WIKIPEDIA
The Free Encyclopedia

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
فارسی
Français
Italiano
日本語
Polski
Русский
Српски / srpski
Edit links

Simple LR parser

From Wikipedia, the free encyclopedia

In computer science, a **Simple LR** or **SLR parser** is a type of LR parser with small parse tables and a relatively simple parser generator algorithm. As with other types of LR(1) parser, an SLR parser is quite efficient at finding the single correct bottom-up parse in a single left-to-right scan over the input stream, without guesswork or backtracking. The parser is mechanically generated from a formal grammar for the language.

SLR and the more-general methods LALR parser and Canonical LR parser have identical methods and similar tables at parse time; they differ only in the mathematical grammar analysis algorithms used by the parser generator tool. SLR and LALR generators create tables of identical size and identical parser states. SLR generators accept fewer grammars than do LALR generators like yacc and Bison. Many computer languages don't readily fit the restrictions of SLR, as is. Bending the language's natural grammar into SLR grammar form requires more compromises and grammar hackery. So LALR generators have become much more widely used than SLR generators, despite being somewhat more complicated tools. SLR methods remain a useful learning step in college classes on compiler theory.

SLR and LALR were both developed by Frank DeRemer as the first practical uses of Donald Knuth's LR parser theory.[*citation needed*] The tables created for real grammars by full LR methods were impractically large, larger than most computer memories of that decade, with 100 times or more parser states than the SLR and LALR methods.[*citation needed*]

Lookahead sets [edit]

To understand the differences between SLR and LALR, it is important to understand their many similarities and how they both make shift-reduce decisions. (See the article LR parser now for that background, up through the section on reductions' **lookahead sets**.)

The one difference between SLR and LALR is how their generators calculate the lookahead sets of input symbols that should appear next, whenever some completed production rule is found and reduced.

SLR generators calculate that lookahead by an easy approximation method based directly on the grammar, ignoring the details of individual parser states and transitions. This ignores the particular context of the current parser state. If some nonterminal symbol *S* is used in several places in the grammar, SLR treats those places in the same single way rather than handling them individually. The SLR generator works out `Follow(S)`, the set of all terminal symbols which can immediately follow some occurrence of *S*. In the parse table, each reduction to *S* uses Follow(S) as its LR(1) lookahead set. Such follow sets are also used by generators for LL top-down parsers. A grammar that has no shift/reduce or reduce/reduce conflicts when using follow sets is called an SLR grammar.

LALR generators calculate lookahead sets by a more precise method based on exploring the graph of parser states and their transitions. This method considers the particular context of the current parser state. It customizes the handling of each grammar occurrence of some nonterminal S. See article LALR parser for further details of this calculation. The lookahead sets calculated by LALR generators are a subset of (and hence better than) the approximate sets calculated by SLR generators. If a grammar has table conflicts when using SLR follow sets, but is conflict-free when using LALR follow sets, it is called a LALR grammar.

Example [edit]

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'. This occurs because, when the action table for an LR(0) parser is created, reduce actions are inserted on a per-row basis. However, by using a follow set, reduce actions can be added with finer granularity. The follow set for this grammar:

symbol	S	E	1
following	$	$	1,$

A reduce only needs to be added to a particular action column if that action is in the follow set associated with that reduce. This algorithm describes whether a reduce action must be added to an action column:

```
function mustBeAdded(reduceAction, action){
 ruleNumber = reduceAction.value;
 ruleSymbol = rules[ruleNumber].leftHandSide;
 return (action in followSet(ruleSymbol))
}
```

for example, mustBeAdded(r2, "1") is false, because the left hand side of rule 2 is "E", and 1 is not in E's follow set. Contrariwise, mustBeAdded(r2, "$") is true, because "$" is in E's follow set.

By using mustBeAdded on each reduce action in the action table, the result is a conflict-free action table:

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

See also [edit]

- LR parser
- LL parser
- LALR parser
- SLR grammar

Categories: Parsing algorithms