzioni mostrate sono reversibili ma la **reverse** non è scriverla al contrario, ovvero non avrebbe senso inizializzare ed eseguire i threads prima che venga inizializzata la coda ed i semafori. Questo è stato uno dei motivi per cui la visita nell'albero sintattico di *Janus*, nel caso della **reverse**, non viene fatta con una **post-visit** di default ma valutato caso per caso, infatti in questo caso è fondamentale che l'ordine delle inizializzazioni venga rispettato in entrambe le direzioni.

Capitolo 5

Come utilizzare il compilatore

In questa sezione verrà illustrato come utilizzare il compilatore *Janus*, i prerequisiti per l'installazione ed alcuni esempi per mostrarne il funzionamento. I comandi quì mostrati sono per distribuzioni **GNU/Linux**.

5.1

Prerequisiti

Per poter utilizzare il compilatore abbiamo bisogno di alcuni strumenti:

- Java (Versione 1.6 o superiore)
- ANTLR4
- Compilatore C/C++

Una volta installata la macchina virtuale per Java ed il compilatore per *C/C++*, la versione più recente ad oggi di **ANTLR4** è ottenibile su GitHub [2]. Mentre il compilatore presentato in questa tesi è disponibile sul mio GitHub [6].

5.2

Installazione del compilatore

Una volta scaricato il progetto bisogna recarsi all'interno della main folder ed eseguire **ANTLR4** per creare la classi necessarie partendo dalla grammatica, la quale si trova all'interno della cartella /**src**. Si utilizza il comando:

```
antlr4 src/janus.g4
```

Una volta che **ANTLR4** ha creato i file necessari possiamo compilare il progetto utilizzando il **Makefile**.

5.3

Utilizzo del compilatore

Una volta compilato il progetto è possibile utilizzare il compilatore. Il programma prende come argomento il file di sorgente *Janus* con estensione ***.jan**, ed eventualmente alcune opzioni. Sono disponibili le seguenti impostazioni:

- **-m**, i threads non condividono memoria (default)
- **-s**, shared memory, i threads condividono la struttura dati

- **-j**, no join threads
- **-n**, non compila il sorgente *C++*
- **-r**, auto-run
- **-h** o –**help**, stampa un piccolo manuale

L'opzione **-m** non consente ai threads di condividere la memoria delle strutture dati, ogni thread ha una sua copia locale, questo è il meccanismo di default illustrato precedentemente. L'opzione **-s** consente ai threads di condividere la memoria della struttura dati, essi hanno un puntatore ciascuno alla stessa struttura, in questa modalità le funzione di **send** e **receive** diventano poco utili. L'opzione **-j** indica al compilatore di togliere l'istruzione *C/C++* **pthread_join()** una volta inizializzati i threads, la quale attende che il thread passato in input termini, questa risulta molto comoda nel caso si voglia utilizzare la **print**, altrimenti si rischia di non vedere nessuna stampa a terminale. Come impostazione di default, l'applicazione *Java* compila il codice una volta generato il sorgente di output, con il tag **-n** si può evitare la compilazione automanica, magari se si desidera di esiminare il sorgente *C++* prima che venga compilato. Si può invece abilitare l'esecuzione automatica del programma con il tag **-r**, il quale indica all'appliazione di eseguire il programma una volta compilato se non ci sono errori. Infine i tag **-h** e –**help** stampano a video un riassunto delle impostazione dell'applicativo per *Janus*.

Una volta esaminate le opzioni del compilatore l'utilizzo risulta immediato, infatti ad esempio possiamo utilizzare il compilatore nel seguente modo:

```
./janus.sh example.jan
```

Lo script **janus.sh**, presente all'interno della main folder del progetto, necessita dei permessi per essere eseguito, al suo interno si trova il comando per l'esecuzione della macchina virtuale **Java** con i classpath corrispondenti. Si noti che nell'esempio non è stata inserita l'opzione **-m** e nemmeno **-s**, perchè il meccanismo di message passing è di default, quindi l'istruzione precedente è equivalente a:

```
./janus.sh example.jan –m
```

mentre per la memoria condivisa va esplicitata. A questo punto si può lanciare l'eseguibile creato all'interno della main folder.

```
./example
```

Come anticipato precedentemente, è possibile definire un limite per le code dei messaggi. Di default il limite è impostato a 65000, ed è anche il valore massimo che può avere una coda. Per cambiare questo valore è sufficiente usare l'impostazione:

```
limit='val'
```

dove **val** è un valore intero. Ad esempio si può limitare la coda a 1000 messaggi in questo modo:

```
./janus.sh example.jan limit=1000 -mj
```

All'interno della main folder del progetto **Janus** c'è una cartella chiamata **concurrency**, nella quale sono contenuti i sorgenti *C++* per la gestione della concorrenza citati precedentemente, **queue.cpp** e **queue.h**. Un'altra cartella di nome **out** è presente all'interno della main folder che conterrà il codice in uscita con estensione **.cpp** ed il file oggetto del programma *Janus* che viene dato in input all'applicativo *Java*. Inoltre all'interno si trova un **Makefile** per la compilazione separata dei sorgenti *C++*, il quale prende come argomento il nome del sorgente di testo *Janus*, questo se si seleziona il tag **-n** per la non compilazione automatica. In questo ultimo caso bisogna recarsi nella cartella **out** ed eseguire il **Makefile** in questo modo:

```
make FILENAME = fibonacci
```

dove il nome a destra dell'uguale deve essere uguale al sorgente *Janus* senza estensione. Tuttavia la compilazione è gestita di default in maniera automatica dall'applicazione *Java*, la quale si occupa di passare correttamente il nome del file. Il **Makefile** consente anche di svuotare la cartella **out** con il comando **clean**:

```
make clean
```

Capitolo 6

Sperimentazione

6.1
Introduzione

In questa parte si vogliono provare alcuni semplici algoritmi concorrenti con lo scambio di messaggi tra processi. Gli esempi citati che quì sono sprovvisti di funzione main sono disponibili su github [6] con estensione .jan.

6.2

Esempio 1 - Sender-receiver

```
1      local port p
2
3      struct mystruct tosend
4
5      tosend.m += n
6
7      fork mystruct tosend
8          local int a = tosend.m
9          print a
10         asend(a,p)
11         delocal int a = 0
12     and
13         local int b = 0
14         arcv(b,p)
15         print b
16         delocal int b = tosend.m
17
18     join
19
20     tosend.m -= n
21
22     delocal port p
```

Esempio 1: single sender-receiver

Questo primo semplice esempio di sender-receiver vuole dimostrare la biettività di una funzione che permette lo scambio di informazioni tra due processi concorrenti, questa prende in input il valore da inviare, apre una *fork and join* con il valore condiviso tra i processi tramite la struttura '*tosend*' di tipo '*mystruct*', ed il primo invia questa variabile in modalità asincrona. Il processo receiver dichiara la sua variabile su cui scrivere e stampa il valore ricevuto.

Nel caso dell'esecuzione della forward il processo va a buon fine. Per quanto riguarda il processo produttore, il valore di '*a*' viene azzerato dopo la send e l'assert della delocal va a buon fine. Anche il ricevente termina con successo le sue operazioni, il valore ottenuto dalla receive deve essere uguale a quello passato alla funzione, e se lo scambio dei messaggi è andato a buon fine, l'assert della delocal termina con successo.

Nel caso della reverse, per come è scritto il codice nell' esempio 1, il processo che diventa il ricevente, cioè quello che inizia a riga 7, soddisfa l'asserzione

```
assert(a == tosend.n)
```

in fase di deallocazione. Il programma stampa i valori inviati e ricevuti, concludendo con successo questo primo esperimento. Si noti l'istruzione a riga 20, fondamentale per la procedura di reverse, dove verrà sostituita con l'istruzione **tosend.n += n** per un corretto incremento della variabile prima di avviare i threads.

6.3 Esempio 2 - Single sender-receiver with multi message

Nel prossimo esempio proviamo lo scambio di un numero arbitrario di messaggi tra due processi in modo concorrente, utilizzando il construtto *from-until* per l'iterazione.

```
1  procedure second(int v, int l)
2      local port p
3
4      struct mystruct tosend
5
6      tosend.m += v
7      tosend.n += l
8
9      fork mystruct tosend
10         local int a = 0
11         local int i = 0
12         from i = 0 loop
13             i += 1
14             a += tosend.m
15             print a
16             asend(a,p)
17         until i = tosend.n
18         delocal int a = 0
19         delocal int i = tosend.n
20     and
21         local int b = 0
```

```
22          local int j = 0
23          from j = 0 loop
24              arcv(b,p)
25              print b
26              j += 1
27          until j = tosend.n
28          delocal int b = tosend.m
29          delocal int j = tosend.n
30
31      join
32
33      tosend.n -= n
34      tosend.n -= l
35
36      delocal port p
```

Esempio 2: single sender-receiver with multi message

Questo codice vuole utilizzare più valori da scambiare sempre con due processi, ognuno dei quali dotato di un proprio ciclo interno. Il valore da inviare, il quale sarà lo stesso per ogni iterazione, ed il numero di iterazioni del *loop* vengono presi in input dalla funzione e condiviso tra i due processi tramite la struttura **tosend**.

Nel caso della **forward** il processo sender inizia ad inviare gli elementi sulla porta '*p*', la variabile contatore '*b*' viene incrementata di uno fino al valore indicato, la sua delocal quindi si conclude con successo. La variabile '*a*' prende il valore che si vuole mandare e l'asserzione sulla sua delocal si conclude senza errori. Questo perchè viene azzerata alla conclusione di ogni send, e così anche il thread termina senza errori. Il thread che riceve fallisce la seconda volta che esegue la **arcv**, perchè dopo aver ricevuto il valore alla prima iterazione, la variabile '*b*' contiene un valore diverso da zero quindi l'asserzione fallisce generando in output l'errore:

Assertion 'b == 0' failed.

Utilizzando le funzioni **ssend** e **srcv** è più facile vedere l'errore perchè il thread che invia sarà bloccato dalla non avvenuta ricezione di conferma, dopo aver inviato per la seconda volta.

Per quanto riguarda la **reverse** il thread che ora riceve, ovvero il primo soddisfa alla prima ed alla seconda iterazione l'asserzione della **arcv**, ma fallisce alla terza. L'istruzione **a += tosend.n** a riga 1 diventa **a -= tosend.n**, alla prima iterazione la variabile '*a*' riceve il valore *tosend.n* e lo decrementa dello stesso, quindi '*a*' contiene ora valore zero e può soddisfare l'assezione della **arcv** alla seconda iterazione. Ma questa volta non riceve valore **tosend.m** ma riceve uno zero. Questo succede perchè il secondo thread, quello che ora invia, ha la variabile '*b*' inizializzata a **tosend.m**, per via della delocal a riga 28, ma questa viene azzerata dopo la **asend**. In questo modo il primo thread riceve il valore zero alla seconda chiamata della **arcv** e decrementando di **tosend.m**

otteniamo un valore pari a *-tosend.m*, quindi con questo valore diverso da zero il terzo assert della **arcv** fallisce.

Adesso provando ad aggiungere l'istruzione:

b -= tosend.m

dopo riga 25, nel caso della **forward** i cicli dei due thread terminano entrambi. Per il primo thread restano valide le considerazioni precedenti poichè non è stato modificato il codice, per il secondo thread come ci si aspettava con l'aggiunta dell'istruzione di decremento la varibile '*b*' viene portata a zero ad ogni iterazione, e quindi soddisfa l'asserzione ad ogni ricezione. Quello che ora genera un errore è la delocal a riga 28 proprio perchè la variabile '*b*' esce dal ciclo con valore zero e non uguale a **tosend.n**.

Per quanto riguarda la **reverse** il primo thread fallisce l'asserzione della **arcv** alla seconda iterazione. Avendo aggiunto l'istruzione *b -= tosend.m*, la quale diventa *b += tosend.m* incrementa il valore di '*b*' dopo che questo è già stato inizializzato a *tosend.m* ottenendo un valore pari a due volte *tosend.m*. L'istruzione a riga 14 ora non basta per portare '*a*' a zero e quindi l'asserzione della **arcv** fallisce alla seconda iterazione.

Adesso vediamo come si comporta il programma se proviamo a soddisfare l'asserzione della delocal sostituendo l'istruzione:

delocal int b = tosend.m

con

delocal int b = 0

Nel caso della **forward** i threads terminano entrambi con successo. Per il primo thread valgono sempre le considerazioni precedenti, e come ci si aspettava valgono le considerazioni anche per il thread che riceve, difatti l'asserzione in fase di deallocazione viene rispettata perchè '*b*', come detto prima, esce dal ciclo con valore zero.

Anche nel caso della **reverse** i threads terminano entrambi con succeso. Questo perchè ora la variabile '*b*' del secondo thread viene inizializzata a zero e poi incrementata di *tosend.m* prima della *ssend* ed azzerata dalla stessa *ssend*, rispettando così tutti i vincoli di asserzione. Abbiamo ottenuto un secondo programma biettivo funzionante.

Le istruzioni a riga 33 e 34 dell' *Esempio 2* sono state aggiunta per lo stesso motivo del primo esempio, ovvero per evitare che i threads facciano le loro operazioni sulle variabili condivise prima che queste vengano assegnate.

Provando ora ad aggiungere più processi riceventi il programma si comporta come atteso, ovvero non ci sono problemi nel caso della forward dove tutti i vincoli sono rispettati, e nel caso della reverse tutti i processi che prima ricevevano ed ora inviano portano a termine con successo le asserzioni.

6.4

Esempio 3 - Single sender-receiver with forwarder

Tornando ora al singolo sender e receiver, aggiungendo un terzo che faccia da forwarder ed una seconda porta, si ottiene un programma di questo tipo:

```
1  procedure third(int v, int l)
2      local port p
3      local port q
4
5      struct mystruct tosend
6
7      tosend.m += v
8      tosend.n += l
9
10     fork mystruct tosend
11         local int a = 0
12         local int i = 0
13         from i = 0 loop
14               i += 1
15               a += tosend.m
16               print a
17               asend(a,p)
18         until i = tosend.n
19         delocal int a = 0
20         delocal int i = tosend.n
21     and
22         fork mystruct tosend
23            local int b = 0
24            local int j = 0
25            from j = 0 loop
26                  arcv(b,p)
27                  print b
28                  asend(b,q)
29                  j += 1
30            until j = tosend.n
31            delocal int b = 0
32            delocal int j = tosend.n
33         and
34            local int c = 0
35            local int h = 0
36            from h = 0 loop
37                 arcv(c,q)
38                 print c
39                 c -= tosend.m
40                 h += 1
```

```
41              until h = tosend.n
42              delocal int c = 0
43              delocal int h = tosend.n
44          join
45      join
46
47      tosend.m -= v
48      tosend.n -= l
49
50      delocal port p
51      delocal port q
```

Esempio 3: single sender-receiver with forwarder

Nell'esperimento appena mostrato si sono usati due costrutti *fork and join* nidificati. Il primo thread si occupa di aggiungere alla sua variabile 'a' il valore preso dalla struttura condivisa ed inviare tale valore sulla porta 'p'. Il secondo thread si limita ad inizializzare gli altri due procecessi. Il terzo thread prende il valore che riceve sulla porta 'p' e lo invia sulla porta 'q', notiamo che in questo caso non è necessaria nessuna istruzione che in qualche modo azzeri la variabile 'b' perchè questa verrà azzerata dopo la funzione send. Il quarto thread infine riceve il valore sulla porta 'q'.

Nel caso della **forward** tutte le asserzioni vengono rispettate ed i threads terminano tutti con successo sia in modalità sincrona sia in modalità asincrona. Anche nel caso della **reverse** il programma si comporta come ci si aspettava, il primo e l'ultimo thread ora si scambiano i compiti, i valori verranno inviati sulla porta 'q' e ricevuti sulla porta 'p'. Il secondo thread non ha nulla da invertire e inizializza come nel caso della **forward** i due threads. Il terzo thread adesso riceve sulla porta 'q' ed invia sulla porta 'p' rispettando tutte le asserzioni. Come nel caso della **forward** l'esperimento termina con successo sia in modalità sincrona sia in modalità asincrona.

Una parte interessante di questo esempio riguarda il secondo thread, questo thread che funge come forwarder non ha bisogno di conoscere il valore che gli altri processi si scambiano a differenza degli esempi precedenti.

6.5

Esempio 4 - Multi sender-receiver

A questo punto se aggiungiamo un costrutto *fork and join* all'esempio 3 otteniamo un programma simile al prossimo. Dove se contiamo i processi in ordine, il primo ed il secondo, i quali corrispondono al costrutto più esterno inizializzano quattro threads. Nel primo *fork and join* interno, quello che inizia a riga 10, abbiamo il processo che invia valori sulla porta 'p' ed il processo che riceve sulla porta 'p' ed invia sulla porta 'q'. Nel secondo costrutto *fork and join* annidato abbiamo due threads che leggono sulla porta 'q'.

Esempio 4: multi sender-receiver

```
1  procedure fourth(int v, int l)
2      local port p
3      local port q
4
5      struct mystruct tosend
6
7      tosend.m += v
8      tosend.n += l
9
10     fork mystruct tosend
11         fork mystruct tosend
12             local int a = 0
13             local int i = 0
14                 from i = 0 loop
15                 i += 1
16                 a += tosend.m
17                 print a
18                 asend(a,p)
19             until i = tosend.n
20             delocal int a = 0
21             delocal int i = tosend.n
22         and
23             local int b = 0
24             local int j = 0
25             from j = 0 loop
26                 arcv(b,p)
27                 print b
28                 asend(b,q)
29                 j += 1
30             until j = tosend.n
31             delocal int b = 0
32             delocal int j = tosend.n
33         join
34     and
35         fork mystruct tosend
36             local int c = 0
37             local int h = 0
38             from h = 0 loop
39                 arcv(c,q)
40                 print c
41                 c -= tosend.m
42                 h += 1
43             until h = tosend.n / 2
44             delocal int c = 0
```

```
45              delocal int h = tosend.n / 2
46          and
47              local int d = 0
48              local int k = 0
49              from k = 0 loop
50                    arcv(d,q)
51                    print d
52                    d -= tosend.m
53                    k += 1
54              until k = tosend.n / 2
55              delocal int d = 0
56              delocal int k = tosend.n / 2
57          join
58      join
59
60       tosend.m -= v
61       tosend.n -= l
62
63      delocal port p
64      delocal port q
```

Il funzionamento di questo programma rimane coerente, nel caso dell'esecuzione in avanti vengono inviati tutti i dati sulla porta 'p' poi ridirezionati verso la porta 'q'. I threads riceventi li leggono quindi sulla porta 'q'. Nel caso della reverse saranno questi ultimi ad inviare dati al processo che ridirige il flusso, e quello che prima inviava ora riceve tutto sulla porta 'p'. Naturalmente perchè tutti i vincoli di asserzione siano rispettati, il valore che negli esempi viene preso in input come contatore di elementi, deve essere scelto in funzione dei processi, ovvero deve essere un intero divisibile per il numero dei processi riceventi, altrimenti una receive potrebbe restare in attesa di un qualcosa che non arriverà mai oppure uno o più valori rimasti in coda senza essere letti e nel caso sincrono, bloccare colui che invia.

6.6

Esempio 5 - Fibonacci with message passing

Il primo esempio a cui ho lavorato nello sviluppo del compilatore per *Janus* è stato un programma che calcola la successione di Fibonacci. Ho trovato questo esempio molto istruttivo poichè è possibile vedere in maniera chiara come lavorano i costrutti in entrambe le direzioni. Per questo motivo si vuole mostrare un ulteriore programma *Janus* che calcola la successione di Fibonacci utilizzando il costrutto *Fork and Join* ed il meccanismo di **message passing**. Il codice del programma è il seguente:

Esempio 5: Fibonacci with message passing

```
1  procedure multifib(int x1, int x2, int n)
2      local port p  // send queue
3      local port q  // rcv queue
4
5      struct mystruct s
6
7      s.y1 += x1
8      s.y2 += x2
9      s.m += n
10
11     fork mystruct s
12       local int i = 0
13       s.y1 += 1
14       s.y2 += 1
15       from i = 0 loop
16          asend(s.y1,p)
17          asend(s.y2,p)
18          arcv(s.y1,q)
19          arcv(s.y2,q)
20          print s.y1
21          i += 1
22       until i = s.m
23       delocal int i = s.m
24    and
25       local int i = 0
26       from i = 0 loop
27          arcv(s.y1,p)
28          arcv(s.y2,p)
29          s.y1 += s.y2
30          s.y1 <=> s.y2
31          asend(s.y1,q)
32          asend(s.y2,q)
33          i += 1
34          until i = s.m
35          delocal int i = s.m
36    join
37
38   s.y1 -= x1
39   s.y2 -= x2
40   s.m -= n
41
42   delocal port p
43   delocal port q
```

Il programma dichiara due porte 'p' e 'q', le quali vengono utilizzate per scam-

biarsi i valori in entrata e uscita. La porta 'p' viene utilizzata dal primo thread per inviare le due variabili della successione, mentre la porta 'q' per ricevere l'ultimo valore della successione ed il precedente. Le porte in questione chiaramente vengono invertite per il processo che esegue le operazioni. La struttura 's', dichiarata localmente, viene utilizzata per l'inizializzazione delle varibiali da passare come argomento alla *fork and join*. L'operazione di somma della successione e lo scambio di varibili viene fatto dal secondo processo, il quale una volta terminate le operazioni invia i valori al primo dove vengono stampati.

Nel caso della **forward** il programma funziona correttamente. Le asserzioni del primo processo della funzione di receive vengono rispettate poichè le variabili vengono azzerate dopo la send. La prima iterazione del secondo processo procede senza errori perchè le variabili sono inizializzate a zero, dalla seconda iterazione in poi le asserzioni sono rispettate, anche in questo caso grazie alla funzione di send.

Nel caso della **reverse** il programma non soddisfa l'assert di ricezione nel secondo processo, questo succede perchè la variabile su cui deve scrivere è quella passata per argomento, quindi diversa da zero nel caso della **reverse**. Infatti nel caso si voglia utilizzare la chiamata **uncall** per la **reverse** devono essere passati parametri diversi dalla funzione di **main**, come nel caso della procedura di **Fibonacci** vista nella sezione di **Introduzione**. In questo caso basterà cambiare i valori di $\mathbf{X}_1$ e $\mathbf{X}_2$ con gli ultimi due valori della successione che vogliamo esaminare, ovvero F_n e $F_n - 1$.

Per soddisfare le asserzioni nel caso della **reverse** il secondo processo può utilizzare due variabili inizializzate a zero, questo perchè il processo in questione in entrambe le direzioni prima riceve dei valori e poi li invia, ricordando che la funzione di *send* azzera le variabili dopo aver terminato la sua esecuzione, l'asserzione viene rispettata. Il secondo processo avrà un corpo del genere:

```
local int i = 0
local int v1 = 0
local int v2 = 0
  from i = 0 loop
      arcv(v1,p)
      arcv(v2,p)
      v1 += v2
      v1 <=> v2
      asend(v1,q)
      asend(v2,q)
      i += 1
until i = s.m
delocal int v1 = 0
delocal int v2 = 0
delocal int i = s.m
```

Capitolo 7

Conclusioni e futuri sviluppi

Lo sviluppo del compilatore per *Janus* è stato particolarmente istruttivo sia dal punto di vista teorico sia da quello pratico. Dal punto di vista teorico sono stati toccati diversi argomenti, in particolare trattati nei corsi di Programmazione e Linguaggi di Programmazione. Con le informazione ottenute dai corsi dell'Università di Bologna ho potuto approfondire argomenti come parsing, grammatiche, interpreti e compilatori. Dal punto di vista pratico il Professore Ivan Lanese mi ha lasciato libertà di scelta per l'utilizzo delle tecnologie, si sono così potuti sperimentare diversi linguaggi e tecniche di sviluppo prima della scelta che è stata presentata in questa tesi. In particolare è stato interessante sviluppare la parte di sperimentazione, oltre che a testare in maniera funzionale l'interprete si è potuto scrivere piccoli programmi concorrenti in linguaggio *Janus* e trarre i primi risultati su questo linguaggio bidirezionale.

Il compilatore quì presentato è ancora in una fase iniziale e potrà essere ampliato da studenti interessati all'argomento. Personalmente per rendere l'interprete più completo penso si possano aggiungere altri tipi di dato, quali stringhe o tipi di valori, come double e float. Inoltre si potrebbe studiare un metodo per poter passare valori ai threads inizializzati dal costrutto *fork and join* che non siano le strutture dati, magari lavorando sullo scope delle funzioni.

Bibliografia

[1] Christopher, Lutz (1986), *Janus: a time-reversible language*. `http://tetsuo.jp/ref/janus.pdf`

[2] Terence Parr (1989), *ANTLR ANother Tool for Language Recognition.* `https://www.antlr.org/`

[3] Yokoyama, T (2010), *Reversible Computation and Reversible Programming Languages*. `https://doi.org/10.1016/j.entcs.2010.02.007`

[4] Robert Glück, Torben Mogensen and Holger Bock Axelsen (2012), *Janus: a reversible imperative programming language online tool.* `http://topps.diku.dk/pirc/?id=janusP`

[5] Robert Gluck, Ivan Lanese and Irek Ulidowski (2016), *Causal-consistent Janus.*

[6] 2019, *Janus reversible computing programming language with antlr4.* `https://github.com/andrebfc/Janus-Parser-with-antlr4`

[7] Elena Giachino, Ivan Lanese and Claudio Antares Mezzina (2016), *Causal-consistent Reversible Debugging.* `http://www.cs.unibo.it/caredeb/fase14.pdf`

[8] Yokoyama, T (2009), *Reversible Computation and Reversible Programming Languages.* `https://pdfs.semanticscholar.org/15e0/733ab60c4e5f47580c81c03cb95c82900630.pdf`

[9] Maurizio Gabrielli, Simone Martini (2006), *Linguaggi di programmazione. Principi e paradigmi*, McGraw-Hill, Milano.