Parsing

This part of the course is dedicated to the parsing problem.

All the material presented can also be found 4th chapter of the textbook:

Compilers: Principles, Techniques, and Tools (2nd Edition)
by Aho, Lam, Sethi, Ullman
Pearson Education.

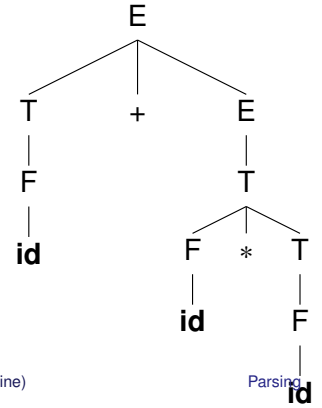
The italian edition has title:
Compilatori. Principi, tecniche e strumenti (seconda editione).

Parsing: example

From a grammar:

$$\begin{array}{lcl} E & \rightarrow & T \mid T + E \\ T & \rightarrow & F \mid F * T \\ F & \rightarrow & \mathbf{id} \mid (E) \end{array}$$

and a string: $\mathbf{id} + (\mathbf{id} * \mathbf{id})$;
build the derivation tree:



Parsing

A fundamental step in the compilation of a program:

- lexical analysis: to recognize token, regular languages, lexer, lex;
- syntax analysis: to construct the derivation tree, context-free languages, parser, Yacc, Bison;
- intermediate code generation;
- ...

Subjects already presented in other courses:

- Foundations of Computer Science (Fondamenti dell'Informatica): (theory) context-free grammars, non-deterministic pushdown automata, pumping lemma, decidability results;
- Programming Languages (Linguaggi di Programmazione): (practice) production and semantics rules in software tools.

What we add in these lectures

A precise description of the algorithm used for parsing

- Programming Languages: description of the software tools,omitting the description of the used algorithms,
  (while the algorithms for lexical analysis are presented).
- Foundations of Computer Science: the presented algorithms are not sufficiently efficient.

In detail:

- we present several parsing algorithms;
- with different level of complexness and efficiency;
- several subsets of context free languages recognizable in linear time;
- algorithms used in software tools.

Parsing algorithms

- Parser top-down: the derivation tree is built starting from the root;
  - non-deterministic, with backtracking: general but not efficient,
  - predictive parsing: possible only for a limited set of languages.
- Parser bottom-up: builds the derivation tree starting from the leaves: deterministic,
  more complicated, but also more general: able to parse a larger class of languages.
  the most used in practice.

Parsing top-down non-deterministico

Costruisco l'albero di parser:

- a partire dalla radice, simbolo iniziale,
- espando sempre il non-terminale più a sinistra
  - uso in maniera parallela tutte le produzioni
  - uso le produzioni in un certo ordine, faccio backtracking in caso di fallimento
- se creo un terminali a sinistra, controllo che questi coincidano con quelli presenti nella stringa,
  se no fallimento.

Stesse idee degli automi a pila non-deterministici.

Esempio

Grammatica:

$$\begin{array}{lcl} E & \rightarrow & T \mid T + E \\ T & \rightarrow & F \mid F * T \\ F & \rightarrow & \mathbf{id} \mid (E) \end{array}$$

Stringa:

$$\mathbf{id} * \mathbf{id}$$

Esempio

Grammatica:

$$\begin{array}{lcl} E & \rightarrow & T \mid T + E \\ T & \rightarrow & F \mid F * T \\ F & \rightarrow & \mathbf{id} \mid (E) \end{array}$$

Stringa:

$$(\mathbf{id} + \mathbf{id})$$

Pseudo-codice

Per ogni non-terminale A definiamo:

```
function parse_A () {
  choose a production  A -> X1 ... Xn
     parse_X1 ();
      .
      .
     parse_Xn ();
}
```

Per ogni terminale t definiamo

```
function parse_t () {
  x = read_token ();
  if (x=t) then return ()
           else error
}
```


Pseudo-codice

Programma:

```
parse () {
  parse_S ();
  match (eof)
}
```

Come implemento l'istruzione

```
choose a production  A -> X1 ... Xn
```

- Esecuzione parallela: automi a pila non deterministici
- Backtracking: può portare a tempi esponenziali.

Trasformazioni di grammatiche

Il parser con backtracking è sensibile al modo in cui è definita la grammatica del linguaggio

- lo stesso linguaggi definito da grammatiche diverse
- per grammatiche ambigue il parsing non ha soluzione unica
- con alcune grammatiche il parsing può non terminare (ricorsione sinistra)
- modificando la grammatica si riducono le scelte da svolgere (left factoring)

Vedremo come trasformare la grammatiche di un linguaggio per ottenere un parser-top down predittivo,
nessuna scelta, complessità lineare (non più potenzialmente esponenziale)

Ricorsione sinistra

Una grammatica è ricorsiva a sinistra se esiste una derivazione del tipo

$$A \rightarrow^{+} A\alpha$$

Esempio:

$$\begin{array}{lcl} E & \rightarrow & T \mid T + E \\ T & \rightarrow & F \mid F * T \\ F & \rightarrow & \mathbf{id} \mid (E) \end{array}$$

non ricorsiva a sinistra

Ricorsione sinistra

Una grammatica è ricorsiva a sinistra se esiste una derivazione del tipo

$$A \rightarrow^{+} A\alpha$$

Esempio:

$$\begin{array}{lcl} E & \rightarrow & T \mid \quad E + T \\ T & \rightarrow & F \mid \quad T * F \\ F & \rightarrow & \mathbf{id} \mid (E) \end{array}$$

ricorsiva a sinistra

Le due grammatiche associano $+$ e $*$ in maniera differente.
Non sono esattamente equivalenti.

Problemi ricorsione sinistra

L'algoritmo di parsing (con backtracking) può divergere se la grammatica ha ricorsione sinistra.

Esempio:

$$\mathbf{id} + \mathbf{id}$$

Eliminare la ricorsione immediata a sinistra

Caso base se A ha produzioni

$$A \rightarrow A\alpha \mid \beta$$

Viene aggiunto un non terminale A' e le produzioni sostituite da

$$\begin{array}{lcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon \end{array}$$

Entrambe le grammatiche da A derivano $\beta\alpha^*$

Esempio

La grammatica ricorsiva a sinistra

$$\begin{array}{lcl} E & \rightarrow & T \mid E + T \\ T & \rightarrow & F \mid T * F \\ F & \rightarrow & \mathbf{id} \mid (E) \end{array}$$

diventa

$$\begin{array}{lcl} E & \rightarrow & T\,E' \\ E' & \rightarrow & +\,T\,E' \mid \epsilon \\ T & \rightarrow & F\,T' \\ T' & \rightarrow & *\,F\,T' \mid \epsilon \\ F & \rightarrow & \mathbf{id} \mid (E) \end{array}$$

Ricorsione sinistra immediata

In generale, se le produzioni per A sono:

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

Viene aggiunto un non-terminale A' e le produzioni sostituite da

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

Eliminazione ricorsione sinistra

L'algoritmo descritto nel dragon-book.
Si definisce un ordine lineare $\leq$ tra i non terminali e in maniera incrementale si riscrivono le produzioni

$$A \rightarrow B\beta$$

se $B \leq A$

Parsing top-down predittivo

Si elimina il non-determinismo del parser top-dow non deterministico.
Con un lookahead di n caratteri (osservando gli n caratteri successivi della stringa non ancora analizzata) si determina la produzione top-down da utilizzare.
Non tutte le grammatiche ammettono un parser top-down predittivo, con lookahead di n simboli.
LL(n) è classe di grammatiche che lo ammettono

- esamina la stringa Left to right
- si cerca la Leftmost derivation, a partire dal simbolo iniziale.
- si usano n simboli di lookahead.

si ha che:

$$LL(1) \subset LL(2) \subset \ldots \subset \texttt{grammatichelibere}$$

Considereremo principalmente le grammatiche LL(1).

Fattorizzazione sinistra

Per un parsing predittivo produzioni nella forma
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$
risultano problematiche,
se α produce una stringa di m terminali devo avere un lookahead di almeno $m+1$ caratteri per decidere la produzione da utilizzare.

Possono essere eliminate riscrivendo le produzioni:

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \mid \gamma \\ A' & \rightarrow & \beta_1 \mid \beta_2 \end{array}$$

Esempio

La grammatica per le espressioni aritmetiche:

$$\begin{array}{lcl} E & \to & T \mid T + E \\ T & \to & F \mid F * T \\ F & \to & \mathbf{id} \mid (E) \end{array}$$

viene riscritta come:

$$\begin{array}{lcl} E & \to & TE' \\ E' & \to & \epsilon \mid +E \\ T & \to & FT' \\ T' & \to & \epsilon \mid *T \\ F & \to & \mathbf{id} \mid (E) \end{array}$$

Funzione FIRST

Un parser predittivo deve scegliere tra le produzioni di un non-terminale

$$A \to \alpha_1 \mid \ldots \mid \alpha_n$$

è utile sapere con quali terminali iniziano le stringhe derivate da α_i.

$$\begin{array}{lcll} \texttt{FIRST}(a) & ::= & \{a\} & a \text{ terminale} \\ \texttt{FIRST}(A) & ::= & \bigcup_{A \to \alpha} \texttt{FIRST}(\alpha) & A \text{ non-terminale} \\ \texttt{FIRST}(\epsilon) & ::= & \{\epsilon\} & \text{stringa vuota} \\ \texttt{FIRST}(X_1 \ldots X_n) & ::= & \texttt{FIRST}(X_1) & \epsilon \notin \texttt{FIRST}(X_1) \\ \texttt{FIRST}(X_1 \ldots X_n) & ::= & \texttt{FIRST}(X_1) \cup \texttt{FIRST}(X_2 \ldots X_n) & \epsilon \in \texttt{FIRST}(X_1) \end{array}$$

Definizione ricorsiva, soluzione minima.

Funzione FOLLOW

Nel caso $A \to \alpha$, $\epsilon \in \texttt{FIRST}(\alpha)$, è utile conoscere quali terminali possono seguire A in una derivazione,

ossia l'insieme $\texttt{FOLLOW}(A)$ formato dai terminali b per cui esiste una derivazione

$$S \to^* \beta A b \gamma,$$

Infatti, se $b \in \texttt{FOLLOW}(A)$ ed in un parsing devo espandere A con un lookahead b, allora la produzione $A \to \alpha$ è un possibile candidato.

Definizione ricorsiva di FOLLOW

I più piccoli insiemi tali che:

$$\begin{array}{lcll} eof \in \texttt{FOLLOW}(S) & & & S \text{ simbolo iniziale} \\ \texttt{FOLLOW}(A) & ::= & \bigcup_{B \to \alpha A \beta} \texttt{FIRST}(\beta) & \\ & & \cup \bigcup_{B \to \alpha A \gamma} \texttt{FOLLOW}(B) & \epsilon \in \texttt{FIRST}(\gamma) \\ & & -\{\epsilon\} & \end{array}$$

possono essere calcolati in maniera incrementale assegnando, inizialmente, per ogni non-terminale B $\texttt{FOLLOW}(B) = \emptyset$ e aggiungendo in seguito nuovi elementi in base alle produzioni.

Esempio:

La grammatica

$$\begin{array}{lcl} E & \rightarrow & T\,E' \\ E' & \rightarrow & +\,T\,E' \mid \epsilon \\ T & \rightarrow & F\,T' \\ T' & \rightarrow & *\,F\,T' \mid \epsilon \\ F & \rightarrow & \textbf{id} \mid (E) \end{array}$$

porta a:

$$\texttt{FIRST}(E) = \texttt{FIRST}(T) = \texttt{FIRST}(F) = \{\,(, id\,\}$$
$$\texttt{FIRST}(E') = \{\,+, \epsilon\,\}$$
$$\texttt{FIRST}(T') = \{\,*, \epsilon\,\}$$

$$\texttt{FOLLOW}(E) = \texttt{FOLLOW}(E') = \{\,), \$\,\}$$
$$\texttt{FOLLOW}(T) = \texttt{FOLLOW}(T') = \{\,+,), \$\,\}$$
$$\texttt{FOLLOW}(F) = \{\,+, *,), \$\,\}$$

Definizione di grammatiche LL(1)

Una grammatica G è LL(1) se per ogni coppia di produzioni $A \rightarrow \alpha \mid \beta$

$$\texttt{FIRST}(\alpha) \cap \texttt{FIRST}(\beta) = \emptyset$$

$$\epsilon \in \texttt{FIRST}(\alpha) \;\Rightarrow\; \texttt{FIRST}(\beta) \cap \texttt{FOLLOW}(A) = \emptyset$$

Ogni grammatica LL(1) ammette un parser predittivo con un unico simbolo di lookahead.

Parser predittivo

Nella funzione

```
function parse_A () {
  choose a production  A -> X1 ... Xn
     parse_X1 ();
      .
      .
     parse_Xn ();
}
```

posso scegliere la produzione in base al simbolo di lookahead.

Parser predittivo

Con singolo di lookahead l scelgo la produzione

$$A \rightarrow \alpha$$

tale che:

$$l \in \texttt{FIRST}(\alpha)$$

o

$$\epsilon \in \texttt{FIRST}(\alpha) \;\wedge\; l \in \texttt{FOLLOW}(A)$$

La condizione LL(1) assicura che ci sia al più una tale produzione.

Parsing table

Una matrice M che mi indica le produzioni da utilizzare nel parsing top-down.
In base al

- non-terminale da espandere A
- il terminale di lookahead a

Descrive quale produzione $A \rightarrow \alpha$ utilizzare.

$$M[A, a] := A \rightarrow \alpha$$

Se

- $a \in \texttt{FIRST}(\alpha)$ oppure
- $a \in \texttt{FOLLOW}(A)$ e $\epsilon \in \texttt{FIRST}(\alpha)$

La condizione LL(1) assicura che, per ogni coppia A, a ci sia al più una produzione possibile.

Esempio di parsing table

consideriamo la grammatica

$$\begin{array}{lcl} E & \rightarrow & T\,E' \\ E' & \rightarrow & +\,T\,E' \mid \epsilon \\ T & \rightarrow & F\,T' \\ T' & \rightarrow & *\,F\,T' \mid \epsilon \\ F & \rightarrow & \textbf{id} \mid (E) \end{array}$$

Esempio di parsing table

consideriamo la grammatica

$$\begin{array}{lcl} E & \rightarrow & T\,E' \\ E' & \rightarrow & +\,T\,E' \mid \epsilon \\ T & \rightarrow & F\,T' \\ T' & \rightarrow & *\,F\,T' \mid \epsilon \\ F & \rightarrow & \textbf{id} \mid (E) \end{array}$$

	id	+	*	(	)	$
E	E→ TE`			E→ TE`		
E`		E`→ +TE`			E`→ ε	E`→ ε
T	T→ FT`			T→ FT`		
T`		T`→ ε	T→ *FT`		T`→ ε	T`→ ε
F	F→ id			F→ (E)		

Esempio di parsing table

consideriamo la grammatica

$$\begin{array}{lcl} E & \rightarrow & T\,E' \\ E' & \rightarrow & +\,T\,E' \mid \epsilon \\ T & \rightarrow & F\,T' \\ T' & \rightarrow & *\,F\,T' \mid \epsilon \\ F & \rightarrow & \textbf{id} \mid (E) \end{array}$$

	id	+	*	(	)	$
E	E→ TE`			E→ TE`		
E`		E`→ +TE`			E`→ ε	E`→ ε
T	T→ FT`			T→ FT`		
T`		T`→ ε	T→ *FT`		T`→ ε	T`→ ε
F	F→ id			F→ (E)		

Parser non ricorsivo, automa a pila

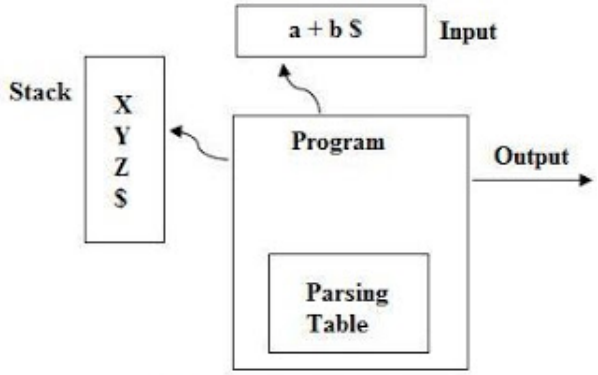


Fig 2.6 Model of a predictive parser

Input:

- la stringa ancora da analizzare,
- una pila contenente la parte finale della stringa derivata, mediante leftmost produzione,
- un tabella di parsing.

Parser automa a pila

Ad ogni passo l'automa può:

- accettare un token della stringa di input, se presente in testa all input e alla pila
- modificare la pila mediante la produzione indicata dalla tabella di parsing
- generare errore, se non esiste nessuna produzione nella tabella di parsing per gli input attuali
- terminare con successo se pila e stringa di ingresso sono entrambe vuote.

Predictive parsing, iterative program

```
repeat
  begin
    let X be the top stack symbol and a the next input;
    if X is a terminal or $ then
      if X = a then
        pop X from the stack and remove a from the input
      else  ERROR( )
    else /* X is a nonterminal */
      if M[X,a] = X -> Y1, Y2, ... , Yk then
        begin
          pop X from the stack;
          push Yk, Yk-1, ... ,Y1 onto the stack, Y1 on top
        end
      else  ERROR( )
  end
until stack is empty
```


Esempio parsing

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \to TE'$
$E'T'F	id + id * id$	$T \to FT'$
$E'T'id	id + id * id$	$F \to id$
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \to \epsilon$
$E'T+	+ id * id$	$E' \to +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \to FT'$
$E'T'id	id * id$	$F \to id$
$E'T'	* id$	
$E'T'F*	* id$	$T' \to *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \to id$
$E'T'	$	
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

Parser LL(1)

Riassumendo:

- parsing in tempo lineare, esamino una volta sola la stringa d'ingresso;
- relativamente semplice costruire, a mano, la tabella di parsing;
- algoritmo non troppo difficile da comprendere e motivare;
- svantaggio principale: poche grammatiche rientrano nella classe LL(1).

Parser LL(K)

La metodologia precedente può essere estesa per costruire un parser LL(K)

- la funzioni `FIRST` e `FOLLOW` restituiscono insiemi di stringe lunghe fino a k caratteri,
- la condizione LL(K) per una grammatica è una variazione più complessa della condizione LL(1)

Principalmente ci si limita alle grammatiche LL(1)

Error recovering

Several types of errors in a program:

- lexical,
- syntactic,
- semantiics,
- logical.

Compilers have to deal with errors by:

- generating error messages,
- continuing the program analysis.

Syntactic errors

Parser need to deal with syntactic errors.
Several techniques:

- Panic mode: to skip some part of the input string till a synchronization item,
- Phase level level recovering: define procedure to be executed when an error in detected,
- Error production: add rules generating incorrect programs, the parsing tree generates the error messages.

Parser LL(1) error recovering

Panic mode:

- if A is in the top of stack, use `FOLLOW`A and synchronization set, and the pop A,
- in hierarchical grammar, use the first character of the next level
  - inside a command: characters terminating a command ;
  - inside an expression: characters terminating an expression)
  - inside a block: tokens closing a block `end` }
- one can also insert `FIRST`(A) in the synchronization set, omitting the pop action.

Parser LL(1) error recovering

Phase level:
introduce actions in the empty elements of the parsing table,
some simple examples:

- skip elements of the inputs, till the next synchronization item;
- insert elements into the input;
- pop or push elements in the stack;
- more elaborated procedures ...

There is no exact recipe, empirical choices.

Esercizi

Costruire le tabelle di parsing per le grammatiche:

$$\begin{array}{lcl} S & \to & L = R \mid R \\ L & \to & *R \mid \mathbf{id} \\ R & \to & L \end{array}$$

$$\begin{array}{lcl} S & \to & C; S' \\ S' & \to & \epsilon \mid C; S' \\ C & \to & \mathbf{id} = R \\ R & \to & \mathbf{id} \mid \mathbf{num} \end{array}$$

Parser bottom-up

Operano nel verso opposto rispetto ai parser top-down.

A partire da una stringa di non terminali cerco una serie di riduzioni (produzioni applicate al contrario) che mi portano al simbolo iniziale.

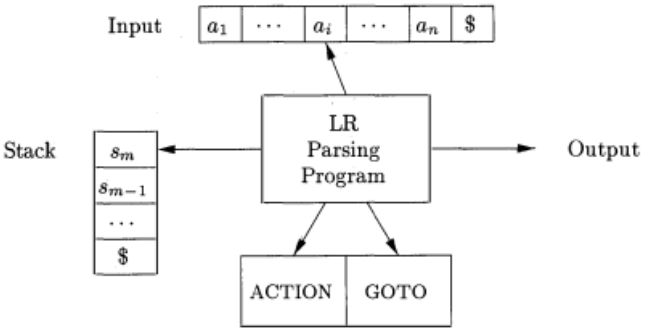
$$\mathbf{id} * \mathbf{id} \leftarrow F * \mathbf{id} \leftarrow T * \mathbf{id} \leftarrow T * F \leftarrow T \leftarrow E$$

La derivazione che ottengo è la rightmost (espando il non-terminale più a destra).

Dalla derivazione posso calcolare l'albero di parsing.

Parsing shift-reduce

Anche un parser bottom-up può essere visto come un automa a pila.



Stack e input ancora da esaminare, definiscono una stringa intermedia (un antecedente), della rightmost derivation, della stringa di input.

Esempio

Consideriamo la grammatica (ricorsiva a sinistra) delle espressioni:

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathbf{id} \end{array}$$

Stack	Input	Action
$	**id** + **id** * **id** $	shift
$ **id**	+ **id** * **id** $	reduce by $F \to$ **id**
$ F	+ **id** * **id** $	reduce by $T \to F$
$ T	+ **id** * **id** $	reduce by $E \to T$
$ E	+ **id** * **id** $	shift
$ E +	**id** * **id** $	shift
$ E + **id**	* **id** $	reduce by $F \to$ **id**
$ E + F	* **id** $	reduce by $T \to F$
$ E + T	* **id** $	shift
$ E + T *	**id** $	shift

Esempio, continua

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathbf{id} \end{array}$$

Stack	Input	Action
$ E + T * **id**	$	reduce by $F \to$ **id**
$ E + T * F	$	reduce by $T \to T * F$
$ E + T	$	reduce by $E \to E + T$
$ E	$	accept

Decisioni

Un parser shift-reduce ad ogni passo deve decidere se:

- inserire un nuovo simbolo in input nella pila: operazione di shift, o
- ridurre la testa della pila applicando una riduzione (produzione al contrario) operazione reduce,

Le decisioni sono prese in base al contenuto della pila, e in base allo prossimo token in input.

$$\$\ E + T \quad * \mathbf{id}\ \$ \quad \text{shift}$$

$$\$\ E + T \quad \$ \quad \text{reduce by } E \to E + T$$

Decisioni

In alcuni casi casi ci possono essere piu produzioni possibili:

$$\$\ E + T * F \quad \$ \quad \text{reduce by } T \rightarrow T * F$$
$$\$\ E + T$$

$$\$\ E + T * F \quad \$ \quad \text{reduce by } T \rightarrow F$$
$$\$\ E + T * T$$

Solo una è corretta.

Algortimi LR

Devo defire un algortimo, che in base al contenuto della pila, sceglie il passo (shift-reduce) da eseguire.

Metodo.
Associo ad ogni parte di iniziale delle pila uno stato che:

- riassume l'informazione contenuto della pila,
- indica quali sono le riduzioni possibili,
- nel caso ci siano piu riduzioni indicando quella da scegliere,
- un numero finito di stati possibili,
- stati menorizzati nella pila,
- calcolo incrementale, se introduco un nuovo elemento nella pila, in base allo stato e al nuovo elemento, posso calcolare il prossimo stato.

Tre tipi di parser

Tre tipi di stati (informazioni aggiuntive) che inserisco nella pila di un parser shift-reduce.
Tre tecniche per dirimere i conflitti.
Tre tipi di parser.

- parser SLR (Simple LR), meno informazione, più semplici da costruire, trattano un insieme ristretto di linguaggi.
- parser LR: più informazione, più complessi da costruire, trattano un insieme più ampio di linguaggi.
- parser LALR (LookAhead LR): un buon compromesso tra semplicità e generalità.

Linguaggi LR(n)

- L esamino la stringa from Left to right
- R costruisco la Rightmost derivation
- n con n simboli di lookahead.

Parsing SLR [DeRemer 1969]

Lo stato, l'informazione aggiuntiva: un insieme di produzioni (più qualche altra informazione)

Una produzione appartiene allo stato se:

- corrisponde alla prima riduzione di una catena di riduzioni, tale che:
  - trasforma la stringa presente in pila, eventualmente seguita da altri simboli;
  - nel simbolo iniziale della grammatica.
  - La catena di riduzione è l'inverso di una rightmost derivation (leftmost reduction);
- la riduzione viene applicata alla testa pila e/o a un gruppo di simboli immediatamente a destra di essa.

Esempio

Con la grammatica

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \textbf{id} \end{array}$$

se la pila contiene i simboli $E + T$,
lo stato è formato dalle produzioni:

- $E \rightarrow E + T$ ovviamente;
- $T \rightarrow T * F$, si applica se la pila è seguita dalla stringe $*F$.

non devo inserire la produzione $E \rightarrow T$ infatti se applicata
porta alla stringa $E + E$ e impedisce la riduzione al simbolo iniziale.

Non ci sono altre produzioni nello stato.

Esempio

Con la stesa grammatica

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \textbf{id} \end{array}$$

se la pila contiene $E +$
lo stato contiene:

- $E \rightarrow E + T$, nel caso la pila sia seguita da T;
- $T \rightarrow F$, nel caso la pila sia seguita da F;
- $T \rightarrow T * F$, nel caso la pila sia seguita da $T * F$;
- e le produzioni $F \rightarrow (E)$ e $F \rightarrow T$.

Item

Più precisamente, lo stato contiene, oltre alle produzioni,
un dato su quanta parte della parte destra di ciascuna produzione
(handle) è già presente in pila,
questo dato viene rappresentato con un punto che separa la parte
presente in pila da quella ancora da inserire.

Definizione

Un item LR(0) di una grammatica G e' un espressione $A \rightarrow \alpha \cdot \beta$ con
$A \rightarrow \alpha\beta$ produzione in G.

Se $A \rightarrow XY$ è una produzione allora $A \rightarrow \cdot XY$, $A \rightarrow X \cdot Y$, $A \rightarrow XY\cdot$,
sono i tre item generati.

Esempio

Con la grammatica

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \textbf{id} \end{array}$$

se la pila contiene $E +$
lo stato contiene gli item:

- $E \rightarrow E + \cdot T$, nel caso la pila sia seguita da T;
- $T \rightarrow \cdot F$, nel caso la pila sia seguita da F;
- $T \rightarrow \cdot T * F$, nel caso la pila sia seguita da $T * F$;
- e gli item $F \rightarrow \cdot(E)$ e $F \rightarrow \cdot T$.

Definizione formale di stato e prefisso ammissibile

Definizione

Una stringa $\alpha\beta$ è un prefisso ammissibile (viable prefix) per un item $A \to \beta \cdot \gamma$ se esiste una stringa di terminali w e una rightmost derivation

$$S \to^*_{rm} \alpha A w \to_{rm} \alpha\beta\gamma w$$

Definizione

Lo stato che descrive una pila contenente la stringa α è l'insieme degli item $A \to \beta \cdot \gamma$ per cui $\alpha = \alpha_1\beta$ è un prefisso ammissibile.

Come calcolare gli stati: Closure

Non tutti gli insiemi di item possono formare uno stato.
Se uno stato contiene un item, lo stesso stato deve contenere anche altri item.

Definizione

Dato un insieme I di item, la sua chiusura $\texttt{CLOSURE}(I)$ è il più piccolo insieme tale che

- $I \subseteq \texttt{CLOSURE}(I)$
- se $(A \to \alpha \cdot B\beta) \in \texttt{CLOSURE}(I)$ e $B \to \gamma$ allora $B \to \cdot\gamma \in \texttt{CLOSURE}(I)$

Un insieme e' chiuso se coincide con la sua chiusura.

Gli stati dell'automa sono sottoinsiemi chiusi di item.

Esempio

Sulla grammatica:

$$\begin{array}{lcl} E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \textbf{id} \end{array}$$

La chiusura di $\{ E \to E + \cdot T \}$ è l'insieme:

$$\begin{array}{rl} \texttt{CLOSURE}(\{ E \to E + \cdot T \}) = & \{ E \to E + \cdot T, \\ & T \to \cdot T * F,\ T \to \cdot F, \\ & F \to \cdot(E),\ F \to \cdot\textbf{id} \} \end{array}$$

Se in testa alla pila è presente la stringa $E +$,
ed è corretto applicare la riduzione $E \to E + T$
(nel caso la pila sia seguita dal simbolo T)
ma potrebbe essere necessario applicare la prima la riduzione $T \to F$
(nel caso la pila sia seguita dal simbolo F).

Grammatica aumentata

Per semplificare la condizione di terminazione del parser LR, si definisce la semantica aumentata caratterizzata da:

- un nuovo simbolo iniziale S'
- con una singola produzione $S' \to S$, dove S è simbolo iniziale originale.

La grammatiche aumentata delle espressioni aritmetiche è

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \textbf{id} \end{array}$$

Funzione GOTO

Dato uno stato I, che descrive una pila con α,
e un simbolo X,
posso calcolare lo stato che descrive la pila αX.

Lo stato successivo è definito dalla funzione GOTO

Definizione

Dato un insieme chiuso I di item e un simbolo della grammatica X, definiamo $\texttt{GOTO}(I, X)$ come

$$\texttt{CLOSURE}(\{A \rightarrow \alpha X \cdot \beta \mid A \rightarrow \alpha \cdot X\beta \in I\})$$

Esempio

Sia I lo stato:

$$\{ E \rightarrow E + \cdot T,\ T \rightarrow \cdot T * F,\ T \rightarrow \cdot F,\ F \rightarrow \cdot(E),\ F \rightarrow \cdot \textbf{id} \}$$

$\texttt{GOTO}(I, T)$ è lo stato:

$$\{ E \rightarrow E + T \cdot,\ T \rightarrow T \cdot * F \}$$

$\texttt{GOTO}(I, F)$ è lo stato:

$$\{ T \rightarrow F \cdot \}$$

$\texttt{GOTO}(I, ( )$ è lo stato:

$$\texttt{CLOSURE}(\{ F \rightarrow (\cdot E) \})$$

$\texttt{GOTO}(I, \textbf{id})$ è lo stato:

$$\{ F \rightarrow \textbf{id} \cdot \}$$

Con gli altri simboli Y la funzione $\texttt{GOTO}(I, Y)$ genera l'insieme vuoto e segnala una condizione di errore.

Collezione canonica

L'insieme stati di un automa SLR per una grammatica G è data dalla collezione canonica di insieme di item, insieme ottenuto

- partendo dallo stato iniziale $\texttt{CLOSURE}(\{S' \rightarrow \cdot S\})$
  All'inizio del parsing, la pila vuota, e ci sia aspetta che la stringa in input, dopo una catena di riduzioni, permetta le riduzione $S' \rightarrow S$.
- aggiungendo tutti gli stati di item ottenibili applicando la funzione GOTO.

L'insieme generato dalla stato iniziale e dalla funzione GOTO

Automa SRL (non compattato)

Parser shift-reduce:

- che oltre ad un simboli della grammatica, inserisce di volta in volta nella pila uno stato (collezione di item)
- usa l'informazione dello stato I, ed un simbolo di lookahead a, per decidere l'azione da svolgere (shift o reduce)

Shift

- Quando: $\texttt{GOTO}(I, a) \neq \emptyset$, ossia I contiene un item nella forma $A \rightarrow \alpha \cdot a\beta$
- Come: consumo il simbolo a dall'input, inserisco in pila in simbolo a e lo stato $\texttt{GOTO}(I, a)$

Reduce, con la riduzione $A \rightarrow \alpha$

- Quando: I contiene l'item $A \rightarrow \alpha \cdot$, $a \in \texttt{FOLLOW}(A)$
- Elimino dalla pila $|\alpha|$ simboli e stati e inserisco il simbolo A e $\texttt{GOTO}(I, A)$, se in I è lo stato in testa alla pila dopo l'eliminazione.

Linguaggio SRL

La costruzione precedente può creare conflitti:

- conflitti shift-reduce: per una coppia I, a, è ammissibile sia un azione di shift, che una reduce.
  I contiene due item, uno nella forma $A \to \alpha \cdot a\beta$, e l'altro nella forma $A' \to \alpha' \cdot$, con $a \in \texttt{FOLLOW}(A')$
- conflitti reduce-reduce: per una coppia I, a, ci sono due riduzioni possibili.

Se non ci sono esistono conflitti, la grammatica si definisce SRL, può essere riconosciuta da un automa SRL.

Automa SRL (compatto)

Per motivi efficienza, l'automa SRL è implementato in modo diverso rispetto alla descrizione precedente

- Uno stato I descrive anche i simboli presenti in testa alla pila.
  Se in testa alla pila c'è I e $A \to \alpha \cdot \beta \in I$,
  allora in testa alla pila deve esserci la stringa di simboli α.
  Nella pila posso inserire solo stati, simboli ridondanti.
- Uno stato può essere codificato con un numero.
  Costruisco la collezione canonica, e numero gli stati ottenuti.
- Le azioni che l'automa deve compire possono essere riassunte in due tabelle
  ACTION
  GOTO

ACTION

Tabella con

- righe con indice stato I (codificati da numeri),
- colonne con indice simboli non terminali a.

Dato lo stato I e il simbolo di lookahead a la tabella indica l'azione da compiere.

- shift I', (s I'), vai uno shift e inserisci in pila lo stato I',
  ($I' = \texttt{GOTO}(I, a)$)
- reduce n, (r n), applica un riduzione secondo la produzione n, (anche le produzione vengono numerate)

Riduzione, GOTO

Applicare la riduzione n corrispondente alla produzione $A \to \alpha$ consiste in

- rimuove $|\alpha|$ stati dalla pila,
- inserire lo stato $\texttt{GOTO}(I, A)$, dove I stato in testa alla pila dopo rimozione,

Tabella GOTO, associa uno stato ad ogni coppia stato, simbolo non-terminale.

Costruzione automa SLR

- Grammatica aumentata, numero le produzioni.
- Collezione canonica, numero gli stati.
- Calcolo la funzione `FOLLOW` e di conseguenza anche la `FIRST`
- Costruisco la tabella GOTO (stato, terminale)
- Costruisco la tabella ACTION (stato, non-terminale)

Parsing LR(1) (Donald Knuth 1965)

Come il parsing SRL, ma gli stati contengono più informazione

Debolezza del parsing SRL:
Nello stato I, con lookahead a, (e simboli della pila che formano la stringa $\alpha'\alpha$)
viene suggerita l'azione reduce, secondo la produzione $A \to \alpha$ se:

- $A \to \alpha\cdot \in I$
- $a \in \texttt{FOLLOW}(A)$

$\texttt{FOLLOW}(A)$ funzione generale: restituisce tutti i simboli che possono seguire A in una qualsiasi derivazione,

Un analisi più accurata dovrebbe limitarsi alle derivazioni che generano una stringa iniziale $\alpha' A$
È possibile che non esista nessuna derivazioni $S \to^* \alpha' A a \gamma$ (per un qualsiasi γ).

Il parsing *SRL* indica delle riduzioni che possono essere escluse da un analisi più accurata.

Esempio

Grammatica

$$\begin{array}{lcl} S & \to & L = R \mid R \\ L & \to & *R \mid \textbf{id} \\ R & \to & L \mid \textbf{int} \end{array}$$

Stack	Input	Action
$\{S' \to \cdot S\}$	**id** $=$ **id** \$	shift
" $\{L \to \textbf{id}\cdot\}$	$=$ **id** \$	reduce by $L \to$ **id**
" $\{R \to L\cdot,\ S \to L\cdot = R\}$	$=$ **id** \$	reduce by $R \to L$
" $\{S \to R\cdot\}$	$=$ **id** \$	deadend

Il simbolo $=$ può seguire solo le istanze di R che appaiono in $*R$

Aggiungendo più informazione nello stato risolvo il conflitto shift-reduce nella terza linea.

Item LR(1)

Un item LR(1) e' una coppia item LR(0), simbolo terminale.

Definizione

Un item LR(1) di una grammatica G e' una coppia $(A \to \alpha\cdot\beta,\ a)$ con $A \to \alpha\beta$ produzione in G e a simbolo terminale in G.

Lo stato associato ad una pila $\alpha'\alpha$, è formato dagli item $(A \to \alpha\cdot\beta,\ a)$ tali che esiste una stringa di terminali w e una rightmost derivation nella forma

$$S \to^*_{rm} \alpha' A a w \to_{rm} \alpha'\alpha\beta a w$$

Nell'item LR(1) introduco un'informazione esplicita su quali sono i simboli terminali a che posso seguire l'istanza sintetizzata di A.

Closure LR(1)

Definizione

Dato un insieme I di item, la sua chiusura $\texttt{CLOSURE}(I)$ è il più piccolo insieme tale che

- $I \subseteq \texttt{CLOSURE}(I)$
- se $(A \to \alpha \cdot B\beta, a) \in \texttt{CLOSURE}(I)$, $B \to \gamma$, $b \in \texttt{FIRST}(\beta a)$ allora $(B \to \cdot\gamma, b) \in \texttt{CLOSURE}(I)$

Funzione GOTO

Definizione

Dato uno stato LR(1) I e un simbolo della grammatica X, definiamo $\texttt{GOTO}(I, X)$ come

$$\texttt{CLOSURE}(\{(A \to \alpha X \cdot \beta, a) \mid (A \to \alpha \cdot X\beta, a) \in I\})$$

Stato iniziale

Anche gli automi LR(1) considerano grammatiche aumentate con un nuovo simbolo iniziale S' e una nuova produzione $S' \to S$.

Lo stato iniziale dell'automa LR(1) è

$$\texttt{CLOSURE}(\{(S' \to \cdot S, \$)\})$$

Comportamento del parser

Nello stato I con lookahead a esegue

- shift se $\texttt{GOTO}(I, a) \neq \emptyset$
- reduce se esiste $(A \to \alpha\cdot,\ a) \in I$
- success nel caso particolare $I = \{S' \to S\cdot, \$\}$ e $a = \$$
- error negli altri casi.

Parsing LALR [DeRemer 1969]

Dal punto di vista pratico, i parser LR(1) hanno troppi stati, tabelle ACTION e GOTO troppo ampie.

Preferibile una tecnica di parsing intermedia tra SLR e LR: tabelle di parsing ridotte, insieme sufficientemente ricco di grammatiche trattabili.

Gli stati di un automa LALR sono ottenuto quozientando gli stati dell'automa LR(1).

Parser LALR

Definiamo core di un LR(1) stato I ($[\![I]\!]$) è l'insieme dei LR(0) item in I

$$[\![I]\!] = \{A \rightarrow \alpha \cdot \beta \mid \exists a \ (A \rightarrow \alpha \cdot \beta, a) \in I\}$$

Definiamo due stati equivalenti $I_j \approx I_k$ se hanno lo stesso core $[\![I_j]\!] = [\![I_k]\!]$

La funzione GOTO preserva l'equivalenza tra stati.

$$[\![I_j]\!] = [\![I_k]\!] \Rightarrow [\![\texttt{GOTO}(I_j, X)]\!] = [\![\texttt{GOTO}(I_k, X)]\!]$$

$$I_j \approx I_k \Rightarrow \texttt{GOTO}(I_j, X) \approx \texttt{GOTO}(I_k, X)$$

Parser LALR

Definizione

Gli stati dell'automa LALR sono costituiti dall'unione dei stati LR(1) equivalenti tra loro.

$$[I] = \bigcup_{I_j \approx I} I_j$$

La funzione GOTO è data da:

$$\texttt{GOTO}([I], X) = [\texttt{GOTO}(I, X)]$$

Il comportamento del parser LALR è:
nello stato $[I]$ con lookahead a esegue:

- shift se $\texttt{GOTO}([I], a) \neq \emptyset$
- reduce se esiste $(A \rightarrow \alpha\cdot, \ a) \in [I]$

Proprietà parser LALR

Proposizione

Quozientare non introduce conflitti shift-reduce.
Se l'automa LALR(1), per la grammatica G ha un conflitto shift-reduce, allora anche l'automa LR(1) per G ha un conflitto shift-reduce.

Prova ...
Quozientare può introdurre conflitti reduce-reduce.

Proposizione

Data una grammatica G, gli stati del parser LALR, sono gli stati del parser SLR arricchiti con un informazione sui simboli di follow di una produzione.

Segue dal fatto che:

$$[\![\texttt{GOTO}(I, X)]\!] = \texttt{GOTO}([\![I]\!], X)$$

e dal fatto che l'insieme degli stati è generato dalla funzione GOTO.

Possibili costruzioni parser LALR

- Costruisco l'automa LR(1) e faccio il quoziente. Inefficiente, devo generare un automa con molti stati.
- Nella costruzione del parser LR(1) evito la costruzione di stati con lo stesso core. Se la funzione GOTO genera uno stato con I equivalente ad uno stato J già generato ($I \approx J$), fondo I con J e rifaccio i calcoli della funzione GOTO. Non genere più stati del necessario ma si può fare meglio.
- Costruisco gli stati SLR e aggiungo in seguito l'informazione di prossimo terminale (di lookahead). Costruzione più efficiente.

Costruzione parser LALR

Costruisco il parser SLR.

Devo distinguere tra i simboli di lookahead in $\texttt{GOTO}(I, X)$ in

- propagati, simboli di lookahead in I, che si spostano in $\texttt{GOTO}(I, X)$.
- quelli generati spontaneamente sono presenti indipendentemente dai simboli di lookahead di I, appartengono a qualsiasi $\texttt{GOTO}(J, X)$ con $J \approx I$

Costruzione parser LALR

Aggiungo un nuovo terminale $\#$, lo inserisco come simbolo di lookahead in un item kernel di uno stato *SLR*, attraveros la funzione CLOSURE LR(1) vedo come si propaga agli item non kernel, e attravero la funzione GOTO LR(1) come si propoga ad item kernel di altri stati.

In questo modo ottengo due informazioni:

- L'insieme di lookahead generati spontaneamente, da CLOSURE LR(1).
- Un mappa di come si propagano i simbolo di lookahead, da ogni item kernel.

Calcolo il punto fisso della propagazione (Dopo aver inserito il simbolo di lookahead \$ nello item $S' \to \cdot S$ dello stato iniziale.

Uso di grammatiche ambigue

Una grammatica ambigua G genera sempre parser con conflitti. Una stringa ha più alberi di parsing, quindi la costruzione dell'albero di parsing deve poter prendere due strade diverse.

Grammatiche ambigue più convenienti perché più sintetiche, potenzialmente generano parser più compatti e veloci.

Nell'uso comune le grammatiche ambigue possono essere disambiguate definendo

- un ordine di precedenza tra gli operatori,
- delle regole di associatività (destra o sinistra) per uno stesso operatore.

Nei parser lo stesso effetto può essere ottenuto specificando come risolvere i conflitti, shift-reduce nel parsing.

Senza conflitti di parsing, ad ogni stringa viene associata al più ad un albero.

Error recovering

I parser LR riconoscono un errore subito.
Non appena la parte dell'input letta non ammette nessun completamento corretto, il parser genera errore.
Non faccio operazioni di shift senza la possibilità di terminare con successo.

Due gestioni degli errori:

- panic mode: scelgo un insieme di non terminali principali (statement, expression, block),
  nel caso di errore smonto lo stack sino a trovare uno stato s con $\texttt{GOTO}(s, A)$ per A principale, inserisco lo stato $\texttt{GOTO}(s, A)$ in pila.
  scarico l'input sino a trovare un simbolo in $\texttt{FOLLOW}(A)$,
- recupero a livello di frase: si scrivono delle procedure che per ogni coppia stato I, lookahea a che porta a errore, determinano le azioni da compiere.
  Eventualmente un azione caso particolare dell'azione precedente.