TOP-DOWN PARSING

Recursive-Descent, Predictive Parsing

Prior to top-down parsing

- Checklist :

1. Remove ambiguity if possible by rewriting the grammar
2. Remove left- recursion, otherwise it may lead to an infinite loop.
3. Do left- factoring.

Left- factoring

- In predictive parsing , the prediction is made about which rule to follow to parse the non-terminal by reading the following input symbols
- In case of predictive parsing, left-factoring helps remove removable ambiguity.
- “Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.”

  - Aho,Ullman,Sethi

Left-factoring

- Here is a grammar rule that is ambiguous:

A -> xP1 | xP2 | xP3 | xP4 ….| xPn

Where x & Pi’s are strings of terminals and non-terminals and x !=ε

If we rewrite it as

A-> xP’

P’ -> P1|P2|P3 …|Pn

We call that the grammar has been “left-factored”, and the apparent ambiguity has been removed. Repeating this for every rule left-factors a grammar completely

Example

- stmt -> *if* exp *then* stmt *endif* |
  if exp *then* stmt *endif else* stmt *endif*

We can left factor it as follows :

stmt -> *if* exp *then* stmt *endif* ELSEFUNC
ELSEFUNC -> *else* stmt *endif* | ε (epsilon)

Thereby removing the ambiguity

Parsers: Recursive-Descent

- Recursive, Uses backtracking
- Tries to find a leftmost derivation
- Unless the grammar is ambiguous or left-recursive, it finds a suitable parse tree
- But is rarely used as programming constructs can be parsed without backtracking

Consider the grammar:

$S \longrightarrow cAd \mid bd$

$A \longrightarrow ab \mid a$

and the string “cad”

Recursive parsing with backtracking : example

Following the first rule,
S->cAd to parse S

$$\begin{tikzcd}
& S \arrow[dl] \arrow[d] \arrow[dr] & \\
c & A & d
\end{tikzcd}$$

S->cAd

The next non=term in line A is parsed using first rule, A -> ab , but turns out INCORRECT, parser backtracks

$$\begin{tikzcd}
& S \arrow[dl] \arrow[d] \arrow[dr] & \\
c & A \arrow[dl] \arrow[d] & d \\
a & b &
\end{tikzcd}$$

A -> ab

$$\begin{tikzcd}
& S \arrow[dl] \arrow[d] \arrow[dr] & \\
c & A \arrow[dl] & d \\
a & &
\end{tikzcd}$$

A -> a

Next rule to parse A is taken A->a, turns out CORRECT , Parser stops

Predictive parser

- It is a recursive-descent parser that needs no backtracking
- Suppose

  A -> A1 | A2 | ….| An
- If the non-terminal to be expanded next is ‘A’ , then the choice of rule is made on the basis of the current input symbol ‘a’ .

Procedure

- Make a transition diagram(like dfa/nfa) for every rule of the grammar.
- Optimize the dfa by reducing the number of states, yielding the final transition diagram
- To parse a string, simulate the string on the transition diagram
- If after consuming the input the transition diagram reaches an accept state, it is parsed.

Example

The grammar is as follows

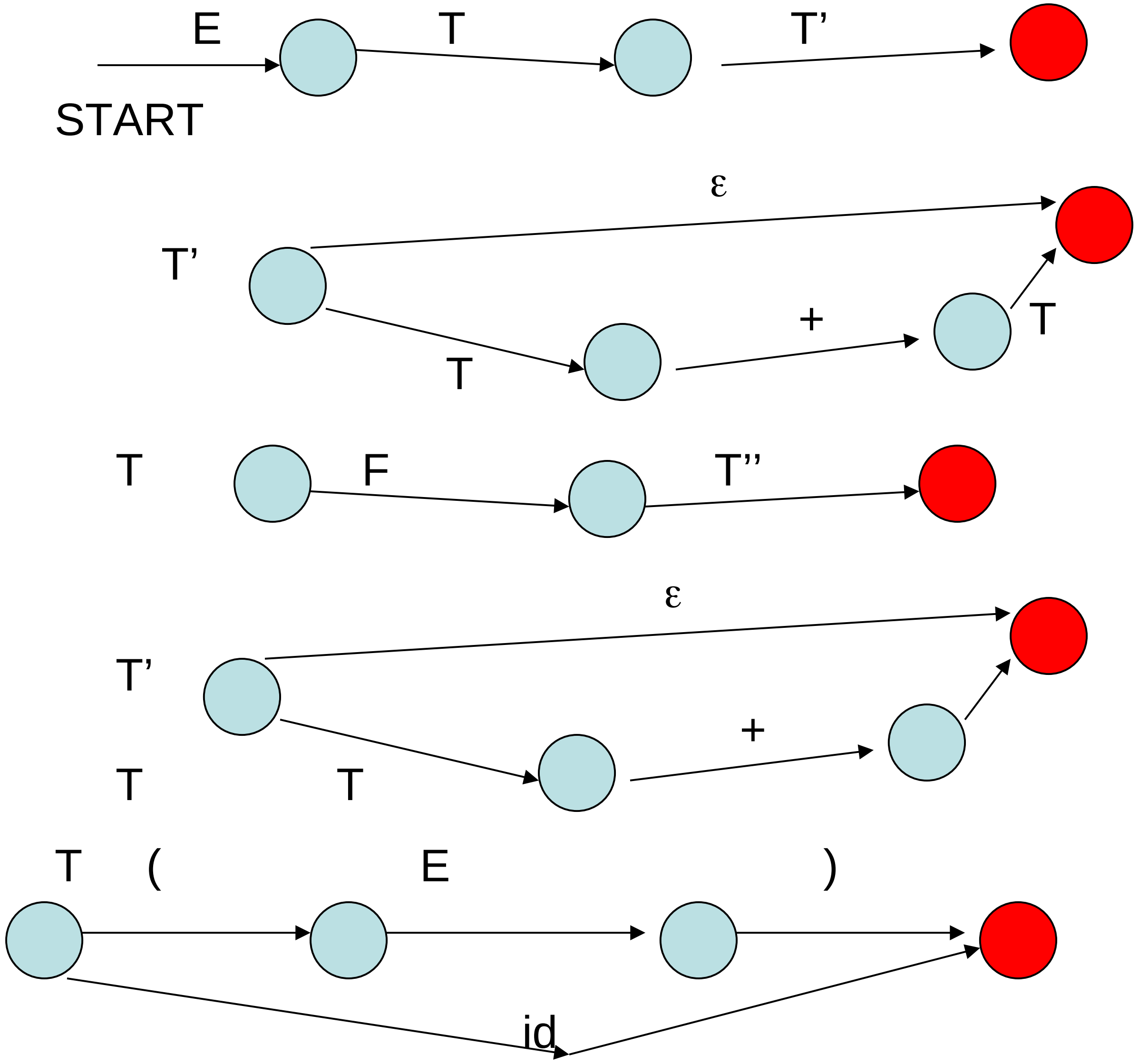
- E -> E + T | T
- T - > T * F | F
- F -> (E) | id

After removing left-recursion , left-factoring

The rules are :

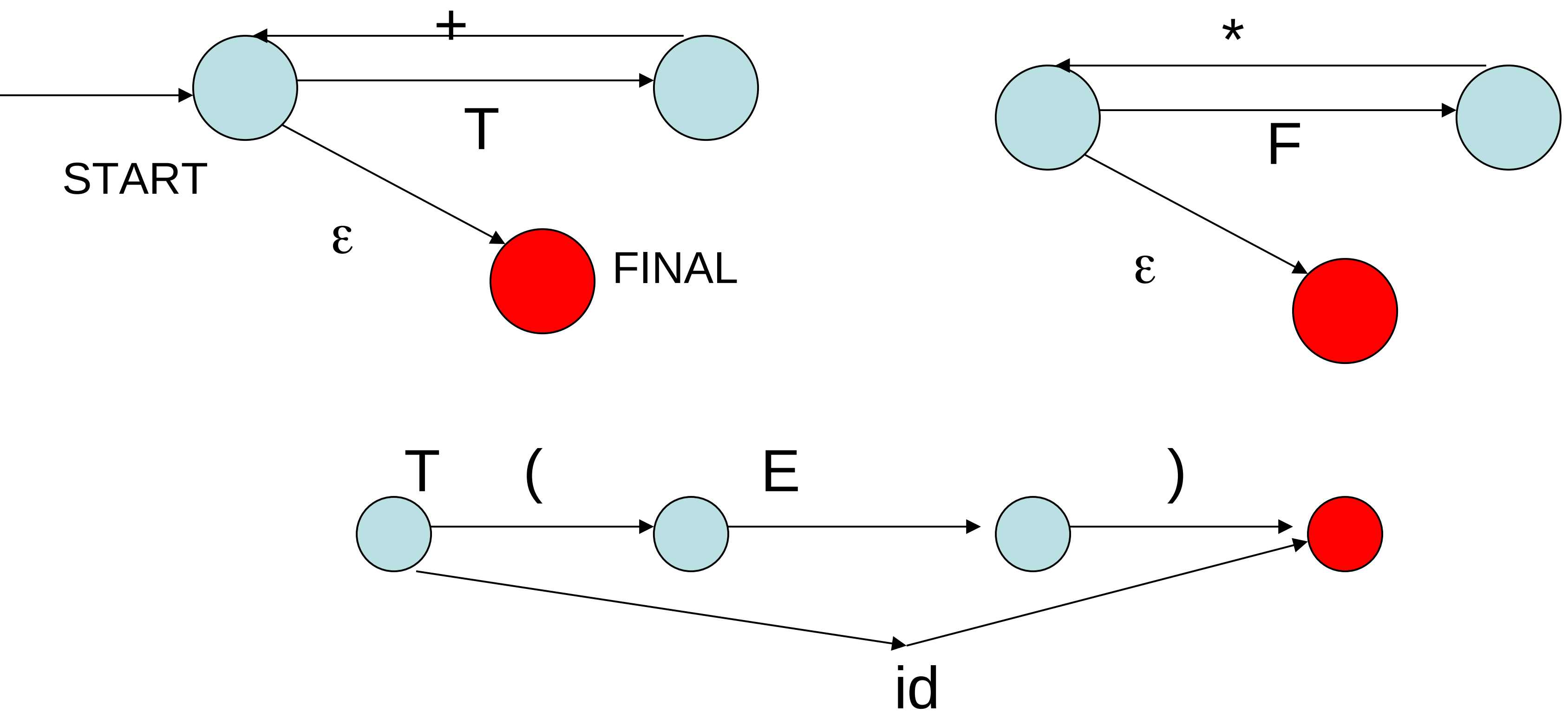
Rules and their transition diagrams

- E->T T'
- T' -> +T T' | ε
- T -> F T''
- T -> *F T'' | ε
- T -> (E) |id



Optimization

After optimization it yields the following DFA like structures:



SIMULATION METHOD

- Start from the start state
- If a terminal comes consume it, move to next state
- If a non – terminal comes go to the state of the “dfa” of the non-term and return on reaching the final state
- Return to the original “dfa” and continue parsing
- If on completion(reading input string completely), you reach a final state, string is successfully parsed.

Disadvantage :

- It is inherently a recursive parser, so it consumes a lot of memory as the stack grows.
- To remove this recursion, we use LL-parser, which uses a table for lookup.

Example of LL(1) grammar

- E -> TE’
- E -> +TE’ | ε
- T -> FT’
- T’ -> *FT’ | ε
- F -> (E) | id

First and Follow

Symbol	FIRST	FOLLOW
E	(,id	$,)
E’	+,ë	$,)
T	(,id	+,$,)
T’	*,ë	+,$,)
F	(,id	*,+,$,)

Algo for Construction of predictive parsing table :

1. For each production A→a of grammar G, do steps 2 and 3
2. For each terminal 'a' in FIRST(a) , add A→a in M[A,a].
3. If e is in FIRST(a) , add A→a to M[A,b] for each terminal b in FOLLOW(A). If ë is in FIRST(a) , and $ is in FOLLOW(A), then add A→a to M[A,$]
4. Make each undefined entry as “ERROR”, i.e. An error entry.

Generated Parser Table For String id + id * id

Non Terminal			**INPUT SYMBOLS**			
	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)		

How to control the parser?

- If X=a=$, parser halts, string accepted.
- If X=a !=$, parser pops X, and advances the input pointer to point to next input symbol.
- If X is a non-terminal, the program consults entry M[X,a] of the parsing table M. Replace the top of stack(X) with production rule corresponding to entry in table. If entry = ERROR, call error recovery routine.

MATCHED	STACK	INPUT	ACTION
	E$	id+id * id$	
	TE′$	id+id * id$	E->TE′
	FT′E′$	id+id * id$	T->FT′
	id T′E′$	id+id * id$	F->id
id	T′E′$	+id * id$	Match id
id	E′$	+id * id$	T′->Є
id	+TE′$	+id * id$	E′-> +TE′
id+	TE′$	id * id$	Match +
id+	FT′E′$	id * id$	T-> FT′
id+	idT′E′$	id * id$	F-> id
id+id	T′E′$	* id$	Match id
id+id	* FT′E′$	* id$	T′-> *FT′
id+id *	FT′E′$	id$	Match *
id+id *	idT′E′$	id$	F-> id
id+id * id	T′E′$	$	Match id
id+id * id	E′$	$	T′-> Є
id+id * id	$	$	E′-> Є

What does LL signify ?

The first L means that the scanning takes place from Left to right.

The second L means that the Left derivation is produced first.

The prime requirements are : -

- Stack
- Parsing Table
- Input buffer
- Parsing program .

What does LL signify ?

The first L means that the scanning takes place from Left to right.

The second L means that the Left derivation is produced first.

The prime requirements are : -

- Stack
- Parsing Table
- Input buffer
- Parsing program .

- Input buffer contains the string to be parsed, followed by $,a symbol used to indicate end of the input string. The stack indicates a sequence of grammar symbols with $ on the bottom,indicating bottom of   the stack. Initially, the stack contains the start symbol of the grammar on the top of $. The parsing table is a 2-D array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $.

- How to control the parser?
  - If X=a=$, parser halts, string accepted.
  - If X=a !=$, parser pops X, and advances the input pointer to point to next input symbol.
  - If X is a non-terminal, the program consults entry M[X,a] of the parsing table M. Replace the top of stack(X) with production rule corresponding to entry in table. If entry = ERROR, call error recovery routine.

Algo for Construction of predictive parsing table :

1. For each production A→a of grammar G, do steps 2 and 3
2. For each terminal 'a' in FIRST(a) , add A→a in M[A,a].
3. If e is in FIRST(a) , add A→a to M[A,b] for each terminal b in FOLLOW(A). If ë is in FIRST(a) , and $ is in FOLLOW(A), then add A→a to M[A,$]
4. Make each undefined entry as "ERROR", i.e. An error entry.

Example:

Grammar

E→TE'

E'→+TE' | ë

T→ FT'

T'→*FT' | ë

F→(E) | id

(ë stands for epsilon)

First and Follow

Symbol	FIRST	FOLLOW
E	(,id	$,)
E’	+,ë	$,)
T	(,id	+,$,)
T’	*,ë	+,$,)
F	(,id	*,+,$,)

Building the table

	Id	+	*	(	)	$
E	E→TE’			E→TE’		
E’		E’→+TE’			E’→ë	E’→ë
T	T→FT’			T→FT’		
T’		T’→ë	T’→*FT’		T’→ë	T’→ë
F	F→id				F→(E)	

Input=id+id*id

Stack	Input buffer
$E	*id+id*id$*
$E'T'	*Id+id*id$*
$E'T'F	*Id+id*id$*
$E'T'id	*Id+id*id$*
$E'T'	*+id*id$*
$E'	*+id*id$*
$E'T+	*+id*id$*
$E'T	*id*id$*

Stack	Input Buffer
$E'T'F	*id*id$*
$E'T'id	*id*id$*
$E'T'	**id$*
*$E'T'F**	**id$*
$E'T'F	*id$*
$E'T'id	*id$*
$E'T'	*$*
$E'	*$*
$	Accepted

Thus, we can easily construct an LL parse with 1 lookahead. Since one look ahead is involved, we also call it an LL(1) parser.

There are grammars which may requite LL(k) parsing.

For e.g. look at next example…..

Grammar:

S→iEtSS’ | a

S’→Es | ë

E→b

Note that this is

If then else statement

	FIRST	FOLLOW
S	a,I	$,ë
S’	$,ë	$,ë
E	b	t

Parse Table

	a	b	e	i	t	$
S	S→a			S→iEtS S’		
S’			S→ë S→e S			S→ë
E		E→b				

Ambiguity

- The grammar is ambiguous and it is evident by the fact that we have two entries corresponding to M[S’,e] containing S →€ and S’ →eS. This ambiguity can be resolved if we choose

S’→eS i.e associating the else’s with the closest previous “then”.

Note that the ambiguity will be solved if we use LL(2) parser, i.e. always see for the two input symbols. How?

When input is ‘e’ then it looks at next input. Depending on the next input we choose the appropriate rule.

LL(1) grammars have distinct properties. -No ambiguous grammar or left recursive grammar can be LL(1).

A grammar is LL(1) if and only if whenever a production $A \rightarrow C \mid D$ the following conditions hold:

…contd

1)For no terminal a both C and D derive strings beginning with a.

Thus First(C) != First(D)

2)At most one of C or D can derive €

3) If C* € then D does not derive any string beginning with terminal Follow(A).