PARSING TOP-DOWN

Deriva tutte le stringhe ricorsivamente finché:

- matcha la stringa derivata con la stringa in input

- è sicuro di avere trovato un errore nella sintassi

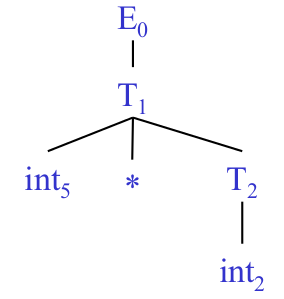
es: si considera la grammatica

E → T + E | T

T → int | int \* T | ( E )

e in input ho la stringa: int(5) \* int(2)

Si trova il seguente parse tree che matcha la stringa, quindi la sintassi è corretta:



SIMPLE PARSING STRATEGY

(Recursive Descent Parsing)

Per ogni variabile, si fa una funzione per ogni produzione che ritorna true solo se la stringa in input matcha quella produzione.

Tutti i match sono controllati da un'altra funzione (col nome della variabile, T in questo caso) che chiama tutte le produzioni e se una è true restituisce true:

bool T 1 () { return term(OPEN) && E() && term(CLOSE); }

bool T 2 () { return term(INT) && term(TIMES) && T(); }

bool T 3 () { return term(INT); }

bool T()

{

int save = next;

return T 1 ()

|| (next = save, T 2 ())

|| (next = save, T 3 ());

}

Funziona solo se non c'è una ricorsione a sinistra, esempio S → Sa.

**Eliminare la ricorsione a sinistra:**

Si usa un algoritmo che funziona solo se non ci sono ne produzioni epsilon ne produzioni unità.

La si trasforma in una ricorsione a destra:

S → S α 1 | ... | S α n | β 1 | ... | β m

diventa:

S → β 1 S’ | ... | β m S’

S’ → α 1 S’ | ... | α n S’ | ε

Una grammatica può essere ricorsiva a sinistra anche in più passi, ad esempio:

S → A α | δ

A → S β

S in più passi produce:

S →+ Sβα

L'algoritmo per eliminare la ricorsione a sinistra dice che:

Per ogni variabile:

Ai →Aj β

si trasforma la produzione con:

Ai →δ1 β | δ2 β |...| δk β

dove A j →δ1 | δ2 |...| δk

e poi si applica la trasformazione della pagina precedente.

-----------------------------------

Ad esempio, la grammatica:

S → A α | δ

A → S β

si trasforma in:

S → Sβα | δ

e poi la si fa diventare lineare a destra:

S → δS'

S' → βα S' | epsilon

**Predictive parsers**

Come gli algoritmi a discesa ricorsiva, ma possono predirre quale produzione usare grazie ai token di look-ahead, quindi la forma è del tipo:

Si sostituisce il codice

return E1() || (next = save, E2());

con

switch ( somhttps://www.facebook.com/?stype=lo&jlou=AfemOErouB3yDzdU-fDWf1pCifTUHwHcwWVugi-kICSeY3HNGjg-PzM7hDYXvTkwvfR7RtsK8630f519wxA9vFgj&smuh=57064&lh=Ac8tDPO8mdsEw2lvething )

{

case L1: return E1();

case L2: return E2();

otherwise: print “syntax error”;

}

Il vantaggio di questa forma è che non bisogna fare il backtracking (next = save; …)

I parser predittivi accettano le grammatiche LL(k):

– L means “left-to-right” scan of input

– L means “leftmost derivation”

– k means “predict based on k tokens of lookahead”

**LL(1)**

LL(1) dice che, per ogni variabile e tolen c'è al massimo una produzione che porta al successo.

Può essere rappresentato come una tabella 2D:

- una dimensione per la variabile corrente da espandere

- una dimensione per il prossimo token (di look-ahead)

- ogni cella della tabella contiene una produzione

**Left-Factoring:**

è la forma richiesta della grammatica per LL(1), si raggruppano le produzioni con un prefisso in comune e il resto viene concatenato con una variabile, es:

E → T + E | T

T → int | int \* T | ( E )

diventa:

E → T X

X → + E | ε

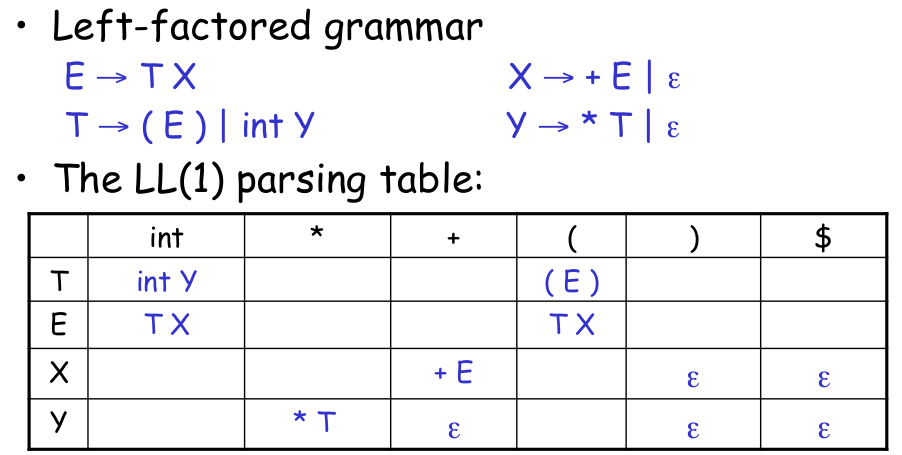
T → ( E ) | int Y

Y → \* T | ε

**LL(1) parsing table:**

al posto dello switch...L1......L2......

si usa una tabella e uno stack, esempio:



Nella cella [E, int] si sostituisce nello stack E con la sua produzione, cioè TX.

Nella cella [Y, +], si elimina Y dallo stack (si scrive epsilon al suo posto)

(per la regola di costruzione delle parsing table, si guardano i Follow di Y)

Ogni cella bianca indica situazioni di errore sintattico.

Per sapere quando finisce l'input, si aggiunge il simbolo **$** alla fine dell'input.

Lo stack, in partenza contiene il simbolo iniziale S, e sotto ad S il $.

Quando si matcha un terminale dell'input con un terminale in cima allo stack, lo si cancella dallo stack e si guarda il prossimo terminale di input.

Il parser accetta lòa stringa in input se e solo se alla fine dell'elaborazione, trova il $ sia nell'input che nello stack (se rimangono altre variabili nello stack e l'input è finito, devono potersi annullare tutte con epsilon).

Nessuna cella può avere definizioni multiple.

I linguaggi riconosciuti da LL(1) sono quelli riconosciuti dall' LL1(1) parsing table.

**First e Follow:**

es: il simbolo iniziale S produce, in zero o più passi:

S →\* βAγ

e l'input da controllare è βbδ (con δ diverso da γ).

ci sono 2 possibilità:

1) b appartiene ad un'espansione di A, tale che A → α e b è il primo terminale di α.

In questo caso, b € First(A)

2) A si annulla e b appartiene ad un'espansione di γ.

In questo caso, b € Follow(A) e epsilon € First(A).

**First:**

**Definizione:**

First(X) = { b | X → \* bα} ∪ {ε | X → \* ε}

**Calcolare i First:**

1. First(b) = { b }

2. per ogni produzione X → A1 ... An:

Add First(A 1 ) – {ε} to First(X). Stop if ε ∉ First(A 1 )

Add First(A 2 ) – {ε} to First(X). Stop if ε ∉ First(A 2 )

...

Add First(A n ) – {ε} to First(X). Stop if ε ∉ First(A n )

Add ε to First(X)

3. ripeti lo step 2 finché l'insieme dei First non cresce più

**Follow:**

**Definizione:**

Follow(X) = { b | S → \* β X b δ }

**Calcolare i Follow:**

1. Add $ to Follow(S) (if S is the start non-terminal)

2. For all productions Y → ... X A1 ... An

Add to Follow(X) First(A1 ... An ) – {ε}

(i … prima della X dicono che devi scorrere l'intero corpo della produzione, e fare quest'operazione per ogni fetta)

and for all productions Y → αX or Y → αXβ with ε ∈ First(β)

Add to Follow(X) the Follow(Y)

3. ripeti lo step 2 finché l'insieme dei Follow non cresce più

**Costruire LL(1) parsing table:**

Ci sono 2 dimensioni: una delle variabili e una dei terminali.

Per ogni produzione A → α in G si fa:

- per ogni b € First( α ) scrivi nella cella T[A,b] = α

- se epsilon € First( α ), allora si scrive α anche per ogni b € Follow( A )

(si applica anche al simbolo $, se fa parte dei Follow di A)

T[A,b] = α, e se anche $ fa parte dei Follow: T[A, $] = α

**Risolvere abmiguità:**

E → E + E | E \* E | ( E ) | int

La grammatica contiene più di un parse tree per una combinazione di tipo

int + int \* int.

Si riscrive la grammatica con associatività a sinistra del + e del \* e con la precedenza del \* sul +:

E → E + T | T

T → T \* int | int | ( E )

**The Dangling Else:**

S → if E then S

| if E then S else S

è ambigua perché l'else può fare riferimento sia al primo if che al secondo if.

Se vogliamo che l'else faccia riferimento solo all'ultimo IF aperto, si modifica la grammatica con i MIF (matched IF) e UIF (unmatched IF):

S → MIF | UIF

MIF → if E then MIF else MIF

UIF → if E then S | if E then MIF else UIF

**Bottom-up parsing**

Il bello degli algoritmi di bottom-up parsing è che possono usare le grammatiche anche ricorsive a sinistra e non fattorizzate a sinistra, tipo la grammatica di prima:

E → E + T | E – T | T

T → T \* int | int

Funziona anche con le grammatiche ambigue.

**Chaotic bottom-up parsing:**

prova una strada e se va bene bene, altrimenti male.

**Non-determinist Chaotic bottom-up parsing:**

si prende in input la stringa da controllare la sintassi con la grammatica.

Si trovano nella stringa tutti i possibili right-hand side di una produzione della grammatica.

Per ogni possibilità, fa una riduzione clonando l'istanza del problema.

Continua così finché non arriva al simbolo iniziale S.

SVANTAGGI:

ad ogni passo crea k copie di istanze, quindi k\*k\*k\*...\*k

Molte istanze possono arrivare alla soluzione, c'è uno spreco.

**Non-deterministic LR parser:**

Crea istanze multiple come l'algoritmo sopra, però solo un'istanza può portare al sibolo iniziale, se ci sono errori di sintassi nessuna.

Si divide l'input in 2 parti, viene separato dal triangolo rosso.

La parte a destra del triangolo deve ancora essere esaminata dal parser, mentre la parte a sinistra è già stata letta dal parser.

La parte a sinistra sta in uno stack, quindi le operazioni di riduzione si possono effettuare solo nei terminali dell'input che si trovano nella parte più a destra del triangolo (uno o più simboli si possono prendere per ogni riduzione, ma leggendo da destra verso sinistra)

Quindi LR sta per:

L = legge l'input da sinistra a destra

R = Right-most derivation

**Azioni:**

- Reduce

- Shift (sposta il triangolo a destra)

entrambe le azioni sono scelte in modo non deterministico.

**Algoritmo:**

1) Trova tutte le riduzioni possibili in cima allo stack (supponiamo che siano k)

2) Crea k copia di istanze, in ogni istanza fa una delle k riduzioni, nell'istanza originale non fa nessuna riduzione e fa shift, poi torna al passo 1.

3) si ferma quando la riduzione trova il simbolo iniziale.

**Generalized LR parser: SLR(1)**

Come l'algoritmo sopra, ma trova in anticipo le istanze che sono condannate a bloccarsi, guardando lo stack.

Fa una lista di tutti i possibili stack legali, tramite un DFA.

Se la configurazione dello stack non è legale, l'istanza è condannata a bloccarsi, quindi la si uccide.

**SLR automaton:**

il DFA che riconosce tutte le istanze legali di stack si chiama SLR automaton.

(tutti gli stati sono accettanti).

Lo stato iniziale è formato dal simbolo iniziale e dalla chiusura di S, cioè tutte le variabili raggiungibili in una solo passo →

Per gli altri stati, si aggiungono sempre le produzioni facendo la chiusura dei termini che seguono il pallino.

(guarda esempio goto.pdf, 30 slide fatte passo per passo)

L'automa dice anche che azioni fare, se fare shift o reduce.

**Azioni**

- Riduzione: quando il pallino arriva alla fine della produzione

- Shift: altrimenti

Possono esserci conflitti, quindi si usa il look-ahead e si cambiano le azioni shift / reduce.

Alla fine si usa la rappresentazione tabellare del SLR automaton.

**Algoritmo per costruire rappresentazione tabellare dell'SLR automaton:**

1) disegna l'SLR automaton usando la Closure in ogni stato e in ogni produzione, nel simbolo dopoil pallino

2) Fai la tabella:

- in una dimensione ci sono gli stati

- nell'altra i terminali e le variabili della grammatica

3) Calcola Follow Set delle variabili della grammatica

4) Per ogni terminale in uscita da ogni stato, fai l'azione di shift.

Quindi nella cella della tabella scrivi:

shift, <nome stato destinazione>

5) Per ogni stato che ha almeno una produzione col pallino alla fine, fai l'azione di riduzione in ogni terminale presente nell'insieme dei Follow della variabile che è nel Left-Hand Side della produzione.

Ad esempio, se sono nello stato 4 con la produzione: V → int G pallino in fondo,

Trovo tutti i Follow di V, e per ogni terminale che appartiene ai Follow di V, scrivo nella cella identificata dalla riga dello stato:

reduce, V → int G

6) Per ogni variabile in uscita dalle transizioni, scrivi nella cella:

goto, <nome stato destinazione>

**Accettare l'input:**

si aggiunge il $ alla fine dell'input e si aggiunge il nuovo simbolo iniziale S' → S.

L'input viene accettato in tutti gli stati che hanno la produzione

S' → S pallino in fondo.

Bisogna poi ricalcolare i Follow Set considerando anche il $ e S'.

La tabella finale è l'algoritmo si parsing SLR(1).

**LR(1) Parsing:**

La grammatica di riferimento è:

E → E + ( E ) | int

I linguaggi accettati dai parser LR sono più ampi rispetto ai linguaggi dei parser LL perché LR legge prima tutti i simboli in input di una produzione e poi fa la riduzione, mentre invece LL legge al massimo k di questi terminali (i caratteri di look-ahead).

[X → α•Yβ, a] è un LR Item

dove α è già stato letto (quindi è nello stack), si trova prima del pallino

Y può essere una variabile che contiene altre produzioni.

Dopo aver fatto il parsing di Yβ, mi aspetto di trovare il carattere a in input, quindi posso ridurre il tutto nella variabile X.

Y → γ, cosa mi aspetto di trovare dopo aver fatto il parsing di γ?

Il First dei simboli che seguono Y nella produzione di X, cioè First(βa).

Quest'operazione è detta Closure(Items), e si trovano tutte le produzioni che stanno dentro ad uno stato del DFA del LR(1).

**Closure(Items):**

repeat

for each [X → α•Yβ, a] in Items

for each production Y → γ

for each b ∈ First(βa)

add [Y → •γ, b] to Items

until Items is unchanged

**Esempio:**

Si parte sempre con la produzione S → E, che sarebbe l'item [S → •E, $]

Il primo stato del DFA dell'LR(1) quindi è la Closure( [S → •E, $] ):

------------------------------------------------------------

- La variabile da espandere è E, perché è dopo il pallino

- Guardo tutte le produzioni di E:

E → E+(E)

E → int

- l'algoritmo dice che, per ogni produzione trovata, devo controllare tutti i First di “Ba” che seguono la variabile nella produzione precedente, in questo caso:

B è epsilon (dopo E non c'è nessun simbolo)

a è il $

Quindi First(Ba) = {$}

Allora aggiungo all'insieme degli item dello stato iniziale questo item:

[E → •E+(E), $]

La stessa cosa vale anche per l'altra produzione, aggiungo quindi l'item:

[E → •int, $]

---------------------------------------------------

Ora il for più esterno cambia item e passa al secondo, es [E → •E+(E), $].

In questo caso:

- la variabile da espandere è E,

- Le produzioni di E sono:

E → E+(E)

E → int

B è “+(E)”

a è $

Quindi First(Ba) = {+}

- Allora si aggiunge l'item [E → •E+(E), +] e l'item [E → •int, +],

perché si aggiunge per ogni produzione, ogni item formato dalla stessa produzione come core e un simbolo diverso dei First(Ba).

------------------------------------------------------

In questo caso ho già finito, se esegui l'algoritmo anche sugli item rimanenti e su quelli futuri non ne vengono creati di nuovi.

Quindi il primo stato contiene i seguenti item:

S → • E, $

E → • E+(E), $

E → • int, $

E → • E+(E), +

E → • int, +

**Transizioni fra stati:**

uno stato che contiene l'item [X → α•yβ, b], ha una transizione etichettata y verso un nuovo stato, che formato dagli item della funzione Transition(State, y)

Dove y può essere sia un terminale che una variabile.

Transition(State, y)

Items ← ∅

for each [X → α • yβ, b] ∈ State

add [X → αy • β, b] to Items

return Closure(Items)

Praticamente si fa andare il pallino avanti in ogni item, tale che il pallino di ognuno è prima y, e poi si fa la chiusura di quell'insieme di item

**Shift / Reduce Conflicts:**

Se uno stato del DFA dell' LR(1) contiene sia

[X → α • aβ, b] che [Y → γ• , a]

allora su input a può fare

- shift: [X → α a•β, b]

- reduce: [Y → γ• , a]

Esempio: Dangling else:

[S → if E then S • , else]

[S → if E then S • else S, $]

Su input else è possibile fare sia shift che reduce.

La stessa cosa vale anche per la grammatica:  
  
E → E + E | E \* E | int

**Gli stati del DFA conterranno questi item:**

[E → E \* • E, +]

[E → • E + E, +]

[E → E \* E•, +]

[E → E• + E, +]

In questo caso basta assegnare una precedenza fra le azioni shift / reduce (es il \* ha precedenza sul +)

**Reduce / Reduce Conflict:**

Se uno stato del DFA contiene sia l'item

[X → α • , a] che [Y → β • , a]

il parser LR(1) non sa quale riduzione fare.

Dipende dalle grammatiche scritte male.

Esempio:

S → ε | id | id S

ha 2 parse tree per produrre la stringa id:

S → id

S → id S → id

**LALR:**

Come LR(1), solo che raggruppa insieme gli stati che hanno lo stesso core, si mettono insieme i look-ahead.

Può generare più conflitti di tipo reduce / reduce, quindi i linguaggi accettati da LALR sono inferiori rispetto a LR(1), però il DFA ha un fattore 10 di stati in meno rispetto al DFA di LR(1).