In [computer science](http://www.answers.com/topic/computer-science), a **Simple LR parser** or **SLR parser** is created by an SLR parser generator which reads a [BNF grammar](http://www.answers.com/topic/backus-naur-form-3) and constructs an [LR(0)](http://www.answers.com/topic/lr-parser) state machine and computes the look-aheads sets for each reduction in a state by examining the Follow Set for the nonterminal associated with the reduction. The Follow Set for each nonterminal symbol is computed by seeing which terminal symbols follow the nonterminal symbol in the rules of the BNF grammar. It is useful to have the First Set already computed when creating the Follow Set.

At runtime, the SLR parser will perform a reduction based on a grammar rule *A* → *w* if the next symbol in the input stream is in the Follow Set of *A* (see [LL parser](http://www.answers.com/topic/ll-parser) for a definition of Follow Set, and [[1]](http://www.jambe.co.nz/UNI/FirstAndFollowSets.html)).

The problem with SLR parsers is that the computation of the look-ahead sets is too simplistic, using only the rules of the grammar to determine look-ahead sets. The more accurate way to determine look-ahead sets is to examine the nonterminal transitions in each state within the LR(0) state machine. These more accurate look-ahead sets are called the [LALR(1)](http://www.answers.com/topic/lalr-parser) look-ahead sets.

An SLR parser will typically have more conflict states than an [LALR parser](http://www.answers.com/topic/lalr-parser). For real-world computer languages, an SLR parser is usually not adequate, but for student projects in a compiler class it is a good learning tool.

A grammar that has no conflicts reported by an SLR parser generator is an [*SLR grammar*](http://www.answers.com/topic/slr-grammar).

**Algorithm**

The SLR parsing algorithm

Initialize the stack with S

Read input symbol

while (true)

if Action(top(stack), input) = S

NS <- Goto(top(stack),input)

push NS

Read next symbol

else if Action(top(stack), input) = Rk

output k

pop |RHS| of production k from stack

NS <- Goto(top(stack), LHS\_k)

push NS

else if Action(top(stack),input) = A

output valid, return

else

output invalid, return

**Example**

A grammar that can be parsed by an SLR parser but not by an LR(0) parser is the following:

(0) S → E

(1) E → 1 E

(2) E → 1

Constructing the action and goto table as is done for LR(0) parsers would give the following item sets and tables:

**Item set 0**

S → • E

+ E → • 1 E

+ E → • 1

**Item set 1**

E → 1 • E

E → 1 •

+ E → • 1 E

+ E → • 1

**Item set 2**

S → E •

**Item set 3**

E → 1 E •

The action and goto tables:

|  |  |  |  |
| --- | --- | --- | --- |
|  | *action* | | *goto* |
| *state* | **1** | **$** | **E** |
| **0** | s1 |  | 2 |
| **1** | s1/r2 | r2 | 3 |
| **2** |  | acc |  |
| **3** | r1 | r1 |  |

As can be observed there is a shift-reduce conflict for state 1 and terminal '1'. However, the follow set of E is { $ } so the reduce actions r1 and r2 are only valid in the column for $. The result is the following conflict-less action and goto table:

|  |  |  |  |
| --- | --- | --- | --- |
|  | *action* | | *goto* |
| *state* | **1** | **$** | **E** |
| **0** | s1 |  | 2 |
| **1** | s1 | r2 | 3 |
| **2** |  | acc |  |
| **3** |  | r1 |  |