Appunti COOOOOOOMPIILATOOOORIIII

A.A. 2022/2023

Indice

1 Compilatori **5**
- 1.1 Introduzione . . . 5
  - 1.1.1 Differenze tra compilatore ed interprete . . . 5
- 1.2 Struttura di un compilatore . . . 6
  - 1.2.1 Fasi del front-end . . . 7
  - 1.2.2 Fasi del back-end . . . 8
- 1.3 Tipi di linguaggi . . . 8
- 1.4 Alcune domande d'esame . . . 9
- 1.5 Esercizi . . . 9
- 1.6 Binding e tempo di binding . . . 10
  - 1.6.1 Alcuni esempi . . . 11
  - 1.6.2 Tipi di Binding . . . 12
- 1.7 Compilatore Just-In-Time . . . 12

2 Analisi lessicale **13**
- 2.1 Analizzatore lessicale . . . 13
  - 2.1.1 Analisi lessicale e parsing . . . 13
  - 2.1.2 Token, patter e lessemi . . . 14
- 2.2 Espressioni regolari . . . 15
  - 2.2.1 Stringhe e linguaggi . . . 15
  - 2.2.2 Definizioni regolari . . . 15
- 2.3 Riconoscimento dei token . . . 16
  - 2.3.1 Riconoscimento identificatori chiave . . . 18
- 2.4 Lavorare con JFlex . . . 19
  - 2.4.1 Sezione usercode . . . 20
  - 2.4.2 Opzioni e dichiarazioni . . . 20
  - 2.4.3 Macro per il metodo di scansione . . . 21
  - 2.4.4 Macro per la fine del file . . . 21
  - 2.4.5 Macro per gli stati . . . 21
  - 2.4.6 Struttura delle espressioni regolari in jflex . . . 22
  - 2.4.7 Altre variabili del lexer . . . 22
  - 2.4.8 Priorità match Espressioni regolari in Jflex . . . 22

3 Analisi sintattica **23**
- 3.1 Parser . . . 23
- 3.2 Grammatiche Context-free . . . 23
  - 3.2.1 Derivazioni . . . 24
  - 3.2.2 Albero di parsing e derivazioni . . . 24
  - 3.2.3 Eliminare ambiguità . . . 25
  - 3.2.4 Grammatica ricorsiva sinistra . . . 25

3.3 Parser top-down . . . 26
3.3.1 First . . . 27
3.3.2 Follow . . . 29
3.3.3 Algoritmo per la costruzione di una tabella per un parsing predittivo . . . 29
3.3.4 Esercizio First, Follow e tabella di parsing . . . 30
3.4 LL(1) . . . 31
3.4.1 Parsing predittivo non ricorsivo . . . 32
3.5 Parsing bottom-up . . . 35
3.5.1 Insieme delle grammatiche (prima parte) . . . 35
3.5.2 Parser shift-reduce . . . 36
3.6 Parser LR . . . 37
3.6.1 Item LR(0) e automa LR(0) . . . 38
3.6.2 Parser SLR(1) . . . 41
3.6.3 Parser LR(1) . . . 42
3.6.4 Grammatiche LALR(1) . . . 44
3.6.5 Precedenze per risolvere i conflitti . . . 44
3.6.6 Esercizio Handle ben Marcato . . . 45
3.6.7 Esercizio handle di questa produzione . . . 45
3.7 Grammatiche ad Attributi . . . 46
3.7.1 Definizioni guidate dalla sintassi . . . 46
3.7.2 Grammatiche S e L attribuite . . . 46
3.7.3 Valutazione di una grammatica ad attributi con un albero . . . 47
3.7.4 Schema di traduzione . . . 50
3.7.5 Esercizi sulle grammatiche ad attributi . . . 51
3.7.6 Grammatiche ad attributi per costruire un albero . . . 54

4 Analisi semantica **57**
4.1 Scoping . . . 57
4.1.1 Scoping statico e struttura a blocchi . . . 57
4.2 Tabella dei simboli . . . 58
4.2.1 Tabella dei simboli e scoping . . . 58
4.3 Type Checking . . . 59
4.3.1 Regole d'inferenze di tipo . . . 59
4.3.2 Esercizio sulla creazione di un type environment . . . 61

5 Generazione codice intermedio **62**
5.1 Struttura del codice a tre indirizzi . . . 62
5.1.1 Grammatica ad attributi per generare codice a tre indirizzi . . . 65
5.2 Backpatching . . . 66
5.2.1 Backpatching sulle espressioni booleane . . . 66
5.2.2 Backpatching su statement per il controllo del flusso . . . 67

6 Generazione codice macchina **69**
6.1 Codici intermedi . . . 69
6.2 Infrastrutture ibride . . . 69
6.3 Run-time environments . . . 70
6.3.1 Organizzazione della memoria . . . 70
6.3.2 Allocazione statica e dinamica . . . 72
6.3.3 Allocazione della memoria a stack . . . 72
6.3.4 Record di attivazione . . . 73
6.4 Stack machine . . . 74
6.5 MIPS . . . 75

6.5.1 Strategia per generare codice MIPS 76

7 Esercizi a cazzo di cane di compilatori **79**

Capitolo 1

Compilatori

1.1 Introduzione

Un compilatore è un programma che prendendo in input un programma in linguaggio sorgente (source program), lo traduce in un programma equivalente scritto però in un altro linguaggio destinazione (target program).

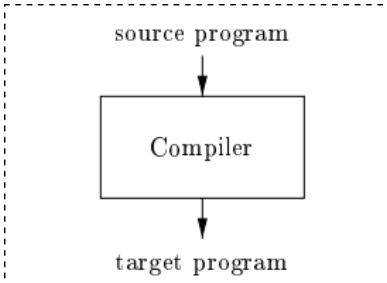
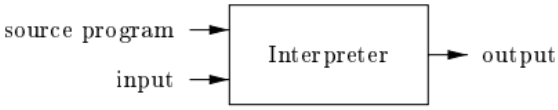


Figure 1.1: A compiler

Un interprete è un altro tipo comune di elaboratore di linguaggi. Invece di produrre un programma destinazione, un interprete esegue direttamente le operazioni specificate nel programma sorgente sugli input forniti dall'utente.



1.1.1 Differenze tra compilatore ed interprete

- **Compilatore**: Un compilatore è molto più efficiente, mostra gli errori in fase di scrittura del codice. L'essere tipizzato migliora le prestazioni perché si sa già quanto spazio dedicare in memoria. Come contro abbiamo che il codice non è portabile su architetture diverse da quelle in cui è compilato.
- **Interprete**: I linguaggi interpretati sono più semplici da utilizzare, gli errori sono segnati a tempo di esecuzione, l'omissione del tipo delle variabili porta ad un calo delle prestazioni perché la memoria sarà allocata e deallocata durante l'esecuzione. Hanno una migliore portabilità visto che devono soltanto essere eseguiti.

Nota: Alcuni linguaggi sono sia compilati che interpretati. Un esempio è Java che viene compilato in bytecode(.class) e poi quest'ultimo viene interpretato dalla JVM.
Compilatore Just in time: è un compilatore integrato nell'interprete che nei momenti in cui la CPU è libera, quest'ultima oltre ad interpretare, compila anche alcune funzioni in bytecode per sfruttare i vantaggi sia della compilazione nativa che quella del bytecode.

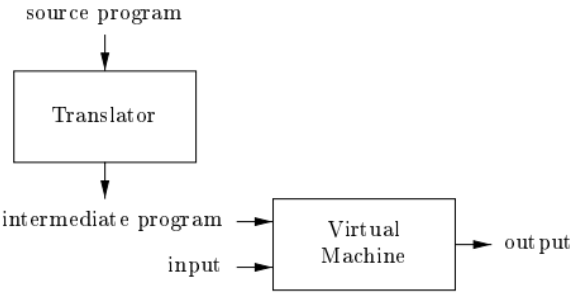


Figure 1.4: A hybrid compiler

Per creare un programma eseguibile, oltre al compilatore, sono necessari anche altri programmi:
Preprocessore: Prende in input il codice sorgente e lo estende con tutte le librerie dichiarate, è anche in grado di espandere le macro presenti all'interno del codice. Questo sorgente sarà poi passato al compilatore;
Compilatore: Prende in input un sorgente e genera codice Assembly;
Assembler: Prende in input il codice Assembly e genera codice macchina rilocabile. Un programma si dice rilocabile quando può essere caricato per l'esecuzione in qualsiasi punto della memoria.
Linker/Loader: Il linker risolve i riferimenti esterni che si presentano quando il codice in un file fa riferimento a una locazione di memoria o a un simbolo definito in un altro file. Il loader, infine, si occupa di caricare in RAM tutti i file eseguibili, rendendoli pronti per l'esecuzione.
Programmi residenti: Programmi che rimangono permanentemente in una zona di memoria precisa del computer. *Esempio: Antivirus.*

1.2 Struttura di un compilatore

Un compilatore si divide in due parti:

- Analisi (Front-end): Suddivide il programma sorgente in lessemi (parole con significato) in modo da controllare la corretta sintassi e semantica del codice. Inoltre tutte le informazioni raccolte sono inserite in una tabella dei simboli (symbol table). Fanno parte di questa fase: analizzatore lessicale, analizzatore sintattico, analizzatore semantico, generatore del codice intermedio. **Il front-end dipende dal linguaggio che si vuole implementare.**
- Sintesi (Back-end): A partire dalle informazioni contenute nella tabella dei simboli e dalla rappresentazione intermedia del codice costruisce il programma di destinazione. Fanno parte di questa fase: ottimizzatore del codice, generatore del codice macchina. **Il back-end dipende dal linguaggio target.**

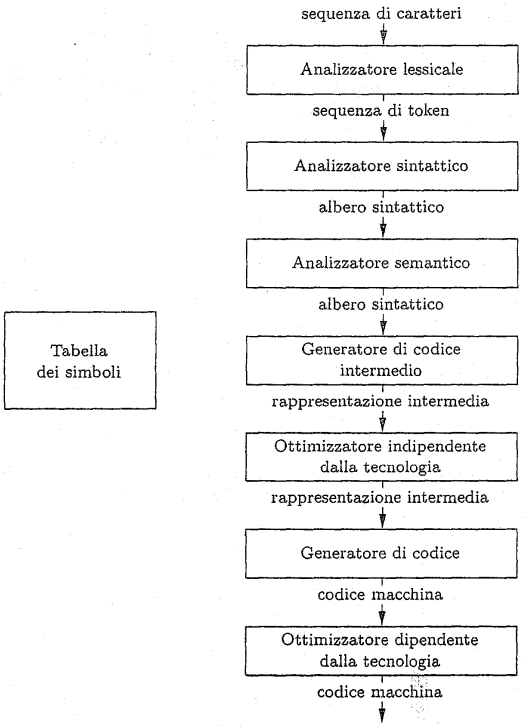


Figura 1.6 Fasi di compilazione.

Di seguito verranno spiegate brevemente le varie fasi tramite un esempio:

$$\text{Position} = \text{initial} + \text{rate} * 60$$

1.2.1 Fasi del front-end

Analizzatore lessicale

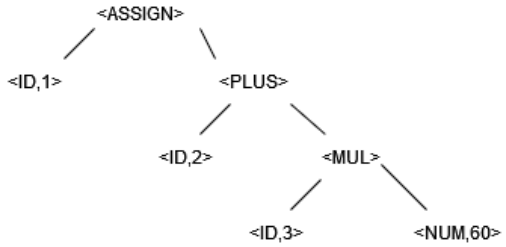
Legge il programma sorgente come flusso di caratteri e li raggruppa in lessemi, per ogni lessema produce un token con forma <nome-token, valore-attributo>, passerà i lessemi alla fase successiva insieme ad una prima tabella dei simboli.
Esempio:

<ID,1><ASSIGN><ID,2>,<PLUS><ID,3><MUL><NUM,60>

Symbol table	
1	Position
2	Initial
3	Rate

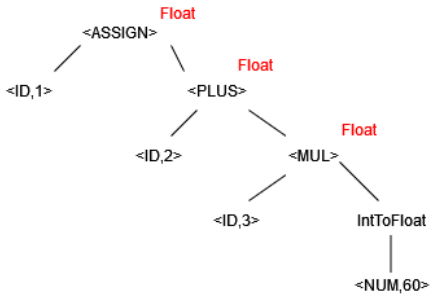
Analizzatore sintattico

Utilizza i token prodotti dalla fase precedente per produrre un syntax tree, dove ogni nodo rappresenta gli argomenti dell'operazione:



Analizzatore semantico

Utilizza il syntax tree e la tabella dei simboli per verificare la consistenza del programma sorgente rispetto alla definizione del linguaggio. Inoltre effettua anche il controllo dei tipi (type checking).



Symbol table		
1	Position	float
2	Initial	float
3	Rate	float

Generatore codice intermedio

Visitando il syntax tree e utilizzando la tabella dei simboli si inizia a generare un codice intermedio di basso livello secondo la rappresentazione nota come *codice a tre indirizzi*:

$$t1 = inttofloat(60)$$
$$t2 = id3 * t1$$
$$t3 = id2 + t2$$
$$id1 = t3$$

1.2.2 Fasi del back-end

Ottimizzatore del codice

Tenta di migliorare il codice intermedio prodotto dalla fase precedente cercando di aumentare le prestazioni:

$$t1 = id3 * 60.0$$
$$id1 = id2 + t1$$

Generatore codice macchina

Prende in input il codice intermedio ottimizzato e lo traduce in codice macchina, in questa fase vengono assegnati i registri per le variabili:
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
Nota: le fasi di un compilatore sono fasi logiche, infatti ognuno potrebbe implementarle nell'ordine che vuole, per esempio invertendo le fasi dell'analisi sintattica e semantica.
Nota: Quando le variabili nel codice Assembly sono in chiaro ed al più 3 si tratta di un'architettura a RAM, con un'architettura a Stack vengono omesse.
Es:
LDF
MULF
ADDF

1.3 Tipi di linguaggi

- Imperativi: sono linguaggi di programmazione in cui i comandi, le istruzioni, sono espressi come una successione di attività sequenziali, articolano quindi i comandi sulla base di una successione temporale. (Es: C)
- Dichiarativi: vengono utilizzati spesso per definire regole di creazione e interrogare delle basi di dati. (Es: SQL)
- Ad oggetti
- Funzionali: Paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche (Es: Lisp)
- Logico: Paradigma di programmazione che differisca dalla programmazione tradizionale in quanto richiede e nello stesso tempo consente al programmatore di descrivere solo la struttura logica del problema anziché il modo di risolverlo (Es: Prolog).

1.4 Alcune domande d'esame

- **Quando si crea un compilatore di cosa si ha bisogno?**
  Per creare un compilatore bisogna sapere le specifiche del linguaggio di programmazione ed in quale linguaggio deve essere tradotto. Alla creazione di nuove architetture sta a chi ha creato l'architettura creare il back- end per un linguaggio intermedio, così tutti quelli che sviluppano un nuovo compilatore devono preoccuparsi di fare solo il front-end e non devono creare un nuovo compilatore per ogni architettura che viene creata.

- Transpiler: Prende in input un codice sorgente di un programma scritto in un linguaggio di programmazione e produce in output un codice sorgente equivalente di un linguaggio di programmazione diverso.

1.5 Esercizi

1)Creare l'analisi lessicale, sintattica e semantica del seguende codice:

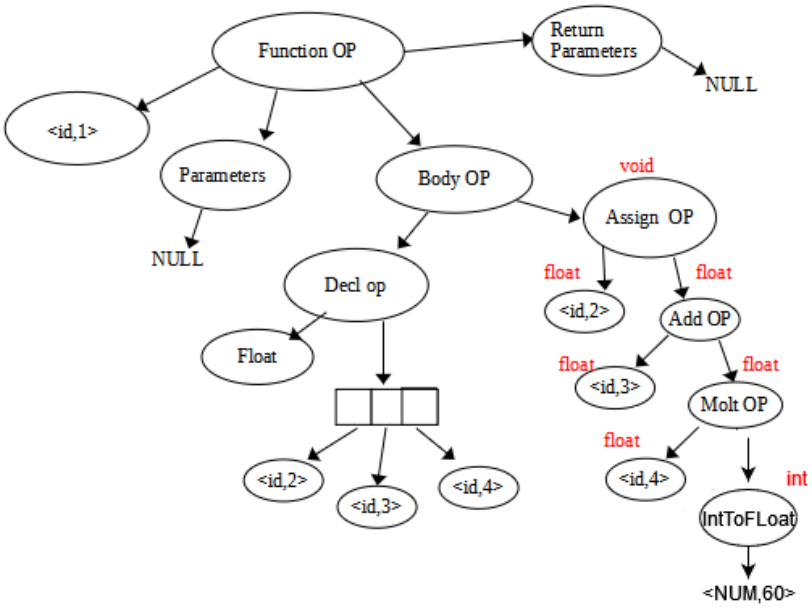
```
main(){
    float position,initial,rate;
    position = initial + rate * 60;
}
```

Analisi lessicale:

```
<ID,1><LPAR><RPAR><LBRAK>
    <FLOAT><ID,2><COMMA><ID,3><COMMA><ID,4><SEMI>
    <ID,2><ASSIGN><ID,3><PLUS><ID,4><MUL><NUM,60><SEMI>
<RBRAK>
```

Analisi sintattica e semantica:

Symbol table		
1	main	void → void
2	Position	float
3	Initial	float
4	Rate	float



2)Analisi lessicale, sintattica e semantica del seguente codice:

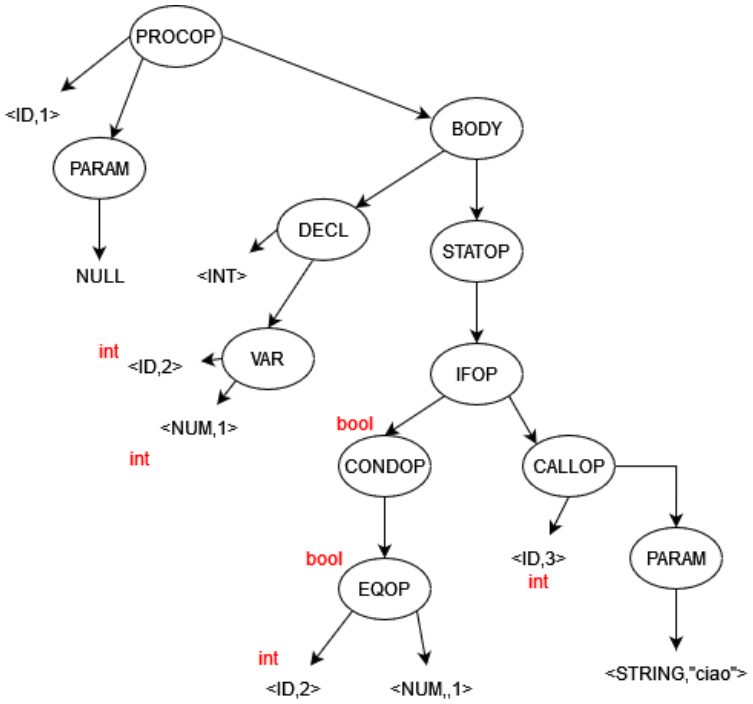
```
main(){
    int a = 1;
    if (a==1) printf("ciao");
}
```

Analisi lessicale:

```
<ID,1><LPAR><RPAR><LBRAK>
    <INT><ID,2><ASSIGN><NUM,1><SEMI>
    <IF><LPAR><ID,2><EQ><NUM,1><RPAR><ID,3><LPAR><STRING,"ciao"><RPAR><SEMI>
<RBRAK>
```

Analisi sintattica e semantica:

Symbol table		
1	main	void → void
2	a	int
3	printf	



1.6 Binding e tempo di binding

Durante il ciclo di vita di un linguaggio di programmazione (dalla definizione del linguaggio all'esecuzione dei suoi programmi) ciascun elemento del linguaggio è **legato** (binding) a delle caratteristiche in un preciso **momento** (binding time) da parte di un certo attore. Esistono diversi momenti di legame:

- Definizione del linguaggio: quanti tipi avrà, come si assegnano i valori, che simbolo avrà un determinato token. **Attore: Creatore del linguaggio**
- Implementazione del linguaggio (ovvero, scrittura del compilatore). Attore: Sviluppatore del compilatore
- Scrittura di un programma: legame di una variabile al tipo. **Attore: Programmatore**
- Compilazione/Traduzione di un programma (compile time): alloca la memoria necessaria per le variabili in base al loro tipo, assegna anche gli indirizzi relativi alle variabili. **Attore: Compilatore/Traduttore**

- Linking/Loader: legami delle variabili al loro indirizzo assoluto. **Attore: S.O.**
- Esecuzione del programma eseguibile (runtime): legame fra variabili e valore. Infatti solo in questa fase posso vedere effettivamente il valore delle variabili. **Attore: Processore**

1.6.1 Alcuni esempi

RIPETIZIONE!
Il binding(legame) è l'atto di associare le proprietà ai nomi.
Il binding times (momento di legame) è il momento nel ciclo di vita del programma in cui si verifica l'associazione. Citando diversi esempi:

1. la parola chiave if è **legata** (bound) al concetto di condizione a **tempo** di definizione del linguaggio.
2. La **scelta** di dove memorizzare i parametri di una funzione è fatta a **tempo** di implementazione di un linguaggio
3. Il **legame** fra una variabile ed il suo tipo è definito a tempo di scrittura del programma (in linguaggi tipati senza inferenze di tipo) e realizzato (bound) a **tempo** di traduzione (analisi semantica)
4. Il **legame** fra una variabile ed il suo indirizzo relativo è definito e realizzato (bound) a **tempo** di traduzione
5. Il **legame** fra una variabile ed il suo indirizzo assoluto è realizzato (bound) a **tempo** di caricamento del programma in memoria
6. Il **legame** fra una variabile ed il suo valore è realizzato (bound) a **tempo** di esecuzione

Esercizio:

Data l'istruzione X = X + 10 individuare quando queste scelte e legami vengono definiti e da chi.

1. Scelta dei possibili tipi primitivi di X: - a tempo di **definizione** - attore **creatore**
2. Scelta del particolare tipo di X: - a tempo di **implementazione** - attore **programmatore**
3. Realizzazione di legame (binding) fra un particolare tipo ed X (in linguaggi compilati): - a tempo di **compilazione** - attore **analizzatore semantico**
4. Realizzazione di legame (binding) fra un particolare tipo ed X (in linguaggi interpretati): - a tempo di **esecuzione** - attore **processore**.
5. Scelta di quali valori possa assumere X se intero: - a tempo di **implementazione** - attore **programmatore**
6. Realizzazione di legame (binding) fra X ed il suo valore: - a tempo di **esecuzione** - attore **processore**
7. Realizzazione di legame (binding) fra 10 e la sua rappresentazione (decimale, ottale, etc.): - a tempo di **definizione del linguaggio** - attore **creatore**
8. Scelta di quanti bits utilizzare per rappresentare 10: - a tempo di **implementazione** - attore **programmatore**
9. Realizzazione di legame (binding) fra l'operatore + ed il/i suo/i significato/i: - a tempo di **definizione** - attore **creatore**
10. Realizzazione di legame (binding) fra l'operatore + ed il codice che lo implementa: - a tempo di **compilatore** - attore **generatore codice macchina**

1.6.2 Tipi di Binding

In generale col binding si analizza e si confrontano i diversi linguaggi di programmazione. Siccome sappiamo che esistono due grandi famiglie di linguaggi, compilati ed interpretati, essi differiscono principalmente nel binding ed è grazie a questo aspetto che possiamo differenziarli.

- **Early binding** (a tempo di compilazione): nei linguaggi compilati la maggior parte dei bindings avviene a tempo di compilazione, rendendo **più efficiente l'esecuzione**, ma rendendo la programmazione **poco flessibile** in quanto ogni variabile ha il proprio tipo e non può essere utilizzata con tipi diversi.
- **Late binding** (a tempo di esecuzione): nei linguaggi interpretati la maggior parte dei bindings avviene a tempo di esecuzione, rendendo il codice **meno efficiente** ma più **flessibile per il programmatore** in quanto potrebbe permettere di utilizzare una variabile e associargli più tipi.

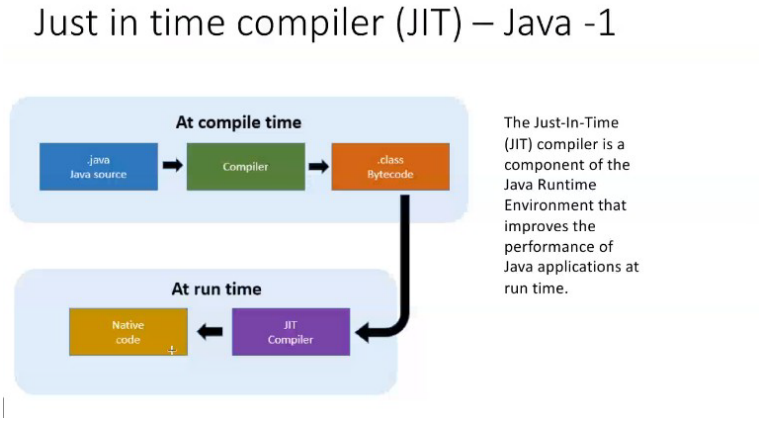
Esempio di binding:

- # define g 100: legame di g al valore 100, il legame è statico;
- int h: avviene un legame tra la variabile e la dimensione del tipo, il legame è statico;
- i = f(10): il legame tra i e la locazione della memoria della funzione è dinamico.

NOTA: Come facciamo a sapere che i tre quesiti precedenti sono trattati in un ambito di compilatore e non di interprete? FACILE! Il corso si chiama COMPILATORI DIO PORCO!

1.7 Compilatore Just-In-Time

Il compilatore Just-In-Time (JIT) è un componente di Java Runtime Environment che migliora le prestazioni delle applicazioni Java in fase di runtime. I programmi Java sono costituiti da classi, che contengono bytecode neutrale dalla piattaforma che può essere interpretato da una JVM su molte architetture di computer differenti. In fase di esecuzione, la JVM carica i file di classe, determina la semantica di ogni singolo bytecode ed esegue il calcolo appropriato. Il processore aggiuntivo e l'utilizzo della memoria durante l'interpretazione indicano che un'applicazione Java viene eseguita più lentamente di un'applicazione nativa. Il compilatore JIT aiuta a migliorare le prestazioni dei programmi Java compilando bytecode in codice macchina nativo in fase di runtime. Quando un metodo è stato compilato, la JVM chiama direttamente il codice compilato di quel metodo invece di interpretarlo. La compilazione JIT è molto onerosa, quando la JVM viene avviata per la prima volta, vengono chiamati migliaia di metodi.



Capitolo 2

Analisi lessicale

Per implementare un analizzatore lessicale a mano conviene iniziare da un'opportuna rappresentazione dei lessemi di ogni token, procedere poi con la scrittura del codice che identifica ogni occorrenza di ciascun lessema nella sequenza di caratteri d'ingresso ed infine restituire le informazioni sui token identificati. Un'alternativa consiste nel generare un analizzatore lessicale in modo automatico a partire da una descrizione della struttura dei lessemi tramite l'uso di espressioni regolari.
Nota: il lessema è una stringa che rispetta un pattern, dove ogni pattern è una sequenza di caratteri che viene successivamente legato ad una categoria chiamata Token.

2.1 Analizzatore lessicale

L'analizzatore lessicale è un componente del compilatore che leggendo una sequenza di caratteri che forma il programma sorgente, li raggruppa in lessemi e ne genera successivamente il token associato ad ogni lessema.

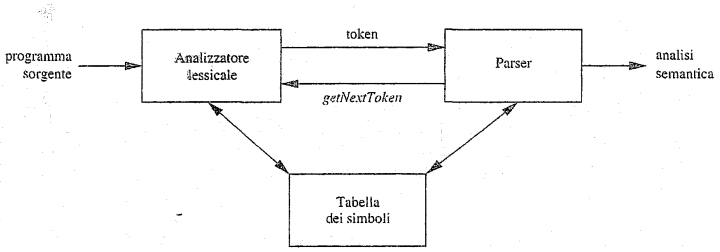


Figura 3.1 Interazioni tra l'analizzatore lessicale e il parser.

Successivamente la sequenza di token sarà inviata al parser che effettuerà l'analisi sintattica. Inoltre, al parser sarà inviata anche la tabella dei simboli che verrà creata ad ogni lettura di identificatori. Parser e analizzatore lessicale comunicano tramite il metodo "getNextToken".

2.1.1 Analisi lessicale e parsing

Ci sono una serie di ragioni per cui la parte di analisi di un compilatore è normalmente separata in fasi di analisi lessicale e di parsing (analisi della sintassi).

- 1. **Semplicità di progettazione e sviluppo**: questa separazione permette di semplificare almeno una della due fasi. Infatti, un compilatore che nella fase di analisi sintattica dovrebbe analizzare anche gli spazi o i commenti sarebbe molto più complicato di un compilatore che li trova già rimossi.

- **L'efficienza del compilatore è migliorata**: Uno strumento progettato soltanto per effettuare la fase di analisi, ci consente di applicare opportuni algoritmi di ottimizzazione e ulteriori tecniche di buffering per migliorare la lettura dell'input senza interessarci dei problemi di del parsing.
- La portabilità del compilatore è migliorata

2.1.2 Token, patter e lessemi

- Token: è una coppia nome-valore. Il nome del token rappresenta una specifica unità lessicale come una parola chiave o una sequenza di caratteri. Il parser riceve in input i nomi dei token e li elabora.
- Pattern: è una descrizione compatta della forma che il lessema di un token può assumere.
- Lessema: sequenza di caratteri del programma sorgente che corrisponde al pattern di un token, viene identificato dall'analizzatore lessicale.

Token	Descrizione informale	Lessemi d'esempio
if	caratteri `i`, `f`	`if`
else	caratteri `e`, `l`, `s`, `e`	`else`
comparison	`<`, `>`, `<=`, `>=`, `==`, `!=`	`<=`, `!=`
id	lettera seguita da lettere e cifre	`pi`, `score`, `D2`
number	costante numerica qualsiasi	`3.14159`, `0`, `6.02e23`
literal	tutto tranne ", racchiuso tra "	`"core dumped"`

Figura 3.2 Esempi di token.

Per la maggior parte dei linguaggi di programmazione i token possono essere suddivisi nelle cinque seguenti classi.

1. Token per le parole chiave: Per ogni parola chiave è richiesto un token distinto. Il pattern per Una parola chiave coincide con la parola chiave stessa.
2. Token per gli operatori: per rappresentare un singolo operatore oppure una classe di operatori, per esempio gli operatori di confronto.
3. Un token per rappresentare tutti gli identificatori.
4. Uno o piu token per rappresentare le costanti. Numeri interi e stringhe di caratteri sono esempi di costanti.
5. Token per i segni d'interruzione: tra questi rientrano, per esempio, i vari tipi di parentesi, la virgola, il punto e virgola, ecc.

Ad ogni token è associato soltanto un attributo ed ha la seguente struttura: <Attributo, Valore>, eventualmente quest'ultimo può essere strutturato in modo da contenere più informazioni. Alcuni attributi più utilizzati sono ID associati agli identificatori e NUM associato ai valori numerici.

2.2 Espressioni regolari

2.2.1 Stringhe e linguaggi

Si definisce **alfabeto** qualsiasi insieme finito di stringhe es: $\{a, b\}$.

Una **stringa** è una sequenza di simboli presi da un alfabeto es: aababa. La stringa vuota o nulla si indica con ϵ. Di seguito alcuni termini legati alle stringhe usando la stringa *asfnazza*:

1. **Prefisso**: è una qualsiasi stringa ottenuta rimuovendo zero o più caratteri dalla fine di una stringa es: asf;

2. **Suffisso**: è una qualsiasi stringa ottenuta rimuovendo zero o più caratteri dall'inizio di una stringa es: nazza;

3. **Sottostringa**: è una qualsiasi stringa ottenuta rimuovendo un qualsiasi suffisso o prefisso da una stringa es: fna;

4. Un suffisso, prefisso, o sottostringa si dicono propri se sono diversi da ϵ e dalla stringa stessa;

5. **Sotto-sequenza**: è una qualsiasi stringa ottenuta rimuovendo zero o più simboli non necessariamente consecutivi es: aszza.

Un linguaggio è un qualsiasi insieme numerabili di stringhe di un dato alfabeto, questa è una definizione molto ampia es: $\{\epsilon\}$ è un linguaggio.

2.2.2 Definizioni regolari

Useremo le espressioni regolari per descrivere i pattern dei nostri lessemi in modo da associare molto facilmente i token.
Importante è dare dei nomi ad alcune espressioni così da semplificarne la creazione, soprattutto con espressioni più complesse. Alcune espressioni regolari:

letter = [A-Za-z];

digit = [0-9]

id = letter_ (letter_ | digit)*

digits = digit$^+$

optimalFraction = . digits | ϵ

optimalExponent = \(E (+ | - | ϵ) digits \) | ϵ

number = digit optimalFraction optimalExponent

Nota significato dei simboli più importanti:

* : 0 o più volte, + : 1 o più volte, ? : 0 o una volta

∧ : negazione

. : qualsiasi carattere escluso "\n"

[\s\S] : qualsiasi carattere incluso "\n"

[\w\W] : qualsiasi carattere alfanumerico

*? : blocca l'espressione regolare al primo match

le espressioni regolari iniziano con ∧ e terminano con $

A-z : continente anche i seguenti caratteri speciali: [,],\,/,∧,_

Esercizi espressioni regolari

1) Scrivere un espressione regolare che accetti tutti i numeri interi, ma non devono iniziare per 0:
zero = 0
unoToNove = [1-9]
zeroToNove = [0-9]
number = zero | (unoToNove)(zeroToNove)* oppure 0 | [1-9][0-9]*

2) Scrivere l'espressione regolare che accetti tutte le stringhe con le vocali nel seguente ordine ... a ... e ... i ... o ...u:
NotV = [∧ aeiouAEIOU]*
expr = **NotV** a **NotV** e **NotV** i **NotV** o **NotV** u

3) Fare in modo che l'espressione precedente accetti più vocali nel loro ordine alfabetico senza ripetizione dopo il loro ordine es: caaaaseeeeeeoddodooucuu
expr = (**NotV** | a)$^+$ a (**NotV** | e)$^+$ e (**NotV** | i)$^+$ i (**NotV** | o)$^+$ o (**NotV** | u)$^+$ u

2.3 Riconoscimento dei token

Per far riconoscere ai pattern quali token assegnare, trasformeremo prima i pattern in Token per gli operatori e poi li implementeremo nel linguaggio di programmazione che desideriamo.
I **diagrammi di transizione** sono costituiti da nodi chiamati **stati**. Gli stati rappresentano una delle possibili condizioni che possono verificarsi durante l'analisi dei lessemi. I nostri diagrammi di transizioni avranno le seguente caratteristiche:

- Il diagramma ha uno stato iniziale indicato con una freccia entrante o con start.
- Stati **finali** o **d'accettazione**: indicano che il lessema è stato riconosciuto, viene indicato con il doppio cerchio.
- Se si ritiene necessario arretrare di una posizione il puntatore forward, annotiamo lo stato finale con un *. **Note:** Questa necessità si presenta quando il simbolo che ha causato la conclusione della transizione (stato finale) non fa parte del lessema riconosciuto.

Esempio di digrammi di stato

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

Figure 3.11: Patterns for tokens of Example 3.8

Mostriamo il diagramma di transizione per *relop*:

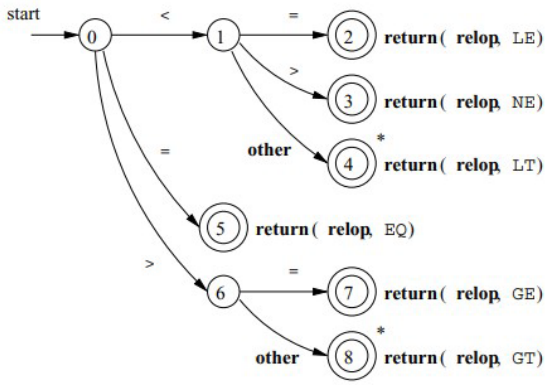


Figure 3.13: Transition diagram for **relop**

Iniziamo dallo stato 0 (stato iniziale). Se vediamo $<$ come primo simbolo di input, allora tra i lessemi che corrispondono al pattern per relop possiamo solo guardare $<$, $<>$ o $<=$. Andiamo quindi allo stato 1 e guardiamo il carattere successivo. Se è $=$, allora riconosciamo il lessema $<=$, entriamo nello stato 2 e restituiamo il token relop con l'attributo LE. Se nello stato 1 il carattere successivo è $>$, allora abbiamo il lessema $<>$, e andiamo nello stato 3 per restituiamo NE. Se nello stato 1 leggiamo un qualsiasi altro carattere entriamo nello stato 4 per restituire LT, notiamo che lo stato 4 ha anche * per indicare che dobbiamo tornare indietro di un carattere.

2.3.1 Riconoscimento identificatori chiave

Il riconoscimento di parole chiavi rappresenta un problema. Di solito, parole if o then sono riservate e quindi non sono identificatori. Quindi un diagramma di transizione che riconosce gli identificatori riconoscerà anche le parole chiavi.

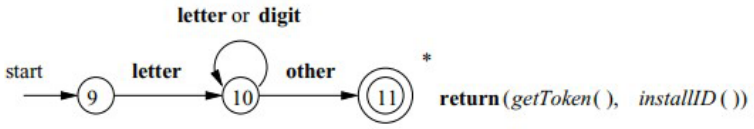


Figure 3.14: A transition diagram for **id**'s and keywords

Abbiamo due modi per gestire questo problema:

1. Nella tabella dei simboli risultano inizialmente inserite tutte le parole chiave del nostro linguaggio. Ogni volta che il lexer riconosce un identificatore, prima di inserire il token nella tabella simboli, controlla prima se quest'ultimo risulta all'interno di essa come identificatore, se è già presente restituisce il suo attributo. Altrimenti inserisce l'identificatore nella tabella e restituisce come attributo l'indice della posizione nella tabella dei simboli.
   Es. con il lessema "if" restituisce IF, mentre con "munnezza" prima lo inserisce e poi restituisce <id,20> dove 20 è l'indice di dove trovarlo nella tabella.

2. Creare per ogni parola chiave un diagramma di transizione.
   Attenzione Se si adotta questo approccio dobbiamo fare in modo che i diagrammi per le parole chiave siano implementati prima di quelli degli identificatori.

Esempio implementazione di un diagramma di transizione

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

2.4 Lavorare con JFlex

```
/* JFlex example: part of Java language lexer specification */
import java_cup.runtime.*;

/**
* This class is a simple example lexer.
*/

%%
%class Lexer
%unicode
%cup
%line
%column

%{
    StringBuffer string = new StringBuffer();

    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }

    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}

LineTerminator =    \r|\n|\r\n
InputCharacter =    [^\r\n]
WhiteSpace     =    {LineTerminator} | [ \t\f]


/* comments */
Comment             = {TraditionalComment} | {EndOfLineComment} | {
    DocumentationComment}
TraditionalComment  = "/*" [^*] ~"*/" | "/*" "*"+ "/"

// Comment can be the last line of the file, without line terminator.
EndOfLineComment     = "//" {InputCharacter}* {LineTerminator}?
DocumentationComment = "/**" {CommentContent} "*"+ "/"
CommentContent       = ( [^*] | \*+ [^/*] )*


Identifier = [:jletter:] [:jletterdigit:]*
DecIntegerLiteral = 0 | [1-9][0-9]*


%state STRING
%%
/* keywords */
<YYINITIAL> "abstract"      { return symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean"       { return symbol(sym.BOOLEAN); }
<YYINITIAL> "break"         { return symbol(sym.BREAK); }

<YYINITIAL> {

    /* identifiers */
    {Identifier}                { return symbol(sym.IDENTIFIER); }

    /* literals */
    {DecIntegerLiteral}         { return symbol(sym.INTEGER_LITERAL); }
    \"                          { string.setLength(0); yybegin(STRING); }
```

```
    /* operators */
    "="                         { return symbol(sym.EQ); }
    "=="                        { return symbol(sym.EQEQ); }
    "+"                         { return symbol(sym.PLUS); }

    /* comments */
    {Comment}                   { /* ignore */ }

    /* whitespace */
    {WhiteSpace}                { /* ignore */ }
}


<STRING> {
    \"                          { yybegin(YYINITIAL);
                                    return symbol(sym.STRING_LITERAL, string.toString
                                        ());
                                }
    [^\n\r\"\\]+                { string.append( yytext() ); }
    \\t                         { string.append('\t'); }
    \\n                         { string.append('\n'); }
    \\r                         { string.append('\r'); }
    \\\"                        { string.append('\"'); }
    \\                          { string.append('\\'); }
}

/* error fallback */
[^]                             {throw new Error("Illegal character <"+yytext()+">");}
```

Da questa specifica JFlex genera un file *.java* con una classe che contiene il codice per lo scanner. La classe avrà un costruttore che prende un *java.io.Reader* da cui viene letto l'input. La classe avrà anche una funzione **yylex()** che esegue lo scanner e che può essere utilizzata per ottenere il token successivo dall'input (in questo esempio la funzione ha effettivamente il nome next token() perché la specifica utilizza l'opzione %cup). La specifica è composta da tre parti, divise per %%:

- usercode (codice utente);
- opzioni e dichiarazioni;
- regole lessicali.

Un file flex è quindi diviso come segue:
Usercode
%% opzioni e dichiarazioni %% regole lessicali

2.4.1 Sezione usercode

Nella prima sezione, "codice utente", il testo fino alla prima riga che inizia con %% viene copiato testualmente all'inizio della classe generata dal lexer (prima della dichiarazione di classe effettiva).

2.4.2 Opzioni e dichiarazioni

La seconda sezione "Opzioni e dichiarazioni" consiste in un insieme di opzioni, codice incluso all'interno della classe scanner generata, stati lessicali e dichiarazioni di macro. Ciascuna opzione JFlex deve sia iniziare una riga della specifica sia iniziare con un %.
Nel nostro esempio vengono utilizzate le seguenti opzioni:

1. **%class**: il Lexer dice a JFlex di dare alla classe generata il nome "Lexer" e di scrivere il codice in un file "Lexer.java";

2. **%unicode**: definisce l'insieme di caratteri su cui lo scanner lavorerà. *Per la scansione di file di testo, dovrebbe essere sempre utilizzato %unicode.* È possibile specificare la versione Unicode, ad es. %unicode. In JFlex 1.6.1, in automatico è Unicode 7.0;
3. **%cup**: da inserire per interfacciarsi con un parser generato da CUP, genera il metodo next_token() e non un main;
4. **%cupsym "classname"**: usato per dare un nome personalizzato alla classe generata da CUP;
5. **%line**: attiva il conteggio delle righe (è possibile accedere al numero di riga corrente tramite la variabile *yyline*);
6. **%column**: attiva il conteggio della colonna (si accede alla colonna corrente tramite colonna *yycolumn*);
7. **%standalone**: se presente genererà un main all'interno della classe Lexer che permetterà di eseguirla passando un file come argomento a linea di comando.

Le macro sono abbreviazioni di espressioni regolare, usate per rendere le specifiche lessicali più facili da leggere e da capire. Una dichiarazione di macro consiste in un identificatore di macro seguito da = seguito da un'espressione regolare che la rappresenta.

2.4.3 Macro per il metodo di scansione

- %Integer o %int cambia il tipo di ritorno del token in int;
- %type "typename" cambia il tipo di ritorno del token nel tipo che specifichiamo noi;

2.4.4 Macro per la fine del file

- ```
  \%eofval{
  ...
  \%eofval}
  ```

  Il codice all'interno di questa macro viene eseguito ogni volta che si raggiunge la fine del file. Può succedere che il metodo di scansione ritorni più volte a leggere la fine del file se richiesto

- ```
  \%eof{
  ...
  \%eof}
  ```

  Il codice all'interno di questa macro viene eseguito quando si raggiunge la fine del file.

2.4.5 Macro per gli stati

- %state STRING dichiara uno stato nell'analizzatore lessicale chiamato STRING. Una dichiarazione di stato è una riga che inizia con %state e seguita dal nome dello stato. Uno stato agisce come una condizione iniziale. Se lo scanner è nello stato lessicale STRING, possono essere abbinate solo le espressioni precedute dalla condizione iniziale <STRING>. Di default è definito lo stato YYINITIAL ed è anche lo stato in cui il lexer inizia la scansione. Se un'espressione regolare non ha condizioni in uno stato, si passa in automatico a controllare gli altri stati.

2.4.6 Struttura delle espressioni regolari in jflex

- a : avviene il match con il carattere a;
- casa : avviene il match con i caratteri casa;
- $[a-z]$: avviene il match con la sequenza di caratteri dalla a alla z;
- Unione (||): [[a-c] ||[d-f]], equivalente a [a-cd-f];
- Intersezione (&&): [[a-f]&&[f-m]], equivalente a [f].
- Insieme differenza(–): [[a-z]–m], equivalente a [a-ln-z];
- Negazione di un carattere ($[^{-}a]$): match con tutti i caratteri tranne a;
- '"' StringCharacter "'": matcha il testo presente nei doppi apici;
- { identifier }: Matcha con l'espressione regolare contenuta in una variabile precedentemente definita;
- [:jletter] e [:jletterdigit]: usa i metodi di diefault di java per controllare la correttezza di una sequenza di caratteri;
- *: prende un carattere 0 o più volte;
- +: prende un carattere 1 o più volte;
- ?: prende un carattere 0 o 1 volta;
- a$\{n\}$: matcha il carattere esattamente n volte;
- a$\{n, m\}$: matcha il carattere da n a m occorrenze;

2.4.7 Altre variabili del lexer

- String yytext(): ritorna l'input presente nel case;
- yylength(): ritorna la lunghezza dell'input matchato;
- yystate(): ritorna lo stato in cui si trova l'analizzatore lessicale.

2.4.8 Priorità match Espressioni regolari in Jflex

In Jflex possiamo avere il match di un pattern con più espressioni regolari, Jflex risolve automaticamente questo problema comportandosi in due modi:

- Match la stringa con il pattern più lungo che ha trovato;
- In caso che ci siano due pattern con la stessa lunghezza vince chi si trova prima.

Capitolo 3

Analisi sintattica

Le grammatiche forniscono una specifica della sintassi di un linguaggio precisa e di facile comprensione, permettono di generare un parser in modo automatico.

3.1 Parser

Il parser secondo il nostro modello riceve una sequenza di token dall'analizzatore lessicale e controlla se questa sequenza può essere generata dalla grammatica del linguaggio che vogliamo implementare. In questa fase viene costruito un albero di derivazione che successivamente si trasformerà in albero sintattico.
Abbiamo tre tipo di parser per le nostre grammatiche:

- Universali: possono trattare qualsiasi tipo di grammatica;
- Top-down: costruiscono l'albero partendo dalla radice;
- Bottom-up: costruiscono l'albero partendo dalle foglie.

I metodi top-down e bottom-up sono i più utilizzati e si applicano alle grammatiche di tipo LL (usate dai perser implementati a mano) e LR (usate dai parser automatici).

3.2 Grammatiche Context-free

Per parlare di grammatiche ci baseremo sul seguente esempio:

$$
\begin{aligned}
&1) E \to E + T \\
&2) E \to T \\
&3) T \to T * F \\
&4) T \to F \\
&5) F \to (E) \\
&6) F \to id
\end{aligned}
$$

Una grammatica Context-Free consiste in terminali e non terminali dove:

- **Terminali**: Sono i simboli di base di cui sono costituite le stringhe (nome dei token), nella nostra grammatica sono: $\{+, * + (,)\}$;
- **Non terminali**: sono variabili sintattiche che denotano un insieme di stringhe, nel nostro esempio: $\{E, T, F\}$;

- **Simbolo iniziale**: è il simbolo iniziale da cui partire per analizzare una grammatica, è un non terminale e tipicamente si indica con S: nel nostro esempio non è indicato ma se vorremmo inserirlo possiamo fare come segue: S $\rightarrow E$.
- **Produzioni**: le produzioni di una grammatica specificano come i terminali e i non-terminali possono essere combinati per formare stringhe. Ogni produzione consiste in un non terminale, una freccia e un corpo composto da terminali e non terminali. Es:
  Mostriamo la produzione di id + id:
  $\mathrm{E} \overset{1}{\Rightarrow} \underline{E} + T \overset{2}{\Rightarrow} T + \underline{T} \overset{4}{\Rightarrow} \underline{T} + F \overset{4}{\Rightarrow} \underline{F} + F \overset{6}{\Rightarrow} id + \underline{F} \overset{6}{\Rightarrow} \mathrm{id} + \mathrm{id}$

3.2.1 Derivazioni

La costruzione di un albero di parsing può essere formalizzata mediante l'uso di derivazioni, basate sulla sostituzione dei non terminali come nell'esempio mostrato sopra.
Le derivazioni possono essere di due tipi:

- Leftmost, viene sempre scelto il non terminale più a sinistra della sequenza.
- Rightmost, viene sempre scelto il non terminale più a destra.

Una frase appartiene ad una grammatica se partendo da una derivazione riesco ad ottenere la frase stessa quindi possiamo definire il linguaggio formalmente come segue: $\mathrm{L(G)} = \{w | S \overset{*}{\Rightarrow} w\}$

3.2.2 Albero di parsing e derivazioni

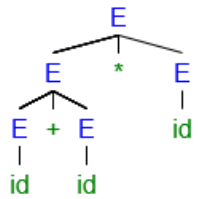
Un albero di derivazione (o di parsing) è una rappresentazione grafica di una derivazione che non dipende dall'ordine in cui le produzioni sono utilizzate per rimpiazzare i non-terminali. Ogni nodo interno di un albero di parsing rappresenta l'applicazione di una produzione.

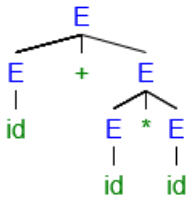
Nota: L'albero di derivazione è una struttura logica, non è una struttura dati, a differenza dell'albero sintattico. Inoltre, l'albero di derivazione ha i terminali e dipende strettamente dalla grammatica, quello sintattico no.

Nella figura a destra è mostrato l'albero sintattico per **"id + id"**.
La grammatica precedentemente mostrata può essere semplificata nel seguente modo:

$$1) E \rightarrow E + E$$
$$2) E \rightarrow E * E$$
$$2) E \rightarrow (E)$$
$$4) E \rightarrow id$$

Il problema di questa grammatica è il seguente: per l'espressione id + id * id produce due diversi alberi sintattici che operano in modo diverso. **ambigua**:





Nel primo albero verrà data priorità alla somma, mentre nel secondo verrà data priorità alla moltiplicazione. Questo è un caso di **grammatica ambigua**, ovvero una grammatica che produce almeno 2 alberi di parsing diversi.

3.2.3 Eliminare ambiguità

Si consideri la seguente grammatica che presenta il tipico problema di ambiguità noto come "dangling-else":

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\ & |\ \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \end{aligned}$$

Questa grammatica è ambigua perché:
per **if** E_1 **then if** E_2 **then** S_1 **else** S_2 è possibile costruire due diversi alberi (provare).
Per risolvere questo problema si può usare la **fattorizzazione sinistra** che permette di ottenere grammatiche utili per i parsing predittivi top-down.

Fattorizzazione sinistra

Quando la scelta tra due produzioni alternative per un non-terminale A non è chiara, possiamo riscriverle in modo da differire tale scelta finché non avremo letto abbastanza simboli d'ingresso da poter prendere la decisione corretta.
Riprendendo la grammatica precedente, nel momento in cui leggiamo il token **if** non siamo in grado di decidere immediatamente quali delle due produzioni prendere.

In generale: Se A $\rightarrow\ \alpha\beta_1\ |\ \alpha\beta_2$ sono due produzioni per A e la stringa d'ingresso inizia con una stringa non vuota derivata da α possiamo riscriverla come:
A $\rightarrow\ \alpha A'$
A' $\rightarrow\ \beta_1\ |\ \beta_2$
Se β_2 manca si aggiunge ϵ (indica stringa vuota), otteniamo quindi: A $\rightarrow\ \alpha A'$
A' $\rightarrow\ \beta_1\ |\ \epsilon$

La nostra grammatica quindi senza ambiguità diventerà:

$$\begin{aligned} stmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \ stmt' \\ stmt' &\rightarrow\ \ \textbf{else}\ stmt|\ \epsilon \end{aligned}$$

3.2.4 Grammatica ricorsiva sinistra

Una grammatica è detta ricorsiva (non accettata da parser top down) a sinistra se ha un non-terminale A per cui esiste una derivazione: A $\stackrel{*}{\Rightarrow} A\alpha$.
Esempio:
A $\rightarrow A\alpha$
A $\rightarrow B$
è ricorsiva sinistra.
Applicando l'algoritmo per la sostituzione della ricorsione a sinistra diventa come segue:
A $\rightarrow \beta A'$
A' $\rightarrow \alpha A'|\ \epsilon$

L'algoritmo per eliminare una ricorsione sinistra immediata è il seguente:

- Si inizia raggruppando tutte le produzioni di A come segue:
  A $\rightarrow A\alpha_1\ \ |A\alpha_2\ \ |...|A\alpha_n\ \ |\beta_1\ \ \beta_2\ \ |...|\beta_n$
  In cui nessuno dei β_i inizia con A. Si sostituiscono le produzioni di A come segue:

- A $\rightarrow \beta_1 A'\ \ |\beta_2 A'\ \ |...|\beta_n A'\ \ |$
  A' $\rightarrow \alpha_1 A'\ \ |\alpha_2 A'\ \ |...|\alpha_n A'\ \ |\epsilon$

Il non-terminale A genera le stesse stringhe di prima, ma non presenta più ricorsione a sinistra.
Questo metodo elimina la ricorsione sinistra da tutte le produzioni per A e A' (a patto che nessuno degli α coincida con ϵ), ma non è grado di eliminarla nel caso di derivazioni che richiedono due o più passi:
S $\rightarrow Aa \mid b$
A $\rightarrow Ac \mid Sd \mid \epsilon$
Il non terminale S presenta una ricorsione sinistra che non è possibile risolvere con l'algoritmo precedente. Un modo per eliminarla è sostituire per prima cosa la S in A come segue:

1. S $\rightarrow Aa \mid b$
   A $\rightarrow Ac \mid Aad \mid bd \mid \epsilon$
   Nel nostro caso si è verificata ambiguità quindi procediamo ad eliminarla
2. S $\rightarrow Aa \mid b$
   A $\rightarrow bdA' \mid A'$
   A' $\rightarrow cA' \mid adA' \mid \epsilon$

L'algoritmo per cui avviene questa sostituzione è il seguente:

```
1)  ordina arbitrariamente i non-terminali come A_1, A_2, ..., A_n.
2)  for ( ogni i da 1 fino a n ) {
3)      for ( ogni j da 1 fino a i − 1 ) {
4)          sostituisci ogni produzione nella forma A_i → A_j γ
                con le produzioni A_i → δ_1 γ | δ_2 γ | ··· | δ_k γ,
                in cui A_j → δ_1 | δ_2 | ··· | δ_k sono tutte le
                produzioni per il non-terminale A_j in esame
5)      }
6)      elimina la ricorsione sinistra immediata dalle produzioni per A_i
7)  }
```

3.3 Parser top-down

Il parsing top down di una sentenza corrisponde a generare una derivazione leftmost della sentenza (o, equivalentemente, un albero di derivazione dal nodo radice alle foglie in depth-first da sinistra verso destra).
Abbiamo due tipi di Top down parsing:

1. **Recursive descent** (può avere backtracking). Viene implementato codificando ogni non terminale in una funzione. Per esempio sulla produzione: $A \rightarrow XY$

   ```
   A() { return X(C) and Y(C) }
   ```

2. **Predictive parsing** ("sbircia" il prossimo simbolo nell'input per decidere quale produzione applicare). Viene implementato creando una tabella le cui entry (Non terminale A da sviluppare, prossimo terminale nell'input) restituiscono la produzione A -> β da applicare.

IMPORTANTE Le grammatiche adatte ad una parser top down (predittivo):

1. devono essere **ben formate**:
   - Ogni non terminale deve avere almeno una produzione che lo definisca (altrimenti non saprei con cosa sostituire un non terminale). Caso negativo : A -> a B C, C-> d (B non ha produzione)
   - Ogni non terminale deve essere raggiungibile dal non terminal iniziale. Ma questa regola non ha importanza nel parsing top-down. Caso negativo: S -> S b | a, B -> c (B non è raggiungibile a partire dal simbolo iniziale S – la produzione B-> c può essere eliminata)
2. devono essere **non ambigue** (dovrei restituire più di un albero di derivazione per alcune frasi). Caso negativo: A -> a B, A -> C b, C -> a, B -> b. (sentenza "a b" ha due alberi di derivazione)
3. devono essere **non ricorsive sinistre** (potrei avere un loop infinito)
   Caso negativo: A -> A b, A -> c
4. devono essere **non fattorizzate a sinistra** (pure "sbirciando" il prossimo simbolo non saprei quale simbolo scegliere).
   Caso negativo: A -> a B, A -> a C, B -> b, C -> c. (se "A" è il simbolo da espandere ed "a" è il simbolo "sbirciato" nell'input comunque non saprei se usare A -> a B o A -> a C. - necessità di backtrack)

Per un **parser recursive descend**, posso costruire un parser con eventuale backtracking su grammatiche che rispettano necessariamente tutte le regole 1-3.

Per un **parser predittivo**, posso costruire un parser senza backtracking su grammatiche che rispettano necessariamente tutte le regole 1-4. Ne devono rispettare comunque altre tre che saranno approfondite successivamente. Tutte queste regole insieme rendono la grammatica LL(k).

3.3.1 First

Definiamo FIRST(α), in cui α è una generica stringa dei simboli della grammatica, come l'insieme dei terminali che costituiscono l'inizio delle stringhe derivabili da α:
Se $\alpha \Rightarrow \epsilon$, allora anche ϵ appartiene all'insieme FIRST(α).
Esempio: Abbiamo la seguente grammatica:

$$S \to AB$$
$$S \to c$$
$$A \to aA$$
$$A \to a$$
$$A \to \epsilon$$
$$B \to dB$$
$$B \to e$$

Calcoliamo il first(S):
$S \Rightarrow c$ oppure
$S \Rightarrow AB \Rightarrow aAB$ oppure
$S \Rightarrow b$ oppure
$S \Rightarrow \epsilon \Rightarrow B \Rightarrow dB$ oppure e
Si prende B perchè A potendo diventare epsilon scompare e quindi si va a prendere first(non terminale) o direttamente l'eventuale terminale.

Quindi first(S) = $\{c, a, b, d, e\}$, first(A)= $\{a, b, \epsilon\}$, first(B) = $\{d, e\}$

Nel FIRST(X) si può avere la presenza di ϵ ma non di $

Formula generale:
FIRST(X) = FIRST$(X_1, ..., X_n)$ = FIRST(X_1) $\cup_\epsilon$ FIRST(X_2) $\cup_\epsilon$ ... $\cup_\epsilon$ FIRST(X_n) $\cup_\epsilon$ $\{\epsilon\}$
Nota: $\cup_\epsilon$ vuol dire "Unito se c'è epsilon"

Altri esercizi

a) Scrivere la grammatica per le stringhe binari dove lo 0 è seguito da un 1.
$S \to 1S \mid 01S \mid \epsilon$

b) Scrivere la grammatica per tutte le sequenze di ((((()))(())):

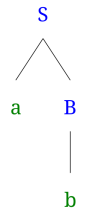
$$S \to S(S)S$$
$$S \to \epsilon$$

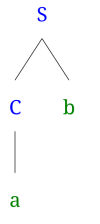
La prova è lasciata a chi legge.

c) Dire se la seguente grammatica è ambigua:

$$A \to aB$$
$$A \to Cb$$
$$C \to a$$
$$B \to b$$

La grammatica è ambigua possiamo infatti mostrare due alberi per la stringa ab





d) Data la seguente grammatica un parser top-down riconosce il suo linguaggio?

$$E \to E + T$$
$$E \to T$$
$$T \to T * F$$
$$T \to F$$
$$F \to num$$

No perché è ricorsiva sinistra. **Devo renderla non ricorsiva sinistra!**

e) Creare una grammatica per il seguente codice:

```
void main(){
    int a,b;
    f(b);
}
```

S → program
program → VOID ID LPAR **functionparams** RPAR LBRAKE body RBRAKE
functionparams → funzcall → ID COMMA **functionparams** | funzcall → ID | ϵ
body → **vardecl** SEMI **funzcalls**
vardecl → funzcall → **decl** | ϵ
decl → ID COMMA **decl** | ID COMMA | ID **vardecl**
funzcalls → **funzcall funzcalls** | funzcall
funzcall → ID LPAR **params** RPAR SEMI
params → ID COMMA **params** | ID | ϵ
type → INT | FLOAT

3.3.2 Follow

L'insieme FOLLOW(X) del non terminale X è l'insieme dei terminali che possono seguire A in una qualsiasi forma sentenziale partendo da S. Si controllano tutte le forme sentenziali dove è presente A.

- Nei follow non deve mai comparire ϵ
- FOLLOW(S) = ha sempre il simbolo $
- Sia S $\underset{*}{\Rightarrow}$ α $A\underline{x}$ β, il FOLLOW(A) = FIRST(x):
  - Se x è un terminale è il non terminale stesso;
  - Se x è un non terminale α allora il follow è FIRST(α);
  - Se il FIRST(α) contiene ϵ si prendono anche i FOLLOW(α), quindi avremo che FOLLOW(A) = FIRST(α) $\cup_\epsilon$ FOLLOW(α);
  - Se A è l'ultimo simbolo allora allora si prendono i follow del Non terminale padre della produzione, nel nostro esempio S.

3.3.3 Algoritmo per la costruzione di una tabella per un parsing predittivo

La tabella a sulle colonne tutti i non terminali della grammatica . Ha sulle righe tutti i terminali della grammatica.

Input: una Grammatica G
Output: Tabella di Parsing M
Metodo: Per ogni produzione $A \to \alpha$ della grammatica G si svolgono i seguenti passi:

1. Per ogni terminale α in FIRST(α) si aggiunge la produzione A → nella cella $M[A, \alpha]$
2. Se ϵ appartiene a FIRST(α) allora per ogni terminale b in FOLLOW(A) si aggiunge una produzione nella cella corrispondente.

Se dopo aver svolto questi passi ci sono delle celle senza produzioni allora quelle celle sono dette **condizioni di errore**.

3.3.4 Esercizio First, Follow e tabella di parsing

La grammatica su cui effettueremo l'esempio è la seguente:
E → T E'
E' → + T E' | ϵ
T → F T'
T' → * F T' | ϵ
F → (E) | id

Iniziamo calcolando i First

1. Al passo 1 Calcoliamo i FIRST di E', T' e F.

	FIRST
E	
E'	{+,ϵ}
T	
T'	{*,ϵ}
F	{(,id}

2. Al passo 2 Calcoliamo i FIRST di T.

	FIRST
E	
E'	{+,ϵ}
T	{(,id)
T'	{*,ϵ}
F	{(,id}

3. Infine calcoliamo i FIRST di E

	FIRST
E	{(,id)}
E'	{+,ϵ}
T	{(,id)
T'	{*,ϵ}
F	{(,id}

Calcoliamo ora i follow:

	FIRST	**FOLLOW**
E	{(,id)}	{), $ }
E'	{+,ϵ}	{), $ }
T	{(,id)	{+,), $}
T'	{*,ϵ}	{+,), $}
F	{(,id}	{*, +,), $}

Generiamo ora la tabella delle produzioni:

Non terminale	Simbolo d'ingresso					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathrm{id}$			$F \to (E)$		

NOTA: se nella tabella in una cella compaiono più di una produzione questo implica che il parser dovrà effettuare un backtrack, **e a noi non piace**.

3.4 LL(1)

Le grammatiche la cui tabella delle produzioni per ogni cella ha **esattamente una** produzione sono dette **grammatiche LL1**.
LL(1) sta per:

- L = left-to-right
- L = leftmost derivation
- (1) = un carattere di lookahead

LL(1) è un parser top-down predittivo senza backtrack, deve rispettare le seguenti regoli già definite in precedenza:

1. G → ben formata
2. Non ambigua
3. Non ricorsiva sinistra
4. No fattorizzazione a sinistra

Vengono poi aggiunte 3 nuove regole, prendiamo in esempio la seguente grammatica:
A $\to \alpha$
A $\to \beta$

5. FIRST(α) ∩ FIRST(β) = $\emptyset$. Questo permette di evitare conflitti nella tabella delle produzioni.
   ***Hack**: Se c'è un ϵ si può saltare la regola.*
6. Se da α posso derivare ϵ oppure se da β posso derivare ϵ la grammatica è LL(1). Se sono entrambe ϵ non lo è.
7. Se $\alpha \overset{*}{\Rightarrow} \epsilon$ allora FIRST(β) ∩ FOLLOW(A) = $\emptyset$.
   OPPURE
   Se $\beta \overset{*}{\Rightarrow} \epsilon$ allora FIRST(α) ∩ FOLLOW(A) = $\emptyset$

NOTA: Un linguaggio è LL(1) se esiste una grammatica LL(1) che lo riconosce.

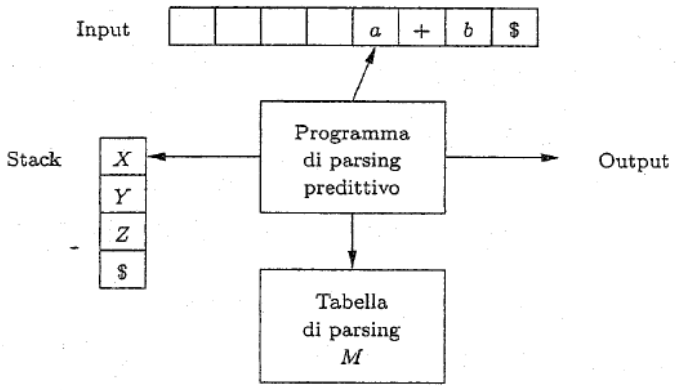
3.4.1 Parsing predittivo non ricorsivo

Un parser predittivo non ricorsivo può essere costruito gestendo uno stack esplicitamente, piuttosto che facendo affidamento sullo stack (implicito) dovuto alle chiamate ricorsive. Il parser riproduce il processo di derivazione sinistra. Se w è la porzione dell'ingresso riconosciuta a un certo momenento, allora lo stack contiene una sequenza di simboli grammaticali α tali che:

$$S \stackrel{*}{\Rightarrow} w\alpha$$

Questo parser è composto come segue:

- un buffer di ingresso, che conterrà la stringa da analizzare ed il simbolo \$ per indicare la fine dello stack;
- uno stack, che contiene una sequenza di simboli grammaticali, in questo caso deve essere gestito dal programmatore;
- Una tabella di parsing;
- Uno stream di output, che può essere un eventuale albero di parsing.



L'algoritmo da applicare per simulare lo stack è il seguente:

ip punta al primo simbolo della sequenza d'ingresso w;
assegna a X il simbolo alla cima dello stack;
while ($X \neq \$$) { /* lo stack non è vuoto */
    if (X è a) rimuovi l'elemento alla cima dello stack e avanza il puntatore *ip*;
    else if (X è un terminale) *errore*();
    else if ($M[X, a]$ indica un errore) *errore*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        produci come uscita la produzione $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        rimuovi l'elemento alla cima dello stack;
        poni $Y_k, Y_{k-1}, \ldots, Y_1$ sullo stack, con Y_1 in cima;
    }
    assegna a X il simbolo alla cima dello stack;
}

Esercizio sullo stack La grammatica usata è quella dell'esercizio 3.3.4 usando anche i FIRST e FOLLOW.

Riconosciuta	Stack	Input	Azione
	$E\$$	$\mathrm{id} + \mathrm{id} * \mathrm{id}\$$	
	$TE'\$$	$\mathrm{id} + \mathrm{id} * \mathrm{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathrm{id} + \mathrm{id} * \mathrm{id}\$$	output $T \to FT'$
	$\mathrm{id}\ T'E'\$$	$\mathrm{id} + \mathrm{id} * \mathrm{id}\$$	output $F \to \mathrm{id}$
id	$T'E'\$$	$+\ \mathrm{id} * \mathrm{id}\$$	consuma id
id	$E'\$$	$+\ \mathrm{id} * \mathrm{id}\$$	output $T' \to \epsilon$
id	$+\ TE'\$$	$+\ \mathrm{id} * \mathrm{id}\$$	output $E' \to +\ TE'$
id +	$TE'\$$	$\mathrm{id} * \mathrm{id}\$$	consuma +
id +	$FT'E'\$$	$\mathrm{id} * \mathrm{id}\$$	output $T \to FT'$
id +	$\mathrm{id}\ T'E'\$$	$\mathrm{id} * \mathrm{id}\$$	output $F \to \mathrm{id}$
id + id	$T'E'\$$	$* \mathrm{id}\$$	consuma id
id + id	$* FT'E'\$$	$* \mathrm{id}\$$	output $T' \to * FT'$
id + id *	$FT'E'\$$	$\mathrm{id}\$$	consuma *
id + id *	$\mathrm{id}\ T'E'\$$	$\mathrm{id}\$$	output $F \to \mathrm{id}$
id + id * id	$T'E'\$$	$\$$	consuma id
id + id * id	$E'\$$	$\$$	output $T' \to \epsilon$
id + id * id	$\$$	$\$$	output $E' \to \epsilon$

- La seguente grammatica è LL(1)?:
S $\Rightarrow AB$
S $\Rightarrow a$
A $\Rightarrow bB$
A $\Rightarrow \epsilon B$
B $\Rightarrow bB$

Per prima cosa calcoliamo i first e i follow:

	FIRST	**FOLLOW**
S	{ a, b}	{ $ }
A	{ b, ϵ}	{ b }
B	{ b }	{ $ }

Calcoliamo anche i First delle produzioni:

	FIRST
S-> A B	{ b }
S -> a	{ a }
A -> b	{ b }
A -> ϵ	{ ϵ }
B -> b	{ b }

Controlliamo se è LL(1)
Per la coppia S verifichiamo le 3 regole:

1. FIRST(AB) $\cap$ FIRST(a) = $\emptyset$
2. La regola 2 e 3 sono verificate

Per la coppia A verifichiamo le 3 regole:

1. FIRST(b) $\cap$ FIRST(ϵ) = $\emptyset$
2. La regola 2 è vera
3. FIRST(b) $\cap$ FOLLOW(A) = $\{b\} \cap \{b\}$. **La grammatica non è LL(1)**

- Verifica se la seguente grammatica è LL(1), se non lo è trasformala Z $\Rightarrow ZB$
Z $\Rightarrow y$
B $\Rightarrow Bx$
B $\Rightarrow Ax$
A $\Rightarrow z$
A $\Rightarrow zZy$

La grammatica non è LL(1) perchè è sia ambigua sia fattorizzabile. RISOLVIAMO:
Z $\Rightarrow y\ S'$
S' $\Rightarrow B\ S' | \epsilon$
B $\Rightarrow A\ x\ B'$
B' $\Rightarrow x\ B' | \epsilon$
A $\Rightarrow z\ R$
R $\Rightarrow Z\ y | \epsilon$

Nonterminal	Nullable?	First	Follow
S	✗	y	
Z	✗	y	y, $
S'	✗	z	y, $
S''	✓		
B	✗	z	z
B'	✓	x	z
A	✗	z	x
R	✓	y	x

	$	y	x	z
S		S ::= Z $		
Z		Z ::= y S'		
S'				S' ::= B S'
S''				
B				B ::= A x B'
B'			B' ::= x B'	B' ::= ε
A				A ::= z R
R		R ::= Z y	R ::= ε	

Controlliamo la correttezza delle tre regole (ANche se si deve fare prima): Per la coppia S' verifichiamo le 3 regole:

1. prima regola valida perchè c'è ϵ
2. La regola 2 è verificata
3. FIRST(BS') $\cap$ FOLLOW(S') $= \{z\} \cap \{y\} = \emptyset$.

Per la coppia B' verifichiamo le 3 regole:

1. prima regola valida perchè c'è ϵ
2. La regola 2 è verificata
3. FIRST(xB') $\cap$ FOLLOW(B') $= \{x\} \cap \{z, y\} = \emptyset$.

Per la coppia R verifichiamo le 3 regole:

1. prima regola valida perchè c'è ϵ
2. La regola 2 è verificata
3. FIRST(ZY) $\cap$ FOLLOW(R) $= \{y\} \cap \{x\} = \emptyset$.

La grammatica è LL(1)

3.5 Parsing bottom-up

Un parsing bottom-up è un parsing che tenta di di costruire un albero di derivazione per una stringa in input iniziando dalle foglie fino ad arrivare alla radice.
Ad ogni passo in cui effettuiamo una riduzione di una sottostringa quando abbiamo un match con una produzione sostituiamo la sottostringa con il non terminale a sinistra di quella produzione. Se ad ogni passo scegliamo bene la produzione arriveremo alla fine ad ottenere il non terminale padre della grammatica e questo corrisponde ad una right-most inversa.
Le decisioni fondamentali nel parsing bottom-up riguardano **quando effettuare una riduzione** e **quale produzione applicare**.

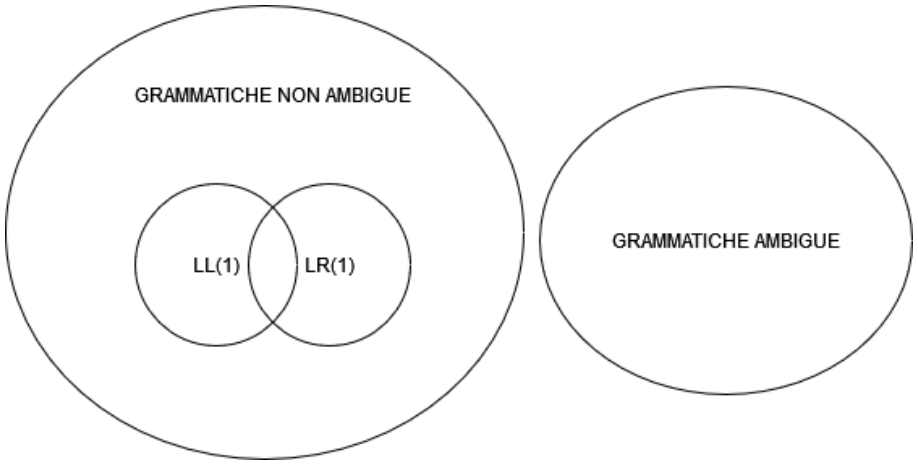
Esempio: Data la seguente grammatica

$$
\begin{aligned}
&0) E' \to E\$ \\
&1) E \to E + T \\
&2) E \to T \\
&3) T \to T * F \\
&4) T \to F \\
&5) F \to (E) \\
&6) F \to id
\end{aligned}
$$

e la seguente stringa id + id:
$\underline{\text{id}} + \text{id} \overset{6}{\Rightarrow} \underline{\text{F}} + \text{id} \overset{4}{\Rightarrow} \underline{\text{T}} + \text{id} \overset{2}{\Rightarrow} \text{E} \underline{+} \text{id} \Rightarrow \text{E} + \underline{\text{id}} \overset{6}{\Rightarrow} \text{E} + \underline{\text{F}} \overset{4}{\Rightarrow} \underline{\text{E} + \text{T}} \overset{1}{\Rightarrow} \text{E}$

Questa è proprio una right-most inversa. Informalmente un "handle" è una sottostringa che corrisponde al corpo di una produzione e la cui riduzione rappresenta un passo di una derivazione più a destra.
Nel nostro caso gli Handle sono:
id perchè $F \to id$
F perchè $T \to F$
id perchè $F \to id$
E + T perchè $E \to E + T$
T perchè $E \to T$

3.5.1 Insieme delle grammatiche (prima parte)



3.5.2 Parser shift-reduce

Un parser shift-reduce è un tipo di parsing bottom-up che sfrutta uno stack ed un buffer:

- Stack: conterrà i simboli della grammatica, la fine dello stack è segnata con $
- Buffer: contiene la stringa di input di cui vogliamo verificare l'appartenenza alla grammatica.

Durante l'analisi dell'input il parser sposterà a destra zero o più simboli finchè non effettuerà una riduzione con un terminale. Una volta che sullo stack ci sarà il terminale iniziale e il buffer conterrà solo $, il parser annuncerà l'accettazione dell'input e in tutti gli altri casi darà errore.
Le possibili azioni che può compiere il parser sono:

1. Shift: Sposta il simbolo di input successivo in cima allo stack.
2. Reduce: si rimpiazza l'handle sul top dello stack con un simbolo non terminale;
3. Accetta: annuncia il completamento dell'analisi con successo.
4. Errore: annuncia il rifiuto dell'input con un errore di sintassi.

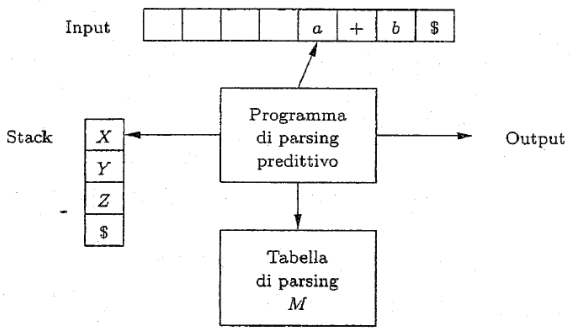
Esempio: La grammatica è quella usata nell'esempio precedente, la stringa di input è la seguente id * id

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \to T * F$
$\$ T$	$\$$	reduce by $E \to T$
$\$ E$	$\$$	accept

L'uso dello stack per un parser shift-reduce è giustificato dal fatto che, con tale struttura un handle è sempre sul top dello stack e mai all'interno.

3.6 Parser LR

Il parser LR riesce a riconoscere gli handle grazie l'utilizzo di un automa push down. Lo schema di un parser LR è il seguente:



- **Driver**: è lo stesso per ogni parser ed esegue le iperazioni di shift, reduce, accept, error;
- **buffer input**: stringa da analizzare;
- **Stack**: utilizzato per immagazzinare una stringa nella forma $S_0X_1....S_nX_n$ dove S_n è il top, ogni X_i è un simbolo della grammatica, ogni S_i è uno stato dell'automa;
- **Tabella di Parsing** (figura 3.1): si costruisce di volta in volta e dipende dalla grammatica che si sta implementando. Le righe saranno gli stati dell'automa. Le colonne sono i non terminali e terminali.
  La tabella è composta da due parti:
  - **action**: sono presenti i terminali, e sono presenti sia operazioni di shift che di reduce;
  - **goto**: sono presenti i non terminali e sono presenti operazioni di reduce. Inserirà sullo steck il non terminale padre della produzione.

  Le operazioni di shift sono indicate con Sx, dove x indica lo stato in si shift. Le operazioni di reduce sono indicate con Rx dove x indica di quale produzione perendere il non terminale padre.

STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			

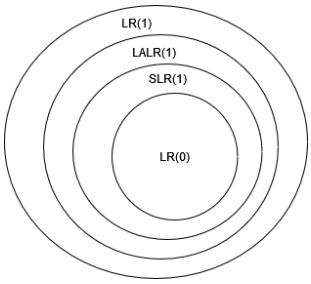
Figura 3.1: struttura tabella di parsing

L'analisi LR è scelta per diverse ragioni:

- I parser LR possono essere costruiti per riconoscere tutti i costrutti dei linguaggi di programmazione per i quali è possibile scrivere grammatiche context free.
- Il metodo di analisi LR è il più generale metodo di analisi shift-reduce senza ritorno indietro.
- La classe delle grammatiche che possono essere analizzate usando i metodi LR è un grosso sottoinsieme proprio della classe delle grammatiche che possono essere analizzate con un parser predittivo.

Le classi sono le seguenti:

- La LR(0) è la parsing table più semplice ma è anche quella che presenta più facilmente conflitti;
- La SLR(1) elimina dei conflitti presenti nella LR(0), ma è più complessa; comunque può presentare dei conflitti;
- La LR(1) o LLR(1) è la più potente nel senso che si eliminano la maggior parte dei conflitti; però la grammatica perde il suo potere descrittivo (motivo per il quale sono state introdotte le grammatiche); inoltre la parsing table può essere molto complessa (molti stati) e molto costosa.
- LALR(1) è la più utilizzata. Poiché l'LR(1) è troppo costosa in termini di stati si rinuncia ad identificare tutti i conflitti accorpando alcuni stati del diagramma che hanno item in comune.



Quindi se un linguaggio è LR(0) sarà anche SLR(1), LALR(1), LR(1). Ci possono essere più grammatiche, anche in una stessa classe, che generano uno stesso linguaggio. Un linguaggio è inerentemente ambiguo se ogni grammatica che lo genera è ambigua (porterà a parser non deterministici, non li vedremo).

- LR(0): è la parsing table più semplice ma è anche quella che presenta più facilmente conflitti;
- SLR(1) o Simple LR(1): elimina dei conflitti presenti nella LR(0), ma è più complessa; comunque può presentare dei conflitti;.
- LR(1): è l'LR canonico, è la più potente in quanto si eliminano la maggior parte dei conflitti; però la grammatica perde il suo potere descrittivo (motivo per il quale sono state introdotte le grammatiche); inoltre la parsing table può essere molto complessa (molti stati) e molto costosa;
- LALR(1) o Look-ahead LR: è la più usata in quanto la LR(1) risulta troppo costosa. In generale fonde alcuni stati del diagramma che hanno item in comune.

Suggerimenti utili per esercizi

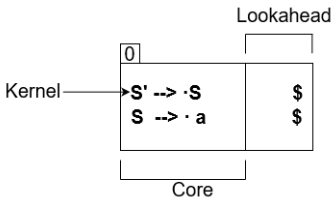
1. Se è LL(1) non è detto che non sia LR(1);
2. Se è LR(1) non è detto sia LALR(1), SLR(1) o LR(0), se non è LR(1) non è SLR(1), LALR(1) o LR(0);
3. Se è SLR(1) è LR(1), se è SLR(1) è LALR(1);
4. Se è LALR è LR(1), se è LALR(1) non è detto sia SLR(1).

3.6.1 Item LR(0) e automa LR(0)

Per prima cosa presa una qualsiasi gramamtica si aggiunge una produzione iniziale S' $\rightarrow X$, dove X è il non terminale iniziale della grammatica.

Un item LR è composto da:

- **Kernel** (o Generatore): è la produzione che genera tutte le produzioni dell'item;
- **Core:** è la parte sinistra dell'item, in cui sono presenti tutte le produzioni;
- **Lookahead:** se presente è la parte destra dell'item ed indica i follow di una produzione.



Successivamente si procede costruendo gli **Item LR(0)**, cioè una produzione al cui interno abbiamo un ·, che indica il prossimo non terminale/terminale da leggere e quelli già letti.
Esempio A $\rightarrow$ X · y.
Un item LR(0) fa uso anche del concetto di chiusura degli insiemi.

Chiusura (clousure) degli item

Se I è un insieme di item della grammatica G, allora CLOSURE(I) è un nuovo insieme di item costruito a partire da I secondo le seguenti regole:

1. Inizialmente si aggiungono tutti gli item di I all'insieme CLOUSURE(I);
   Esempio: Abbiamo la grammatica :
   S $\rightarrow aB$
   B $\rightarrow c$
   Iniziamo aggiungendo S':
   S' $\rightarrow S$
   S $\rightarrow aB$
   B $\rightarrow c$
   Ora eseguiamo i passi di una chiusura in un item:

   - Nel primo passo abbiamo:
     S' $\rightarrow \cdot S$
     S $\rightarrow \cdot aB$
   - Immaginando di leggere "a" e quindi di cambiare stato:
     S $\rightarrow a \cdot B$
     B $\rightarrow \cdot c$

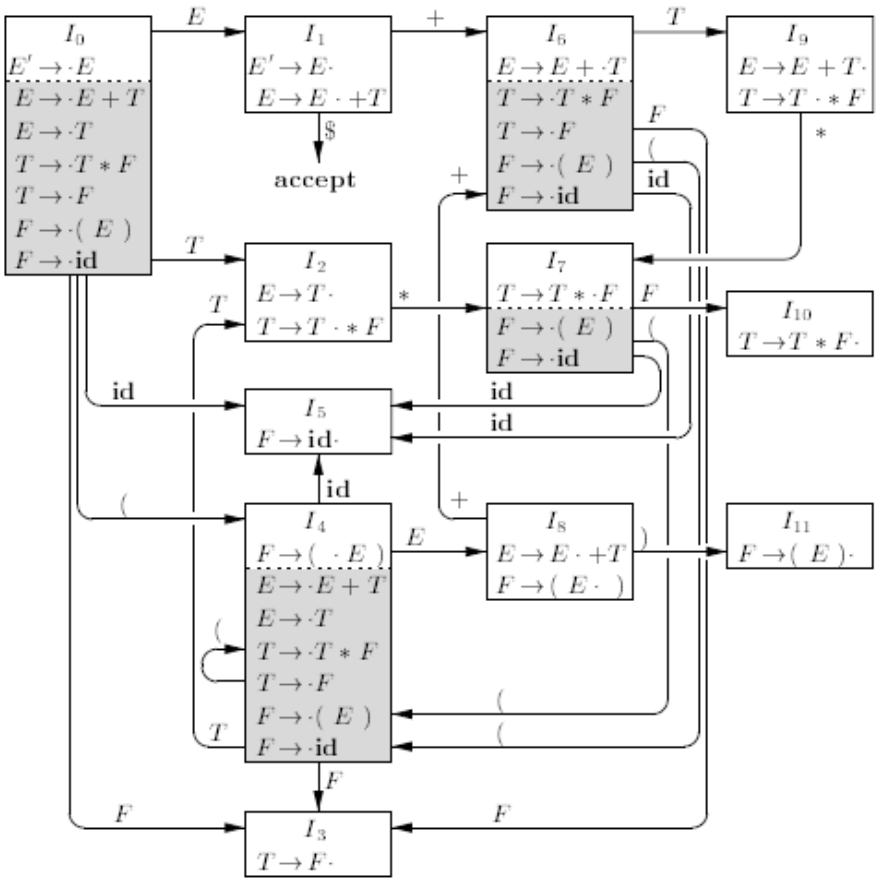
   Si procede così finchè non si hanno solo item completi.

2. In caso abbiamo una produzione S $\rightarrow a\epsilon$, nell'item avremo $\rightarrow a\cdot$, questo corrisponde ad un item completo e all'interno dello stack non avremo cambiamenti. Quando nell'SLR(1) implementeremo i follower, quando avremo $S \rightarrow \cdot$ all'interno della tabella inseriremo delle reduce sotto i terminali corrispondenti.

Esercizio parsing LR(0)

La grammatica su cui creare il parsing LR(0)è la seguente:
1) E $\rightarrow$ E + T
2) E $\rightarrow$ T
3) T $\rightarrow$ T * F
4) T $\rightarrow$ F
5) F $\rightarrow$ (E)
6) F $\rightarrow$ id

Iniziamo inserendo E':
0) E'$\rightarrow$ E
1) E $\rightarrow$ E + T
2) E $\rightarrow$ T
3) T $\rightarrow$ T * F
4) T $\rightarrow$ F
5) F $\rightarrow$ (E)
6) F $\rightarrow$ id



Creiamo la parsing table:

Stato	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Eseguiamo lo stack sull'input id * id + id $

Passo	Stack	Simboli	Input	Azione
(1)	0		id * id + id $	shift 5
(2)	0 5	id	* id + id $	reduce $F \rightarrow$ id
(3)	0 3	F	* id + id $	reduce $T \rightarrow F$
(4)	0 2	T	* id + id $	shift 7
(5)	0 2 7	T *	id + id $	shift 5
(6)	0 2 7 5	T * id	+ id $	reduce $F \rightarrow$ id
(7)	0 2 7 10	T * F	+ id $	reduce $T \rightarrow T * F$
(8)	0 2	T	+ id $	reduce $E \rightarrow T$
(9)	0 1	E	+ id $	shift 6
(10)	0 1 6	E +	id $	shift 5
(11)	0 1 6 5	E + id	$	reduce $F \rightarrow$ id
(12)	0 1 6 3	$E + F$	$	reduce $T \rightarrow F$
(13)	0 1 6 9	$E + T$	$	reduce $E \rightarrow E + T$
(14)	0 1	E	$	accept

3.6.2 Parser SLR(1)

Una parser SLR(1) sfrutta gli Item LR(0) ma ha un'aggiunta; quando incontra un item completo (S $\rightarrow a\cdot$) aggiunge i follow del non terminale all'item completo, nel nostro caso aggiungerà il FOLLOWER(S) all'item completo. Questo ci permette in determinati casi di individuare eventuali conflitti.
I conflitti in un parser LR possono essere di due categorie:

- **Shift/Reduce:** si verifica quando bisogna fare uno shift su un determinato terminale (es: =) ed allo stesso momento bisogna fare una reduce in quanto risulta esserci un item completo che nel follow avrà lo stesso simbolo sul quale effettuare l'azione shift (es ($, =).
- **Reduce/Reduce:** Si verifica nel momento in cui abbiamo all'interno di uno stato due o più item completi che nel loro follow hanno lo stesso non terminale (es: primo_insieme(a, b) - secondo_insieme(c, b).

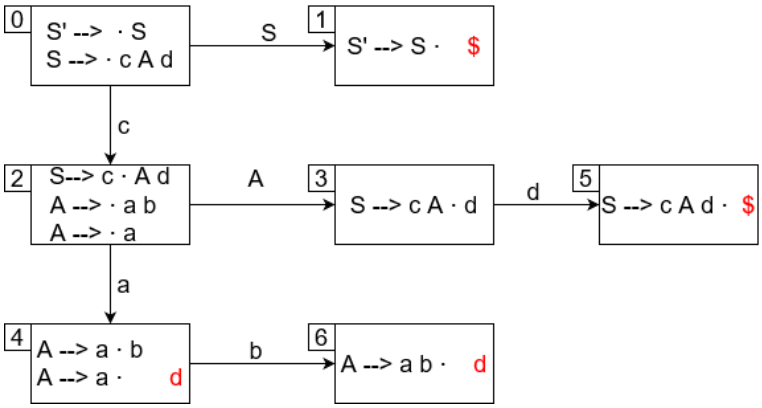
NOTA 1: Tutte le grammatiche SLR(1) non sono ambigue, ma non tutte le grammatiche non ambigue sono SLR(1).

NOTA 2: Se la domanda di esame è: La seguente grammatica è LL(1)? Ci si può fermare appena si trova un conflitto nell'automa.

Esercizio parsing SLR(1)

La grammatica è la seguente:
0) S' → S
1) S → c A d
2) A → a b
3) A → a

L'automa SLR(1) è il seguente:



La parsing table è la seguente: L'esecuzione dello stack sull'input c a d:

State	ACTION					GOTO		
	c	d	a	b	$	S'	S	A
0	s2						1	
1					acc			
2			s4					3
3		s5						
4		r_3		s6				
5					r_1			
6		r_2						

Trace			
Step	Stack	Input	Action
1	0	c a d $	s2
2	0 c 2	a d $	s4
3	0 c 2 a 4	d $	r_3
4	0 c 2 A	d $	3
5	0 c 2 A 3	d $	s5
6	0 c 2 A 3 d 5	$	r_1
7	0 S	$	1
8	0 S 1	$	acc

3.6.3 Parser LR(1)

Una grammatica SLR(1) è soggetta a conflitti. Questo è dovuto all'uso di follow generali, e non contestuali.

- Nel caso di grammatica SLR(1) il follow si indica solo quando mi trovo in un item di riduzione;
- Nel caso di grammatica LR(1) il follow si indica in tutti gli item, anche quelli di non riduzione.

In LR(1) si parla di **Follow Contestuale** per definire quel sottoinsieme di FOLLOW costituito dai simboli che in quel particolare stato potrebbero seguire la produzione completa dell'item.

Capire meglio il Follow Contestuale

Sia la grammatica: S → C C
C → c C
C → d
e considerando lo stato 0 dell'automa sottostante.

1. Il primo item LR(0) è S'→•S con Follow Contestuale banalmente $, in cui S segue il •. Quindi aggiungeremo item per ogni produzione per S, e a tutti questi nuovi item verrà assegnato l'insieme di follow contestuali **$**, perché questo è ciò che segue la S qui. Quindi questo ci fa aggiungere il secondo item LR(0) S→•CC con Follow Contestuale **$**;
2. Il secondo item LR(0) S→•CC ha ancora un non terminale, C in questo caso, che segue il •. Quindi aggiungeremo item per ogni produzione per C, e a tutti questi nuovi item verrà assegnato l'insieme di follow contestuali **{c,d}** (S→•C<u>C$</u>, il first di questa <u>C$</u>);
3. Il terzo ed il quarto item LR(0) generati, dal passo 2, saranno rispettivamente C→•cC e S→•d con Follow Contestuale ereditato **{c,d}**.

Modifichiamo la grammatica nel modo seguente:
S → C C
C → c C
C → d
C → ϵ

1. Il primo item LR(0) è S'→•S con Follow Contestuale banalmente $, in cui S segue il •. Quindi aggiungeremo item per ogni produzione per S, e a tutti questi nuovi item verrà assegnato l'insieme di follow contestuali **$**, perché questo è ciò che segue la S qui. Quindi questo ci fa aggiungere il secondo item LR(0) S→•CC con Follow Contestuale **$**;
2. Il secondo item LR(0) S→•CC ha ancora un non terminale, C in questo caso, che segue il •. Quindi aggiungeremo item per ogni produzione per C, e a tutti questi nuovi item verrà assegnato l'insieme di follow contestuali **{c,d,$}** (S→•C<u>C$</u>, il first di questa <u>C$</u> che siccome C può andare in ϵ deve comprendere anche il follow che risulta essere $);
3. Il terzo ed il quarto item LR(0) generati, dal passo 2, saranno rispettivamente C→•cC e S→•d con Follow Contestuale ereditato **{c,d$}**.

L'automa LR(1) per la grammatica riporta sopra è:

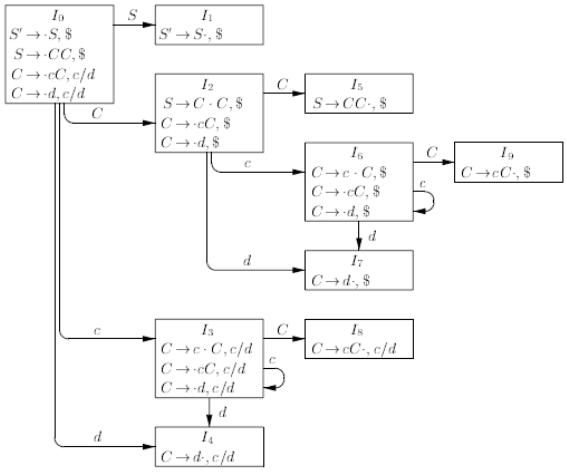
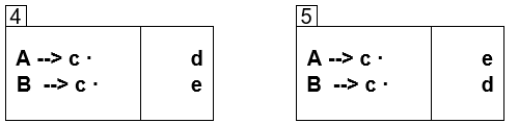


Figura 3.2: Automa LR(1)

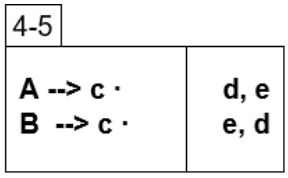
Possiamo subito notare che gli stati esplodono già per grammatiche piccole. LR(1) inizialmente non era usato per limitazioni di memoria, e quindi si è arrivati alla creazione delle grammatiche LALR(1).

3.6.4 Grammatiche LALR(1)

Le grammatiche LALR(1) si occupano di effettuare un operazione di merge degli stati di un parser LR(1) che hanno core uguale. ES:



Avendo core uguale si possono unire, il risultato sarà un nuovo stato avendo il core di uno dei due stati (quale prendo è ininfluente sono uguali PORCO DIO) e per ogni produzione dei core unisco i follow:



NOTA 1: Ai fini dell'esame possiamo non rifare l'automa, e indicare soltanto le coppie che conterranno gli automi da unire. Nel caso dell'automa nella figura 3.2 le coppie sono: (3,6) (4,7) (8,9).
NOTA 2: Se la traccia chiede di verificare se una grammatica è LR(1) basta controllare che quest'ultima sia SLR(1), a meno di infamate (NON SEMPRE).

3.6.5 Precedenze per risolvere i conflitti

Sappiamo che una grammatica ambigua non è LR(1) perchè alla creazione della parsing table avremo sicuramente un conflitto Shift/Reduce oppure Reduce/Reduce. Nella teoria non possiamo fare nulla per risolvere questo problema, ma nell'implementazione della nostra parsing table possiamo indicare la precedenza o l'associatività di una produzione.
Infatti la grammatica: **E** $\rightarrow$ **E** + **E** | **E** * **E** | **(E)** | **id** è ambigua perché non c'è nulla che indica se deve essere eseguito prima l'operatore + o *. Questo porterà ad avere nella parsing table in uno stato sicuramente un conflitto.
Nell'implementazione possiamo usare due modi per risolvere questi conflitti, entrambi faranno si che nella tabella dove è presente il conflitto si cancelli una delle due azioni incriminate:

- **Risoluzione standard:** in presenza di un conflitto Shift/Reduce vince lo shift, in presenza di un conflitto Reduce/Reduce vince la prima reduce. (Questa risoluzione standard non ci piace per la grammatica che abbiamo preso in esempio, Questo perchè porterebbe a dare la precendenza all'operatore +). (Può essere usata in CUP)
- **Risoluzione con precedenze:** indicando all'atto dell'implementazione quale operatore deve avere la precedenza, così quando la parsing table viene generata automaticamente l'azione che ha meno priorità viene cancellata in caso di conflitto.

NOTA 1: Se la tabella LR(1) non presenta conflitti Shift/Reduce allora anche la tabella LALR(1) non presenta conflitti Shift/reduce. Questo non vale per il conflitto Reduce/Reduce.

Dangling-Else Il dangling-else è un problema di ambiguità noto nella creazione di parser. Si verifica quando usiamo if innestati. Infatti, quando ci sono più istruzioni "if", la parte "else" non ha una visione chiara con quale "if" dovrebbe combinarsi. Infatti presa la seguente grammatica:
stmt $\rightarrow$ IF cond THEN IF cond THEN stmt ELSE stmt
stmt $\rightarrow$ IF cond THEN stmt

Possiamo avere due alberi di parsing, che possono portarci a dire che l'ultimo else può far parte o dell'if più vicino a lui o dell'if iniziale. Usando le precendenze possiamo risolvere questo problema, di norma si tende a dire che l'ultimo else si deve attaccare all'if a lui più vicino.

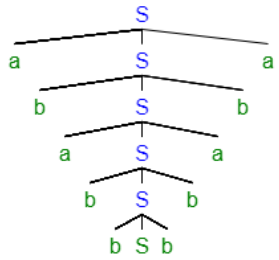
3.6.6 Esercizio Handle ben Marcato

Si consideri la grammatica
S → aSa | bSb | aa | bb
Si identifichi quale delle seguenti forme sentenziali destre derivabili da questa grammatica hanno handle ben marcato. (il marcamento degli handle nell'esercizio avviene tramite parentesi quadre):

- ○ abab[aababa]
- ○ ababaS[ababa]
- ◉ abab[bSb]baba
- ○ ababbS[bb]aba

Spiegazione della soluzione

La prima e la seconda scelta sono false perché nella grammatica considerata non esistono handle del genere.
La quarta scelta contiene un handle ma creando l'albero di derivazione destra e quindi riducendo l'handle in S otterrò la seguente derivazione ababbSSaba, in questo caso non riesco più a ridurre.
La terza scelta è quella giusta in quanto produce l'albero di derivazione corretto:



3.6.7 Esercizio handle di questa produzione

Si consideri la grammatica
S → CAd
A → ab
A → a
C → cC
C → c
Si indichi quale tra i seguenti è l'handle per la sentenza "ccabd":

- ○ la prima c che viene ridotta con C → c
- ○ la sotto-stringa "ab" che viene ridotta con A → ab
- ○ la seconda c che viene ridotta con C → cC

Per trovare la soluzione a questo quesito basta effettuare una right-most fino alla sentenza "ccabd": S → CAd → Cabd → cCabd → ccabd.
La risposta corretta è quindi la seconda c che viene ridotta con C → cC.

3.7 Grammatiche ad Attributi

Per costruire l'albero sintattico dato in output dal parser facciamo uso delle grammatiche ad attributi.
Una grammatica ad attributi è una grammatica context-free che non fa altro che aggiungere 0 o più attributi per ogni simbolo della grammatica. Per ogni produzione si possono avere 0 o 1 regola associata. Per esempio:

$$\text{E} \rightarrow E_1 + T \qquad ||E.s = E_1.val + T.val||$$

Nota: Il non terminale E e E_1 sono lo stesso non terminale si inserisce il pedice solo per distinguere gli attributi nella regola associata alla produzione.
Alcuni esempi sull'uso delle grammatiche ad attributi:

- Per un compilatore si potrebbero utilizzare per costruire un albero sintattico;
- Per un interprete si potrebbero utilizzare per fare eseguire direttamente il risultato finale di un'espressione, ad esempio alla lettura di 3 + 4 da direttamente il risultato della somma.

Le grammatiche ad attributi si dividono in:

- **Definizioni guidate dalla sintassi (SDD):** le regole vengono definite in modo dichiarativo cioè non è definito l'ordine in cui devono essere eseguite, se ne occuperà il processore; sono sempre poste alla fine delle produzioni; sono definite tra i seguenti due simboli ||...||. Sono chiamate azioni semantiche.
- **Schemi di traduzione**: le regole sono operative cioè viene specificato l'ordine in cui eseguirle; sono poste all'interno delle produzioni nel punto esatto in cui devono essere eseguite durante il riconoscimento. Sono chiamate regole semantiche. Sono definite tra i seguenti due simboli {...}

3.7.1 Definizioni guidate dalla sintassi

In una definizione guidata dalla sintassi gli attributi sono associati ai simboli grammaticali, mentre le regole alle produzioni. Es:

$$\text{A.val} \rightarrow x_1.val\ x_2.val\ x_3.val \qquad ||x_2.val = (A.val, x_3.val)||$$

Gli attributi di una grammattica possono essere di due tipi:

- **Sintetizzati**:
  1. Un attributo di un non terminale X padre della produzione deve dipendere solo dai suoi figli;
  2. Un attributo di un non terminale X padre di una produzione può dipendere da se stesso.

  Es:

$$A.s \rightarrow x_1.val\ x_2.val\ x_3.val \qquad ||A.s = x_3.val||$$
$$A.val \rightarrow \epsilon \qquad ||A.s = A.val||$$

- **Ereditati**: Usati per calcolare gli attributi di un non terminale figlio di una produzione, può dipendere dal padre, dai fratelli o da se stesso. Es:

$$A.val \rightarrow x_1.val\ x_2.val\ x_3.val \qquad ||x_2.val = (A.val, x_3.val)|| \tag{3.1}$$

  Questa regola va bene nel caso in cui abbiamo visto prima il padre e poi il fratello destro. In un parser bottom-up da sinistra verso destra non va bene perchè abbiamo visto soltanto il valore di x_1. Esiste un mode per rendere accettabili queste tipi di attributi tramite le grammatiche L-Attribuite.

3.7.2 Grammatiche S e L attribuite

- S-attribuite: una definizione guidata dalla sintassi è detta S-attribuita se e solo se ogni suo attributo è sintetizzato;
- L-attribuite: una definizione guidata dalla sintassi è detta L-attribuita se i suoi attributi sono sia sintetizzati che ereditati a sinistra. Questi tipi di attributi prendono il nome di L-ereditato.

Per una grammatica L-attribuita va bene un parser top-down per verificare le dipendenze. Per una grammatica sintetizzata o S-attribuita va bene un parser bottom-up. Per una grammatica che non è nessuna delle due si utilizza l'ordine topologico.

Dipendenze cicliche tra gli attributi:

Le grammatiche con dipendenze cicliche tra gli attributi non sono buone, questo perché sono irrisolvibili:
A -> B || A.s = B.i e B.i = A.s + 1 ||
è L-attribuita ma anche ciclica perché per calcolare l'attributo A.s abbiamo bisogno di B.i e viceversa.

3.7.3 Valutazione di una grammatica ad attributi con un albero

Prendiamo la seguente grammatica ad attributi con definizioni guidate dalla sintassi:

Produzioni	Regole
S → F T'	|| T'.i = F.val ; S.val = T'.s ||
T' → * F T'_1	|| T'_1.i = T'.i * F.val ; T'.s = T'_1.s||
T' → ϵ	|| T'.s = T'.i ||
F → digit	|| F.val = digit.val ||

Per prima cosa controlliamo che tipo di grammatica ad attributi è.
Costruiamo quindi una tabella degli attributi con 2 colonne; una colonna in cui avremo i non terminali ed un'altra con i loro attributi:

Non Terminali	Tipo degli attributi
S	
T'	
F	

Individuiamo nel primo passo tutti gli attributi che hanno i non terminali:

Non Terminali	Tipo degli attributi
S	val
T'	i s
F	val

Per capire gli attributi di che tipo sono dobbiamo analizzare le regole in cui sono calcolati; per la grammatica riportata sopra analizzeremo le seguenti regole:

- Per S.val analizziamo la regola S.val = T'.s nella produzione S -> F T'. Scopriamo così che S.val è sintetizzato perché dipende dal suo figlio T';
- Per T'.i dobbiamo analizzare due regole:
  - T'.i = F.val nella produzione S -> F T'. Scopriamo che è ereditato perché dipende dal suo fratello sinistro F;
  - T'_1.i = T'.i * F.val nella produzione T' -> * F T'_1. Scopriamo che è ereditato perché dipende dal suo fratello sinistro F e da suo padre T';

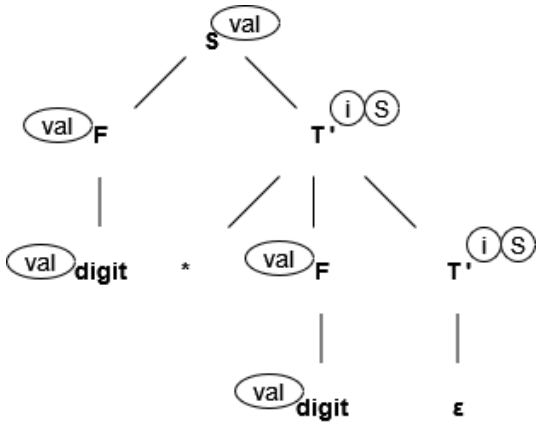
  Le due regole hanno entrambi attributi L-ereditati e quindi sono consistenti tra di loro. Quindi T'.i è L-ereditato.
- Per T'.s dobbiamo analizzare due regole:
  - T'.s = T_1'.s nella produzione T' -> * F T'_1. Scopriamo che è sintetizzato perché dipende dal figlio T'_1 ;
  - T'.s = T'.i nella produzione T' -> ϵ. Scopriamo che è sintetizzato perché dipende da se stesso;

  Le due regole hanno entrambi attributi sintetizzati e quindi sono consistenti tra di loro. Quindi T'.s è sintetizzato.
- Per F.val analizziamo F.val = digit.val nella produzione F -> digit. Scopriamo così che F.val è sintetizzato perché dipende dal suo figlio digit.

La grammatica è L-attribuita.

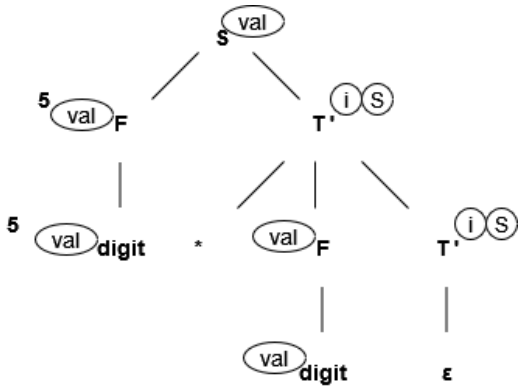
Non Terminali	Tipo degli attributi
S	val : sintetizzato
T'	i : L-ereditato s : sintetizzato
F	val : sintetizzato

Ora possiamo passare alla costruzione dell'albero e ad osservare come evolve il valore degli attributi dell'albero. Proviamo l'esecuzione sull'input 5 * 3.
Inizialmente costruiamo l'albero e per ogni nodo aggiungiamo anche i loro eventuali attributi:



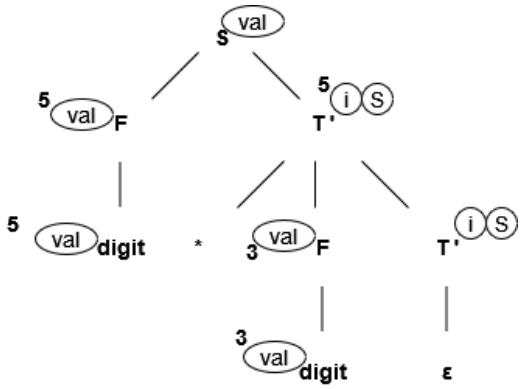
Durante l'analisi dell'albero passeremo per attributi sintetizzati e ereditati. Gli attributi ereditati saranno risolti per primi.

1. Partendo da S osserviamo che S.val è sintetizzato quindi dobbiamo prima scoprire il valore di F e T';
2. Andando in F vediamo che F.val è anch'esso sintetizzato, quindi ci serve scoprire prima il valore di digit;
3. Nel nodo di digit scopriamo che digit.val ha valore 5. Quindi risalendo l'albero e usando la regola "F.val = digit.val" anche F.val avrà valore 5.

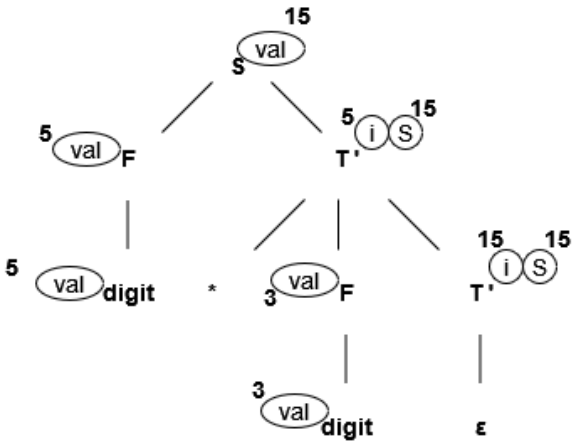


4. Risalendo l'albero siamo tornati ad S, non possiamo ancora calcolare il valore di S.val perché essendo sintetizzato dobbiamo ancora calcolare gli attributi di T'; T' ha due attributi i ereditato e s sintetizzato. Per i ereditato abbiamo diverse scelte per calcolarlo. T'.i = F.val e T'_1.i = T'.i * F.val. Essendo che il nostro T' è figlio di S usiamo la regola T'.i = F.val perché si trova nella prima produzione. Il valore di T'.val è 5 quindi.

5. Per calcolare T'.s iniziamo a scendere l'albero dal ramo di T' ci troviamo di fronte a <*, F e T'>.
   Il carattere * è un terminale e non ha attributi, possiamo tralasciarlo. Passiamo a calcolare gli attributi di F.
6. Calcoliamo gli attributi di F come per il punto 2 e 3. Da qui otteniamo che F.val ha valore 3.



7. Ora passiamo a calcolare T'; come abbiamo visto precedentemente T' ha due attributi: l'attributo **i** ereditato e l'attributo **s** sintetizzato. Per T'.i sappiamo che è figlio della produzione T' -> * F T'_1, quindi utilizziamo la regola T'_1.i = T'.i * F.val. Abbiamo quindi che T'.i assume valore 5 * 3 = 15. Da qui possiamo ora calcolare il valore dell'attributo T'.s grazie alla regola della produzione T' -> ϵ, e otteniamo T'.s = 15. Ora possiamo iniziare a risalire
8. Risalendo al primo T' possiamo calcolare anche il suo attributo T'.s usando la stessa regola utilizzata sotto e otteniamo anche qui T'.s = 15.
9. Tornati ad S osserviamo che gli attributi dei suoi due figli sono calcolati, Quindi possiamo ora calcolare il valore dell'attributo S.val utilizzando la regola S.val = T'.s; otteniamo così che S.val = 15.



Nota: la visita che abbiamo usato per calcolare gli attributi sintetizzati nell'albero prende il nome di visita dell'albero top-down con risalita.

3.7.4 Schema di traduzione

Ricorda che cenzino ti vuole bene e che cane bianco in realtà fa finta di avere la 104

Per costruire una grammatica ad attributi secondo uno schema di traduzione, basta specificare l'ordine in cui eseguire le regole. Un modo per costruire uno schema di traduzione si basa sull'aver prima costruito una definizione guidata dalla sintassi:

1. Si prende una definizione guidata dalla sintassi;
2. Per ogni attributo di un Non Terminale si indica se l'attributo è sintetizzato o ereditato;
3. Successivamente si scrivono le regole secondo il seguente ordine:
   - Le regole che riguardano gli attributi ereditati devono essere inserite prima del Non Terminale a cui fanno riferimento;
   - Le regole che riguardano gli attributi sintetizzati devono essere inserite alla fine; come per le regole di una definizione guidata dalla sintassi.

Esempio di uno schema di traduzione
Data la seguente grammatica ad attributi scritta secondo una definizione guidata dalla sintassi, riscriverla secondo uno schema di traduzione:

Produzioni	**Regole**
S → F T'	|| T'.i = F.val ; S.val = T'.s ||
T' → * F T'$_1$	|| T'$_1$.i = T'.i * F.val ; T'.s = T'$_1$.s||
T' → ϵ	|| T'.s = T'.i ||
F → digit	|| F.val = digit.val ||

Gli attributi dei non terminali sono del seguente tipo:

Non Terminali	**Tipo degli attributi**
S	val : sintetizzato
T'	i : L-ereditato s : sintetizzato
F	val : sintetizzato

Possiamo ora quindi passare alla creazione di uno schema di traduzione:

- Nella prima produzione abbiamo un attributo ereditato T'.i e uno sintetizzato S.val; inseriremo la regola per l'attributo ereditato T'.i prima del non terminale T', mentre la regola dell'attributo sintetizzato verrà inserita alla fine;
- Nella seconda produzione abbiamo la stessa situazione, inseriremo quindi T'$_1$.i prima del non terminale T'$_1$, mentre T'$_1$ alla fine;
- La terza e quarta produzione hanno entrambi attributi sintetizzati quindi scriveremo le regole alla fine.

Produzioni
S → F { T'.i = F.val } T' { S.val = T'.s }
T' → * F { T'$_1$.i = T'.i * F.val } T'$_1$ { T'.s = T'$_1$.s }
T' → ϵ { T'.s = T'.i }
F → digit { F.val = digit.val }

Nota: in una regola associata ad una produzione si possono usare solo i simboli (terminali o non terminali) di quella produzione:

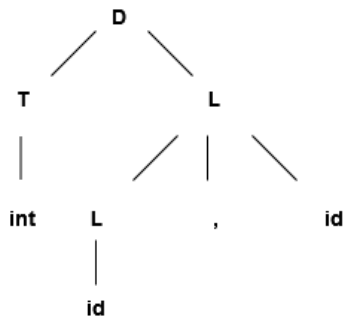
$$A \to x_1 x_2 B \qquad ||A.val = x_1.val + x_3.val||$$
$$B \to x_3 \qquad ||B.val = x_3.val||$$

La prima regola non va bene, mentre la seconda si.

3.7.5 Esercizi sulle grammatiche ad attributi

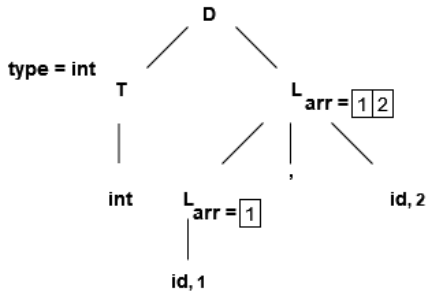
1) Data la seguente grammatica:
D → T L
T → int
T → float
L → L_1 , id
L → id
Vogliamo creare una grammatica ad attributi che data una stringa di input con la seguente forma <int id,id> costruisca una tabella dei simboli. Si può ipotizzare l'esistenza di una funzione addType(id.entry, type).

Come prima cosa possiamo procedere a costruire l'albero per l'input che ci è stato fornito:



Proviamo prima a creare una grammatica S-attribuita utilizzando anche la memoria oltre agli attributi. Partiremo dal basso e andremo verso l'alto:

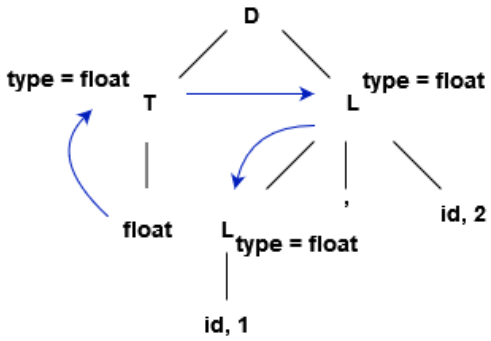
- Come prima cosa diamo a T un attributo type assegnando come valore int.
- Passiamo poi ad assegnare un attributo ad L, utilizzeremo come attributo un array; Per la produzione L → id avremo un array con un singolo valore in cui inseriremo id (inseriamo l'indice della tabella dei simboli), mentre per la produzione L → L_1 , id avremo un array con 2 valori, inseriamo l'array proveniente da L_1 e l'indice di id.
- Una volta ottenuti gli attributi di T e L , in D possiamo far lanciare la funzione addType().



Produzioni	Regole
D → T L	|| foreach el in L.arr : addType(el, T.type) ||
T → int	|| T.type = "int" ||
T → float	|| T.type = "float" ||
L → L_1 , id	|| L.arr = L_1.arr.add(id.entry) ||
L → id	|| L.arr.add(id.entry) ||

Abbiamo visto che siamo riusciti a creare regole sintetizzate quindi la grammatiche è S-attribuita. Ora proviamo a verificare se è possibile creare delle regole L-attribuite. Per le regole L-attribuite possiamo usare solo i singoli attributi, niente utilizzo di array. In questo caso come input abbiamo <float id, id>:

1. Partiamo dal nodo "float" e saliamo fino al nodo T, qui ricaviamo in modo sintetizzato il valore dell'attributo T.type = float;
2. Risaliti alla radice vogliamo far si che il type float passi da T a L, quindi ad L assegniamo un attributo type con valore float (da questo momento l'attributo type esiste per ogni L);
3. Scendiamo nell'albero nella produzione L → L_1 , id; qui per prima cosa L_1 erediterà il tipo dal padre; come seconda cosa L si occuperà di eseguire addType su < id, 2>;
4. Infine il figlio L_1 si occuperà di invocare addType su < id, 1>.



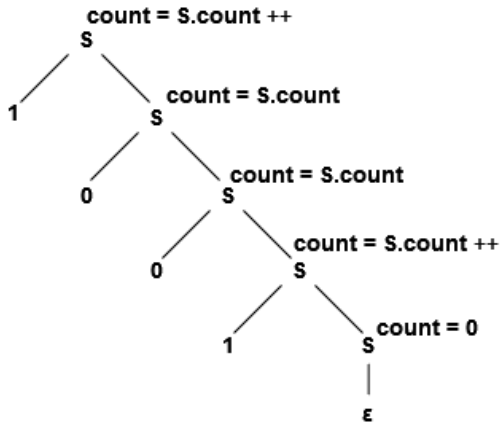
Produzioni	Regole
D → T L	|| L.type = T.type ||
T → int	|| T.type = "int" ||
T → float	|| T.type = "float" ||
L → L_1 , id	|| L_1.type = L.type ; addType(id.entry, L.type) ||
L → id	|| addType(id.entry, L.type) ||

Le regole le abbiamo definite secondo una definizione guidata dalla sintassi. Proviamo ora a definirle secondo uno schema di traduzione:

Schema di traduzione
D → T { L.type = T.type }
T → int { T.type = "int" }
T → float { T.type = "float" }
L → { L_1.type = L.type } L_1 , id { addType(id.entry, L.type) }
L → id { addType(id.entry, L.type) }

2) Creare una grammatica ad attributi tramite uno schema di traduzione che prende in input un numero binario e ne conti gli 1:
- Per prima cosa scriviamo una grammatica che legga i numeri binari (Il tipo di grammatica che si crea può facilitare o complicare l'esercizio):
S → 0 S
S → 1 S
S → ϵ

Per creare la grammatica ad attributi che ci conti gli uno possiamo procedere inizialmente sviluppando un albero per un input, ad esempio < 1 0 0 1>. Dall'albero notiamo che quando ci troviamo in una produzione con il valore 1 aggiorniamo il counter di S con il precedente incrementato di 1; mentre quando siamo in una produzione con 0 aggiorniamo il counter di S con quello precedente. Per ϵ impostiamo il counter a 0.

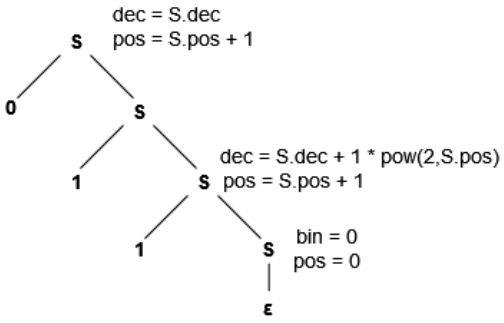


Schema di traduzione
S → 0 S_1 { S.count = S_1.count }
S → 1 S_1 { S.count = S_1.count++ }
S → ϵ { S.count = 0 }

3) Fare in modo che la grammatica precedente stavolta converti un numero binario in decimale.
Creando l'albero per l'input 011, vediamo che il valore meno significativo sta alla fine e che se eseguiamo in visita top down possiamo riuscire ad implementare l'algoritmo di conversione da binario a decimale con le potenze. Quindi supponiamo di disporre di una funzione pow per effettuare l'elevamento a potenze, e due attributi:

- dec : rappresenta il valore decimale:
- pos : rappresenta il nostro esponente ed aumenterà man mano che saliremo nell'albero.

Quindi quando siamo nel caso di S → ϵ avremo entrambi i due valori a 0. Nel caso in cui c'è 1 calcoliamo il valore dell'attributo S.dec come S_1.dec + 1 * pow(2, S_1.pos) e successivamente incrementiamo il valore dell'attributo di S.pos. Quando siamo del caso 0 il valore di S.bin viene soltanto copiato da S_1, mentre S.pos viene incrementato come nel caso precedente.



Schema di traduzione
S → 0 S_1 { S.dec = S_1.dec ; S.pos = S_1.pos++ }
S → 1 S_1 { S.dec = S_1.dec + 1 * pow(2, S_1.pos) ; S.pos = S_1.pos++ }
S → ϵ { S.dec = 0 ; S.pos = 0 }

3.7.6 Grammatiche ad attributi per costruire un albero

Implementeremo i nodi di un albero sintattico mediante oggetti dotati di un numero opportuno di campi. Ogni oggetto ha un campo op che costituisce l'etichetta del nodo. I nodi possono essere di due tipi:

- Foglia: se è un nodo foglia avrà un campo aggiuntivo per mantenere il valore dell'attributo. La foglia viene creata tramite la funzione: *Leaf(Token.entry, Token.value)*, l'invocazione di questa funzione restituirà un riferimento;
- Nodo: in questo caso si creerà un nuovo nodo che avrà tanti campi quanti sono i nodi figli del non terminale. Il nodo viene creato tramite la funzione: Node(op, ptr1, ptr2, ...);

Esempio: Data la seguente grammatica:

	Produzioni	Regole semantiche
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf(\mathbf{num}, \mathbf{num}.val)$

costruiamo l'albero sintattico per l'input: (x + 3) - 2.
I token lessicali saranno: (id + num_3) - num_2
Mostriamo la costruzione tramite l'uso dello stack.

1) Inizialmente leggiamo "(" e la inseriamo nello stack:

(	

2) Viene letto id e viene inserito nello stack con la sua entry (riferimento alla tabella dei simboli):

id	1
(	

3) Si controlla se è possibile effettuare una reduce, in questo caso è possibile, prima che venga eseguita la reduce viene prima eseguita la regola ad essa associata:

Creazione foglia:

id	1

viene restituito x come riferimento e sarà inserito nello stack come valore del non terminale inserito:

T	x
(	

Si procede così finchè non arriviamo alla radice.

4) Nuova riduzione di T in E:

E	x
(	

5) Non si può ridurre si aggiunge il simbolo più sullo stack, e saltando al passo successivo inseriamo anche il simbolo num con valore 3:

num	3
+	
E	x
(	

6) Effettuiamo una riduzione di num in T, creando prima la foglia, e inserendo poi nello stack il non terminale T con il riferimento alla foglia:

id	1

il riferimento sarà y;

T	y
+	
E	x
(	

7) Riduzione di E + T in E, in questo caso viene creato un nuovo nodo contenente come etichetta l'operazione '+', e i riferimenti ai suoi due figli E e T; il riferimento del nuovo nodo sarà inserito nello stack con E.

+	x	y

come riferimento avremo z.

E	z
(	

8) Inseriamo il simbolo) nello stack:

)	
E	z
(	

9) Effettuiamo una riduzione di (E) in T, in questo caso come possiamo osservare dalla regola copieremo il nodo, quindi copiamo semplicemente il riferimento:

T	z

10) Inseriamo il simbolo -, e il token num con valore 2:

num	2
-	
E	z

11) Effettuiamo una riduzione di num in T ed assegniamo il riferimento della foglia creata a T:

num	2

ha come rifermento t;

T	t
-	
E	z

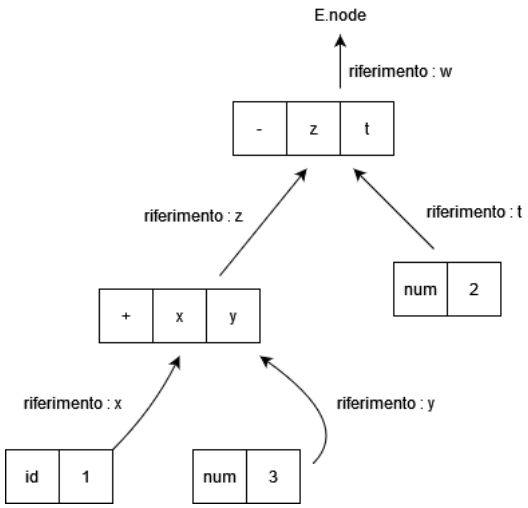
12) Ci comportiamo come per il punto 7:

-	z	t

come riferimento avremo w.

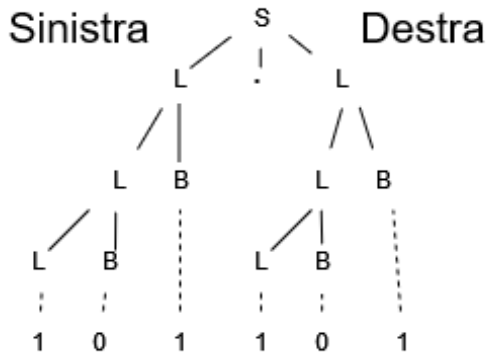
E	w

Siamo arrivati alla radice, l'albero sintattico è costruito.



Esercizio: Schema di traduzione per convertire un binario in virgola mobile in decimale

La grammatica è :
S → L . L
L → L B
L → B
B → 0
B → 1
L'albero che otteniamo esegue l'input 101.101 è il seguente:



Possiamo procedere come segue:

1. Per creare distinzione tra la parte sinistra e destra del numero binario possiamo usare un attributo che indichi la posizione Sinistra (S) o Destra (D) del non terminale L e tramite un if decidere che operazione effettuare, ovviamente questo attributo deve essere dato ai figli.
2. Dalla figura dell'albero possiamo osservare che è sbilanciato verso sinistra. Quindi possiamo applicare il seguente algoritmo:
   - Per la parte sinistra: abbiamo rispettivamente $2^2 - 2^1 - 2^0$, possiamo raggruppare le operazioni da effettuare a coppia di 2 ed ad ogni livello moltiplicare per 2

NOTA: JavaCup può usare precedenze ed associazioni per creare parser per grammatiche non LALR(1) ma che si prestano

Capitolo 4

Analisi semantica

Durante lo sviluppo di un compilatore una parte importante è capire il compilatore come deve prendere le decisioni:

- Se è il compilatore a dover prendere decisioni viene detto che ha una politica statica o a tempo di compilazione;
- Se invece si possono prendere decisioni quando il programma è eseguito viene detto che ha una politica dinamica o a run-time.

4.1 Scoping

Un altro aspetto fondamentale è lo scoping cioè l'ambiente o il blocco di codice dove una variabile è definita e pronto all'utilizzo. Se per un determinato linguaggio per capire lo scoping di una variabile basta analizzare il programma allora si parla di **scoping statico**. Si parla di **scoping dinamico** invece, se si deve aspettare che il programma sia in esecuzione questo perchè una variabile può essere usata in blocchi di codice diversi e quindi può cambiare valore.

4.1.1 Scoping statico e struttura a blocchi

Il linguaggio C e i suoi derivati prevedono lo scoping statico. In C lo scope di una variabile è determinato automaticamente in base alla posizione in cui appare, in linguaggi come Java il programmatore può dichiarare lo scope di una variabile ad esempio tramite public, private, ecc.
Nello scoping statico si usa il concetto di blocco; un blocco è un raggruppamento di dichiarazioni e istruzioni. Per delimitare un blocco il linguaggio C utilizza le parentesi graffe { e }; altri linguaggi, a partire dall' Algol, utilizzano le parole chiave begin ed end.
In C la sintassi di un blocco e definita dalle regole seguenti.

1. Un blocco è un tipo di statement. Un blocco può apparire ovunque sia ammesso un qualsiasi altro tipo di statement.
2. Un blocco è costituito da una sequenza di dichiarazioni seguita da una sequenza di statement, tutte racchiuse tra parentesi graffe.

Si noti che una tale descrizione sintattica permette l'annidamento di blocchi. Questa proprieta di annidamento prende il nome di struttura a blocchi. I linguaggi della famiglia del C hanno una struttura a blocchi, con un'unica limitazione: una funzione non può essere definita all'interno di un'altra funzione.

In informatica, ed in particolare in programmazione, lo **shadowing** è la regola di visibilità secondo la quale una variabile locale "nasconde", all'interno di un blocco, una variabile con lo stesso nome definita nel blocco superiore.
Formalmente definiamo lo scope di una variabile in un **blocco** come segue: Se la dichiarazione D di un nome x appartiene al blocco B allora lo scope di D è tutto il blocco B tranne ogni blocco B' annidato in B a qualunque livello di profondità in cui il nome x venga ridefinito. Si dice che x è ridefinito in B' se esiste in B' una nuova dichiarazione D' dello stesso nome x.

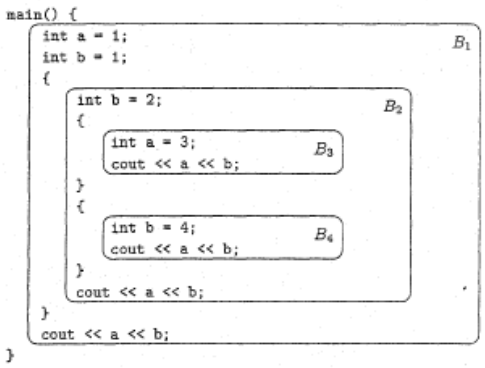


Figura 1.10 Blocchi in un programma C++.

Dichiarazione	Scope
int a = 1;	$B_1 - B_3$
int b = 1;	$B_1 - B_2$
int b = 2;	$B_2 - B_4$
int a = 3;	B_3
int b = 4;	B_4

Figura 1.11 Scope delle dichiarazioni dell'Esempio 1.6.

4.2 Tabella dei simboli

Le tabelle dei simboli sono strutture dati utilizzate dai compilatori per mantenere informazioni relative ai costrutti del linguaggio sorgente. Tali informazioni sono raccolte in modo incrementale durante le fasi di analisi del processo di compilazione e quindi utilizzate nelle fasi di sintesi, per la generazione del codice destinazione. Un elemento di una tabella dei simboli raccoglie informazioni relative a un identificatore: la stringa che lo rappresenta, cioè il lessema, il tipo, la sua posizione in memoria, e qualsiasi altro data rilevante. Tali tabelle devono in genere essere in grado di gestire dichiarazioni multiple dello stesso identificatore in un programma. Questo è possibile realizzando una tabella dei simboli per ogni singolo scope individuato.

4.2.1 Tabella dei simboli e scoping

Nell'implementazione di un compilatore lo scoping viene implementato realizzando una tabella dei simboli per ogni blocco individuato. Normalmente questo scoping è implementato tramite l'uso di uno stack:

1. All'inizio di un nuovo blocco si crea una nuova tabella dei simboli;
2. Tutte le volte che si incontra una variabile si controlla se è presenta nella tabella dei simboli del blocco o nelle altre tabelle presenti nello stack.
3. Quando si definisce una nuova variabile la si aggiunge nella tabella dei simboli del blocco.
4. Quando si incontra un nuovo blocco si crea una nuova tabella dei simboli e viene pushata nello stack. Questo è il caso di blocchi annidati.
5. Alla fine del blocco si effettua il pop della tabella.

La prima cosa da fare durante l'analisi semantica è quindi quella di cercare quali costrutti costruiscono una nuova tabella dei simboli. Nel nostro compilatore eseguiremo due visite una per costruire la tabella dei simboli (dall'alto verso il basso) e poi una visita dal basso verso l'alto per controllare i tipi.
Prende il nome di **Type environment** l'insieme di tutte le tabelle dei tipi su una path o catena partendo da un blocco e sono utili per ricavare il tipo di una variabile, funzione, ecc.. utilizzata in un blocco.
Se una variabile, funzione, ecc.. è utilizzata in un blocco ma non è presente nel type environent di quel blocco allora è fuori dallo scope del blocco e non può essere utilizzata.
In Java avviene tramite delle tabelle concatenate, ed assumono una struttura ad albero.

Esempio di scoping errato in java:

```
class X{
    public static void main(String args[]){
        int a = 2;
        {
            int a = 3;
        }
    }
}
```

Java non permette lo shadowing di una variabile dichiarata all'interno di un blocco statico. Ma se invece non è dichiarata al'interno di un blocco statico lo permette:

```
class X{
    public static void main(String args[]){
        int a = 2;
        {
            int a = 3;
        }
    }
}
```

Nota: Java non permette ai blocchi all'interno di un metodo di effettuare lo shadowing.

4.3 Type Checking

Per poter effettuare il controllo dei tipi, un compilatore deve prima assegnare un tipo ad un'espressione, poi procede verificare che tali espressioni siano conformi rispetto a un insieme di regole logiche comunemente detto **type system** di un linguaggio. Tutti i controlli di tipo possono essere effettuati a run-time a patto che si conservi il valore di ogni elemento. Un complitare efficiente però effettua questo controllo al tempo di compilazione. Se un compilatore permette l'esecuzione di programmi senza errori di tipo si dice che l'implementazione del linguaggio è fortemente tipizzata.
In Java questo controllo è effettuato anche per evitare che il bytecode prodotto effettui azioni malevole.

Il controllo dei tipi può assumere due forme:

- **Sintesi**: prevede la costruzione di un tipo di un'espressione a partire dal tipo delle sue sottoespressioni. Tale approccio richiede che tutti i nomi siano dichiarati prima di poter essere utilizzati.
- **Inferenza**: determina il tipo di un costrutto del linguaggio in base al modo in cui esso è utilizzato, cioè un linguaggio anche se è fortemente tipizzato, non richiede di dichiarare i nomi prima dell'uso.

4.3.1 Regole d'inferenze di tipo

I simboli usati per le regole d'inferenza di tipo sono i seguenti:

- $\wedge$ oppure uno spazio vuoto: è l'and logico;
- $=>$ oppure $\frac{\quad}{\quad}$: significano allora;
- x: T oppure x : τ: x ha tipo T;
- `oppure $\vdash$: è possibile provare che;
- O oppure Γ: indica il type environment e si legge nel contesto nell'ambiente.

- Esempio:

$$\frac{(x:\tau) \in \Gamma \wedge \Gamma \vdash e:\tau}{\Gamma \vdash x = e} \tag{4.1}$$

La lettera greca "Gamma" indica il contesto in cui vale il vincolo che andiamo a creare, mentre la lettera "tau" indica qual è il tipo che deve essere rispettato dalla variabile e.
La regola sopra riportata si applica all'operazione di assegnazione, infatti i vincoli posti sono i seguenti:

- x di tipo "tau" appartiene al contesto "Gamma"
- la variabile e nel contesto "Gamma" è di tipo "tau".

- Esempio sulla somma :

- $\frac{5{:}intero}{`5{:}integer}$
- $\frac{\frac{5{:}intero}{`5{:}integer} \; \frac{2{:}intero}{`2{:}integer}}{`5{+}2{:}integer}$

- Esempio assegnazione:

$$\frac{\Gamma(id) = \tau \qquad \Gamma \vdash e:\tau}{\Gamma \vdash id := e : \text{nottype}} \tag{4.2}$$

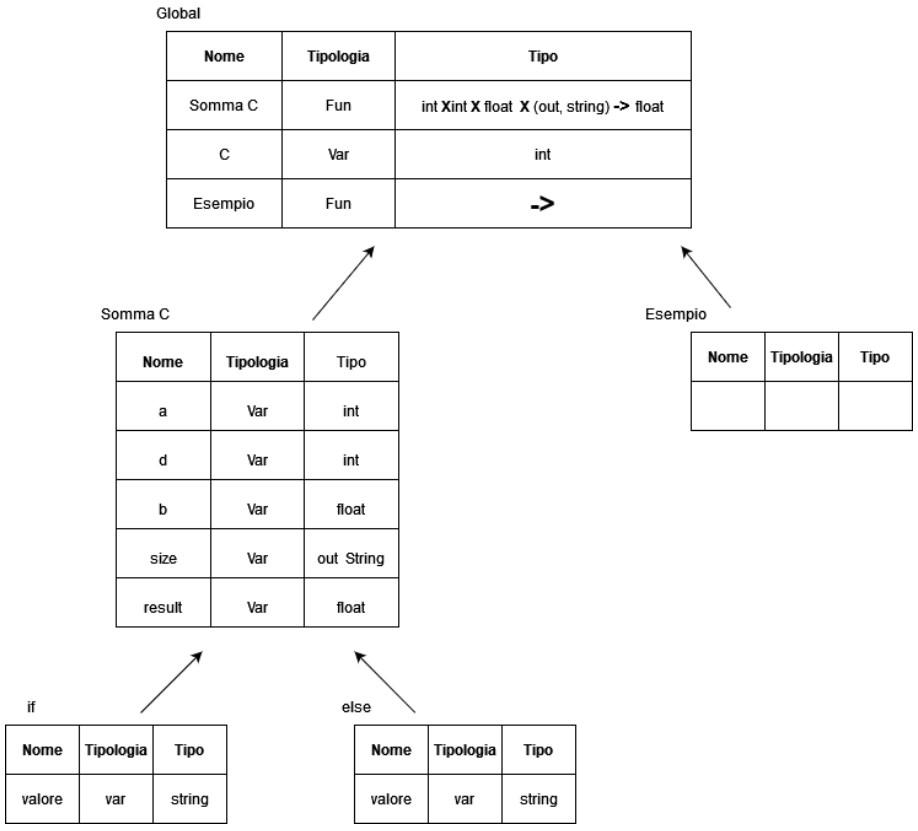
Esistono molte regole d'inferenza ognuna diversa in base al linguaggio che si vuole implementare. L'insieme di queste regole prende il nome di **type system**.
Se per ogni operazione del linguaggio c'è una regola d'inferenza allora si dice cha abbiamo un sound type system.

NOTA: Nell'analisi semantica è costruita la vera e propria tabella dei simboli, infatti fino ad ora è stata sempre utilizzata la tabella delle stringhe.

4.3.2 Esercizio sulla creazione di un type environment

Dato il seguente codice creare il type environment:

```
def sommac(integer a, d | real b | out string size): real {
        real result;

        result  <<  a + b + c + d;

        if result > 100 then{
                var valore << 'grande';
                size << valore; }
        else {
                var valore << 'piccola';
                size << valore;
        }

        return result;
}

var c << 1;

start:
def esempio(){}
```



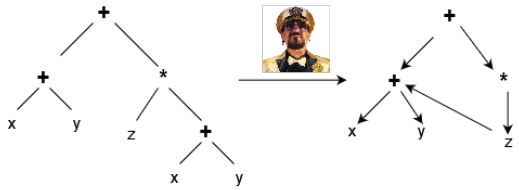
Capitolo 5

Generazione codice intermedio

Questa fase prende in input l'albero semantico e la tabella dei simboli. Viene dato in output il codice a tre indirizzi inserito all'interno di un array.
Durante la creazione di un compilatore molte volte si pensa già a questa fase durante la costruzione dell'albero sintattico, infatti molte volte invece di un albero viene costruito un DAG cosicché si possano riutilizzare gli stessi nodi e quindi durante la generazione del codice generare del codice già ottimizzato in minima parte.
Esempio: Data la seguente espressione x + y + z * x + y.
Come prima cosa dobbiamo costruire l'albero per farlo iniziamo ad inserire le parentesi in questa espressione:

1. x + y + z * (x + y)
2. x + y + (z * (x + y))
3. (x + y)+ (z * (x + y))

Successivamente facciamo l'albero sintattico e vediamo se è possobile ottimizzarlo in un dag:



5.1 Struttura del codice a tre indirizzi

Un indirizzo può assumere tre forme:

- Può avere un nome, per comodità viene usato il nome delle variabili nel codice, nella realtà si inseriscono dei puntatori;
- Può assumere il valore di una costante;
- Può assumere un nome temporaneo generato dal compilatore.

Le istruzioni più comuni generate dal codice a tre indirizzi sono:

- x = y op z;
- x = op z;
- x = y;
- goto L (dove L è un etichetta);
- if x goto L;
- if False x goto L;

- if x relop y goto L;
- chiamete a funzione del tipo:

  x_1

  x_2

  ...

  x_n

  call p,n

- x = y[i] o y[i] = x dove i è riferito non alla seconda cella dell'array ma al byte, quindi per accedere alla seconda cella se è un intero bisogna fare i+4;
- x = *y;
- x = &y.

NOTA: Un codice a tre indirizzi deve usare sempre e solo indirizzi.
Esercizio: dato il codice:

```
do i = i +1; while (a[i] > v);
```

Generare il codice a tre indirizzi:

```
L : i = i+1
    t = i *8
    t2 = a[t]
    if t2 > v goto L
```

Un altro modo è quello di usare un array d'istruzioni.
NOTA: nelle prossime implementazioni l'istruzione if sarà tralasciata perchè parole del prof: Il libro non la mostra e quindi all'esame non ci sarà.

100	i = + 1
101	t = i *8
102	t2 = a[t]
103	if t2 > v goto L

Esistono ben 4 modi per implementare questo array.

Array di quadruple

È una tabella composta di quattro colonne:

- La prima contiene l'operatore;
- La seconda il primo argomento;
- La terza il secondo argomento;
- La quarta contiene il risultato

0	+	i	1	t1
1		t1		i
2	*	i	8	t2
3	[ ]	a	t2	t3

Array di triple

Elimina la colonna del risultato perché quest'ultima è una ripetizione, infatti si potrebbe utilizzare direttamente l'indirizzo della cella di memoria:

0	+	i	1
1		(0)	
2	*	(1)	8
3	[ ]	a	(2)

Utilizzando un array di triple sorge un problema però durante la fase di ottimizzazione del codice l'algoritmo utilizzato potrebbe effettuare degli swap e questo potrebbe portare ad avere risultati diversi nelle celle di memoria. Come risolvere?

Array di triple indirette

Utilizza sempre un array di triple, ma anche un vettore in cui è contenuto l'ordine delle operazioni da eseguire cosicché anche se si effettua uno swap questo avviene solo sull'ordine delle operazioni e non anche sul risultato delle celle di memoria.

0	+	i	1
1		(0)	
2	*	(1)	8
3	[ ]	a	(2)

100	(0)
101	(1)
102	(2)
103	(3)

SSA (Static Single-Assignment

Pone dei vincoli su come deve essere costruito il codice a tre indirizzi:

- Se si assegna un valore ad una variabile questa non deve aver avuto già un valore assegnato precedentemente:

```
p = a +b
q = p - c
p = q * d
p = e - p
q = p + a
```

  Non rispetta la forma SSA. Trasformiamola nella forma SSA cosicché l'ottimizzazione del codice migliori. Per farlo basta assegnare degli indici ad ogni occorrenza di p e q:

```
p1 = a +b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + a
```

- Risolve il problema del valore di una variabile contenuta in un if e in un else:

```
if(flag) x = -1
else x = 1
y = x +a
```

  Non sappiamo infatti y quale valore possa assumere, cambia in base a se il flag sia vero o falso. SSA risolve il problema portandosi dietro l'inferenza di tipo con $\phi(x_1, x_2$, così il valore dipende dal flusso di esecuzione:

```
if(flag) x1 = -1
else x2 = 1
x3 = φ(x1,x2)
y = x3 +a
```

5.1.1 Grammatica ad attributi per generare codice a tre indirizzi

Vogliamo generare il codice a tre indirizzi per la seguente espressione: x = x + y * z.
Scriviamo prima la grammatica per riconoscere l'espressione:
S → id = E
E → E + E
E → E * E
E → - E
E → (E)
E → id

Vogliamo quindi ora che venga prodotto il seguente codice a tre indirizzi:

```
t1 = y * z
t2 = x + t1
t3 = t2
```

Supponiamo di avere le seguenti funzioni:

- Gen: che genera il codice intermedio;
- Temp() che permette di creare una nuova variabile temporanea;
- Top punta al top del type environment;
- Top.get() prende l'indirizzo di memoria di una variabile;
- E.addr: è l'attributo che contiene l'indirizzo di memoria.

S → id = E	{ gen(Top.get(id.lessema) = E.addr) }
E → E_1 + E_2	{ E.addr = new Temp(); gen(E.addr = E_1.addr + E_2.addr) }
E → E_1 * E_2	{E.addr = new Temp(); gen(E.addr = E_1.addr * E_2.addr) }
E → - E_1	{E.addr = new Temp(); gen(E.addr = - E_1.addr) }
E → (E_1)	{ E.addr = E_1.addr }
E → id	{ E.addr = Top.get(id.lessema) }

Traduzione delle espressioni booleane

Le espressioni booleane sono molto utili per modificare il flusso di un if-then-else o while, oppure per calcolare valori logici.

Quando si decide di tradurre un espressione booleana si può procedere in due modi:

- Eseguirla tutta;
- Utilizzare uno short circuit, cioè inserire dei salti incondizionati in modo da non vedere tutta l'espressione, Ad esempio con uno short circuit l'operazione B || B, se la prima B è true si potrebbe direttamente saltare alla prossima istruzione, invece senza short circuit si è obbligati ad osservare anche il valore della seconda B.

Una prima traduzione delle espressioni booleane è quella di rappresentarle come istruzioni per il controllo del flusso:
if (x < 100 || x > 200 && x != y) x = 0;
Possiamo scriverla come:

```
    if x < 100 go to L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2: x = 0
L1
```

Il codice sopra riportato rappresenta un'implementazione di uno shortcircuit.

Sorge un problema in questo caso noi sapevamo già dove le goto dovessero saltare, ma un compilatore nel momento della generazione del codice lascia un buco: "goto_". Per riempire questo buco e capire quale indirizzo dare alla goto, si utilizza la tecnica del **backpatching**.

5.2 Backpatching

Risolve il problema di quale etichetta assegnare a una goto in un'istruzione di controllo di flusso.
Ad esempio l'espressione booleana B nell'istruzione "if (B) S" nel caso in cui B risulti falsa contiene un salto alla prima istruzione che segue il codice di S.
La tecnica del backpatching viene utilizzata tramite l'ausilio di attributi, useremo due attributi sintetizzati: truelist e falselist del non terminale B:

- B.truelist è una lista di indirizzi di istruzioni che hanno un goto vuoto, si inserisce l'etichetta in cui saltare in caso di condizione B vera.
- B.falselist è una lista di indirizzi di istruzioni che hanno un goto vuoto, si inserisce l'etichetta in cui saltare in caso di condizione B falsa.

Supporremo di inserire le istruzioni generate in un array e di utilizzare l'indice di un'istruzione come sua etichetta.
Definiamo anche altre tre istruzioni che useremo per manipolare i salti:

1. makelist(i): crea una nuova lista contente solamente l'indice i di un elemento di un'istruzione dell'array e restituisce un puntatore alla lista appena creata;
2. merge(p_1, p_2): concatena le liste puntate da p_1 e p_2 e restituisce un puntatore alla lista risultante;
3. backpatch(p,i): inserisce i come etichetta di destinazione in ognuna delle istruzioni nella lista puntata da p;
4. emit(goto_): utilizzata per inserire un salto incondizionato.

5.2.1 Backpatching sulle espressioni booleane

```
1) B → B1 || M B2        { backpatch(B1.falselist, M.instr);
                           B.truelist = merge(B1.truelist, B2.truelist);
                           B.falselist = B2.falselist;
                         }
```

Se B_1 è vera le goto che si trovano nella sua truelist diventano parte della truelist di B. Se B_1 è falsa dobbiamo valutare anche B_2; quindi le goto nella falselist di B_1 devono puntare all'inizio di B_2, tale etichetta è ottenuta grazie al non terminale M. M è un non terminale con produzione epsilon, anche detto **marker**, è utilizzato come indicatore di prossima istruzione booleana. Grazie a lui quindi sappiamo dove inizia B_2, e tramite backpatching riempiamo le goto di B_1.falselist.

```
2) M → ϵ                 {M.instr = nextrinstr;}

3) B → B1 && M B2        { backpatch(B1.truelist, M.instr);
                           B.falselist = merge(B1.falselist, B2.falselist);
                           B.truelist = B2.truelist;
                         }

4) B → !B1               { B.falselist = B1.truelist;
                           B.truelist = B1.falselist;
                         }

5) B → (B1)              { B.truelist = B1.truelist;
                           B.falselist = B1.falselist;
                         }
```

```
6) B → E1 rel E2          { B.truelist = makelist(nextrinstr);
                            B.falselist = makelist(nextrinstr);
                            emit("if" E1.addr, rel.op E2.addr 'goto_');
                            emit('goto_');
                          }
```

emit è utilizzata per generare del codice a tre indirizzi, nel nostro caso un if e una goto senza etichetta, sarà riempita successivamente tramite l'uso della tecnica di backpatching da altre produzioni.

```
7) B → true               { B.truelist = makelist(nextrinstr);
                            emit('goto_');
                          }

8) B → false              { B.falselist = makelist(nextrinstr);
                            emit('goto_');
                          }
```

5.2.2 Backpatching su statement per il controllo del flusso

```
1) S → if (B) then M S1         { backpatch(B.truelist, M.instr);
                                  S.nextlist = merge(B.falselist, S1.nextlist);
                                }

1) S → if (B) then M1 S1 N else M2 S2
                          { backpatch(B.truelist, M1.instr);
                            backpatch(B.falselist, M2.instr);
                            S.nextlist = merge (S1.nextlist, N.nextlist, S2.nextlist);
                          }
```

Perchè abbiamo aggiunto il non terminale N? N è un non terminale che avrà una produzione che punta a epsilon, quello che farà è creare una lista ed emettere una goto senza etichetta "goto_". È utilizzata perchè S_1 può terminare il suo ciclo in due modi:

- Termina perché effettua un salto tramite una goto: le etichette delle goto le ritroviamo in S_1.nextlist;
- Termina per morte naturale: cioè esegue tutte le operazioni senza effettuare nessuna goto; questo però porta ad un problema, quando termina come prossima istruzione si trova l'else ma non esiste nessun caso in cui un if vero una volta terminato entra in un else. N è usato quindi proprio per evitare questo, S_1 vedrà come prossima istruziona quella di N e poi si bloccherà perchè ci sarà una goto senza etichetta che successivamente sarò riempità con l'etichetta di salto dopo l'else.

```
2) N → ϵ                  { M.nextlist = makelist(nextrinstr);
                            emit('goto_');
                          }

3) S → while (M1 B) do M2 S1
                          { backpatch(S1.nextlist, M1.instr);
                            backpatch(B.truelist, M2.instr);
                            S.nextlist = B.falselist;
                            emit('goto M1_instr');
                          }
```

emit('goto M_1_instr') è utilizzata per far si che quando S_1 termina per "morte naturale" non esca fuori dal while ma salti alla prossima istruzione puntata da M_1 e cioè B.
Ora mostriamo il do while in due modi: esce quando il while è falso ed esce quando il while è vero.
While esce con falso

```
4) S → do M1 S1 while (M2 B)
                          {
                            backpatch(B.truelist, M1.instr);
                            backpatch(S1.nextlist, M2.instr);;
                            S.nextlist = B.falselist;
                          }
```

While esce con vero

```
5) S → do M1 S1 while (M2 B)
                          {
                            backpatch(B.falselist, M1.instr);
                            backpatch(S1.nextlist, M2.instr);;
                            S.nextlist = B.truelist;
                          }

6) S → ( L )              { S.nextlist = L.nextlist; }

7) S → A                  { S.nextlist = null; }

8) L → L1 M S             { backpatch(L1.nextlist, M.instr);
                            S.nextlist = L.nextlist;
                          }

9) L → S                  { L.nextlist = S.nextlist; }
```

Ciclo for (soluzione proposta da Carmine D'Angelo)

```
6) S → for ( init; M1 B; M2 Step ) M3 S1 N
                { temp = merge(S1.nextlist, N.nextlist);
                  backpatch(t.nextlist, M2.instr);
                  backpatch(B.truelist, M3.instr);
                  Step.instr = M1.instr;
                  S.nextlist = B.falselist;
                }
```

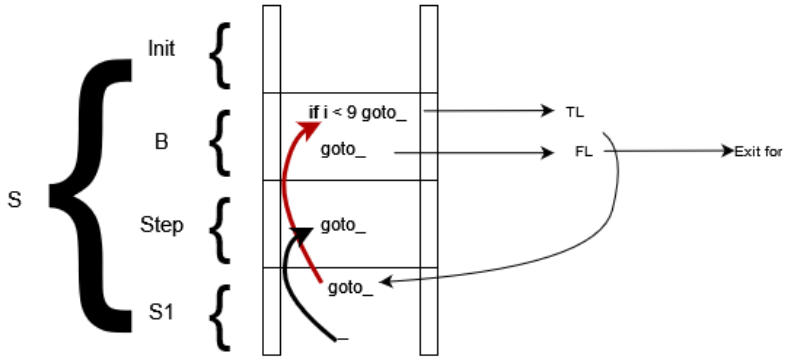


Figura 5.1: Funzionamento logico del for

Capitolo 6

Generazione codice macchina

Il compilatore che stiamo costruendo all'interno di questo corso prende il nome di compilatore a due passi e porta alla creazione di compilatori multipiattaforma e multilinguaggi.
Infatti ad oggi si fissa un codice intermedio e lo si rende noto a chi scrive il front-end ed a chi scrive il back-end di un compilatore:

- Chi scrive il front end? Chi ha un linguaggio di programmazione (sorgente) da implementare. Non deve preoccuparsi delle specifiche del linguaggio macchina target
- Chi scrive il back end? Chi ha un'architettura hardware su cui vuole far eseguire programmi. Non deve preoccuparsi delle specifiche del linguaggio di programmazione sorgente.

6.1 Codici intermedi

Un codice intermendio è LLVM IR è un'infrastruttura di compilazione, scritta in C++, progettata per l'ottimizzazione di programmi in fase di compilazione, di linking, di esecuzione e di non utilizzo, genera codice macchina per varie architetture. Di seguito alcuni linguaggi che compilano in LLVM IR: C/CLang, C++, Swift, Rust, Kotlin nativo, Objective C, Ada, Julia e Fortran.

6.2 Infrastrutture ibride

Il back-end è eliminato dal compilatore ed è sostituito da una macchina virtuale (interprete software) resa più efficiente da un JIT compiler. In pratica un JIT compiler implementa parti del back-end che però in questo caso sono eseguite durante l'esecuzione del codice intermedio (a run-time). La macchina virtuale ed il suo jit compiler sono sempre specifici di un'architettura hardware (come d'altronde il back-end). Quelle più note sono Java e .NET, che presentano rispettivamente i seguenti codici intermedi:

- Bytecode: eseguito da Java Virtual Machine (JVM) che contiene un JIT compiler. Linguaggi che compilano in bytecode: Java, Python, Kotlin, Clojure, Groovy, Scala, etc.
- Common Intermediate Language (CIL): parte della infrastruttura .NET. Eseguito dal CLR che contiene un JIT compiler. Linguaggi che compilano in CIL: C#, F#, Visual Basic, C++, etc.
- Dart platform: permette di compialre il codice in due modi:
  - Nativo: applicazioni mobile o desktop, vegono compilate tramite la Dart VM con un compilatore JIT e un AOT per produrre codice macchina;
  - Web platform: per app web, include un compilatore a tempo di sviluppo (dartdevc) e un compilatore a tempo di produzione (dart2js). Entrambi i compilatori traducono dart in js.

Le infrastrutture ibride sono molto utili perchè si possono sviluppare varie parti di un software in linguaggi diversi ed unirle, questo funziona perchè vengono tutti tradotti nello stesso linguaggio.

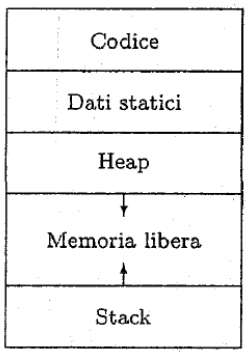
NOTA AOT sta per Ahead-Of-Time (oppure Attack On Titan) ed è rappresentata dai normali compilatori che generano codice macchina, i sono stati categorizzati per via della nascita di nuove tipologie di compilatori.

6.3 Run-time environments

Un compilatore deve implementare in modo accurato le astrazioni definite dalla semantica del linguaggio. Inoltre, esso deve cooperare con il sistema operativo e con altri software di sistema al fine di realizzare tali astrazioni sulla macchina target.
A questo scopo il compilatore crea e gestisce un supporto run-time o ambiente runtime (runtime environment) dove assume che i programmi siano eseguiti. Questo ambiente riguarda una varietà di aspetti tra cui l'organizzazione e l'allocazione della memoria richiesta dagli oggetti che appaiono nel programma, il meccanismo utilizzato dal programma target per accedere alle variabili, i collegamenti fra le procedure, il meccanismo di passaggio dei parametri e l'interfacciamento con il sistema operativo, con i dispositivi di I/0 e con gli altri programmi.
Esempio di run-time environments: il garbage collector di java.

6.3.1 Organizzazione della memoria

Dalla prospettiva dello sviluppatore di compilatori, il programma target e in esecuzione in un suo spazio d'indirizzamento logico in cui ad ogni valore è associato un indirizzo.
Nel nostro corso assumeremo sempre che la memoria a run-time appaia costituita da una sequenza di blocchi di byte contigui e che il byte sia la piu piccola porzione di memoria indirizzabile.
Assumeremo che un byte sia formato da 8 bit e che 4 byte formino una parola o word della macchina e, inoltre, che oggetti di dimensione maggiore del byte siano memorizzati in locazioni consecutive e sia stato assegnato loro l'indirizzo del primo byte. Se non si occupa tutto lo spazio viene aggiunto un padding.
Un tipo di base, per esempio un carattere, un intero o un valore in virgola mobile è sempre memorizzato in un numero intero di byte consecutivi. Un tipo composto, per esempio un array o una struttura, avrà una dimensione tale da consentire la memorizzazione di tutti i suoi elementi.
Un compilatore per un linguaggio come il C/C++ con un sistema operativo come Linux suddivide la memoria in questa modo:



- **Codice**: è dove viene inserito il codice eseguibile del programma, la dimensione è determinata a tempo di compilazione. Si conosce all'interno dello stack l'indirizzo della prima zona libera;
- **Dati statici/memoria statica**: In alcuni casi la dimensione dei dati di un programma potrebbe già essere nota nel momento della compilazione (variabili globali, ecc.), in questo caso i dati potrebbero essere inseriti in questa memoria definita staticamente. Una ragione per fare ciò è che l'indirizzo di tali oggetti, essendo noto al momento della compilazione, può essere direttamente inserito nel codice target. Inizialmente in Fortran tutti i tipi di dati potevano essere allocati staticamente. Avendo dimensione statica si può utilizzare un offset per spostarsi al suo interno.

Stack e heap sono posizionati agli estremi dello spazio rimanente per massimizzare il loro utilizzo. Possono essere disposti in due modi Stack che cresce verso il basso e Heap verso l'alto o viceversa.

- **Stack**: è utilizzato per memorizzare strutture dati dette **record d'attivazione** (activation record) che vengono generati durante le chiamate di procedura.Un registro noto come stack pointer (SP) punta alla cima dello stack. *Il libro è un confuso del cazzo e anche se sa che normalmente lo stack cresce verso gli indirizzi bassi qua lo presenta che cresce verso gli indirizzi alti.* Avendo una dimensione dinamica per

spostarsi all'interno dello stack si utilizzano i **Frame pointer**. I frame pointer sono salvati all'interno di un registro;

- Heap: è uno spazio di memoria che possiamo allocare e deallocare. Questa allocazione di memoria non è sicura a causa della sua accessibilità o visibilità dei dati archiviati a tutti i thread. Il tempo di accesso o di elaborazione è piuttosto lento rispetto alla memoria dello stack. L'allocazione della memoria heap può essere suddivisa in:
  - Young generation: in questa parte della memoria, tutti i nuovi oggetti o dati allocano lo spazio. In java nel caso in cui la memoria sia piena il garbage collector aiuta a memorizzare il resto dei dati.
  - Old or tenured generation: in questa parte sono archiviati i dati più vecchi che non vengono spesso utilizzati o non vengono utilizzati affatto.
  - Permanent generation: questa parte è costituita dai metadati di JVM per i metodi dell'applicazione e le classi di runtime.

  Esistono diversi vantaggi nell'utilizzo della memoria heap, ad esempio è possibile accedere alle variabili a livello globale, nessun limite alla dimensione della memoria. Gli svantaggi riguardano le tempistiche elevate dell'esecuzione, la gestione della memoria risulta più complicata e richiede più tempo per il calcolo. Infatti, in C allocare con malloc e non deallocare mai può portare ad una saturazione della memoria

Stack e Heap crescono finchè c'è memoria libera o non si toccano.

Stack vs Heap

1. Lo stack è utile per l'allocazione della memoria statica, quindi di conseguenza prima che il programma venga eseguito, la memoria viene allocata in fase di compilazione. Considerando che l'heap è vantaggioso per l'allocazione dinamica della memoria, ciò implica che la memoria può essere liberata e allocata in ordine casuale.
2. In termini di disposizione, l'ordine dello stack è last-in-first-out e le variabili nella memoria del computer vengono memorizzate direttamente. Mentre in heap la memoria non viene gestita automaticamente, ma manualmente.
3. Nel contesto thread-safety, lo stack thread ha il suo stack, ed è per questo che risulta thread-safe. Il rovescio della medaglia è che l'heap non è sicuro per il threading a causa della necessità di un codice di sincronizzazione appropriato.
4. La memoria stack è utile per memorizzare chiamate di funzione e variabili locali. Ma la memoria heap è utile per archiviare oggetti in Java. Non importa in quale sezione di codice viene creato l'oggetto, verrà sempre creato all'interno dell'heap spaziale in Java.
5. Negli stack, le variabili memorizzate sono visibili al thread del proprietario o, in Java, è una sorta di memoria privata. D'altro canto, negli oggetti heap creati e resi visibili a tutti i thread, la memoria heap è condivisa.

6.3.2 Allocazione statica e dinamica

L'organizzazione e l'allocazione dei dati in specifiche locazioni di memoria a run-time sono due aspetti fondamentali nell'ambito della gestione della memoria. Si tratta di problemi complessi e delicati, poiché uno stesso nome nel codice sorgente di un programma può fare riferimento a più locazioni di memoria durante l'esecuzione. I due aggettivi statica e dinamica si riferiscono alla gestione della memoria rispettivamente a compile-time e a run-time:

- **Allocazione statica**: è l'allocazione della memoria in fase di compilazione, prima che il programma associato venga eseguito. Nell'allocazione della memoria statica, la dimensione dei dati richiesti dal processo deve essere nota prima che venga avviata l'esecuzione del processo. Se le dimensioni dei dati non sono note prima dell'esecuzione del processo, è necessario indovinarle. Se la dimensione dei dati calcolata è maggiore di quella richiesta, si verifica uno spreco di memoria. Se la dimensione ipotizzata è inferiore, porta a un'esecuzione inappropriata del processo. Nell'allocazione della memoria statica, una volta allocate le variabili rimangono permanenti. Dopo l'assegnazione iniziale, il programmatore non può ridimensionare la memoria.
- **Allocazione dinamica**: si riferisce alla gestione della memoria di sistema in fase di esecuzione. In questo processo la memoria viene allocata alle entità del programma quando devono essere utilizzate per la prima volta mentre il programma è in esecuzione. La dimensione effettiva dei dati richiesti è nota in fase di esecuzione e quindi il processo alloca lo spazio di memoria esatto al programma riducendo così lo spreco di memoria, un fattore che migliora le prestazioni del sistema. L'allocazione dinamica della memoria di solito crea un sovraccarico sul sistema. Alcune operazioni di allocazione vengono eseguite ripetutamente durante l'esecuzione del programma creando più overhead con conseguente esecuzione lenta del programma. L'allocazione dinamica della memoria fornisce flessibilità durante l'allocazione della memoria, come se il programma fosse abbastanza grande, eseguirà operazioni di allocazione della memoria su parti diverse dei programmi e ridurrà lo spreco di memoria.

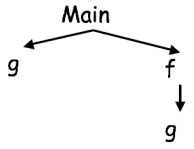
6.3.3 Allocazione della memoria a stack

Tutti i linguaggi che usano procedure, funzioni o metodi come elementi di base per la costruzione dei programmi gestiscono almeno una parte della loro memoria a run-time come uno stack. Ogni volta che viene chiamata una procedura una certa quantità di memoria per le sue variabili locali viene impilata sullo stack; quando la procedura termina, tale spazio viene rimosso dallo stack. Come vedremo questa scelta non solo permette la condivisione dello stesso spazio tra le procedure che non si sovrappongono nel tempo, ma ci permette anche di compilare il codice della procedura in modo tale che l'indirizzo relativo delle sue variabili locali sia sempre lo stesso, indipendentemente dalla sequenza di chiamate di procedura.

Alberi di attivazione

Possiamo quindi rappresentare le attivazioni delle diverse procedure durante l'esecuzione di un dato programma mediante un albero, detto **albero di attivazione**. Ogni nodo corrisponde ad un'attivazione e la radice dell'albero rappresenta l'attivazione della procedura principale, cioè quella da cui ha inizio l'esecuzione del programma. I figli di un nodo che rappresenta l'attivazione di una procedura p corrispondono alle attivazioni delle procedure richiamate dalla specifica attivazione di p. Tali attivazioni sono riportate sull'albero seguendo da sinistra a destra l'ordine in cui sono state chiamate. Si noti che ogni figlio deve terminare prima che possa iniziare l'attivazione alla sua destra.
Esempio 1:

```
class Main{
    g(): Int {1}
    f(): Int  {g()}
    main(): Int { {g(); f();}
}
```



Esempio 2:

```
class Main{
    g(): Int {1}
    f(x : Int): Int  {if x = 0 then g() else f(x-1) fi}
    main(): Int { {g(); f();}
}
```

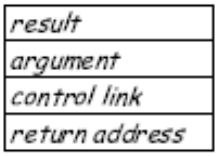
main -> f(3) -> f(2) -> f(1) -> f(0) -> g()

6.3.4 Record di attivazione

Le chiamate e i ritorni da procedura sono generalmente gestiti a run-time da uno speciale stack detto stack di controllo. Ogni attivazione aperta ha un record d'attivazione (a volte anche detto frame d'attivazione) sullo stack di controllo. Abbiamo diverse rappresentazioni dei record d'attivazione:

Record di attivazione a 4 campi

- valori di ritorno (result): eventuali valori di ritorno di una procedura può essere vuoto;
- parametri (argument): sono i parametri che può avere una procedura;
- control link: puntatore all'inizio del record di attivazione;
- return address: indirizzo a cui ritornare alla fine della procedura.



Record di attivazione a 5 campi

- Frame pointer: viene utilizzato per riferirsi a variabili presenti nel frame dello stack corrente;
- Parametri: sono i parametri che può avere una procedura;
- Valore di ritorno: eventuali valori di ritorno di una procedura può essere vuoto;
- stato macchina: un campo in cui è memorizzato il precedente stato della macchina al momento della chiamata alla procedura;
- variabili temporali: è dove vengono conservate tutte le variabili temporali della procedura.

Record di attivazione a 7 campi

- parametri attuali: sono i parametri che può avere una procedura;
- Valore di ritorno: eventuali valori di ritorno di una procedura può essere vuoto;
- control link: puntatore all'inizio del record di attivazione;
- access link: un riferimento per localizzare i dati richiesti dalla procedura corrente ma situati altrove, per esempio in un altro record d'attivazione.
- stato macchina (stato salvato): un campo in cui è memorizzato il precedente stato della macchina al momento della chiamata alla procedura;
- dati locali: dati locali della procedura;
- valori temporali: è dove vengono conservate tutte le variabili temporali della procedura.

- Quando si vuole liberare lo spazio allocato per una funzione si utilizza la cancellazione virtuale, cioè sposto il puntatore al top dello stack all'indirizzo di ritorno della funzione cosicché possa essere sovrascritto.
- Se si implementa un linguaggio in cui si definisce che una funziona deve avere sempre lo stesso indirizzo di ritorno si perde la possibilità di poter effettuare chiamate ricorsive.

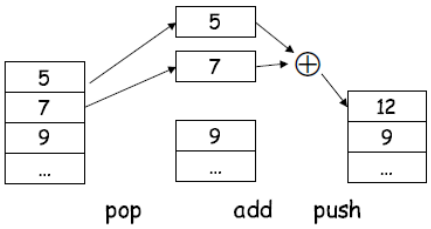
6.4 Stack machine

A differenza di una macchina a RAM dove abbiamo variabili o registri:
addr $t, $t1, $t2

Qui abbiamo uno stack e nient'altro. Per ogni istruzione avremo il seguente schema:

- Gli operandi sono sul top dello stack;
- Si rimuovono gli operandi desiderati dal top dello stack (pop);
- Si effettua il calcolo con gli operandi rimossi;
- si inserisce il risultato sul top dello stack (pop).

Di seguito un esempio di addizione con una macchina a stack:



Le stack machine sono molto comode per diversi motivi:

- perché ogni operazione prende l'operando da una posizione e inserisce i risultati nella stessa posizione. È uno schema di compilazione uniforme e rappresenta un semplice compilatore.
- La locazione di un operando è sempre sul top dello stack;
- non abbiamo bisogno di specificare gli operando e la locazione dei risultati;
- l'operazione "add" corrisponde all'istruzione "add r1. r2", è quindi più compatta.

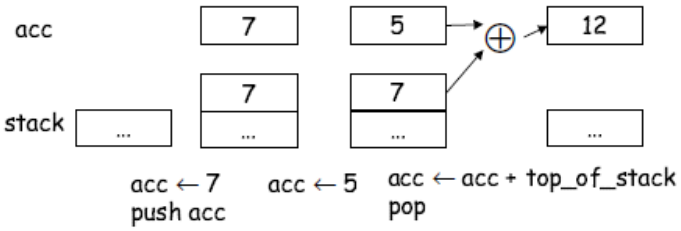
Java per il suo bytecode utilizza una macchina a stack.

Ottimizzazione della macchina a stack:
Una macchina a stack per effettuare l'operazione di add richiede 3 operazioni di memoria:

- 2 letture per prendere gli operandi dallo stack;
- 1 scrittura per inserire il risultato sullo stack.

Il top dello stack è frequentemente acceduto.
Un idea è quello di utilizzare un registro (hanno accesso veloce) chiamato accumulatore (acc) dove andremo a conservare il top dello stack:
acc <- acc + top_of_stack

Otteniamo così l'esecuzione di una sola operazione di memoria.



Lo stato iniziale dello stack è init, al termine dell'operazione rimarrà init.

Esercizio: Eseguire l'operazione 3 + (7 + 5) con una macchina a stack con accumulatore:

Code	**Acc**	**Stack**
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

6.5 MIPS

Abbiamo generato tramite il compilatore codice per una stack machine con accumulatore, vogliamo ora che il codice risultante sia eseguibile sul processore MIPS. Per farlo useremo i seguenti registri del MIPS (ricordando che in questo caso lo stack cresce verso gli indirizzi bassi):

- $a0: è il registro che rappresenta il nostro accumulatore;
- $sp è il registro contenente l'indirizzo del prossimo elemento nello stack;
- top: $sp + 4 punta al top dello stack;
- push: $sp + 4;
- pop: $sp - 4;
- $ra: indirizzo per l'indirizzo di ritorno dello stack;
- $fp: indirizzo per il frame pointer.

Ora elenchiamo le istruzioni che utilizzeremo del MIPS:

- lw reg_1 offset(reg_2): carica una parola di 32bit in reg_1 da un offset indicato da noi;
- sw reg_1 offset(reg_2): salva una parola di 32bit da reg_1 ad un registro indicato da un offset;
- add reg_1 reg_2 reg_3: inserisce in reg_1 la somma tra reg_2 e reg_3;
- addiu reg_1 reg_2 imm: effettua una somma senza effettuare il controllo sull'overflow;
- li reg imm: carica una costante in un registro.

Mostriamo ora la differenza tra stack machine e MIPS:

Stack Machine	**MIPS**
acc ← 7	li $a0,7
push acc	sw $a0, 0($sp) addiu $sp, $sp-4
acc ← 5	li $a0,5
acc ← acc + top_of_stack	lw $t1, 4($sp) add $a0, $a0 ,$st1
pop	addiu $sp, $sp 4

6.5.1 Strategia per generare codice MIPS

Per ogni espressione di una grammatica noi vogliamo generare del codice macchina che ne calcoli il valore e lo inserisca in $a0, preservi il registro $sp e il contenuto dello stack.
Indicheremo questro tramite una funzione cgen(e) che da in output il codice MIPS per e.

- **Per valutare una costante**: cgen(i) = li $a0 i
- **Codice per add**:
  cgen($e_1 + e_2$)=
  - cgen(e_1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e_2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
- **Codice per sub**:
  cgen($e_1 - e_2$)=
  - cgen(e_1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e_2)
    lw $t1 4($sp)
    sub $a0 $t1 $a0
    addiu $sp $sp 4

Per generare codice per una condizione abbiamo bisogno di due nuove istruzioni per il controllo del flusso:

- Salta all'etichetta l se reg_1 = reg_2: beq reg_1 reg_2 label;
- salta ad una nuova istruzione: b label.

Generiamo ora il codice per il costrutto if:

- **cgen(if $e_1 = e_2$ then e_3 else e_4)**
  - cgen(e_1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e_2)
    lw $t1 4($sp)
    addiu $sp $sp 4
    beq $a0 $t1 true_branch
    false_branch:
      cgen(e_4)
      b end_if
    true_branch:
      cgen(e_3)
    end_if:

Per generare codice per una chiamata di una funzione abbiamo bisogno di una nuova istruzione:

- jal label: salta ad una laber e salva in $ra l'indirizzo della prossima istruzione

Genereiamo ora il codice per una chiamata a funzione:

- **cgen(f($e_1, ..., e_n$))**=
  - sw $f0 0($sp)
    addiu $sp $sp -4
    cgen(e_n)
    sw $a0 0($sp)
    addiu $sp $sp -4
    ...
    cgen(e_1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry

Il chiamate salva il suo valore del frame pointer. Salva successivamente i parametri in ordine inverso. Il chiamante salva il suo indirizzo di ritorno nel registro $ra.

Codice per generare una definizione di funzione. Abbiamo anche qui bisogno di una nuova istruzione:

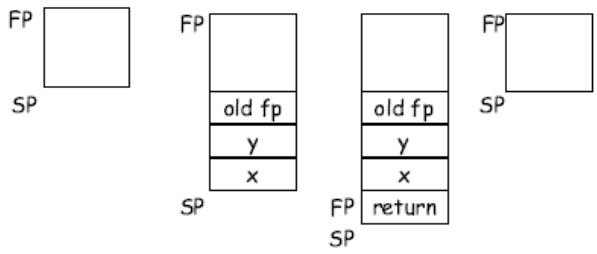
- jr reg: salta all'indirizzo reg

Generiamo quindi il codice per una chiamata a funzione:

- **cgen(def f(x_1, ..., x_n) = e)** =
  - move $fp $sp
    se $ra 0($sp)
    addiu $sp $sp -4
    cgen(e)
    lw $ra 4($sp)
    addiu $sp $sp z
    lw $fp 0($sp)
    jr $ra

Ricordiamo il frame pointer punta al top e non alla fine del frame. Il chiamato estrae l'indirizzo di ritorno, gli argomenti effettivi e il valore salvato del puntatore al frame.

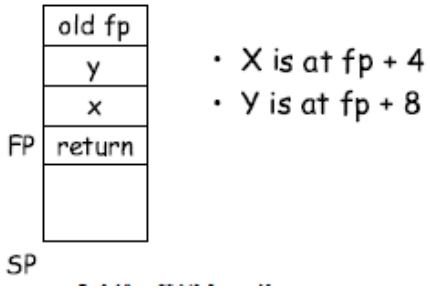
Esempio di un chiamata per f(x,y)



Generiamo ora il codice per le variabili:

- cgen(x_i) = lw \$a0 z(\$fp)

Utilizziamo il frame pointer per poter prendere la variabile dallo stack.



Capitolo 7

Esercizi a cazzo di cane di compilatori

- Effettuare il backpatching per la seguente regola:
$S \rightarrow$ if B_1 then S_1 else if B_2 then S_2

```
S → if B1 then M1 S1 N1 else if M1 B2 then M3 S2
            {
              backpatch(B1.truelist, M1.instr);
              backpatch(B1.falselist, M2.instr);
              backpatch(B2.truelist, M1.instr);
              S.nextlist = merge(S1.nextlist, N1.nextlist, B2.falselist, S2.nextlist);
            }
M → epsilon       { M.instr = nextinstr(); }

N → epsilon       { N.nextlist = makelist(); emit(goto_) }
```