Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

DEFINIZIONE DI UNA GRAMMATICA PER IL LINGUAGGIO JOLIE

Tesi di Laurea in Paradigmi di Programmazione

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Stefano Pascali

Sessione Prima
2010 - 2011

A mio padre . . .

eroe della mia vita . . .

Introduzione

Da alcuni anni, nel panorama tecnologico globale, un nuovo paradigma organizzativo emerge per consentire lo sviluppo di sistemi distribuiti basati sul concetto di servizio.

Questo paradigma, sintetizzato con l'acronimo SOA, prevede che i servizi possono essere composti al fine di progettare altri servizi sempre più complessi, sfruttando orchestratori. Gli orchestratori sono entità in grado di richiamare e coordinare altri servizi, sfruttando modelli di composizione di workflow.

Ed è proprio in questo contesto in cui troviamo JOLIE, un linguaggio di programmazione che permette l'orchestrazione di servizi. Il progetto è open-source e nasce presso l'Università di Bologna grazie a Fabrizio Montesi e Claudio Guidi.

Il punto di forza di questo linguaggio, rispetto ad altri come WS-BPEL, è la sua sintassi simile al C/Java, già familiare alla maggior parte degli sviluppatori.

JOLIE è un linguaggio di programmazione innovativo e ancora in fase di sviluppo ma già utilizzato per applicazioni commerciali. I continui cambiamenti nel linguaggio possono nascere per soddisfare le esigenze commerciali che il paradigma SOA offre, oppure per l'aggiunta di nuovi costrutti, semplificando il lavoro al programmatore che decide di sviluppare applicazioni in questo contesto.

Il parser di JOLIE è stato scritto a mano, senza l'ausilio di traduttori automatici, il che rende più difficile la ridefinizione del parser nel caso in cui vengano aggiunti dei nuovi costrutti o delle operazioni al linguaggio.

Dopo aver scritto un manuale del linguaggio, che rappresenta un punto di partenza per sviluppatori che si affacciano a questo nuovo paradigma di programmazione, il mio lavoro di tesi si è concentrato sullo sviluppo di una definizione della grammatica attraverso l'utilizzo di ANTLR, un generatore automatico di parser.

La parte iniziale della tesi è incentrata sul paradigma SOA, elencandone i vantaggi e le prospettive innovative. Saranno definite le caratteristiche dei servizi e dei sistemi service oriented, a cui seguirà un approfondimento sul Web Service, che rappresenta attualmente la scelta più diffusa per l'implementazione di sistemi distribuiti.

Il secondo capitolo illustra le caratteristiche di JOLIE, analizzando in maniera generale sintassi, operazioni e architettura del linguaggio.

Il terzo capitolo descrive le varie fasi che devono essere sviluppate per descrivere un linguaggio di programmazione, nel nostro caso il linguaggio JOLIE. Dopo aver approfondito il concetto generale di grammatica e sintassi, sarà presentata una breve panoramica sulle tecniche di parsing disponibili.

Il quarto capitolo introduce ANTLR, lo strumento utilizzato per la definizione della nostra grammatica sviluppata, capace di generare parser per grammatiche LL(k). In questo contesto verranno argomentati i principali punti di forza di questo particolare software che produce parser predittivi a discesa ricorsiva strutturalmente simili ai parser scritti a mano.

Infine l'appendice A contiene il sorgente della definizione della grammatica con ANTLR.

Indice

Introduzione **i**

1 SOA:
Un Paradigma Organizzativo **1**

1.1 Service Oriented Architecture 1

1.1.1 Servizi 4

1.2 Vantaggi e Profitti nelle SOA 4

1.3 Web Service 6

1.4 Orchestration o Coreography ? 11

1.4.1 Orchestration 12

1.4.2 Choreography 12

2 JOLIE:
Un Nuovo Linguaggio **13**

2.1 SOA e Servizi in JOLIE 14

2.1.1 Behaviour 14

2.1.2 Engine 16

2.1.3 Service Description 16

2.2 Composizione di servizi in JOLIE 18

2.3 L'architettura di JOLIE 19

2.3.1 Analisi del codice 19

2.3.2 Albero di interpretazione a oggetti 20

2.3.3 Ambiente runtime 20

2.3.4 Gestore della comunicazione 20

3 Descrivere un linguaggio di programmazione 21
3.1 Livelli di descrizione 21
3.2 Grammatica e sintassi 22
3.2.1 Definizione della sintassi 23
3.2.2 Definizione delle grammatiche 24
3.2.3 Derivazioni 25
3.2.4 Alberi di Parsing 25
3.3 Parsing 26
3.3.1 Parsing top-down 27
3.3.2 Parsing bottom-up 29

4 ANTLR:
Un Generatore di Parser 33
4.1 Aspetti positivi di ANTLR 33
4.1.1 Lexer 34
4.1.2 Parser 35
4.2 Definire la grammatica di Jolie attraverso ANTLR 36
4.2.1 EBNF: Simboli e notazioni in ANTLR 37
4.2.2 Differenza tra lexer e parser in ANTLR 38
4.2.3 Token in ANTLR 39
4.2.4 Opzioni in ANTLR 40
4.3 ANTLRWorks 40
4.3.1 Syntax Diagram 41
4.3.2 Parse Tree 42

Conclusioni e sviluppi futuri 45

A La definizione della grammatica per JOLIE 49

Bibliografia 67

Elenco delle figure

1.1 Identificazione dei Servizi 3
1.2 Processo con WSBPEL 10
1.3 Orchestration e Coreography 11

2.1 Schema dell'architettura di JOLIE 19

3.1 Albero di parsing per il costrutto if-else 26
3.2 Schema di un processo con unità di lexer e parser 27

4.1 Start rule program in ANTLR 41
4.2 Parse Tree per la definizione di inputPort 43

ACRONIMI UTLIZZATI

ANTLR *ANother Tool for language Recognition*

AST *Abstract Syntax Tree*

BNF *Backus-Naur Form*

EBNF *Extended Backus-Naur Form*

HTTP *HyperText Transfer Protocol*

IDE *Integrated Development Environment*

JOLIE *Java Orchestration Language Interpreter Engine*

LGPL *Lesser General Public License*

OASIS *Organization for the Advancement of Structured Information Standards*

OOIT *Object Oriented Interpretation Tree*

SOA *Service Oriented Architecture*

SOAP *Simple Object Access Protocol*

SOC *Service Oriented Computing*

SOCK *Service Oriented Computing Kernel*

UDDI *Universal Description, Discovery and Integration*

URL *Uniform Resource Locator*

W3C *World Wide Web Consortium*

WB *Web Service*

WS-BPEL *Web Services Business Process Execution Language*

WS-CDL *Web Services Choreography Description Language*

WSDL *Web Service Description Language*

XML *eXtensible Markup Language*

Capitolo 1

SOA:
Un Paradigma Organizzativo

Non occorre essere sociologi per capire che il prorompente avanzare delle applicazioni Web 2.0 sta letteralmente rivoluzionando i rapporti sociali. Poiché circa il 35% della nostra giornata è praticamente dedicata ad assolvere mansioni lavorative, il mondo del lavoro non è certo escluso da questa palingenesi: questi fenomeni dunque possono comportare la necessità di rivisitare l'intero modello organizzativo della piattaforma gestionale.

Ci serve un paradigma nuovo. Ma non occorre inventarlo, esiste già: si chiama SOA(o SOC).

1.1 Service Oriented Architecture

Con SOC si intende il paradigma di programmazione basato sui servizi. Un servizio è considerato l'elemento centrale della programmazione di applicazioni, e SOC utilizza i servizi per comporre sistemi distribuiti dinamici, ai quali ci si può riferire con il termine SOA.

Nel linguaggio generale, con SOA intendiamo l'ambizioso tentativo di ricondurre le applicazioni Web al sistema gestionale classico o a qualcosa ad esso correlato. Dunque un'unica architettura orientata ai servizi Web. Possiamo pensare all'aggiunta di un

ulteriore strato applicativo software che, con appositi connettori, permette di agganciare il sistema gestionale aziendale ai servizi Web.

Entrando nello specifico, una SOA è un'architettura software che garantisce l'interoperabilità tra diversi sistemi mediante l'uso dei Web Service. Ogni singola applicazione può essere vista come un componente del processo di business da eseguire. Le SOA sono quindi un insieme di servizi che cooperano per realizzare le funzionalità richieste.

Una definizione più scientifica è resa nota dal consorzio OASIS [1, Mlm06], il quale afferma che un'architettura SOA è un paradigma per l'organizzazione e l'utilizzazione di *capabilities* che possono essere sotto il controllo di domini di proprietà differenti. Tale architettura fornisce un mezzo uniforme per offrire, scoprire, interagire ed usare le capacità di produrre gli effetti voluti consistentemente con presupposti e aspettative misurabili.

Per capability intendiamo la capacità di svolgere un'operazione da parte di un'entità.

Nella vita quotidiana, le entità (persone e organizzazioni) creano capabilities per soddisfare i servizi di cui si occupano. Tali entità hanno bisogno di collaborare con altre entità per adempiere i servizi richiesti.

Per avere un'idea più chiara sul concetto di SOA mostriamo un esempio semplificato di un processo di business per la validazione di un ordine per un articolo da acquistare. Tale processo riguarda la verifica per l'accettazione di un ordine richiesto. Semplifichiamo il processo ipotizzando che sia sufficiente un controllo sulla validità della carta di credito dell'acquirente e sulla sua email: dobbiamo quindi considerare valido un ordine che sia relativo ad una email valida e ad una carta di credito valida.

Dal processo identifichiamo le operazioni che devono essere modellate che nel semplice esempio proposto sono:

- CheckEmail
- CheckCreditCard

Questi due operazioni ci aiutano a identificare i candidati per i servizi (un candidato ad un servizio è un contesto che raggruppa una o più operazioni). I servizi diventano quindi le unità per il design e l'implementazione di architetture software distribuite,

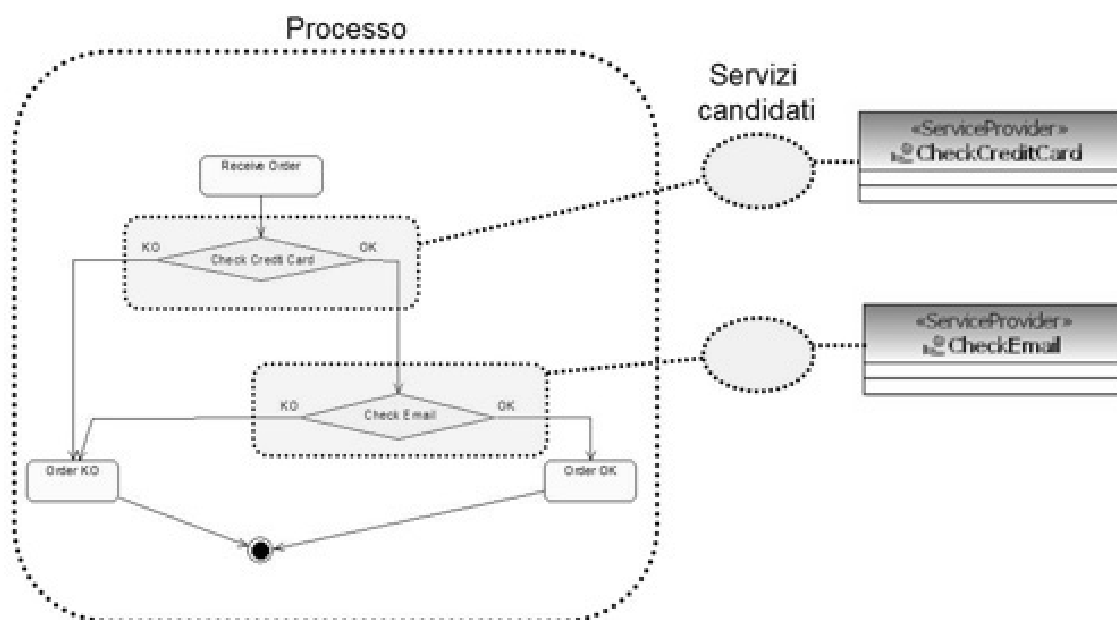

Figura 1.1: Identificazione dei Servizi

che rispondono ad un'esigenza di processi di business anche molto complessi e inoltre potenzialmente decentralizzati ed eterogenei.

Secondo OASIS i tre concetti principali del paradigma SOA sono:

- **Visibilità**: indica la capacità di trovare il servizio più idoneo alle proprie necessità. L'entità che fornisce una capability è definita come *Service Provider*, mentre la controparte che utilizza il servizio *Service Consumer*. Entrambe le parti devono vedersi reciprocamente per poter scambiare informazioni.

- **Interazione**: indica la capacità di richiesta di un servizio e conseguentemente al completamento della richiesta mediante scambio di messaggi.

- **Effetto**: indica la capacità di dare i risultati dell'interazione.

Si nota come tali concetti non definiscano altro che le proprietà di un servizio, che nel contesto SOA è inteso come il meccanismo con il quale bisogni e capability vengono in contatto.

1.1.1 Servizi

Un servizio viene erogato quando si svolgono attività per conto di un richiedente (Service Consumer). Il servente (Service Provider) garantisce la propria offerta di servizio, la propria capacità di svolgerlo e la descrizione delle specifiche con le quali si può invocare correttamente il servizio offerto.

Secondo [2, Pap07], i servizi coinvolti nelle SOA devono essere *trovabili, invocabili e componibili.*

La trovabilità è strettamente legata al concetto di visibilità definito secondo OASIS e comprende quindi un sistema che garantisca questa caratteristica. Questo sistema può essere rappresentato da *Service Registry*, cioè tassonomie che i service consumer possono interrogare per conoscere quale sia il provider più adatto alle proprie esigenze.

L'invocabilità dei servizi è garantita da un sistema di comunicazione standardizzato basato su XML che garantisce l'indipendenza da piattaforme o protocolli di comunicazione. Ogni servizio è in questo modo invocabile da chiunque (indipendenza da piattaforme) e in ogni luogo (indipendenza da protocollo comunicativo).

I servizi devono essere infine componibili: un servizio rappresentante il workflow di un business process può infatti includere ricerca e invocazione di numerosi altri servizi(come nell'esempio di Figura1.1 in cui il servizio di CheckCreditCard può includere al suo interno altri servizi).

1.2 Vantaggi e Profitti nelle SOA

La SOA porta benefici organizzativi in diversi modi, secondo gli obiettivi in cui la SOA e le sue tecnologie vengono applicate. Possiamo analizzare alcuni vantaggi comuni che questa piattaforma architetturale offre.

In [2, Pap07] sono inoltre evidenziati otto principali benefici dovuti al paradigma SOA:

1. **Integrazione:** Una SOA può portare alla creazione di soluzioni composte da servizi intrinsecamente interoperabili. La standardizzazione di componenti e tecnologie permette una rapida integrazione con risorse interne. Un sistema che funziona internamente con il paradigma SOA è facilmente integrabile con i servizi esterni.

2. **Riuso:** SOA permette di sviluppare delle soluzioni che soddisfano i requisiti di un particolare cliente, ma possono essere riutilizzate da potenziali futuri richiedenti. Questo punto di forza stabilisce un ambiente in cui gli investimenti in sistemi esistenti possono essere sfruttati per la costruzione di nuove soluzioni.

3. **Composizione:** Ogni servizio è una composizione di altri servizi più piccoli, permettendo quindi una maggiore modularità. Ma il concetto di composizione è una parte fondamentale di un sistema SOA e non si limita all'aggregazione di un insiemi di servizi. Questo aspetto delle SOA può infatti portare alla costruzione di ambienti di automazione ottimizzati, in cui solo le tecnologie richieste faranno parte dell'architettura.

4. **investimenti legacy:** L'accettazione del settore delle tecnologie dei WS ha generato un grande mercato, consentendo ad ambienti legacy di partecipare alle SOA. In questo modo, ambienti che prima rimanevano isolati, ora possono interoperare contribuendo alla fornitura di nuovi servizi.

5. **XML Standard:** Ogni dato nelle SOA è rappresentato mediante XML, uno standard affermato per l'interoperabilità fra piattaforme, che permette di minimizzare i costi di conversione delle informazioni.

6. **Comunicazione:** Poiché i WS stabiliscono standard di comunicazione, una SOA può centralizzare applicazioni di comunicazione interne ed esterne. Questo consente alle organizzazioni di espandere l'infrastruttura investendo in una sola tecnologia responsabile per la comunicazione, riducendo così i costi. La standardizzazione di un'architettura orientata ai servizi permette ad un'organizzazione di investire in un unico sistema di comunicazione, interno ed esterno.

7. **Switching:** Gran parte delle SOA è basata sul presupposto che quello che viene costruito oggi subirà cambiamenti in futuro. Pertanto una SOA ben progettata deve proteggere le organizzazioni tenendo conto dell'impatto che avrà con l'evoluzione. I cambiamenti possono essere distruttivi, costosi e potenzialmente dannosi per organizzazioni rigide. Servizi che fornisco le proprietà elencate costituiscono un ambiente che è più facilmente adattabile ai cambiamenti. Con questa proprietà,

le SOA permettono alle imprese di evolvere senza restare legate a particolari tecnologie. È possibile cambiare fornitore di servizi agilmente, senza dover sostenere elevati costi di transizione.

8. **Libera Scelta:** Tramite la descrizione standardizzata dei servizi offerti dai vari Service Provider, le SOA permettono ai Service Consumer di scegliere sempre il migliore fornitore disponibile o quello più conveniente alle loro esigenze.

Da quanto affermato, le SOA permetto quindi di creare a bassi costi sistemi distribuiti agili e flessibili, modellati su misura per le funzionalità richieste e ampiamente riusabili. Ora, esaminati questi aspetti, vediamo quali sono le tecnologie che permettono alle architetture orientate ai servizi di funzionare.

1.3 Web Service

Un WS è definito dal W3C come un sistema software in grado di offrire funzionalità ad altri elaboratori tramite interfacce di rete. In base a quanto detto in precedenza, un WS rappresenta quindi l'implementazione di un servizio coinvolto in una SOA.

In altri termini un WS è un'architettura che permette alle applicazioni di parlare l'una con l'altra, definendo un insieme di standard che favorisce l'interoperabilità. I WS offrono la possibilità di soddisfare i requisiti necessari per implementazione di una SOA.

Ancora secondo [2, Pap07], il paradigma service oriented nasce grazie alla crescita della tecnologia dei WS.

Esistono cinque standard riguardanti i WS.I primi due sono standard generali:

- **XML:** Utilizzato come formato generale per descrivere modelli, formati e tipi.
- **HTTP(S):** È il protocollo di basso livello usato per Internet, ed è uno dei più comuni protocolli usati per l'invio di Web services in rete.

Gli altri tre standard fondamentali sono specifici ai WS:

- **UDDI:** È uno standard per la gestione di WS, come ad esempio la registrazione e la ricerca di servizi. UDDI è un sistema per standardizzare i registry dei WS, una sorta di elenco dei WS disponibili.

- **SOAP:** Mentre HTTP è il protocollo di basso livello, SOAP è uno specifico formato per lo scambio dei dati dei WS. Il protocollo SOAP è lo standard di comunicazione dei WS e permette la formattazione di messaggi XML-based. Questi messaggi sono inviati ad altri WS tramite HTTP/S. Il ricevente estrae le informazioni dal messaggio e le trasforma in un protocollo conosciuto dal sistema residente.

- **WSDL:** Nei WS la caratteristica di trovabilità del servizio è garantita dal WSDL e dal protocollo UDDI. WSDL permette di descrivere con una sintassi XML i requisiti funzionali che un particolare servizio offre. Rappresenta quindi l'interfaccia pubblica con la quale altri servizi possono comunicare. Permette quindi di definire interfacce di servizi. Può descrivere tre aspetti distinti di un servizio: la sua signature (nome e parametri), il suo binding (protocollo) e i dettagli di deployment (location).

Utilizzare lo standard WSDL è generalmente la caratteristica chiave dei WS. SOAP e HTTP non sono gli unici standard utilizzati per inviare richieste ai servizi e UDDI non è indispensabile e ricopre un ruolo secondario.

Per ogni messaggio scambiato tra WS deve essere definito il tipo, e per definire i tipi WSDL usa l'elemento *types*. Gli elementi *message* definiscono i messaggi, la base della costruzione di un WS con WSDL. I messaggi sono gli elementi che costituiscono input e output dei servizi, e possono contenere tipi di dato complessi, definiti nella sezione *Type*, oppure tipi di dato primitivi. Gli elementi *operation* definiscono le operazioni, le funzionalità che saranno esposte nell'interfaccia del servizio. Tali elementi contengono elementi di input, output e fault che specificano i messaggi scambiati durante l'operazione. I collegamenti, definiti dagli elementi *binding*, eseguono la mappatura tra il servizio ed il protocollo di comunicazione SOAP, e sono essenzialmente istanze degli elementi *portType*. L'ultimo elemento di un file WSDL è la definizione del servizio, l'elemento *service*, che consente di raccogliere tutte le operazioni sotto un unico nome.

Per comprendere meglio i diversi concetti sopracitati, torniamo al nostro processo di business preso come esempio di SOA.

A questo punto bisogna ricontrollare gli scenari di composizione dei servizi progettati ed, eventualmente, rivedere il raggruppamento delle operazioni (cioè i servizi candidati) o il flusso del processo stesso.

Il passo successivo è la definizione dei service contract, che vanno espressi coerentemente con la tecnologia che si è deciso di adottare: nel nostro caso utilizziamo i WS e, di conseguenza, disegniamo i service contract tramite WSDL.

Il WSDL ricopre un ruolo importante nell'architettura dei WS perché descrive il contratto tra il Service Consumer (l'entità che richiede il servizio) ed il Service Provider (l'entità che fornisce il servizio) in modalità indipendente dal linguaggio di sviluppo e dalla piattaforma utilizzata.

In pratica, WSDL è una grammatica che descrive (integrando XML) come interagire con il servizio, definendo le operazioni e i messaggi relativi e specificando i binding delle operazioni sui protocollo esposti.

Possiamo suddividere il service contract WSDL in due parti: una astratta, che rappresenta la parte del contratto che non varia al variare della modalità di esposizione, ed una concreta, che specifica su quale protocollo è accessibile il servizio e come le informazioni vengono trasmesse.

Gli elementi della parte astratta sono:

- types: descrive la definizione dei tipi di dati scambiati tra client e server usando un formato XML (facoltativo per la descrizione dei tipi primitivi, obbligatorio nel caso di tipi di dato complessi).
- message: descrive la definizione dei messaggi astratti (sia request che response) che possono contenere zero o più elementi, ognuno di un possibile tipo diverso.
- portType: descrive un insieme astratto di operazioni supportato da uno o più endpoint (interfaccia); le operazioni sono definite dallo scambio di più messaggi. Per esempio l'elemento portType può combinare un messaggio request ed un messaggio response in un'unica operazione Request-Response. Una portType generalmente definisce più di un'operazione.

La parte concreta specifica su quale protocollo è accessibile il servizio e come le informazioni vengono trasmesse.

Gli elementi della parte concreta sono:

- binding: specifica il protocollo e il formato dei dati di una particolare portType.

- service: definisce l'indirizzo del Web Service. È un'associazione di un URL e di un binding.

Nel nostro esempio, la fase di design si conclude definendo nel WSDL del servizio di controllo email (CheckEmailService) un'operazione di business di nome checkMail, che prevede una stringa in ingresso e restituisce un boolean in uscita.

```
<wsdl:message name="checkMailResponse">
   <wsdl:part name="checkMailReturn" type="xsd:boolean"/>
</wsdl:message>
<wsdl:message name="checkMailRequest">
   <wsdl:part name="arg" type="xsd:string"/>
</wsdl:message>
<wsdl:portType name="CheckEmail">
   <wsdl:operation name="checkMail" parameterOrder="arg">
      <wsdl:input message="impl:checkMailRequest"
                                 name="checkMailRequest"/>
      <wsdl:output message="impl:checkMailResponse"
                                 name="checkMailResponse"/>
   </wsdl:operation>
</wsdl:portType>
```

Nel caso del servizio di controllo validità della Carta di Credito si definisce un'operazione di business di nome checkCdC, che prevede un oggetto di tipo creditCard in ingresso e restituisce un tipo creditCardResponse in uscita.

```
<wsdl:message name="creditCard">
   <wsdl:part name="creditCard" element="x1:creditCard"/>
</wsdl:message>
<wsdl:message name="creditCardResponse">
   <wsdl:part name="creditCardResponse"
                        element="x1:creditCardResponse"/>
</wsdl:message>
```

```
<wsdl:portType name="CheckCdC">
   <wsdl:operation name="checkCdC">
      <wsdl:input name="creditCard"
                        message="tns:creditCard"/>
      <wsdl:output name="creditCardResponse"
                        message="tns:creditCardResponse"/>
   </wsdl:operation>
</wsdl:portType>
```

Definiti i Service Contract passiamo a dettagliare il processo di orchestrazione.

Per orchestrare i Web services attraverso WSBPEL, si presuppone che i servizi che devono essere orchestrati siano descritti secondo WSDL (ed eventualmente memorizzati in un registry UDDI).

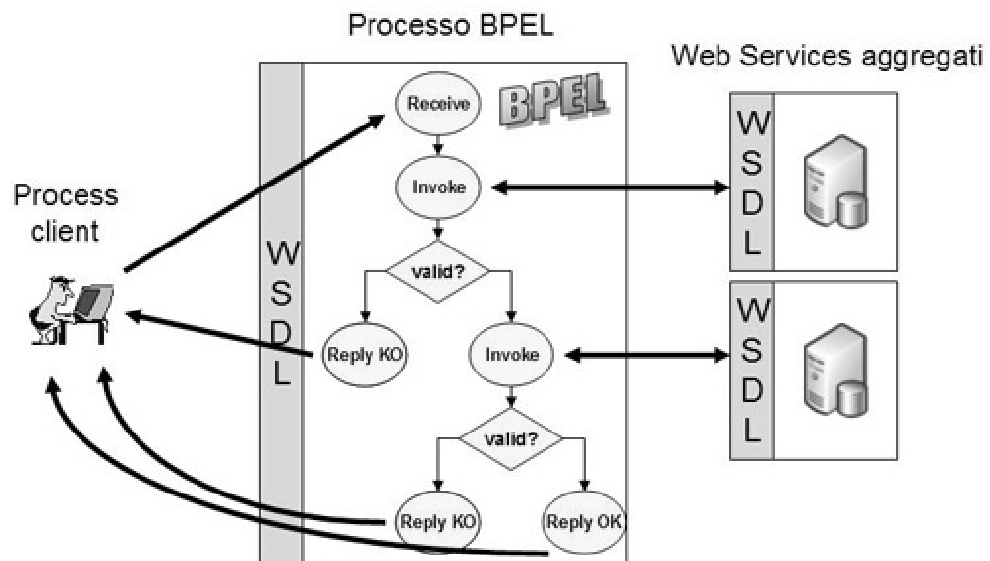

Figura 1.2: Processo con WSBPEL

L'interazione fra Service Consumer e Service Provider, nel contesto dei WS, può essere molto complessa a causa delle numerose componenti (pubblicazione/consumo WSDL, invio messaggi, trasporto) che entrano in gioco.

In contesti reali inoltre i WS coinvolti possono essere numerosi e per ridurre la complessità è stato introdotto un componente all'architettura che lavora ad un livello d'astrazione più alto, un *service aggregator*. Un service aggregator permette di comporre servizi per crearne di nuovi, mascherando la complessità interna di un sistema. Sono nati a questo scopo aggregator language, come WS-BPEL e JOLIE, descritti nel prossimo capitolo.

1.4 Orchestration o Coreography ?

In un sistema basato sulla SOA i servizi possono essere composti tra loro al fine di progettare servizi più complessi.

Sono stati ideati due approcci con lo scopo di ridurre la complessità di connessione fra servizi: *orchestration* e *choreography*. Gli orchestrator sono in grado di richiamare e coordinare altri servizi sfruttando tipici modelli di workflow come la composizione parallela, sequenziale e di scelta. Diverso invece è l'approccio della choreography, che permette di progettare un sistema distribuito dall'alto.

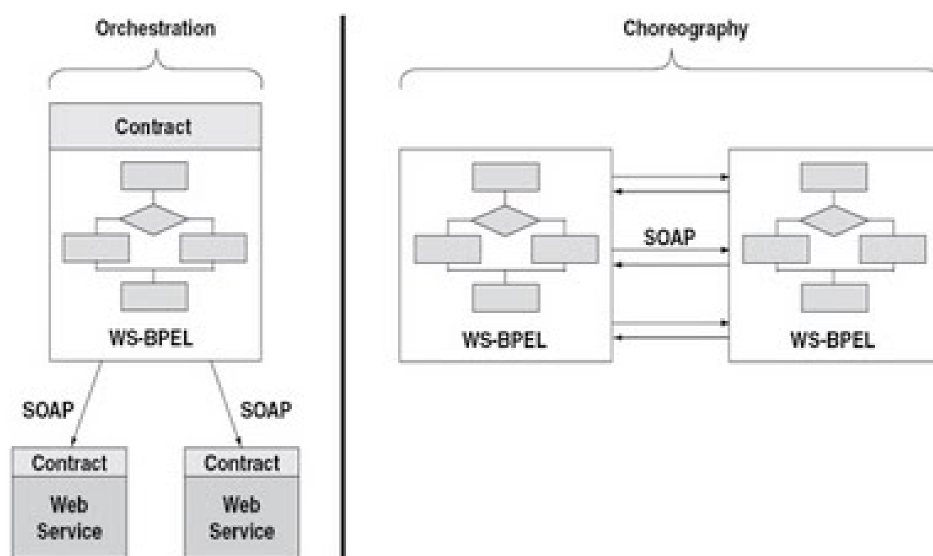

Figura 1.3: Orchestration e Coreography

1.4.1 Orchestration

Nell'Orchestration vi è un singolo processo produttivo che interagisce con i Web Service interni ed esterni. Il processo comprende un workflow in cui è specificato il modo in cui i servizi comunicano tra di loro, ovvero la logica dei messaggi che si scambiano per arrivare a terminare un processo. L'esecuzione del processo ed il controllo del workflow (sequenza e ordine delle operazioni) è affidato ad un singolo gestore logico. Esiste un direttore (sistema centrale) che dice agli orchestrali (servizi o sistemi remoti) come comportarsi. La sincronizzazione è gestita centralmente.

Tra i linguaggi che adottano questo approccio possiamo trovare WS-BPEL e il nuovo linguaggio JOLIE.

1.4.2 Choreography

Con Choreography si intende la descrizione ad alto livello delle interazioni tra i servizi.

La choreography definisce le regole con le quali i messaggi vengono scambiati. Con tale approccio ciascuna entità descrive una parte del processo produttivo responsabile della parte che svolge. Nessuna delle entità possiede una visione globale del processo, ciascuna collabora alla sua esecuzione. In questo caso, ogni sistema remoto(o servizio) sa cosa deve fare e si sincronizza con gli altri servizi per eseguire nuove operazioni. Quindi ogni singolo sistema remoto/servizio conosce cosa fare e chi chiamare alla fine della propria attività.

Il linguaggio attualmente più accreditato per la coreografia è WS-CDL.

Capitolo 2

JOLIE: Un Nuovo Linguaggio

Attraverso l'analisi di entrambi gli approcci è emerso che l'orchestration rappresenta un passo in avanti verso il perfezionamento della progettazione di applicazioni service oriented, in quanto permette di gestire ogni servizio coinvolto nell'applicazione.

Il progetto nasce come risposta ad alcune problematiche che si incontrano nell'orchestrazione delle SOA: connettività, esecuzione parallela e comunicazione. È da questi prerequisiti che nasce un nuovo linguaggio orientato ai servizi: JOLIE.

JOLIE è stato realizzato concentrandosi su funzionalità come la sincronizzazione dei processi o la gestione delle comunicazioni. Si tratta di un progetto tutto italiano, open source e rilasciato sotto la licenza LGPL , frutto di una tesi del corso di laurea in Scienze dell'Informazione presso l'Università di Bologna.

Il progetto JOLIE, iniziato nel marzo 2006, è il prodotto finale di una tesi di laurea sviluppata da Fabrizio Montesi. La base teorica del progetto, tuttavia, può essere fatta risalire agli inizi del 2005, quando Claudio Guidi ha iniziato la sua tesi di dottorato.

JOLIE [3, MGLZ06] è un linguaggio open source di orchestrazione, basato sul paradigma di programmazione service-oriented. Offre la possibilità di creare interamente servizi nuovi o di comporre quelli esistenti per ottenere nuove funzionalità.

JOLIE pone le sue basi teoriche sul calcolo formale SOCK, sviluppato per formalizzare tutte le primitive essenziali per la descrizione dei meccanismi del paradigma SOC, e di

cui JOLIE ne rappresenta un'implementazione concreta. Grazie alle regole semantiche ben definite di SOCK, la progettazione di servizi con JOLIE avviene ad alto livello, astraendo dalle problematiche di comunicazione.

JOLIE è stato sviluppato interamente in Java, in maniera da permetterne l'esecuzione su diverse piattaforme.

Quello che caratterizza il linguaggio JOLIE rispetto a WS-BPEL, oltre al sistema formale sottostante, è la sintassi Java/C-like, senza dubbio più comprensibile rispetto a quella XML-based di WS-BPEL.

Nel corso del capitolo esamineremo una parte della sintassi del linguaggio per favorirne la comprensione. La sintassi utilizzata nelle successive formalizzazioni, prevede i simboli `<` e `>` per le variabili specificate dall'utente, mentre si racchiudono tra i simboli `[` e `]` i componenti opzionali. Data la semplicità, si preferisce adottare questa sintassi, in quanto risulta essere di più facile lettura rispetto alla grammatica formale del linguaggio.

2.1 SOA e Servizi in JOLIE

In JOLIE, SOA e servizi assumono un significato più concreto rispetto al reference model OASIS, e per comprenderlo occorre definire tre importanti concetti: *Behaviour*, *Engine* e *Description*.

2.1.1 Behaviour

Il behaviour di un servizio è definito come la descrizione delle attività composte in un workflow. Le attività rappresentano gli aspetti funzionali (come la logica di business di un servizio), mentre la loro composizione all'interno di un workflow rappresenta la sequenza temporale di esecuzione di queste attività.

Il workflow ha quindi lo stesso significato di WS-BPEL. Le attività si dividono in funzionali, che riguardano la logica di business interna, attività per la gestione dei guasti (molto importante nelle SOA) e attività di comunicazione. Queste ultime sono ovviamente le più importanti e in JOLIE prendono il nome di *operation*, come in WSDL.

In termini pratici, un servizio è un'applicazione attiva su un macchina, e tramite le porte (InputPort/OutputPort)riesce a comunicare con altri servizi mediante lo scambio di messaggi. Le operazioni si dividono in operazioni di input e output.

In conformità alle specifiche WSDL, le operazioni di input possono essere di tipo *One-Way* oppure *Request-Response*.

One-Way rimane in attesa di un messaggio da un altro servizio;

Request-Response rimane in attesa di un messaggio da un altro servizio e prevede anche una risposta per il chiamante.

Un'operazione One-Way ha la seguente sintassi:

```
<nomeOperazione>(<oggettoRichiesto>)
```

Un'operazione Requeste-Response ha invece la sintassi:

```
<nomeOperazione>(<oggettoRichiesto>)(<oggettoSpedito>) {
    <processoDiRisposta>
}
```

Lo standard WSDL definisce anche le operazioni speculari di output: *Notification* e *Solicit-Response*.

Notification invia un messaggio in modo da attivare la relativa One-Way in ascolto;

Solicit-Response invia il messaggio alla relativa Request-Response, e blocca l'esecuzione del servizio chiamante in attesa di una risposta dal servizio in ascolto.

La sintassi di un'operazione Notification è la seguente:

```
<nomeOperazione>@<nomePortaOutput>(<oggettoSpedito>)
```

Un comando di un'operazione Solicit-Response ha la sintassi:

```
<nomeOperazione>@<nomePortaOutput>(<oggettoSpedito>)(<oggettoRisposta>)
```

2.1.2 Engine

In JOLIE un engine è un sistema in grado di gestire sessioni di servizi sfruttando:

Creazione di sessione . La creazione di una sessione è la fase di inizializzazione di un servizio e può essere definita in maniera esplicita da parte dell'utente o implicitamente durante la ricezione di un messaggio in una operation.

Supporto per gli stati . Come supporto per gli stati si intende la capacità di una sessione di accedere a dati non locali. A tale scopo esistono tre stati: uno stato locale che comprende i dati non visibili all'esterno della sessione, uno stato globale che permette di condividere dati fra le sessioni attive nell'engine e uno stato di memorizzazione che garantisce la persistenza dei dati.

Instradamento dei messaggi . In JOLIE ogni sessione è identificata tramite correlation set, un concetto già presente in WS-BPEL che permette all'engine di instradare i messaggi verso la sessione corretta. È possibile definire il correlation set per ogni sessione e influenzare quindi l'instradamento dei messaggi.

Esecuzione di sessioni . L'esecuzione riguarda l'avvio di una sessione creata con il supporto degli stati corrispondenti. L'esecuzione di sessioni può essere sequenziale o concorrente.

2.1.3 Service Description

La descrizione di un servizio riguarda l'interfaccia esterna, il modo in cui le funzionalità vengono esposte.

In JOLIE la descrizione di un servizio si compone di due parti, interfaccia e deployment.

Interfaccia

Il concetto d'interfaccia può essere identificato in tre differenti livelli.

1. **livello funzionale**, che permette di definire quali siano le operazioni di un servizio in grado di ricevere e inviare messaggi;

2. **livello di workflow**, che descrive il workflow del behaviour;

3. **livello semantico** contenente informazioni sulla semantica delle operazioni.

La sintassi utilizzata per creare un'interfaccia è la seguente:

```
interface <nomeInterfaccia>
{
    [OneWay]:
        <operazione_1> , ... [<operazione_n>]
    [RequestResponse]:
        <operazione_1> , ... [<operazione_n>]
}
```

Le interfacce sono collegate al concetto di trovabilità dei servizi, altro aspetto cruciale delle SOA.

Deployment

L'altro componente della descrizione di un servizio è rappresentato dal deployment, ovvero l'implementazione che collega un interfaccia con un protocollo di rete. In JOLIE il deployment di un'interfaccia avviene tramite la dichiarazione di porte di input o output. Una porta è, formalmente, un endpoint dotato di un indirizzo di rete e un protocollo di comunicazione collegato a un'interfaccia le cui operazioni saranno utilizzate per inviare e ricevere messaggi.

La sintassi per la dichiarazione di porte di input (o di output) è la seguente:

```
    [inputPort] [outputPort] <nomePorta> {
        Location: <uri>:<numeroPorta>
        Protocol: [sodep] [soap] [http] [btl2cap]
        [OneWay:
            <operazione_1>] , ... [<operazione_n>]
        [RequestResponse:
            <operazione_1>] , ... [<operazione_n>]
```

```
  [Interfaces:
      <nomeInterfaccia_1>] , ... [<nomeInterfaccia_n>]
}
```

2.2 Composizione di servizi in JOLIE

Come sappiamo la composizione di servizi per crearne nuovi è una caratteristica fondamentale del paradigma SOA.

In JOLIE i servizi possono essere composti tramite *composizione semplice*, *embedding*, *redirecting* e *aggregation*.

La composizione semplice avviene tramite l'esecuzione parallela di service container che comunicano in rete.

Nell'embedding più servizi sono inseriti nello stesso container e sono quindi in grado di comunicare fra loro senza l'utilizzo di una rete.

L'embedding è l'ideale per servizi interni di manipolazione di dati che non richiedono visibilità esterna.

Nella composizione tramite redirecting un solo servizio, chiamato master, viene utilizzato per ricevere tutti i messaggi e per instradarli ai servizi sottostanti, chiamati risorse. Il servizio master sarà quindi dotato di un'unica input port e di tante output port quante le risorse utilizzate. Il client di una risorsa specifica al master il nome del servizio corrispondente senza doversi preoccupare di quali protocolli di rete saranno utilizzati dal master per raggiungere la risorsa.

Tramite l'aggregation più servizi possono essere composti mascherando completamente la struttura sottostante. Se nel redirecting un client conosce tutte le risorse subordinate al servizio master, in questo caso è visibile un unico servizio principale. Per realizzare le diverse funzionalità offerte ai client il master utilizza altri servizi, ma questi sono completamente invisibili al client.

2.3 L'architettura di JOLIE

L'algoritmo dell'interpretazione di JOLIE e la parti che compongono l'interprete sono illustrate nella Figura 2.1.

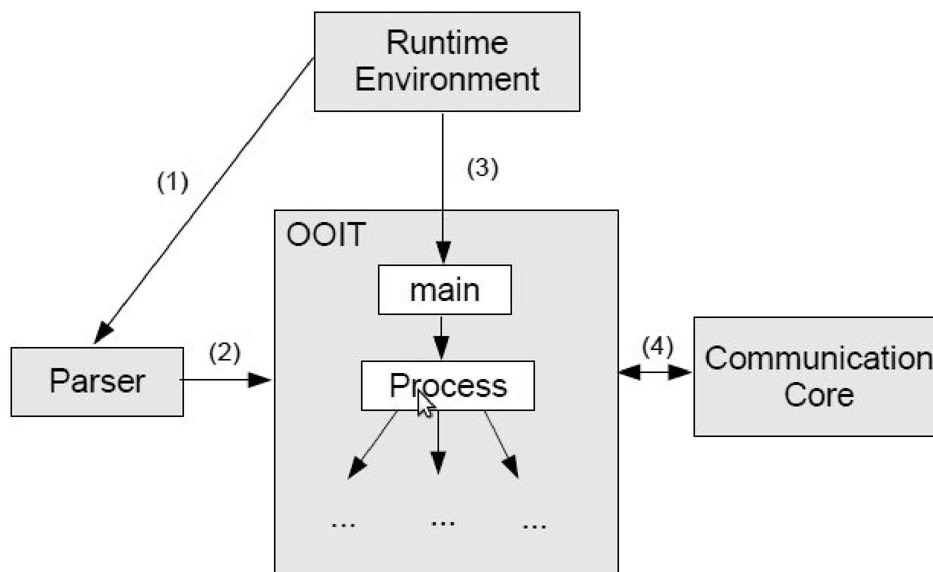

Figura 2.1: Schema dell'architettura di JOLIE

2.3.1 Analisi del codice

JOLIE ha una libreria per il parsing e analisi del codice, con la quale costruisce un albero di sintassi astratta. L'utilizzo del design pattern Visitor permette la separazione dell'algoritmo di parsing dalla struttura degli oggetti composti a cui è applicato. In questo modo è possibile aggiungere operazioni e comportamenti senza dover modificare la struttura. Anche il controllo della semantica è effettuato con questo approccio. Il parser è stato quindi scritto a mano e l'assenza di generatori di parser automatici rende la modifica di questo blocco più complicata.

2.3.2 Albero di interpretazione a oggetti

Le varie unità di esecuzione sono organizzate nell'albero di interpretazione a oggetti OOIT, che viene creato dall'albero di sintassi astratta ottimizzato.

2.3.3 Ambiente runtime

L'ambiente di esecuzione di JOLIE gestisce la creazione di nuove sessioni, sincronizzazione dei processi e interazione con altri componenti.

2.3.4 Gestore della comunicazione

Il communication core di JOLIE mantiene separato l'albero OOIT dalle problematiche relative alla comunicazione. Tale componente si occupa di instradare i messaggi alle giuste sessioni, ponendosi di fronte alle interfacce di input senza preoccuparsi del mezzo di comunicazione utilizzato e dal protocollo con cui vengono codificati i dati.

Arrivati a questo punto, dopo aver analizzato in dettaglio il contesto delle SOA e di JOLIE, concentriamoci sull'obiettivo della tesi, ovvero descrivere una grammatica per il linguaggio JOLIE.

Capitolo 3

Descrivere un linguaggio di programmazione

Secondo [4, GM06], un linguaggio di programmazione è un formalismo artificiale nel quale poter esprimere algoritmi. Ma per quanto artificiale, questo formalismo è tuttavia pur sempre un linguaggio.

Questo capitolo si pone il problema di cosa voglia dire "dare"(definire, apprendere) un linguaggio di programmazione e di quali strumenti ci si possa avvalere in quest'impresa.

3.1 Livelli di descrizione

Sempre secondo [4, GM06], in un lavoro ormai classico della linguistica, vengono studiati i vari livelli ai quali può avvenire la descrizione di un linguaggio, individuando tre grandi ambiti: quello della *grammatica*, quello della *semantica* e quello della *pragmatica*.

La grammatica è quella parte della descrizione di un linguaggio che risponde alla domanda "Quali frasi sono corrette ?". Una volta che sia definito l'alfabeto del linguaggio ad un primo stadio, quello *lessicale*, utilizzando questo alfabeto, sono individuate le sequenze corrette di simboli che costituiscono le parole(o *token*) del linguaggio. Definito alfabeto e parole, la *sintassi* passa a descrivere quali sequenze di parole costituiscono frasi legali. La sintassi è dunque una relazione tra segni: tra tutte le possibili sequenze

di parole(su un dato alfabeto), seleziona un sottoinsieme di sequenze che costituiscono le frasi del linguaggio stesso.

La semantica è quella parte della descrizione di un linguaggio che cerca di dare risposta alla domanda “Cosa significa una frase corretta ?”. La semantica attribuisce un significato ad una frase corretta. Non è difficile supporre che la semantica sia una relazione tra segni(le frasi corrette) e significati (entità autonome che esistono indipendentemente dai segni che usiamo per descriverle).

È al terzo livello che troviamo la pragmatica. Quest'ultima, infatti, è quella parte della descrizione di un linguaggio che si chiede “Come usare una frase corretta e sensata ?”. Frasi con lo stesso significato possono essere usate in modo diverso da utenti diversi. Contesti linguistici diversi possono richiedere l'uso di frasi diverse; alcune sono più eleganti di altre ed è compito del programmatore avere uno stile che rispetti la pragmatica del linguaggio.

3.2 Grammatica e sintassi

Abbiamo già detto che la grammatica di un linguaggio stabilisce prima alfabeto e lessico e quindi definisce, con la sintassi, quali sequenze di simboli corrispondano a frasi ben formate. Evidentemente, almeno dal punto di vista del linguaggio naturale, la definizione dell'alfabeto è cosa immediata. Anche il lessico potrebbe essere definito in modo semplice, ovvero elencando un insieme di vocaboli che ci interessano.

Ma come viene descritta la sintassi ? Nel caso di un linguaggio naturale, è lo stesso linguaggio naturale che viene usato(un esempio potrebbe essere un testo della grammatica delle varie lingue). Anche la sintassi dei linguaggi di programmazione può essere descritta usando il linguaggio naturale, ma questa modalità introduce spesso ambiguità nella descrizione della sintassi e, soprattutto, non è di alcun aiuto nel processo di traduzione (compilazione).

La linguistica ha sviluppato (per opera del linguista americano Noam Chomsky) alcune tecniche per descrivere i fenomeni sintattici in modo formale, cioè usando dei formalismi pensati apposta per limitare l'ambiguità che sempre si rivela nel linguaggio naturale.

Queste tecniche, note sotto il nome di *grammatiche generative*, non sono di grande aiuto nella descrizione sintattica dei linguaggi naturali, ma costituiscono uno strumento fondamentale per descrivere la sintassi dei linguaggi di programmazione.

3.2.1 Definizione della sintassi

In questo paragrafo introduciamo una notazione utilizzata per specificare la sintassi di un linguaggio.

Per specificare la sintassi, presentiamo una notazione largamente utilizzata detta grammatica libera da contesto (context-free) o BNF.

La BNF è una metasintassi, ovvero un formalismo attraverso il quale è possibile descrivere la sintassi di linguaggi formali (il prefisso meta ha proprio a che vedere con la natura circolare di questa definizione). Si tratta di uno strumento molto usato per descrivere, in modo preciso e non ambiguo, la sintassi dei linguaggi di programmazione. Benché non manchino, in letteratura, esempi di sue applicazioni a contesti anche non informatici e addirittura non tecnologici. La BNF viene usata nella maggior parte dei testi sulla teoria dei linguaggi di programmazione (e in molti testi introduttivi su specifici linguaggi).

Quella che sarà utilizzata nel corso di questa tesi è una EBNF. La EBNF è una delle varianti più conosciute della BNF. Essa è la sua forma estesa (il termine può trarre in inganno, in quanto la descrizione di un dato linguaggio redatta utilizzando la EBNF sarà tipicamente meno estesa della descrizione dello stesso utilizzando solo la BNF).

Secondo [5, ALS09], una grammatica descrive in modo naturale la struttura gerarchica dei costrutti di molti linguaggi di programmazione. Per esempio, in Java il costrutto if-else può avere la seguente forma:

if (expression) statement **else** statement

Questa scrittura indica che il costrutto è composto dalla parola chiave **if**, una parentesi aperta, un'espressione, una parentesi chiusa, uno statement, la parola chiave **else** e un altro statement. Utilizzando la variabile *expr* per indicare una generica espressione e la variabile *stmt* per indicare uno statement, questa regola di strutturazione del costrutto

può essere espressa come:

$$stmt \quad \rightarrow \textbf{if} \; (expr) \; stmt \; \textbf{else} \; stmt$$

in cui la freccia può essere letta come "può avere la forma". Una tale regola prende il nome di *produzione*. In una produzione le parole chiave come **if** e le parentesi prendono il nome di *terminali*, mentre le variabili come *stmt* ed *expr* sono dette *non-terminali*.

3.2.2 Definizione delle grammatiche

Sempre secondo [5, ALS09], una grammatica libera dal contesto ha quattro componenti.

1. Un insieme di *simboli terminali*, detti anche *token*. I terminali sono i simboli elementari con cui la grammatica definisce un linguaggio.

2. Un insieme di *simboli non-terminali*, a volte detti anche variabili sintattiche. Ogni non-terminale rappresenta un insieme di stringhe di terminali, secondo un meccanismo descritto di seguito.

3. Un insieme di *produzioni*, ognuna costituita da un non-terminale, detto *testa* o *lato sinistro* della produzione, una freccia e una sequenza di terminali e o non-terminali detta *corpo* o *lato destro* della produzione. L'obiettivo di base di una produzione è quello di indicare una possibile forma di un costrutto; in particolare, se il non-terminale della testa rappresenta un costrutto, il corpo indica una possibile forma del costrutto stesso.

4. Un *simbolo iniziale*, scelto tra i non-terminali della grammatica.

Una grammatica è quindi descritta dalla lista delle sue produzioni, a cominciare da quella relativa al simbolo iniziale, che convenzionalmente viene riportata per prima.

Assumiamo che cifre, simboli, e stringhe riportate in grassetto come per esempio **while** siano terminali. I nomi in corsivo come *expr* rappresentano non-terminali. Per convenienza notazionale, le produzioni con lo stesso non-terminale come testa possono

essere raggruppate indicando i diversi corpi come alternative separate dal simbolo |, da leggersi come "oppure".

Diciamo che una produzione è per un dato non-terminale se il non-terminale in questione è la testa della produzione. Una stringa di terminali è una sequenza di zero o più terminali. La stringa composta da zero terminali, indicata dal simbolo ϵ è detta *stringa vuota*.

3.2.3 Derivazioni

Una grammatica consente la derivazione di stringhe partendo dal simbolo iniziale e sostituendo ripetutamente un non-terminale con il corpo della produzione corrispondente. Le stringhe di terminali che possono essere derivate a partire dal simbolo iniziale formano il linguaggio definito dalla grammatica.

Il *parsing* ha l'obiettivo di scoprire come derivare una data stringa di terminali partendo dal simbolo iniziale e, nel caso in cui ciò non fosse possibile, di individuare e segnalare gli errori di sintassi rilevati nella stringa.

In generale, un programma consiste in un insieme di lessemi(unità lessicali complete), riconosciuti dalla fase di analisi lessicale e raggruppati in token. I nomi dei token sono appunto i simboli terminali elaborati dal parser.

3.2.4 Alberi di Parsing

Un *albero di parsing* rappresenta graficamente il modo in cui una stringa del linguaggio può essere derivata dal simbolo iniziale.

Formalmente, data una grammatica libera dal contesto, l'albero di parsing corrispondente è un albero con le seguenti proprietà:

1. La radice dell'albero è etichettata con il simbolo iniziale.

2. Ogni foglia dell'albero è etichettata con un terminale o con ϵ.

3. Ogni nodo interno è etichettato con un non-terminale.

4. Se A è un non-terminale associato a un nodo interno e x,y,z sono le etichette dei figli di A, ordinate da sinistra a destra, allora deve esistere nella grammatica la

produzione $A \rightarrow x\ y\ z$. Ognuna delle variabili x,y,z sta per un simbolo terminale o non-terminale. Come caso speciale, se $A \rightarrow \epsilon$ è una produzione, allora il nodo A avrà un solo figlio etichettato con ϵ.

Un esempio di albero di parsing, relativo al costrutto if-else, è mostrato in Figura 3.1

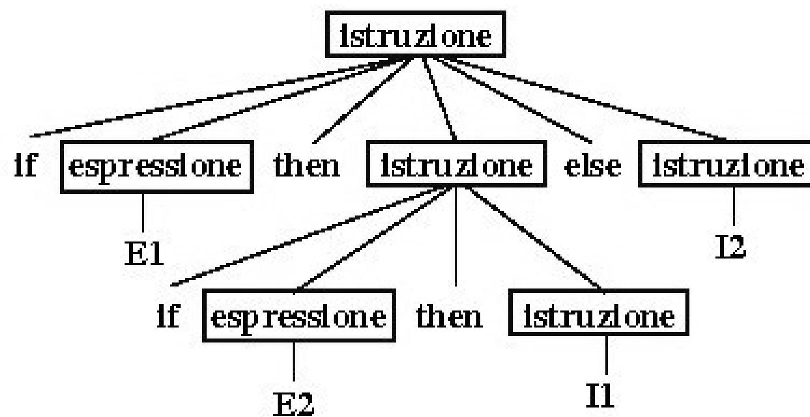

Figura 3.1: Albero di parsing per il costrutto if-else

3.3 Parsing

Il parsing è il processo che ha lo scopo di determinare come una stringa di terminali può essere generata da una grammatica.

Nel discutere questo problema è utile immaginare di costruire l'albero di parsing anche se in pratica può non essere costruito effettivamente.

Il parser riceve una sequenza di token dall'analizzatore sintattico(*lexer*), e verifica se tale sequenza può essere generata dalla grammatica del linguaggio sorgente. Ci aspettiamo che il parser sia in grado di segnalare in una forma chiara e intellegibile gli eventuali errori e, dopo quelli più comuni, sia in grado di riprendere l'analisi della parte restante del programma. Concettualmente, per i programmi ben formati(cioè sintatticamente corretti), il parser costruisce una albero di parsing. Di fatto, non è necessario costruire l'albero.

Un classico processo che ha come protagonisti lexer e parser è raffigurato nella Figura 3.2

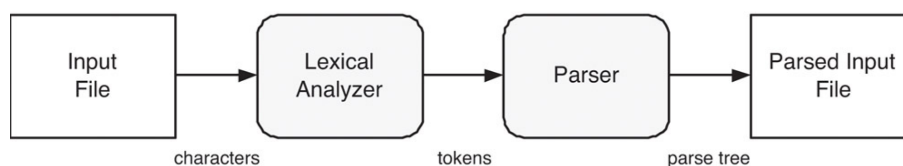

Figura 3.2: Schema di un processo con unità di lexer e parser

la maggior parte dei metodi di parsing ricade in due classi note come *top-down* e *bottom-up*. Questi termini indicano l'ordine in cui sono costruiti i nodi dell'albero.

- Nei parser top-down la costruzione inizia dalla radice e procede verso le foglie. La loro popolarità ricade principalmente alla possibilità di costruirli manualmente in modo semplice.
- Nei parser bottom-up la costruzione parte dalle foglie e procede verso la radice. Benché più complessi sono in grado di trattare una classe più ampia di grammatiche e di schemi di traduzione.

3.3.1 Parsing top-down

Come indica il rispettivo nome, i metodi top-down costruiscono l'albero di parsing partendo dalla radice(top) e scendendo verso le foglie(down). Ad ogni passo del parsing il problema cruciale sta nel determinare la procedura corretta da applicare a un determinato non-terminale A. Una volta scelta una produzione per A, il resto del processo consiste nel verificare la corrispondenza tra i simboli terminali nel corpo della produzione e la stringa d'ingresso.

Una forma generale di parsing top-down, detto anche *parsing a discesa ricorsiva* che può richiedere *backtracking* per trovare la produzione corretta da applicare per un non-terminale A. Il concetto di backtracking è quello di scegliere una certa produzione ed

eventualmente ritornare indietro (effettuare quindi backtracking) se tale produzione si rilevasse non adatta.

Questo tipo di parsing è sempre in grado di scegliere la produzione corretta analizzando il numero prefissato di simboli successivi a quello corrente, tipicamente un solo simbolo.

Parser a discesa ricorsiva

Un parser a discesa ricorsiva consiste in un insieme di procedure, una per ogni non-terminale. L'esecuzione inizia dalla procedura relativa al simbolo iniziale, che termina con successo se il suo corpo scandisce correttamente tutta la stringa d'ingresso. Il metodo generale di discesa ricorsiva può richiedere backtracking, cioè può richiedere di rileggere più di una volta una parte della stringa d'ingresso. Per implementare un parser a discesa ricorsiva la grammatica non deve presentare ricorsione sinistra, non deve cioè contenere produzioni in cui il primo termine della derivazione è lo stesso simbolo iniziale della produzione. La ricorsione sinistra potrebbe infatti causare cicli infiniti all'interno del parser.

Un parser a discesa ricorsiva è solitamente la scelta effettuata in caso d'implementazione manuale, come nel caso di JOLIE.

Parser predittivi

Una sottoclasse dei parser a discesa ricorsiva è rappresentata dai cosiddetti *parser predittivi*, che permettono di rendere più efficiente l'analisi lessicale non utilizzando backtracking. La diminuzione dell'efficienza di un parser a discesa ricorsiva è data proprio dal dover eseguire numerosi backtracking per riuscire a riconoscere la giusta produzione per un particolare input. Per risolvere questo problema, i parser predittivi utilizzano il concetto di *lookahead*. Il concetto di lookahead consiste nell'analizzare un numero arbitrario di token che seguono quello corrente per decidere che produzione selezionare. È sempre possibile costruire un parser predittivo, cioè un parser a discesa ricorsiva senza backtracking a partire da una grammatica della classe **LL(1)**.

Classi di grammatiche LL(1)

La prima “L” indica che la sequenza d’ingresso viene analizzata da sinistra (left appunto) verso destra, la seconda “L” specifica che si costruisce una derivazione sinistra (*leftmost derivation*) e infine il numero “1” fra parentesi indica che le decisioni durante il parsing vengono prese analizzando un solo simbolo di lookahead.

Per poter utilizzare un solo simbolo di lookahead senza backtracking occorre che la grammatica, oltre a non presentare ricorsione sinistra, sia caratterizzata da fattorizzazione sinistra, che permette di eliminare un eventuale prefisso comune a due parti destre della stessa produzione. Una grammatica così ottenuta è detta LL(1). L’implementazione classica di un parser LL(1) avviene mediante una tabella di parsing, che permette di indicare che produzione utilizzare per costruire l’albero in base all’input e una pila di supporto.

La costruzione dei parser top-down utilizza due funzioni, *First* e *Follow* associate a una grammatica. Queste due funzioni ci permettono di scegliere quale produzione applicare basandoci sul simbolo d’ingresso successivo.

Definiamo *First(a)*, in cui a è una generica stringa di simboli della grammatica, come l’insieme dei terminali che costituiscono l’inizio delle stringhe derivabili da a.

Definiamo *Follow(A)*, in cui A è un non-terminale, l’insieme dei simboli terminali che possono apparire immediatamente alla destra di A nelle stringhe derivate da tutte le produzioni della grammatica.

L’algoritmo classico per costruire insieme first, follow e tabella di parsing, a partire da una grammatica BNF(o EBNF) che non presenti ricorsione sinistra e fattorizzazione sinistra, è relativamente semplice ma non verrà indicato, perché non interessante ai fini di questo capitolo ed in particolar modo di questa tesi.

3.3.2 Parsing bottom-up

Il parsing bottom-up procede alla costruzione di un albero di parsing per una data stringa d’ingresso cominciando dalle foglie(bottom) e procedendo verso l’alto(up) fino alla radice, che rappresenta il simbolo iniziale della grammatica.

In linea di principio, si cerca di individuare all'interno di un input le sequenze che rappresentano la parte destra di una produzione, sostituendo poi la sequenza con il non terminale corrispondente fino a raggiungere il simbolo iniziale.

Parser shift-reduce

Il parsing shift-reduce è una forma di parsing bottom-up in cui uno stack mantiene i simboli grammaticali e un buffer contiene i simboli della parte di stringa ancora da analizzare. Scorrendo la stringa d'ingresso da sinistra a destra, il parser impila(*shift*) zero o più simboli sullo stack finché non è pronto per ridurre (*reduce*) una stringa di simboli grammaticali che si trovano sulla cima dello stack. Riducendo tale stringa, la sostituisce con la testa di un'opportuna produzione. Il parser ripete questo processo finché non rileva un errore oppure finché lo stack contiene il simbolo iniziale e il buffer di input è vuoto. Poiché l'analisi dell'input avviene tramite uno stack, questo tipo di parsing produce derivazioni *right most*, e viene indicato con la sigla **LR(k)**.

Classi di Grammatiche LR(k)

La "L" significa che la stringa di input viene scorsa da sinistra a destra, la "R" indica che si costruisce una derivazione destra in ordine inverso e infine k indica il numero di simboli di lookahead utilizzati per prendere le decisioni durante il parsing. In base al numero di simboli di lookahead e all'algoritmo di costruzione della tabella di parsing, è possibile suddividere parser e grammatiche bottom-up in:

LR(0) Un parser LR(0) non ha nessun simbolo di lookahead. Prende le decisioni shift/reduce mantenendo memorizzate informazioni di stato che gli permettono di tenere traccia di dove si trova durante l'analisi.

LR(1) Un parser LR(1) utilizza un simbolo di lookahead. Il metodo per la costruzione della collezione di insiemi di LR(1) è essenzialmente la stessa vista per la costruzione della collezione di LR(0), ma con opportune modifiche alle funzioni First e Follow.

LR(k) Un parser LR(k) è equivalente ad un parser LR(1) ma utilizza k simboli di lookahead.

SLR(1) Un Simple LR(SLR) è un parser LR che riconosce tabelle di parsing generate come per un parser LR(0), ma che effettua una riduzione (reduce) solo se il simbolo successivo in input è nell'insieme Follow. Questo parser può quindi funzionare con un numero maggiore di grammatiche. Non è tuttavia in grado di analizzare tutte le grammatiche libere dal contesto, come può invece fare un parser LR(1). Una grammatica correttamente riconosciuta da un parser SLR viene detta grammatica SLR.

LALR(1) Un lookahead LR parser o LALR parser è un parser specializzato che può trattare sia grammatiche libere dal contesto generali sia le più semplici Simple LR (SLR) parser. Un LALR parser viene costruito partendo dal parser LR(1) della stessa grammatica.

A differenza del parsing LL(k), la grammatica libera dal contesto non deve avere particolari caratteristiche, ed in particolar modo può presentare ricorsione e fattorizzazione sinistra.

È possibile quindi implementare un parser LR per qualsiasi linguaggio di cui sia disponibile una grammatica BNF. La classe di linguaggi definibili con grammatiche LR è maggiore della classe di grammatiche LL. Implementare manualmente un parser LR, specialmente nel caso dell'utilizzo di simboli di lookahead, è comunque molto complicato per via del grande numero di stati che possono essere richiesti dall'automa.

Generatori di parser come Yacc per il linguaggio C o LPG per Java entrano in questo contesto.

Capitolo 4

ANTLR: Un Generatore di Parser

ANTLR (ANother Tool for Language Recognition) è uno strumento generatore di parser e traduttori che permette di definire grammatiche sia nella sintassi ANTLR (simile alla EBNF) sia in una speciale sintassi astratta per AST (Abstract Syntax Tree). ANTLR può creare scanner, parser e AST.

ANTLR è stato sviluppato da Terence Parr, docente di informatica presso l'Università di San Francisco.

Dato che ANTLR è in competizione con i generatori di parser LR, la lettura alternativa ANT(i)-LR può non essere accidentale. Le regole in ANTLR sono espresse in un formato deliberatamente simile all'EBNF al posto della sintassi leggermente diversa utilizzata dagli altri generatori di parser.

4.1 Aspetti positivi di ANTLR

Possiamo riassumere i principali punti di forza di ANTLR:

- ANTLR è un generatore di parser che fa uso del sistema di parsing LL(k), dove k è il simbolo di lookahead.

- ANLTR integra la definizione di lessico e grammatica di un linguaggio in un unico documento;
- Altri parser generator richiedono che la sezione di input sia precedentemente suddiviso in token da uno scanner;
- ANTLR invece produce il lexer e il parser direttamente dalle regole e le produzioni della grammatica;
- Il parser generato è un parser a discesa ricorsiva, che utilizza una procedura per ogni produzione o regola della grammatica. Questa caratteristica rende il codice prodotto maggiormente comprensibile;
- ANTLR utilizza grammatiche in EBNF, che permettono l'utilizzo di operatori di cardinalità nella definizione di regole e produzioni. La grammatica prodotta risulta quindi di dimensioni più ridotte della corrispondente in BNF.
- ANTLR è un software open source, godendo di tutti i benefici che questa classe di software offre.

ANTLR nasce dall'esigenza di un parser generator di facile comprensione sia dal punto di vista della sintassi utilizzata per la definizione della grammatica e la fase di configurazione, sia dal punto di vista della struttura dell'output, cioè il `lexer` e il `parser` per un determinato linguaggio.

4.1.1 Lexer

Altri sinonimi utilizzati per riferirsi alla fase di Lexer possono essere scanner, analizzatore lessicale o tokenizzatore.

I linguaggi di programmazione sono fatti di parole chiave(keywords) e costrutti definiti in modo preciso.

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 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. Di solito 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.

Analogamente, il vostro cervello sta probabilmente raggruppando in token (parole, in tal caso, con un valore semantico per voi) i singoli caratteri che compongono questa frase, il vostro lavoro di determinare dove finisce un token e ne inizia un altro è facilitato, tuttavia, dal fatto che le parole nella frase sono già separate dagli spazi, in tal senso si potrebbe affermare che una frase in italiano è già tokenizzata, tuttavia si può assumere che una forma di raggruppamento e di riconoscimento avvenga anche a livello di parole.

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

4.1.2 Parser

Un altro sinonimo utilizzato per riferirsi alla fase di Parser può essere quello di analizzatore sintattico.

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, quest'ultime determinano 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. Il parser controlla che il token sia conforme alla sintassi del linguaggio definita dalla grammatica.

Analogamente, il vostro cervello, conosce quali tipi di frase sono validi in un dato linguaggio naturale come la lingua italiana. Si potrebbe dire che, in questo preciso momento, il vostro cervello sta parserizzando le parole di questa frase e le sta raggruppando in quello che si intende come frasi valide. Per esempio, il cervello sa che una sentenza termina quando si incontra un punto, allora non si dovrebbe considerare il testo che segue il punto come parte della stessa frase. Oltre a questo, il cervello estrae dalla frase delle informazioni significative. 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 informazione aggiuntiva per la natura della sua struttura.

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. Essi sono essenzialmente la stessa cosa, però tradizionalmente gli analizzatori sintattici sono associati ai flussi di elaborazione dei caratteri mentre i parser sono legati ai flussi di token.

4.2 Definire la grammatica di Jolie attraverso ANTLR

L'appendice A di questa tesi riporta in maniera completa la definizione della grammatica di JOLIE attraverso ANTLR. In questa sezione, invece, cerchiamo di descrivere le regole d'uso generali per utilizzare ANTLR.

Ogni definizione contiene la dichiarazione del nome della grammatica che si vuole implementare.

```
grammar jolie;
```

`jolie` è il nome del file con il suffisso `.g`, ovvero l'estensione utilizzata da ANTLR per la definizione di un file di grammatica.

4.2.1 EBNF: Simboli e notazioni in ANTLR

Facendo riferimento a [6, Par07] i simboli utilizzati da ANTLR, rispettano a grandi linee la notazione EBNF, fatta eccezione di alcune componenti introdotte nella versione 3.1 (ovvero la versione utilizzata per questo scopo).

Le notazioni descritte verranno analizzate in maniera più approfondita riportando in esempio la definizione della grammatica di JOLIE.

- `()` - Le parentesi vengono utilizzate per raggruppare elementi diversi, in modo che siano trattati come un singolo token.
- `?` - Qualsiasi token seguito da questo simbolo può verificarsi 0 o 1 volta;
- `*` - Qualsiasi token seguito da questo simbolo può verificarsi 0 o più volte;
- `+` - Qualsiasi token seguito da questo simbolo può avvenire 1 o più volte;
- `.` - Ogni token può avvenire una sola volta;
- `~` - Qualsiasi token (not) seguito da questo simbolo non può avvenire nessuna volta;
- `..`- Tale simbolo (range) abbraccia due tokens ai suoi lati, ed ha il significato di accettare una vasta gamma di caratteri tra i due confini.

L'uso di lettere minuscole e maiuscole ha una notevole importanza sotto l'aspetto implementativo della definizione di lexer e parser, in quanto differenzia regole e token della grammatica.

Lo standard di una regola EBNF è semplice:

```
program     :
        (constants |
        include |
        ports|
```

```
    interfaces|
    types |
    init|
    execution |
    define |
    embedded)*
    main?
;
```

La regola presa in considerazione rappresenta lo stato di `start rule`, ovvero lo stato iniziale di un processo di definizione.

Un file sorgente in JOLIE può contenere una serie di possibili alternative. Tali alternative sono a loro volta delle regole di produzione(nodi di un Parse Tree) che possono ricorsivamente contenerne altre, fino a quando vengono raggiunti i simboli terminali o token (foglie di un Parse Tree) rappresentati nella sezione del lexer.

Nella struttura sintattica della definizione, l'elemento a sinistra dal simbolo `:` è sostituito con un simbolo del lato destro. Il gruppo di regole, racchiuso dalle parentesi `()`, può verificarsi zero o più volte, seguita appunto dal simbolo `*`. Il simbolo `|` rappresenta le possibili alternativi, tale simbolo può essere letto o interpretato come `OR`. Difatti in un file sorgente JOLIE si possono esprimere più parti di definizione di vari costrutti o macro.

Possiamo invece notare che la regola del `main`, seguita dal simbolo `?`, può verificarsi zero o una volta, in quanto non è possibile definire, all'interno di un sorgente JOLIE (come in tutti i linguaggi), più definizioni della procedura main.

4.2.2 Differenza tra lexer e parser in ANTLR

L'uso di EBNF non distingue regole del lexer con le regole del parser, ma solo simboli terminali e simboli non terminali.

ANTLR adotta la convenzione del `case sensitive`. Le regole del lexer devono iniziare con una lettera maiuscola, mentre quelle del parser con una lettera minuscola.

Le regole del lexer contengono solo letterali o riferimenti ad altre regole del lexer. Le regole del parser possono fare riferimento ad altre regole del parser o indifferentemente a regole del lexer. Tali regole possono contenere dei letterali, ma non possono contenere unicamente letterali.

Prendendo in esempio la nostra definizione della grammatica per JOLIE, si può osservare come le regole del lexer siano concettualmente ben diverse dalle regole del parser.

```
ID        : ('a'..'z' | 'A'..'Z' | '_')
            ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
;

constants : CONSTANT '{' ID ASSIGN ( INT | REAL | ID | STRING )
            (COMMA  ID ASSIGN (INT| REAL | ID | STRING ) )* '}'
;
```

4.2.3 Token in ANTLR

Ci sono due modi per esprimere il concetto di token. Il primo metodo è quello di esprimere i tokens attraverso le regole del lexer. Una seconda alternativa, quella più utilizzata e consigliata dall'autore, è usare il comando speciale di `token{}`.

Tale comando ha una serie di vantaggi rispetto alla regole del lexer.

1. Le definizioni dei nomi simbolici sono tutti raccolti in un unico spazio della definizione.

2. Le parole chiave (keywords) definite in questo modo hanno la priorità sulla regola per i normali nomi di variabile.

Ad esempio, la keyword `include` può anche corrispondere con la regola più generale `ID : "('a' .. 'z')+"`. Nel caso in cui un linguaggio offre una vasta gamma di keywords, come JOLIE, la soluzione più efficace è quella di utilizzare il comando `token{}`.

Tornando al nostro caso, questo comando è stato d'ausilio per definire le keywords di JOLIE.

```
token{
    CONSTANT  = 'constants';
    EXECUTION = 'execution';
    INCLUDE   = 'include';
    DEFINE    = 'define';
    INIT      = 'init';
    MAIN      = 'main';
    ..
    ..
}
```

4.2.4 Opzioni in ANTLR

ANTLR permette il corpo delle opzioni da inserire all'inizio del file della grammatica.

```
options {
    backtrack = true;
    k = 1;
}
```

La nuova grande funzionalità nella nuova versione di ANTLR è l'opzione per la grammatica`backtrack = true`. In questo caso per risalire ad una regola si va a ritroso effettuando la tecnica del backtracking analizzata nel precedente capitolo.

Il termine in esame durante la scansione è detto simbolo di lookahead e tramite il comando `k=1` si assegna un solo simbolo per la scansione da effettuare. Per la grammatica in questione è stato ritenuto sufficiente un solo simbolo di lookahead.

4.3 ANTLRWorks

ANTLRWorks è un nuovo ambiente di sviluppo per la grammatica ANTLR v3. Esso combina un eccellente editor di grammatica e un diagramma di sintassi che aiuta a

evidenziare i percorsi delle regole del parser e del lexer associate ad una grammatica. ANTLRWorks ha come obiettivo quello di migliorare la manutenibilità e la leggibilità delle grammatiche, fornendo una navigazione eccellente e una serie di strumenti di refactoring.

4.3.1 Syntax Diagram

Uno strumento molto importante all'interno di ANTLRWorks è il Syntax Diagram che consente di visualizzare, in maniera quasi simile ad un albero sintattico, le regole del lexer e del parser. La regola iniziale (start rule) è rappresentata attraverso il Syntax Diagram in figura 4.1.

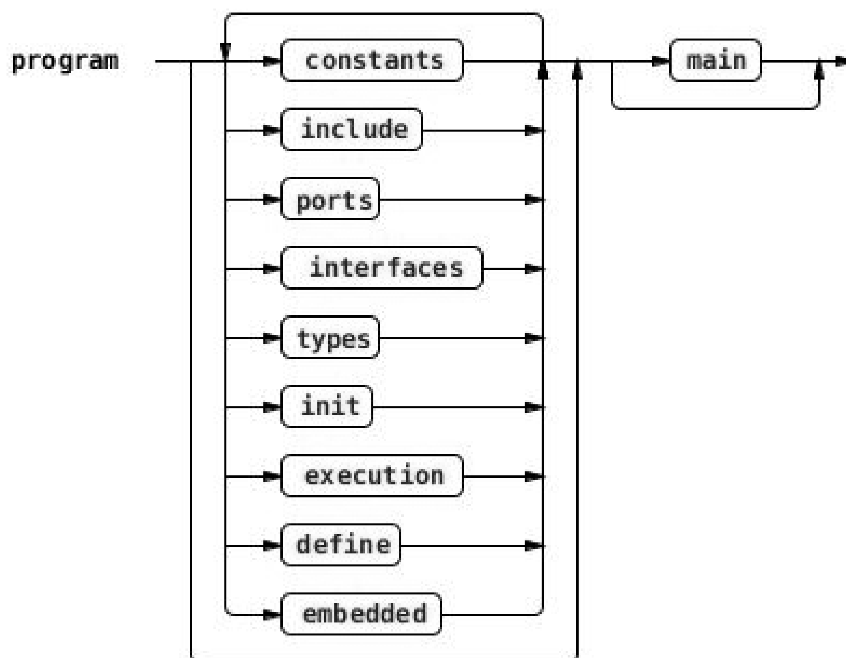

Figura 4.1: Start rule program in ANTLR

Dopo aver definito in maniera esatta la grammatica di JOLIE, siamo riusciti ad avere come output il nostro lexer e il nostro parser (nel linguaggio java). Ogni piccola modifica alla grammatica si ripercuoteva nel codice del parser.

Attraverso uno strumento di debug, all'interno di ANTLRWorks, si è verificato che il parser generato accettava in ingresso istruzioni, costrutti e keywords specifici di JOLIE.

4.3.2 Parse Tree

Per la rappresentazione del Parse Tree, utilizziamo uno strumento all'interno di ANTLRWorks : ANTLR Parse Tree. Scrivendo un frammento di codice di esempio, il parser analizza le regole necessarie e restituisce come output il Parse Tree di riferimento.

Il frammento di codice utilizzato come esempio è il seguente:

```
inputPort MathService {
Location: "socket://localhost:8000"
Protocol: sodep
RequestResponse:
twice
}
```

Nella costruzione del Parse Tree possiamo notare come la radice rappresenta lo start rule, ovvero la regola iniziale le parser (`program`). Tale regola seleziona come nodo un'altra sottoregola, fino ad arrivare alla regola per la dichiarazione di porte (`inputportstatement`). A questo punto i rami del Parse Tree vengono smistati in più regole che riguardano le varie componenti che compongono la definizione di una porta in JOLIE, ovvero `location`, `protocol`, `requestresponseoperation`.

Si può osservare come i nodi dell'albero sono costituiti dalle regole del parser, mentre le sue foglie vengono costruite attraverso le regole del lexer. Il Parse Tree corrispondente è visualizzato in figura 4.2.

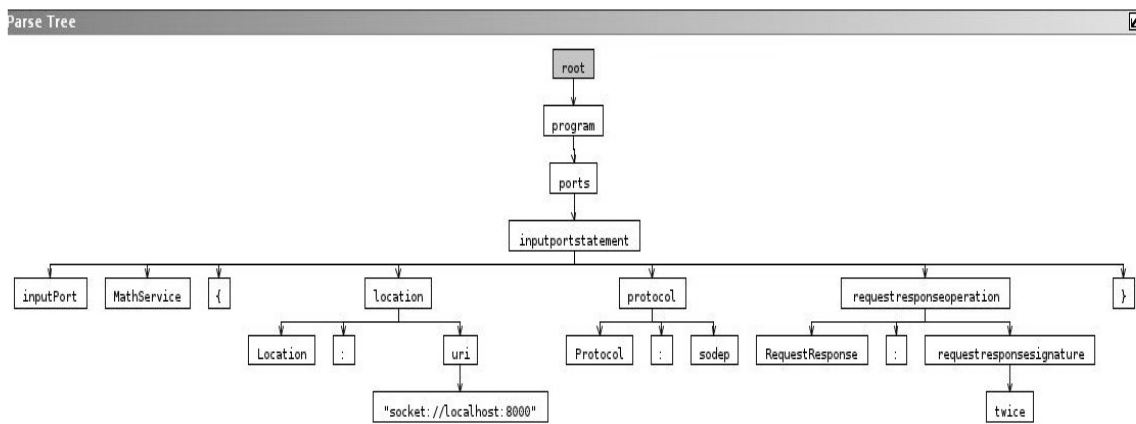

Figura 4.2: Parse Tree per la definizione di inputPort

Conclusioni e sviluppi futuri

Questo lavoro di tesi è nato nel contesto dell'emergente paradigma SOA e del linguaggio JOLIE. L'obiettivo principale è stato definire una grammatica che riconoscesse nel parser i costrutti del linguaggio.

La definizione della grammatica EBNF del linguaggio JOLIE ha richiesto un'analisi del parser attuale di JOLIE, che ricordiamo è stato scritto a mano senza l'ausilio di generatori automatici come ANTLR.

La necessità di sostituzione del parser attuale con uno generato attraverso uno strumento automatico, è dovuto al fatto di poter estendere o modificare nuovi costrutti del linguaggio JOLIE, concentrandosi solo sulla grammatica e sull'interprete, senza più dover considerare l'analisi sintattica e lessicale. La potenzialità risiede nel fatto che JOLIE risulta essere modularmente estendibile e in continuo sviluppo, riuscendo a soddisfare le esigenze del paradigma SOA. Una semplice modifica alla definizione della grammatica risulterebbe potenzialmente utile alla rigenerazione del parser in maniera automatica.

Alcuni problemi riscontrati riguardano le regole del parser nella dichiarazione di operazioni OneWay e RequesteResponse. Tali regole devono essere ridefinite in modo da accettare qualunque frammento di codice ed essere compatibile al 100% con il parser attuale di JOLIE.

Il Parse Tree viene utilizzato per effettuare la fase di parsing ed è la stringa di input trasformata in un albero (togliendo dal codice tutte le informazioni non necessarie all'interpretazione del linguaggio). Quello che servirebbe a noi per proseguire il progetto è l'albero di sintassi astratta, cioè l'output della fase di parsing. Per rendere utilizzabile il nuovo parser all'interno di JOLIE occorre che l'AST prodotto dal parser sia identico all'AST utilizzato all'interno dell'interprete di JOLIE. Nella grammatica definita non sono

definite le azioni semantiche, che sono i costrutti che permettono di modellare l'AST a livello della grammatica.

La caratteristica che ANTLR offre, rispetto ad altri parser generator, è proprio questa. ANTLR permette, in un unico file, di costruire lexer, parser e definire la struttura dell'AST.

Le regole semantiche permettono di definire gli oggetti restituiti da una regola.

Prendendo in esempio una regola del parser, `protocol`, possiamo aggiungere informazioni semantiche semplicemente aggiungendo del codice al costrutto :

```
protocol return Protocol {...}:
        PROTOCOL COLON ID
        protocolconfiguration?
    ;
```

Il significato della regola ora presenta un'aggiunta di informazioni:

1. La regola specifica la sintassi per la dichiarazione di un protocollo;

2. Nel momento in cui ANTLR riconosce la regola `protocol` aggiunge all'AST un oggetto di tipo Protocol che avrà le variabili settate secondo le direttive tra parentesi `{...}`, in questo modo, alla fine della fase di parsing, l'albero sarà aumentato con degli oggetti che rappresentano i costrutti del linguaggio.

Lo sviluppatore del linguaggio, oltre ad aver scritto il codice del parser, ha creato una serie di classi che rappresentano gli elementi del linguaggio.

Sfruttando ANTLR a 360 gradi, possiamo continuare il progetto aggiungendo le azioni semantiche, permettendo in questo modo l'istanziazione di nuovi oggetti.

Lo sviluppo successivo estremamente utile è quello di inserire nelle azioni semantiche della grammatica di ANTLR le classi del sorgente di JOLIE, le stesse classi definite dallo sviluppatore del linguaggio.

In questo modo, quando il parser generato da ANTLR conclude il suo lavoro, ci troviamo con un AST identico a quello generato dal parser attuale ed il compito dell'interprete, sarà semplificato dall'AST preso in input per produrre il risultato.

Un altro uso allo stesso tempo valido potrebbe essere quello di continuare il progetto di Joliepse, un IDE per JOLIE. Joliepse è stato sviluppato attraverso il framework Xtext, ma come è stato evidenziato dai test eseguiti tramite strumenti di profiling del codice, il framework risulta essere particolarmente lento. Il passo successivo per lo sviluppo del progetto Joliepse è la progettazione di un traduttore automatico da una grammatica ANTLR ad una grammatica Xtext.

Appendice A

La definizione della grammatica per JOLIE

In questa appendice è stata riportata per esteso la definizione della grammatica di JOLIE attraverso l'uso di ANTLR. Alcuni concetti base sono stati introdotti durante la stesura del quarto capitolo.

```
/*
   ANTLR v3 Grammar for Jolie
   file version 1.00, July 17, 2011
   strict LL(1)
*/

grammar jolie; // language jolie

options {
    backtrack = true;
    k =1;
}


tokens {
```

```
//KEYWORDS

CONSTANT          = 'constants';
EXECUTION         = 'execution';
INCLUDE           = 'include';
TYPE              = 'type';
DEFINE            = 'define';
INIT              = 'init';
MAIN              = 'main';
EMBEDDED          = 'embedded';
LINKIN            = 'linkIn';
LINKOUT           = 'linkOut';
CH                = 'cH';
EXIT              = 'exit';
SCOPE             = 'scope';
THROW             = 'throw';
THROWS            = 'throws';
INSTALL           = 'install';
COMPENSATE        = 'comp';
SYNCHRONIZED      = 'synchronized';
UNDEF             = 'undef';
IF                = 'if';
ELSE              = 'else';
FOR               = 'for';
WHILE             = 'while';
WITH              = 'with';
FOREACH           = 'foreach';
THIS              = 'this';
INPUTPORT         = 'inputPort';
OUTPUTPORT        = 'outputPort';
```

```
ONEWAY          = 'OneWay';
REQUESTRESPONSE = 'RequestResponse';
LOCATION        = 'Location';
PROTOCOL        = 'Protocol';
INTERFACES      = 'Interfaces';
INTERFACE       = 'interface';
AGGREGATES      = 'Aggregates';
REDIRECTS       = 'Redirects';
CONCURRENT      = 'concurrent';
SEQUENTIAL      = 'sequential';
JAVA            = 'Java';
JOLIE           = 'Jolie';
JAVASCRIPT      = 'Javascript';
ISDEFINED       = 'is_defined';
ISSTRING        = 'is_string';
ISDOUBLE        = 'is_double';
ISINT           = 'is_int';
GLOBAL          = 'global';
NULLPROCESS     = 'nullProcess';

//TERMINAL

SEMICOLON       = ';';
COLON           = ':';
PLUS            = '+';
MINUS           = '-';
VERT            = '|';
ASSIGN          = '=';
DOT             = '.';
COMMA           = ',';
AT              = '@';
```

```
    CHOICE           = '++';
    DECREMENT        = '--';
    ASTERISK         = '*';
    QUESTION         = '?';
    DIVIDE           = '/';
    POSTINTO         = '->';
    DEEPCOPYLEFT     = '<<';
    PERCENT          = '%';
    EQUAL            = '==';
    LANGLE           = '<';
    RANGLE           = '>';
    HASH             = '#';
    MAJOR_OR_EQUAL   = '>=';
    MINOR_OR_EQUAL   = '<=';
    NOT_EQUAL        = '!=';
    NOT              = '!';
    AND              = '&&';
    OR               = '||';
    REDIR            = '=>';

}


// -------------------------------------------------------------
// L E X E R   G R A M M A R
// -------------------------------------------------------------


INT:
        ('0'..'9')+
    ;
```

```
ID:
        ('a'..'z' | 'A'..'Z' | '_')
        ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
    ;

STRING:
        '"' ('a'..'z' | 'A'..'Z' | '_' |'0'..'9' | '.' |
        ':' | '/' | '\\' |' '| ','| ';' )+ '"'
    ;

REAL:
        ('0'..'9')* DOT ('0'..'9')+
        ( ('E'|'e') ('0'..'9')+ )?
    ;

WS:
        (' '|'\r'|'\t'|'\u000C'|'\n')
        {$channel=HIDDEN;}
    ;

ML_COMMENT:
        '/*' ( options {greedy=false;} : . )* '*/'
        {$channel=HIDDEN;}
    ;

SL_COMMENT:
        '//' ~('\n'|'\r')* '\r'? '\n'
        {$channel=HIDDEN;}
    ;
```

```
// -------------------------------------------------------------
// PARSER  G R A M M A R
// -------------------------------------------------------------


program:
        (constants |
        execution |
        include |
        types |
        embedded |
        define |
        init |
        ports |
        interfaces |
        declar_interface)*
        main?

    ;

constants:
        CONSTANT '{' ID ASSIGN ( INT | REAL | ID )
        (COMMA  ID ASSIGN (INT| REAL | ID ) )* '}'
    ;

execution:
        EXECUTION '{'(CONCURRENT | SEQUENTIAL) '}'
    ;

include:
```

```
        INCLUDE STRING
    ;

types:
        TYPE ID COLON native_type  (typedef)?
    ;

typedef:
         '{' subtypes '}'
    ;

subtypes:
         (DOT  ID cardinality? COLON native_type typedef * )*
        | QUESTION
    ;

native_type:
        'any' |'int' | 'real' | 'string' | 'void' |
        'undefined' | 'double' | 'raw' |
        types
    ;

cardinality:
        QUESTION |  ASTERISK |
        ('[' INT COMMA (INT| ASTERISK) ']')
    ;

embedded:
        EMBEDDED '{'  ( JAVA | JOLIE | JAVASCRIPT  )
        COLON STRING (ID ID)?
        (COMMA STRING (ID ID)? )* '}'
```

```
    ;

define:
        DEFINE  ID mainprocess
    ;

init:
        INIT mainprocess
    ;

main:
        MAIN mainprocess
    ;

mainprocess:
        '{' parallelstatement '}'
    ;

process:
        '{' parallelstatement '}' |
        '('parallelstatement ')'
    ;

parallelstatement:
           (sequencestatement (VERT sequencestatement)*)
    ;

sequencestatement:
        (basicstatement (SEMICOLON basicstatement)*)
    ;
```

```
ports:
        (inputportstatement |
        outputportstatement)
    ;

inputportstatement:
        INPUTPORT  ID '{' ((location)?
        (protocol)?  (onewayoperation)?
        (requestresponseoperation)?
        (redirects)?  (aggregates)?
        (interfaces)?) '}'
    ;

redirects:
        REDIRECTS COLON redirectdef
        (COMMA redirectdef)*
    ;

redirectdef:
        ID REDIR ID
    ;

aggregates:
        AGGREGATES COLON ID (COMMA ID)*
    ;

outputportstatement:
        OUTPUTPORT  ID '{' ((location)?
        (protocol)?
        (onewayoperation)?
        (requestresponseoperation)?
```

```
        (interfaces)?)  '}'
    ;

interfaces:
        INTERFACES COLON declar_interface
        (COMMA declar_interface)*
    ;

declar_interface:
        INTERFACE  ID '{'
        requestresponseoperation?
        onewayoperation? '}'
    ;

location:
        LOCATION COLON uri
    ;

uri:
        ID | STRING
    ;

protocol:
        PROTOCOL COLON ID
        protocolconfiguration?
    ;

protocolconfiguration:
        mainprocess
    ;
```

```
onewayoperation:
        ONEWAY COLON
        onewayoperationsignature
    ;

requestresponseoperation:
        REQUESTRESPONSE COLON
        requestresponsesignature
    ;

onewayoperationsignature:
        ID ('(' typeofthrow ')')?
        (COMMA onewayoperationsignature)?
    ;
typeofthrow:
        types | native_type
    ;

requestresponsesignature:
        ID ('(' typeofthrow ')'
        '(' typeofthrow ')')?
       (THROWS (throwsclause)+)?
       (COMMA requestresponsesignature)?
    ;

throwsclause:
        ID '(' (typeofthrow)? ')'
    ;

basicstatement:
        process |
```

```
        assignstatementorpostincrementdecrementorinputoperation |
        ndchoicestatement |
        preincrementdecrement |
        with |
        declar_synchronized |
        undef |
        declar_for |
        declar_if |
        foreach |
        declar_while |
        NULLPROCESS |
        linkIn |
        linkOut |
        define |
        onewayoperationsignature |
        requestresponsesignature |
        inputoroutputoperationdeforcall |
        scoping |
        compensate |
        declar_throw |
        install |
        ch |
        exit
    ;


ndchoicestatement:
        ('['  LINKIN '(' ID ')' |
        (onewayoperationsignature |
        requestresponsesignature
        inputoroutputoperationdeforcall ) ']'
```

```
        mainprocess)+
    ;

is_function:
        (ISDEFINED | ISSTRING | ISDOUBLE | ISINT)
        '('variablepath ')'
    ;

declar_throw:
        THROW '(' ID (COMMA expression)* ')'
    ;

install:
        INSTALL '(' installfunction ')'
    ;

compensate:
        COMPENSATE ID mainprocess
    ;

scoping:
        SCOPE '(' ID ')' mainprocess
    ;

linkIn:
        LINKIN '(' ID ')'
    ;

linkOut:
        LINKOUT '('ID ')'
    ;
```

```
declar_synchronized:
        SYNCHRONIZED '(' ID ')' mainprocess
    ;

inputoroutputoperationdeforcall:
        '(' variablepath? ')' inputoperation
        AT outputportcall
    ;

installfunction:
        (ID | THIS)
        REDIR parallelstatement (COMMA (ID | THIS)
        REDIR parallelstatement)*
    ;

expression:
        terminalexpression
        ((PLUS | MINUS | ASTERISK | DIVIDE) expression)?
    ;

assignstatementorpostincrementdecrementorinputoperation:
        variablepath rightside
    ;

rightside:
        ASSIGN expression |
        CHOICE |
        DECREMENT |
        POSTINTO variablepath |
        DEEPCOPYLEFT variablepath
```

```
    ;

terminalexpression:
        '(' expression ')' |
        MINUS? INT |
        MINUS? REAL |
        STRING |
        (CHOICE | DECREMENT) variablepath |
        variablepath (CHOICE | DECREMENT)?
    ;

variablepath:
        HASH? DOT? ID
        ('[' expression ']')?
        ( ( (DOT (ID |GLOBAL) ('['expression ']')?) |
        (DOT '(' expression')') ) )* |
        GLOBAL ( ( (DOT (ID | GLOBAL) ('[' expression ']')?) |
        (DOT '(' expression ')') ) )*
    ;

with:
        WITH '(' variablepath ')' mainprocess
    ;

declar_while:
        WHILE '(' condition ')' mainprocess
    ;

undef:
        UNDEF '(' variablepath ')'
    ;
```

```
preincrementdecrement:
        (CHOICE | DECREMENT) variablepath
    ;

condition:
        NOT condition |
        '(' condition')' ((AND |OR) condition)? |
        (variablepath | INT | STRING | is_function)
        rightcondition? ((AND | OR) condition)?
    ;

rightcondition:
        (EQUAL | LANGLE | RANGLE | MAJOR_OR_EQUAL |
        MINOR_OR_EQUAL | NOT_EQUAL) expression
    ;

declar_for:
        FOR '(' parallelstatement COMMA
        condition COMMA parallelstatement body
    ;

foreach:
        FOREACH '(' variablepath COLON variablepath body
    ;

declar_if:
        IF '(' condition ')' basicstatement
        (ELSE basicstatement | declar_if)?
    ;
```

```
body:
        ')' basicstatement
    ;

inputoperation:
        ('(' expression? ')' mainprocess)?
    ;

outputportcall:
       inputportstatement | outputportstatement
        '(' expression? ')'
        ('(' (variablepath)? ')'
        ('[' installfunction ']')?)?
    ;

ch:
        CH
    ;

exit:
        EXIT
    ;
```

Bibliografia

[1] [Mlm06] MacKenzie C.M, Laskey K., McCabe F., Brown P.F., Metz R. ,Hamilton B.A., OASIS SOA Reference Model TC. Reference model for service oriented architecture, Technical report, OASIS, 2006.

[2] [Pap07] Papazoglou M.P, Willem-Jan van den Heuvel W., Service oriented architectures: approaches, technologies and research issues, 2007.

[3] [MGLZ06] Montesi F., Guidi C., Lucchi R., Zavattaro G.. JOLIE: a Java Orchestration Language Interpreter Engine.

[4] [GM06] Gabbrielli M., Martini S., Linguaggi di programmazione: Principi e paradigmi, McGraw-Hill , 2006.

[5] [ALS09] Aho A.V., Lam M.S, Sethi R., Ullman J.D., Compilatori: principi tecniche e strumenti , Pearson, Addison Wesley, 2009.

[6] [Par07] Parr T., The definitive ANTLR Reference, Building Domain-Specific Languages , The pragmatic programmers, 2007.