Azzolini Riccardo 2020-12-10

Applicazioni dei linguaggi context-free: i compilatori

1 Struttura di un compilatore

Visto ad alto livello, un **compilatore** è un programma che prende in input il *codice sorgente* di un programma e, se questo ha la forma corretta rispetto alle regole del linguaggio, produce in output del *codice oggetto* eseguibile su una certa macchina (che può anche essere una macchina virtuale, come ad esempio nel caso di Java).

All'interno di un compilatore si possono identificare due principali componenti:

- il **reader**, che legge i sorgenti del programma, riconosce al loro interno le strutture del linguaggio, e costruisce una **rappresentazione interna** (**IR**, *Intermediate Representation*) di tali strutture, verificando nel frattempo la correttezza dell'input;
- il **generatore** o **traduttore**, che a partire dalla IR genera il codice oggetto.

1.1 Reader

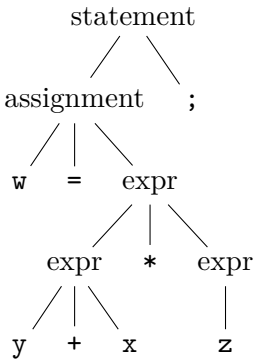
Si supponga di voler compilare il seguente frammento di codice sorgente in linguaggio C:

```
double f(int x, int y, double z) {
    double w;
    w = (x + y) * z;
    return w;
}
```

Gli elementi significativi "più piccoli", indivisibili, della struttura che il reader deve riconoscere in questo codice, ovvero i *simboli* del linguaggio, sono i **token** o **lessemi**: le parole chiave (`double`, `return`, …), gli identificatori (`f`, `x`, `y`, …), gli operatori (`=`, `+`, …), i separatori (`,`, `;`, …), ecc. Tuttavia, il codice sorgente viene fornito in input sotto forma di un file di testo, cioè una sequenza di singoli caratteri (`d`, `o`, `u`, `b`, `l`, `e`, spazio, `f`, ecc.): i simboli della stringa in input non corrispondono ai simboli del linguaggio, sono unità più piccole. Allora, il primo passo per il riconoscimento della struttura è raggruppare i caratteri in token. Poi, i token vengono a loro volta raggruppati in **frasi** secondo le *regole sintattiche* del linguaggio, espresse da una grammatica libera dal contesto avente i token

come terminali. Infine, queste frasi vengono organizzate nella rappresentazione interna, che in sostanza è l'albero sintattico corrispondente alle frasi identificate, semplificato rimuovendo informazioni ridondanti e arricchito invece con informazioni aggiuntive utili per il generatore. Nel contesto della teoria dei compilatori, gli alberi sintattici sono chiamati **alberi di parsing**.

Ad esempio, la struttura riconosciuta dal reader per la riga di codice `w = (x + y) * z;` del frammento mostrato prima potrebbe essere la seguente (qui mostrata in forma semplificata):



1.2 Lexer e parser

Le due fasi di "raggruppamento" necessarie per il passaggio dai caratteri all'IR vengono gestite da due diversi componenti del reader:

1. il **lexer** riceve in input uno stream (flusso) di caratteri, li raggruppa in token, e produce come output uno stream di token;
2. il **parser** riceve in input lo stream di token emesso dal lexer, organizza i token in frasi, e produce in output la rappresentazione interna.

Siccome i token di un tipico linguaggio di programmazione costituiscono un linguaggio regolare, l'implementazione dei un lexer si basa solitamente su un DFA (con qualche piccola aggiunta per eseguire l'output dei token riconosciuti). In particolare, esistono degli strumenti che, data una definizione dei token sotto forma di espressioni regolari, generano automaticamente il DFA che li riconosce. Inoltre, questo è uno dei casi "fortunati" in cui la dimensione del DFA rimane sempre proporzionale a quella dell'NFA, ovvero a quella dell'espressione regolare di partenza, senza crescere in modo esponenziale.

Analogamente, siccome le frasi di un tipico linguaggio sono descritte da una CFG, i parser sono implementati mediante vari tipi di riconoscitori per i linguaggi context-free (adattati in modo da costruire l'IR per le frasi riconosciute), che possono essere generati automaticamente, tramite appositi strumenti, a partire da CFG scritte in sintassi BNF o simile. Uno di questi riconoscitori sono gli automi a pila (deterministici), che, come

visto in precedenza, corrispondono alle derivazioni leftmost su una grammatica, dunque operano in modo top-down (applicano le produzioni partendo dal simbolo iniziale e procedendo verso i terminali). L'approccio basato su di essi (che verrà illustrato a breve) è molto usato al momento,[1] in quanto semplice e intuitivo, ma in passato si preferiva un metodo bottom-up,[2] corrispondente all'inferenza ricorsiva, perché esso richiede meno memoria. Entrambi questi algoritmi hanno però delle limitazioni sulle grammatiche che sono in grado di gestire.

1.3 Symbol table e analizzatore semantico

Le frasi di un linguaggio di programmazione sono soggette non solo alle regole sintattiche, ma anche alle regole di **semantica statica**, che non vengono espresse tramite una CFG, perché farlo è in molti casi impossibile, e se anche fosse possibile complicherebbe troppo la grammatica. Due esempi tipici di regole che non possono essere espresse tramite una CFG, ma che devono essere rispettate dalle frasi di un linguaggio staticamente tipizzato,[3] sono le seguenti:

- le variabili devono essere dichiarate prima di essere utilizzate;
- le espressioni devono rispettare le regole sui tipi.

Ad esempio, la frase `w = (x + y) * z` è corretta solo se:

- `x`, `x`, `y` e `z` sono definite;
- i tipi di `x`, `y` e `z` rispettano le regole imposte dagli operatori dell'espressione `(x + y) * z`;
- l'espressione `(x + y) * z` è assegnabile a `w`, cioè ha un tipo compatibile con il tipo di `w`.

Si noti in particolare che, per verificare quest'ultima condizione, bisogna prima determinare il tipo dell'espressione, secondo i tipi delle variabili coinvolte e le regole degli operatori.

Le regole semantiche vengono verificate dall'**analizzatore semantico**, un terzo componente del reader[4] che opera sull'albero di parsing generato dal parser. Esso si appoggia a una struttura dati chiamata **symbol table**, la quale contiene le informazioni sugli elementi definiti in un programma (variabili, funzioni/metodi, ecc.) che sono necessarie

[1] Un esempio di strumento per generare questo tipo di parser è *ANTLR* (ANother Tool for Language Recognition).

[2] Il classico strumento Unix per la costruzione dei parser, *yacc* (Yet Another Compiler Compiler), è basato su tale metodo.

[3] Un linguaggio staticamente tipizzato è uno in cui il tipo di ciascun'espressione dipende solo da ciò che è scritto nel programma, e non da ciò che avviene in fase di esecuzione.

[4] La suddivisione tra parser e analizzatore semantico non è sempre netta: a volte, la verifica della semantica statica avviene parzialmente o totalmente in contemporanea alla fase di riconoscimento delle frasi.

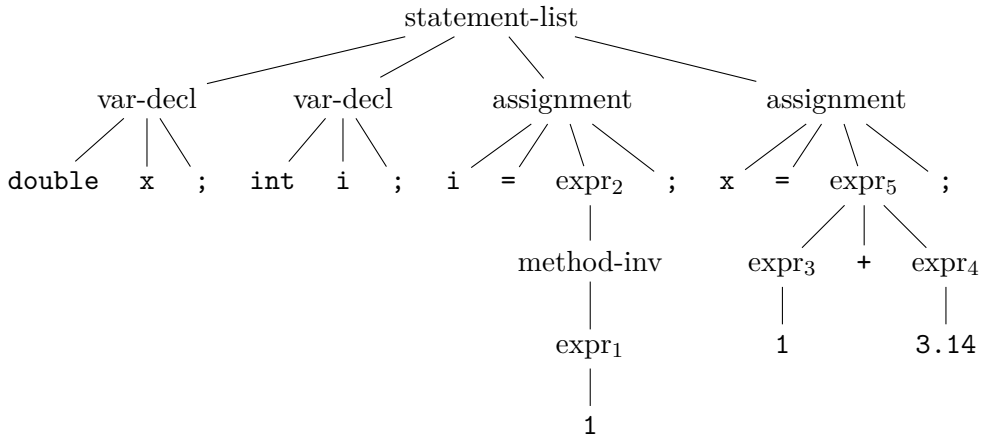
a decidere le regole semantiche del linguaggio. In genere, la symbol table viene creata dal parser, man mano che incontra le definizioni dei vari elementi, e utilizzata dall'analizzatore semantico. Poi, durante l'analisi, quest'ultimo deduce nuove informazioni (ad esempio i tipi delle espressioni) che potrebbero essere utili per il generatore, e perciò le aggiunge come annotazioni sull'albero di parsing.

1.3.1 Esempio: costruzione e uso della symbol table

Si consideri questo frammento di codice:

```
double x;
int i;
i = p(1);
x = i + 3.14;
```

Dato lo stream di token individuati dal lexer, il parser costruisce il seguente albero di parsing (qui semplificato),



e inoltre aggiunge alla symbol table le informazioni relativi alle dichiarazioni di variabili `double x;` e `int i;`. Se si suppone che sia già stata incontrata in precedenza la definizione del metodo `int p(int k)`, al termine del parsing la symbol table ha i seguenti contenuti:

Identificatore	Categoria	Altre informazioni
`p`	metodo	prototipo `int` → `int`
...	...	...
`x`	variabile	tipo `double`
`i`	variabile	tipo `int`

Successivamente, l'analizzatore semantico visita l'albero di parsing (in questo caso in postordine) per verificare le regole semantiche del linguaggio. Nel fare ciò, esso deduce il tipo di ciascuna espressione (a partire dai tipi degli operandi), e lo annota sull'albero di parsing, indicando eventuali conversioni di tipo (dettate dalle regole degli operatori) che dovranno essere inserire nel codice oggetto dal generatore. Le annotazioni aggiunte dall'analizzatore semantico in questo esempio potrebbero essere le seguenti:

Nodo	Informazioni
expr_1	tipo `int`
expr_2	tipo `int`
expr_3	tipo `int`, conversione a `double`
expr_4	tipo `double`
expr_5	tipo `double`

In particolare, alcuni casi interessanti sono:

- expr_2, che ha tipo `int` perché è un'invocazione del metodo `p`, che — come indicato nella symbol table — restituisce appunto `int`;
- expr_3, che ha tipo `int` in quanto `1` è un letterale intero, ma deve essere convertita a `double`, il tipo dell'altro operando di expr_5, per poter eseguire l'addizione.

2 Parser a discesa ricorsiva

Un parser a **discesa ricorsiva** (*recursive descent*) è un programma, scritto in un normale linguaggio di programmazione, che implementa il PDA corrispondente a una grammatica, usando lo stack di esecuzione del linguaggio di programmazione per rappresentare implicitamente lo stack dell'automa.

Data una grammatica $G = \langle V, T, \Gamma, S \rangle$, un parser a discesa ricorsiva è realizzato definendo, per ogni simbolo non-terminale $F \in V$, un metodo `void F()` che "implementa" nel modo seguente le regole di produzione per F:

- In corrispondenza di un simbolo non-terminale A nel corpo di una regola di produzione, si invoca il metodo `A()` associato a tale simbolo.
- In corrispondenza di un simbolo terminale T nel corpo di una regola di produzione, viene invocato il metodo `match(`t`)`:

  ```
  void match(String terminal) throws ParseException {
    if (il primo token dello stream di input coincide con terminal) {
      rimuovi il token dallo stream di input
      return
    } else {
      solleva l'eccezione ParseException
  ```

```
    }
}
```

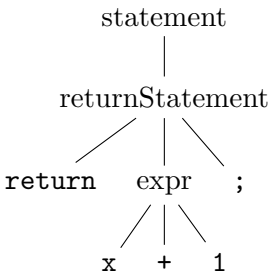
esso verifica che il prossimo terminale in input sia quello previsto dalla regola di produzione, e in caso contrario dà un errore.

2.1 Esempio

Si consideri una grammatica G_{simple} con le seguenti regole di produzione:

$$\begin{aligned} \text{statement} &\to \text{returnStatement} \\ \text{returnStatement} &\to \texttt{return}\ \text{expr}\ \texttt{;} \\ \text{expr} &\to \texttt{x + 1} \end{aligned}$$

Questa è una grammatica molto semplice, che genera unicamente la frase `return x + 1;`, corrispondente all'albero sintattico:



In base alle regole descritte prima, il codice che implementa un parser a discesa ricorsiva per questa grammatica è:

```
void statement() {
    returnStatement();
}

void returnStatement() {
    match("return"); expr(); match(";");
}

void expr() {
    match("x"); match("+"); match("1");
}
```

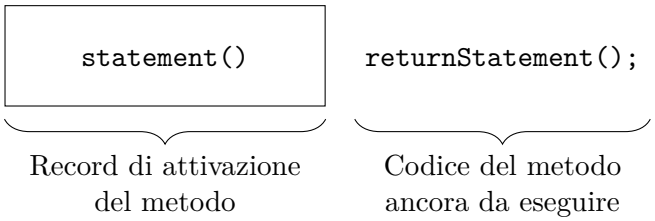
Adesso, si osserverà che il funzionamento di questo codice riproduce le computazioni del PDA associato a G_{simple}. Allora, è utile vedere prima come è fatta la computazione accettante di tale PDA sulla frase `return x + 1;` (che viene fornita in input come di stream di token: `return`, `x`, `+`, `1`, `;`):

Passo di computazione	Regola simulata
(q, `return x + 1 ;`, statement)	
$\vdash$ (q, `return x + 1 ;`, returnStatement)	statement $\rightarrow$ returnStatement
$\vdash$ (q, `return x + 1 ;`, `return` expr `;`)	returnStatement $\rightarrow$ `return` expr `;`
$\vdash$ (q, `x + 1 ;`, expr `;`)	match su `return`
$\vdash$ (q, `x + 1 ;`, `x + 1 ;`)	expr $\rightarrow$ `x + 1`
$\vdash$ (q, `+ 1 ;`, `+ 1 ;`)	match su `x`
$\vdash$ (q, `1 ;`, `1 ;`)	match su `+`
$\vdash$ (q, `;`, `;`)	match su `1`
$\vdash$ (q, ϵ, ϵ)	match su `;`

Siccome l'ID alla fine della computazione è (q, ϵ, ϵ), il PDA accetta per stack vuoto.

Ciascun passo della computazione appena mostrata corrisponde a un passo di esecuzione del parser a discesa ricorsiva:

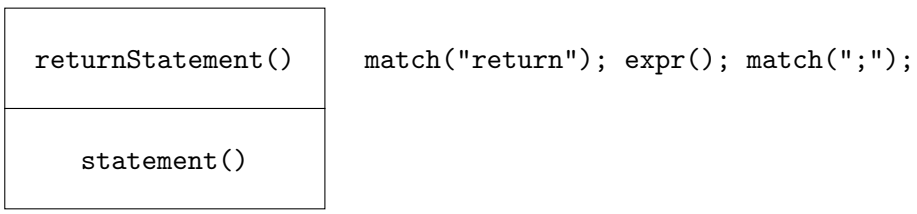
1. All'inizio, viene dato in input al parser lo stream di token della frase `return x + 1;`, e viene invocato il metodo `statement()`, corrispondente al simbolo non-terminale iniziale della grammatica G_{simple}. Allora, sullo stack di esecuzione (del linguaggio di programmazione con cui è implementato il parser) viene creato un record di attivazione per tale metodo,



   e la prossima istruzione che il programma dovrà eseguire, corrispondente al simbolo in cima allo stack del PDA, è l'invocazione di `returnStatement()`. La situazione a questo punto corrisponde dunque a quella dopo il primo passo di computazione del PDA, nel quale viene simulata l'applicazione della regola di produzione statement $\rightarrow$ returnStatement e non vengono consumati token in input:

   (q, `return x + 1 ;`, statement) $\vdash$ (q, `return x + 1 ;`, returnStatement)

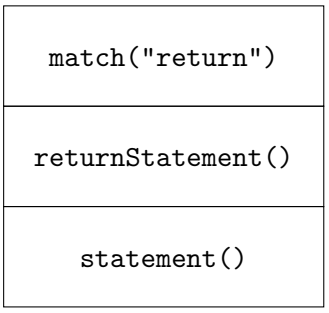
2. Viene invocato il metodo `returnStatement()`, per il quale si crea un nuovo record di attivazione:



Ciò equivale al secondo passo di computazione del PDA, che simula l'applicazione della regola di produzione returnStatement → `return` expr `;`, ancora senza consumare input:

$$(q,\ \texttt{return x + 1 ;},\ \text{returnStatement}) \vdash (q,\ \texttt{return x + 1 ;},\ \texttt{return}\ \text{expr}\ \texttt{;})$$

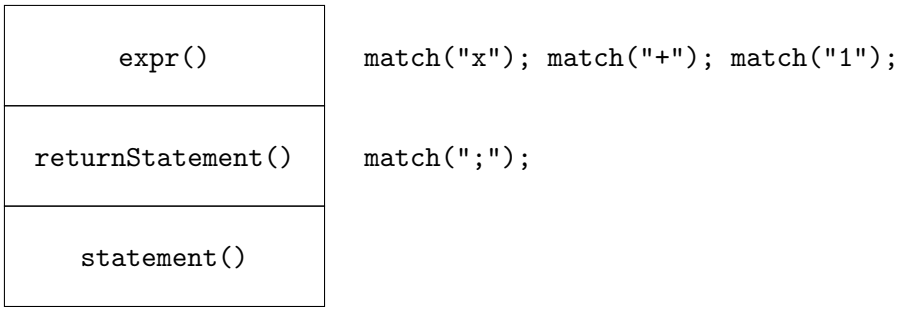
3. Adesso si ha la prima invocazione del metodo `match`,



che consuma il token `return`, simulando il passo di computazione

$$(q,\ \texttt{return x + 1 ;},\ \texttt{return}\ \text{expr}\ \texttt{;}) \vdash (q,\ \texttt{x + 1 ;},\ \text{expr}\ \texttt{;})$$

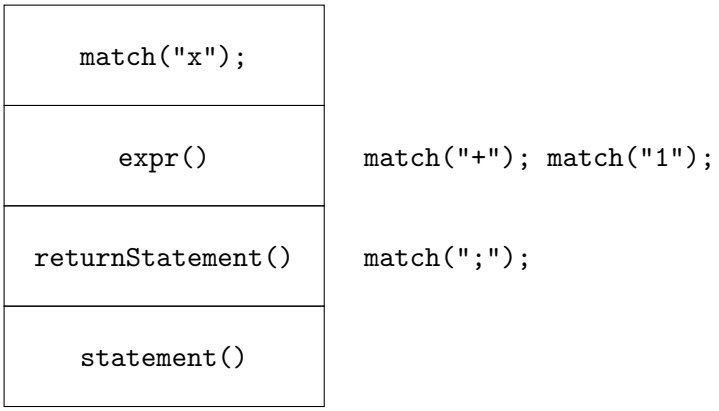
4. L'istruzione successiva di `returnStatement()` è l'invocazione di `expr()`, dunque viene creato un nuovo record di attivazione, e intanto rimane ancora da eseguire l'ultima istruzione di `returnStatement()`:



Il passo di esecuzione corrispondente è

$$(q,\ \texttt{x + 1 ;},\ \text{expr}\ \texttt{;}) \vdash (q,\ \texttt{x + 1 ;},\ \texttt{x + 1 ;})$$

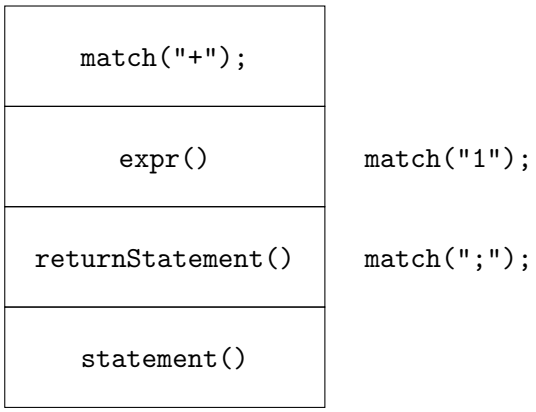
5. Invocando `match`,



il metodo `expr()` consuma il token `x`:

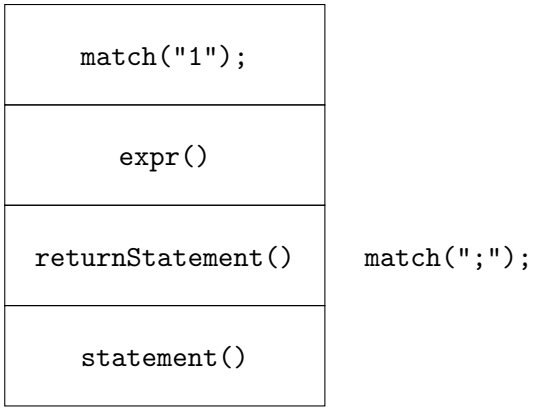
$$(q,\ \texttt{x + 1 ;},\ \texttt{x + 1 ;}) \vdash (q,\ \texttt{+ 1 ;},\ \texttt{+ 1 ;})$$

6. Poi consuma anche il token `+`:



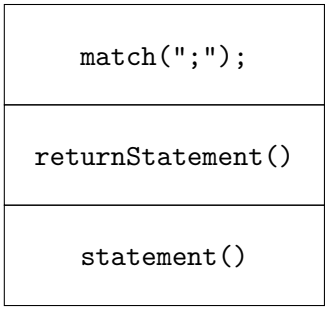
$$(q,\ \texttt{+ 1 ;},\ \texttt{+ 1 ;}) \vdash (q,\ \texttt{1 ;},\ \texttt{1 ;})$$

7. E infine consuma il token `1`:



$$(q,\ \texttt{1}\ \texttt{;},\ \texttt{1}\ \texttt{;}) \vdash (q,\ \texttt{;},\ \texttt{;})$$

8. Essendo ora terminata l'esecuzione di `expr()`, il suo record di attivazione viene rimosso dallo stack, e si torna a `returnStatement()`, che esegue la sua ultima istruzione `match`,



consumando così l'ultimo token in input, `;`. Infine, non essendoci più istruzioni da eseguire, `returnStatement()` e `statement()` terminano, lasciando uno stack di esecuzione vuoto. Tutto ciò corrisponde al passo di computazione

$$(q,\ \texttt{;},\ \texttt{;}) \vdash (q,\ \epsilon,\ \epsilon)$$

che porta il PDA ad accettare per stack vuoto.

2.2 Limiti

Data una qualunque CFG, si può sempre applicare la costruzione mostrata prima per implementare un parser a discesa ricorsiva, ma *non è garantito* che tale parser riconosca effettivamente il linguaggio generato dalla grammatica di partenza. Infatti, possono essere riconosciute da parser a discesa ricorsiva solo le grammatiche **LL(k)**,[5] una *sottoclasse propria* delle CFG; in particolare, esse sono le grammatiche:

- non ambigue;
- prive di ricorsione a sinistra;
- per cui è possibile risolvere il non-determinismo nell'applicazione delle regole di produzione guardando al più $k \geq 1$ simboli successivi dell'input.

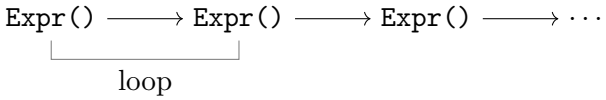
[5] Le lettere LL si riferiscono al fatto che il parser legge l'input "Left-to-right" (da sinistra a destra) e segue la strategia di derivazione Leftmost.

2.2.1 Ricorsione a sinistra

Si parla di **ricorsione a sinistra** (**left-recursion**) **diretta** quando il corpo di una regola di produzione contiene come *primo* simbolo lo stesso simbolo non-terminale che tale regola definisce, cioè il simbolo in testa alla regola: $A \to A\alpha$. Ad esempio, nella grammatica delle espressioni semplificate, si ha ricorsione a sinistra diretta nella regola di produzione Expr → Expr `+` Expr. Applicando a questa produzione la tecnica di costruzione di un parser a discesa ricorsiva, si ottiene il metodo

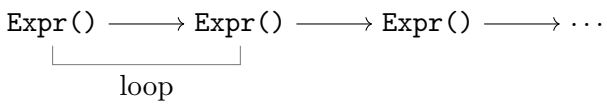
```
void Expr() {
    Expr(); match("+"); Expr();
}
```

che quando viene invocato va immediatamente in loop infinito, perché la prima cosa che fa è chiamare ricorsivamente se stesso (senza un caso base per arrestare la ricorsione):

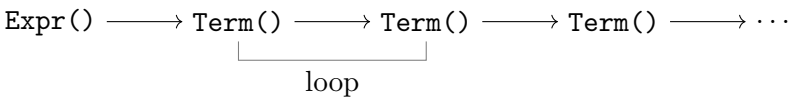


Si noti che questo problema non è in alcun modo legato all'ambiguità della grammatica delle espressioni da cui è tratta questa produzione, tanto è vero che si ha ricorsione a sinistra diretta anche nella grammatica non ambigua:

- la produzione Expr → Expr `+` Term da luogo a un loop infinito esattamente come nella grammatica ambigua:



- anche la produzione Term → Term `*` Factor può dar luogo a un loop infinito:



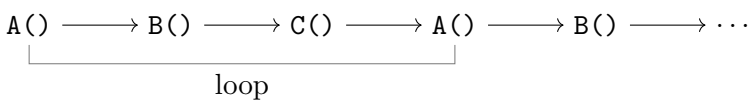
Un'altra situazione problematica è la **ricorsione a sinistra indiretta**, nella quale il loop infinito di chiamate ricorsive coinvolge più metodi. Ad esempio, essa è presente nella grammatica

$$A \to B\texttt{r}$$
$$B \to C\texttt{s}$$
$$C \to A\texttt{t}$$

il cui parser a discesa ricorsiva è:

```
void A() { B(); match("r"); }
void B() { C(); match("s"); }
void C() { A(); match("t"); }
```

Quando si invoca il metodo `A()`, questo chiama immediatamente `B()`, che a sua volta chiama subito `C()`, e quest'ultimo richiama subito `A()`, dando luogo a un loop infinito:



In generale, perché un parser a discesa ricorsiva funzioni, bisogna evitare la ricorsione a sinistra, sia diretta che indiretta. Fortunatamente, è sempre possibile eliminarla, applicando alla grammatica delle opportune trasformazioni (che però, purtroppo, determinano una perdita di leggibilità[6]). Ad esempio, date le regole di produzione

$$\begin{aligned}
\text{Expr} &\to \textbf{Expr}\ \texttt{+}\ \text{Expr} \\
\text{Expr} &\to \textbf{Expr}\ \texttt{*}\ \text{Expr} \\
\text{Expr} &\to \texttt{(}\ \text{Expr}\ \texttt{)} \\
\text{Expr} &\to \texttt{INT}
\end{aligned}$$

la ricorsione a sinistra (qui evidenziata in grassetto) può essere eliminata trasformando la grammatica nella seguente:

$$\begin{aligned}
\text{Expr} &\to \texttt{(}\ \text{Expr}\ \texttt{)}\ \text{Expr}' \\
\text{Expr} &\to \texttt{INT}\ \text{Expr}' \\
\text{Expr}' &\to \texttt{+}\ \text{Expr}\ \text{Expr}' \\
\text{Expr}' &\to \texttt{*}\ \text{Expr}\ \text{Expr}' \\
\text{Expr}' &\to \epsilon
\end{aligned}$$

(qui per semplicità si è lavorato sulla grammatica ambigua, ma una trasformazione analoga può essere applicata alla versione non ambigua).

2.2.2 Eliminazione del non-determinismo

Essendo un programma per un calcolatore reale, un parser a discesa ricorsiva viene eseguito da un modello di computazione deterministico. Allora, quando esistono più regole di produzione per un non-terminale, il programma deve aver modo di "capire" deterministicamente, in base all'input, quale regola applicare.

[6] Molti strumenti per la generazione di parser hanno il vantaggio di essere in grado di eliminare automaticamente la ricorsione a sinistra (almeno nei casi più semplici, se non in generale): così, la grammatica scritta dall'utente rimane leggibile, e tutte le complicazioni sono gestite internamente dallo strumento usato.

Ad esempio, si consideri la grammatica

$$\begin{aligned}
\text{statement} &\to \text{returnStatement} \mid \text{ifStatement} \mid \text{assignment} \\
\text{returnStatement} &\to \texttt{return}\ \text{expr}\ \texttt{;} \\
\text{ifStatement} &\to \texttt{if}\ \text{expr}\ \texttt{then}\ \text{statement}\ \texttt{else}\ \text{statement} \\
\text{assignment} &\to \texttt{ID}\ \texttt{=}\ \text{expr} \\
\text{expr} &\to \ldots
\end{aligned}$$

Nell'implementazione di un parser a discesa ricorsiva, il metodo `statement()` deve decidere se invocare `returnStatement()`, `ifStatement()` o `assignment()`. Per fare ciò, si guarda il primo token rimasto in input *senza consumarlo*; quest'azione è chiamata **lookahead**, e il token "guardato" prende il nome di **lookahead token**.

```
void statement() {
  if (il lookahead token è "return") {
    returnStatement();
  } else if (il lookahead token è "if") {
    ifStatement();
  } else if (il lookahead token è ID) {
    assignment();
  } else {
    parse error
  }
}
```

Qui è sufficiente guardare un singolo token per determinare quale regola di produzione applicare, dunque il parser ha lookahead pari a $k = 1$, ovvero è un parser LL(1). In generale, un parser LL(k) è tanto più efficiente quanto più piccolo è k, perché il lookahead fa "perdere tempo" a leggere più volte gli stessi token.

Data una qualunque CFG, esiste un algoritmo per determinare se essa abbia lookahead k, mentre non è in generale possibile trasformarla in una grammatica con lookahead k che generi lo stesso linguaggio.

2.3 Costruzione dell'IR

Per costruire la rappresentazione interna della struttura sintattica riconosciuta da un parser a discesa ricorsiva, si aggiunge semplicemente ai vari metodi il codice per le operazioni di costruzione, che verranno così eseguite per ciascun non-terminale riconosciuto. Ad esempio:

```
void returnStatement() {
    // ...
    // Una volta riconosciuto, esegui:
    System.out.println("found return");
}

void ifStatement() {
    // ...
    // Una volta riconosciuto, esegui:
    System.out.println("found if statement");
}
```

(qui, per semplicità, viene solo stampato un messaggio informativo, ma si potrebbero invece aggiungere dei nodi a un albero, ecc.).