Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Čeština
Deutsch
Español
فارسی
Français
한국어
日本語
Polski
Português
Русский
Српски / srpski
Українська
中文
Edit links

LL parser

From Wikipedia, the free encyclopedia

In computer science, an **LL parser** is a top-down parser for a subset of context-free languages. It parses the input from **L**eft to right, performing **L**eftmost derivation of the sentence.

An LL parser is called an LL(*k*) parser if it uses *k* tokens of lookahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar without backtracking then it is called an LL(*k*) grammar. LL(*k*) grammars can generate more languages the higher the number *k* of lookahead tokens.[1] A corollary of this is that not all context-free languages can be recognized by an LL(k) parser. An LL parser is called an LL(*) parser (an LL-regular parser[2]) if it is not restricted to a finite *k* tokens of lookahead, but can make parsing decisions by recognizing whether the following tokens belong to a regular language (for example by means of a Deterministic Finite Automaton).

LL grammars, particularly LL(1) grammars, are of great practical interest, as parsers for these grammars are easy to construct, and many computer languages are designed to be LL(1) for this reason. LL parsers are table-based parsers, similar to LR parsers. LL grammars can also be parsed by recursive descent parsers.

Contents [hide]

General case [edit]

The parser works on strings from a particular context-free grammar.

The parser consists of

- an *input buffer*, holding the input string (built from the grammar)
- a *stack* on which to store the terminals and non-terminals from the grammar yet to be parsed
- a *parsing table* which tells it what (if any) grammar rule to apply given the symbols on top of its stack and the next input token

The parser applies the rule found in the table by matching the top-most symbol on the stack (row) with the current symbol in the input stream (column).

When the parser starts, the stack already contains two symbols:

```
[ S, $ ]
```

where '$' is a special terminal to indicate the bottom of the stack and the end of the input stream, and 'S' is the start symbol of the grammar. The parser will attempt to rewrite the contents of this stack to what it sees on the input stream. However, it only keeps on the stack what still needs to be rewritten.

Concrete example [edit]

Set up [edit]

To explain an LL(1) parser's workings we will consider the following small LL(1) grammar:

1. S → F
2. S → (S + F)
3. F → a

and parse the following input:

(a + a)

We construct a parsing table for this grammar by expanding all the terminals by column and all nonterminals by row. Later, the expressions are numbered by the position where the columns and rows cross. For example, the terminal '(' and non-terminal 'S' match for expression number 2. The table is as follows:

	(	**)**	**a**	**+**	**$**
S	2	-	1	-	-
F	-	-	3	-	-

(Note that there is also a column for the special terminal, represented here as **$**, that is used to indicate the end of the input stream.)

Parsing procedure [edit]

In each step, the parser reads the next-available symbol from the input stream, and the top-most symbol from the stack. If the input symbol and the stack-top symbol match, the parser discards them both, leaving only the unmatched symbols in the input stream and on the stack.

Thus, in its first step, the parser reads the input symbol '**(**' and the stack-top symbol 'S'. The parsing table instruction comes from the column headed by the input symbol '**(**' and the row headed by the stack-top symbol 'S'; this cell contains '2', which instructs the parser to apply rule (2). The parser has to rewrite 'S' to '**(** S **+** F **)**' on the stack by removing 'S' from stack and pushing '(', 'S', '+', 'F', ')' onto the stack and this writes the rule number 2 to the output. The stack then becomes:

```
[ (, S, +, F, ), $ ]
```

Since the '**(**' from the input stream did not match the top-most symbol, 'S', from the stack, it was not removed, and remains the next-available input symbol for the following step.

In the second step, the parser removes the '**(**' from its input stream and from its stack, since they now match. The stack now becomes:

```
[ S, +, F, ), $ ]
```

Now the parser has an '**a**' on its input stream and an 'S' as its stack top. The parsing table instructs it to apply rule (1) from the grammar and write the rule number 1 to the output stream. The stack becomes:

```
[ F, +, F, ), $ ]
```

The parser now has an '**a**' on its input stream and an 'F' as its stack top. The parsing table instructs it to apply rule (3) from the grammar and write the rule number 3 to the output stream. The stack becomes:

```
[ a, +, F, ), $ ]
```

In the next two steps the parser reads the '**a**' and '**+**' from the input stream and, since they match the next two items on the stack, also removes them from the stack. This results in:

```
[ F, ), $ ]
```

In the next three steps the parser will replace **'F'** on the stack by **'a'**, write the rule number 3 to the output stream and remove the **'a'** and **')'** from both the stack and the input stream. The parser thus ends with **'$'** on both its stack and its input stream.

In this case the parser will report that it has accepted the input string and write the following list of rule numbers to the output stream:

[2, 1, 3, 3]

This is indeed a list of rules for a leftmost derivation of the input string, which is:

S → **(** S **+** F **)** → **(** F **+** F **)** → **(a +** F **)** → **(a + a)**

Parser implementation in C++ [edit]

Below follows a C++ implementation of a table-based LL parser for the example language:

```
#include <iostream>
#include <map>
#include <stack>

enum Symbols {
 // the symbols:
 // Terminal symbols:
 TS_L_PARENS, // (
 TS_R_PARENS, // )
 TS_A,  // a
 TS_PLUS, // +
 TS_EOS,  // $, in this case corresponds to '\0'
 TS_INVALID, // invalid token

 // Non-terminal symbols:
 NTS_S,  // S
 NTS_F  // F
};

/*
Converts a valid token to the corresponding terminal symbol
*/
enum Symbols lexer(char c)
{
 switch(c)
 {
  case '(':  return TS_L_PARENS;
  case ')':  return TS_R_PARENS;
  case 'a':  return TS_A;
  case '+':  return TS_PLUS;
  case '\0': return TS_EOS; // end of stack: the $ terminal symbol
  default:   return TS_INVALID;
 }
}

int main(int argc, char **argv)
{
 using namespace std;

 if (argc < 2)
 {
  cout << "usage:\n\tll '(a+a)'" << endl;
  return 0;
 }

 // LL parser table, maps < non-terminal, terminal> pair to action
 map< enum Symbols, map<enum Symbols, int> > table;
 stack<enum Symbols> ss; // symbol stack
 char *p; // input buffer

 // initialize the symbols stack
 ss.push(TS_EOS); // terminal, $
 ss.push(NTS_S);  // non-terminal, S

 // initialize the symbol stream cursor
 p = &argv[1][0];

 // setup the parsing table
 table[NTS_S][TS_L_PARENS] = 2;
        table[NTS_S][TS_A] = 1;
 table[NTS_F][TS_A] = 3;

 while(ss.size() > 0)
 {
  if(lexer(*p) == ss.top())
  {
   cout << "Matched symbols: " << lexer(*p) << endl;
   p++;
   ss.pop();
  }
  else
  {
   cout << "Rule " << table[ss.top()][lexer(*p)] << endl;
   switch(table[ss.top()][lexer(*p)])
   {
    case 1: // 1. S → F
     ss.pop();
     ss.push(NTS_F); // F
     break;

          case 2: // 2. S → ( S + F )
     ss.pop();
     ss.push(TS_R_PARENS); // )
     ss.push(NTS_F);  // F
     ss.push(TS_PLUS); // +
     ss.push(NTS_S);  // S
     ss.push(TS_L_PARENS); // (
     break;

          case 3: // 3. F → a
     ss.pop();
     ss.push(TS_A); // a
     break;

    default:
     cout << "parsing table defaulted" << endl;
     return 0;
     break;
   }
  }
 }

 cout << "finished parsing" << endl;

 return 0;
}
```

Parser implementation in Python [edit]

```
#All constants are indexed from 0

Term = 0
Rule = 1

# Terminals
T_LPAR = 0
T_RPAR = 1
T_A = 2
T_PLUS = 3
T_END = 4
T_INVALID = 5
```

```
# Non-terminals
N_S = 0
N_F = 1

#parse table
table = [[ 1, -1, 0, -1, -1, -1],
         [-1, -1, 2, -1, -1, -1]]

rules = [[(Rule,N_F)],
         [(Term,T_LPAR), (Rule,N_S), (Term,T_PLUS), (Rule,N_F), (Term,T_RPAR)],
         [(Term,T_A)]]

stack = [(Term,T_END), (Rule,N_S)]

def lexicalAnalysis(inputstring):
    print('Lexical analysis')
    tokens = []
    #cdict = {'+': T_PLUS, '(': T_LPAR, ')': T_RPAR, 'a': T_A}
    #for c in inputstring:
    #    tokens.append(cdict.get(c, T_INVALID))
    #
    # in the meantime it has been changed on wikipedia to simple mapping above,
    # but the original if-elif-elif-else could be indented to make further
distinction
    # for multi-character terminals like between '-' and '->' .
    for c in inputstring:
        if c   == '+': tokens.append(T_PLUS)
        elif c == '(': tokens.append(T_LPAR)
        elif c == ')': tokens.append(T_RPAR)
        elif c == 'a': tokens.append(T_A)
        else: tokens.append(T_INVALID)
    tokens.append(T_END)
    print(tokens)
    return tokens

def syntacticAnalysis(tokens):
    print('Syntactic analysis')
    position = 0
    while len(stack) > 0:
        (stype, svalue) = stack.pop()
        token = tokens[position]
        if stype == Term:
            if svalue == token:
                position += 1
                print('pop', svalue)
                if token == T_END:
                    print('input accepted')
            else:
                print('bad term on input:', token)
                break
        elif stype == Rule:
            print('svalue', svalue, 'token', token)
            rule = table[svalue][token]
            print('rule', rule)
            for r in reversed(rules[rule]):
                stack.append(r)
        print('stack', stack)

inputstring = '(a+a)'
syntacticAnalysis(lexicalAnalysis(inputstring))
```

Remarks [edit]

As can be seen from the example the parser performs three types of steps depending on whether the top of the stack is a nonterminal, a terminal or the special symbol **$**:

- If the top is a nonterminal then it looks up in the parsing table on the basis of this nonterminal and the symbol on the input stream which rule of the grammar it should use to replace it with on the stack. The number of the rule is written to the output stream. If the parsing table indicates that there is no such rule then it reports an error and stops.

- If the top is a terminal then it compares it to the symbol on the input stream and if they are equal they are both removed. If they are not equal the parser reports an error and stops.
- If the top is **$** and on the input stream there is also a **$** then the parser reports that it has successfully parsed the input, otherwise it reports an error. In both cases the parser will stop.

These steps are repeated until the parser stops, and then it will have either completely parsed the input and written a leftmost derivation to the output stream or it will have reported an error.

Constructing an LL(1) parsing table [edit]

In order to fill the parsing table, we have to establish what grammar rule the parser should choose if it sees a nonterminal A on the top of its stack and a symbol a on its input stream. It is easy to see that such a rule should be of the form $A \rightarrow w$ and that the language corresponding to w should have at least one string starting with a. For this purpose we define the *First-set* of w, written here as **Fi**(w), as the set of terminals that can be found at the start of some string in w, plus ε if the empty string also belongs to w. Given a grammar with the rules $A_1 \rightarrow w_1$, ..., $A_n \rightarrow w_n$, we can compute the **Fi**(w_i) and **Fi**(A_i) for every rule as follows:

1. initialize every **Fi**(A_i) with the empty set
2. set **Fi**(w_i) to *Fi*(w_i) for every rule $A_i \rightarrow w_i$, where *Fi* is defined as follows:
   - $Fi(a\,w') = \{ a \}$ for every terminal a
   - $Fi(A\,w')$ = **Fi**(A) for every nonterminal A with ε not in **Fi**(A)
   - $Fi(A\,w')$ = **Fi**(A) \ { ε } ∪ $Fi(w')$ for every nonterminal A with ε in **Fi**(A)
   - $Fi(\varepsilon) = \{ \varepsilon \}$
3. add **Fi**(w_i) to **Fi**(A_i) for every rule $A_i \rightarrow w_i$
4. do steps 2 and 3 until all **Fi** sets stay the same.

Unfortunately, the First-sets are not sufficient to compute the parsing table. This is because a right-hand side w of a rule might ultimately be rewritten to the empty string. So the parser should also use the rule $A \rightarrow w$ if ε is in **Fi**(w) and it sees on the input stream a symbol that could follow A. Therefore we also need the *Follow-set* of A, written as **Fo**(A) here, which is defined as the set of terminals a such that there is a string of symbols $\alpha Aa\beta$ that can be derived from the start symbol. We use **$** as a special terminal indicating end of input stream and S as start symbol.

Computing the Follow-sets for the nonterminals in a grammar can be done as follows:

1. initialize **Fo**(S) with { **$** } and every other **Fo**(A_i) with the empty set
2. if there is a rule of the form $A_j \rightarrow wA_iw'$, then
   - if the terminal a is in $Fi(w')$, then add a to **Fo**(A_i)
   - if ε is in $Fi(w')$, then add **Fo**(A_j) to **Fo**(A_i)
   - if w' has length 0, then add **Fo**(A_j) to **Fo**(A_i)
3. repeat step 2 until all *Fo* sets stay the same.

Now we can define exactly which rules will be contained where in the parsing table. If $T[A, a]$ denotes the entry in the table for nonterminal A and terminal a, then

$T[A,a]$ contains the rule $A \rightarrow w$ if and only if
a is in **Fi**(w) or
ε is in **Fi**(w) and a is in **Fo**(A).

If the table contains at most one rule in every one of its cells, then the parser will always know which rule it has to use and can therefore parse strings without backtracking. It is in precisely this case that the grammar is called an *LL(1) grammar*.

Constructing an LL(k) parsing table [edit]

Until the mid-1990s, it was widely believed that LL(k) parsing (for $k > 1$) was impractical[*citation needed*], since the parser table would have exponential size in k in the worst case. This perception changed gradually after the release of the Purdue Compiler Construction Tool Set around 1992, when it was demonstrated that many programming languages can be parsed efficiently by an LL(k) parser without triggering the worst-case behavior of the parser. Moreover, in certain cases LL parsing is feasible even with unlimited lookahead. By contrast, traditional parser generators like yacc use LALR(1) parser tables to construct a restricted LR parser with a fixed one-token lookahead.

Conflicts [edit]

As described in the introduction, LL(1) parsers recognize languages that have LL(1) grammars, which are a special case of context-free grammars (CFGs); LL(1) parsers cannot recognize all context-free languages. The LL(1) languages are a proper subset of the LR(1) languages which in turn are a proper subset of all context-free languages. In order for a CFG to be an LL(1) grammar, certain conflicts must not arise, which we describe in this section.

Terminology[3] [edit]

Let A be a non-terminal. FIRST(A) is (defined to be) the set of terminals that can appear in the first position of any string derived from A. FOLLOW(A) is the union over FIRST(B) where B is any non-terminal that immediately follows A in the right hand side of a production rule.

LL(1) Conflicts [edit]

There are 2 main types of LL(1) conflicts:

FIRST/FIRST Conflict [edit]

The FIRST sets of two different grammar rules for the same non-terminal intersect. An example of an LL(1) FIRST/FIRST conflict:

```
S -> E | E 'a'
E -> 'b' | ε
```

FIRST(E) = {'b', ε} and FIRST(E 'a') = {'b', 'a'}, so when the table is drawn, there is conflict under terminal 'b' of production rule S.

Special Case: Left Recursion [edit]

Left recursion will cause a FIRST/FIRST conflict with all alternatives.

```
E -> E '+' term | alt1 | alt2
```

FIRST/FOLLOW Conflict [edit]

The FIRST and FOLLOW set of a grammar rule overlap. With an empty string (ε) in the FIRST set it is unknown which alternative to select. An example of an LL(1) conflict:

```
S -> A 'a' 'b'
A -> 'a' | ε
```

The FIRST set of A now is {'a', ε} and the FOLLOW set {'a'}.

Solutions to LL(1) Conflicts [edit]

Left Factoring [edit]

For a general method, see removing left recursion.

A common left-factor is "factored out".

```
A -> X | X Y Z
```

becomes

```
A -> X B
B -> Y Z | ε
```

Can be applied when two alternatives start with the same symbol like a FIRST/FIRST conflict.

Another example (more complex) using above FIRST/FIRST conflict example:

```
S -> E | E 'a'
E -> 'b' | ε
```

becomes (merging into a single non-terminal)

```
S -> 'b' | ε | 'b' 'a' | 'a'
```

then through left-factoring, becomes

```
S -> 'b' E | E
E -> 'a' | ε
```

Substitution [edit]

Substituting a rule into another rule to remove indirect or FIRST/FOLLOW conflicts. Note that this may cause a FIRST/FIRST conflict.

Left recursion removal[4] [edit]

A simple example for left recursion removal: The following production rule has left recursion on E

```
E -> E '+' T
  -> T
```

This rule is nothing but list of Ts separated by '+'. In a regular expression form T ('+' T)*. So the rule could be rewritten as

```
E -> T Z
Z -> '+' T Z
  -> ε
```

Now there is no left recursion and no conflicts on either of the rules.

However, not all CFGs have an equivalent LL(k)-grammar, e.g.:

```
S -> A | B
A -> 'a' A 'b' | ε
B -> 'a' B 'b' 'b' | ε
```

It can be shown that there does not exist any LL(k)-grammar accepting the language generated by this grammar.

See also [edit]

- Comparison of parser generators
- Parse tree
- Top-down parsing
- Bottom-up parsing

Notes [edit]

1. ^ Rosenkrantz, D. J.; Stearns, R. E. (1970). "Properties of Deterministic Top Down Grammars" (PDF). *Information and Control* **17**: 226–256. doi:10.1016/s0019-9958(70)90446-8.
2. ^ Dick Grune; Ceriel J.H. Jacobs (29 October 2007). *Parsing Techniques: A Practical Guide*. Springer. pp. 585–. ISBN 978-0-387-68954-8.
3. ^ http://www.cs.uaf.edu/~cs331/notes/LL.pdf
4. ^ Modern Compiler Design, Grune, Bal, Jacobs and Langendoen

External links [edit]

- A tutorial on implementing LL(1) parsers in C#
- Parsing Simulator This simulator is used to generate parsing tables LL(1) and to resolve the exercises of the book.
- LL(1) DSL PEG parser (toolkit framework)

- Language theoretic comparison of LL and LR grammars

Categories: Parsing algorithms