

## **Compiler Design — Lecture notes week 6**

- Author: Ruben Schenk
- Date: 26.10.2021
- Contact: [ruben.schenk@inf.ethz.ch](mailto:ruben.schenk@inf.ethz.ch)

## 8. LL & LR Parsing

---

### 8.1 LL(1) Grammars

One problem with grammars is that not all grammars can be parsed "top-down" with a *single lookahead*. *Top-down* means that we start from the start symbol, i.e. the root of the parse tree, and go down.

**LL(1)** means:

- Left-to-right scanning
- Left-most derivation
- 1 lookahead symbol

#### 8.1.1 Making a Grammar LL(1)

The main problem is that we can't decide which  $S$  production to apply until we see the symbol after the first expression. The solution is to *left-factor* the grammar. There is a common  $S$  prefix for each choice, so add a new non-terminal  $S'$  at the decision point:

This means that we transform our example grammar

$$\begin{aligned} S &\rightarrow E + S \mid E \\ E &\rightarrow \text{number} \mid (S) \end{aligned}$$

to the following "left-factor" grammar:

$$\begin{aligned} S &\rightarrow ES' \\ S' &\rightarrow \epsilon \\ S' &\rightarrow +S \\ E &\rightarrow \text{number} \mid (S) \end{aligned}$$

However, we also need to *eliminate left-recursion* somehow. In general, this is done by rewriting the following left-recursive rule

$$S \rightarrow S\alpha_1 \mid \dots \mid S\alpha_n \mid \beta_1 \mid \dots \mid \beta_m$$

to a rule of the form:

$$\begin{aligned} S &\rightarrow \beta_1 S' \mid \dots \mid \beta_m S' \\ S' &\rightarrow \alpha_1 S' \mid \dots \mid \alpha_n S' \mid \epsilon \end{aligned}$$

In our running example, this would mean to rewrite

$$S \rightarrow S + E \mid E$$

$$E \rightarrow \text{number} \mid (S)$$

to the following left-recursion-eliminating grammar:

$$S \rightarrow ES'$$

$$S' \rightarrow +ES' \mid \epsilon$$

$$E \rightarrow \text{number} \mid (S)$$

## 8.1.2 Predictive Parsing

Given an LL(1) grammar:

- For a given non-terminal, the lookahead symbol uniquely determines the production to apply
- The parsing is driven by a predictive parsing table
- It is convenient to add a special *end-of-file token and a start symbol* that requires

*Example:* Let us look at the following LL(1) grammar:

$$T \rightarrow S\$$$

$$S \rightarrow ES'$$

$$S' \rightarrow \epsilon$$

$$S' \rightarrow +S$$

$$E \rightarrow \text{number} \mid (S)$$

We then propose the following **predictive parsing table**:

	NUMBER	+	(	)	\$ (EOF)
$T$	$\rightarrow S$			$\rightarrow S$	
$S$	$\rightarrow ES'$			$\rightarrow ES'$	
$S'$		$\rightarrow +S$			$\rightarrow \epsilon$
$E$	$\rightarrow \text{number}$		$\rightarrow (S)$		

## 8.1.3 Construction of Parse Table

How do we construct the parse table? We examine two possible cases by considering the following production:  $A \rightarrow \gamma$

## Case 1

Construct the set of all input tokens that may appear *first* in strings that can be derived from  $\gamma$ . Then add the production  $\rightarrow \gamma$  to the entry  $(A, \text{token})$  for each such token.

## Case 2

If  $\gamma$  can derive  $\epsilon$ , then we construct the set of all input tokens that may *follow* the nonterminal  $A$  in the grammar. We then add the production  $\rightarrow \gamma$  to the entry  $(A, \text{token})$  for each such token.

*Note:* If there are two different productions for a given entry, then the grammar is not LL(1).

*Example:*

- $\text{First}(T) = \text{First}(S)$
- $\text{First}(S) = \text{First}(E)$
- $\text{First}(S') = \{ +, \epsilon \}$
- $\text{First}(E) = \{ \text{number}, '(' \}$

- $\text{Follow}(S') = \text{Follow}(S)$
- $\text{Follow}(S) = \{ \$, ')' \} \cup \text{Follow}(S')$

**Note:** we want the *least* solution to this system of set equations... a *fixpoint* computation. More on these later in the course.

$T \mapsto S\$$   
 $S \mapsto ES'$   
 $S' \mapsto \epsilon$   
 $S' \mapsto + S$   
 $E \mapsto \text{number} \mid ( S )$

	number	+	(	)	\$ (EOF)
T	$\mapsto S\$$		$\mapsto S\$$		
S	$\mapsto ES'$		$\mapsto ES'$		
S'		$\mapsto + S$		$\mapsto \epsilon$	$\mapsto \epsilon$
E	$\mapsto \text{number}$		$\mapsto ( S )$		

### 8.1.4 Converting The Parsing Table to Code

When we want to convert a parsing table to code, we proceed as follows:

- Define  $N$  mutually recursive functions, one for each non-terminal  $A$ : `parse_A`
- `parse_A` is of type `unit -> ast` if  $EA\$$  is not an auxiliary non-terminal
- Otherwise, `parse_A` takes additional ast's as inputs, one for each non-terminal in the

"factored" prefix

Then, each function `parse_A`:

- "Peeks" at the lookahead token
- Follows the production rule in the corresponding entry
- Consumes the terminal tokens from the input stream
- Calls `parse_x` to create the sub-tree for the non-terminal  $X$
- If the rule ends in an auxiliary non-terminal, it is called with the appropriate ast's
- Otherwise, this function builds the ast tree itself and returns it

### 8.1.5 LL(1) Summary

**LL(1)** is top-down based parsing that finds the left-most derivation. The process proceeds with the following steps:

1. Language grammar =>
2. LL(1) grammar =>
3. Prediction table =>
4. Recursive-descent parser

## 8.2 LR Grammars

### 8.2.1 Bottom-up Parsing

**LR(k) parser** are *bottom-up parser*:

- Left-to-right scanning
- Right-most derivation
- k lookahead symbols

LR grammars are *more expressive* than LL grammars:

- They can handle left-recursive (and right-recursive) grammars (i.e. virtually all programming languages)
- They make it easier to express programming language syntax (e.g. no left factoring)

The most common technique are **Shift-Reduce parsers**:

- Work bottom up instead of top down
- Construct the right-most derivation of a program in the grammar
- Better error detection/recovery

## 8.2.2 Shift/Reduce Parsing

In shift/reduce parsing, the parser has a **parser state** described as follows:

- Stack of terminals and non-terminals
- Unconsumed input is a string of terminals
- The current derivation step is `stack + input`

**Parsing** is a sequence of `shift` and `reduce` operations:

- Shift: Move lookahead token to the stack
- Reduce: Replace symbols  $\gamma$  at the top of the stack with a non-terminal  $X$  s.t.  $X \rightarrow \gamma$  is a production, i.e. `pop gamma, push X`

*Example:* We consider our previous example

$$\begin{aligned} S &\rightarrow S + E \mid E \\ E &\rightarrow \text{number} \mid (S) \end{aligned}$$

Stack	Input	Action
	(1 + 2 + (3 + 4)) + 5	shift (
(	1 + 2 + (3 + 4)) + 5	shift 1
(1	+ 2 + (3 + 4)) + 5	reduce: $E \mapsto \text{number}$
(E	+ 2 + (3 + 4)) + 5	reduce: $S \mapsto E$
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: $E \mapsto \text{number}$

## 8.3 LR(0) Grammars

### 8.3.1 LR Parser States

Our goal it is to know *what set of reductions are legal* at any given point. The idea to solve this problem is to summarize all possible stack prefixes  $\alpha$  as a finite parser state:

- The parser state is computed by a DFA that reads the stack  $\sigma$
- Accept states of the DFA correspond to unique reductions that apply

### 8.3.2 Example LR(0) Grammar: Tuples

The following grammar is an example grammar for non-empty tuples and identifiers:

$$S \rightarrow (L) \mid \text{id}$$
$$L \rightarrow S \mid L, S$$

Now, if we apply parsing as a sequence of shift and reduce operations, we end up with the following parse operation:

- **Shift:** Move look-ahead token to the stack

Stack	Input	Action
	(x, (y, z), w)	shift (
(	x, (y, z), w)	shift x

- **Reduce:** Replace symbols  $\gamma$  at top of stack with nonterminal  $X$  s.t.  $X \mapsto \gamma$  is a production, i.e., pop  $\gamma$ , push  $X$

Stack	Input	Action
(x	, (y, z), w)	reduce $S \mapsto \text{id}$
(S	, (y, z), w)	reduce $L \mapsto S$

### 8.3.3 Action Selection Problem

Given a stack  $\sigma$  and a lookahead symbol  $b$ , should the parser either:

- Shift  $b$  onto the stack (new stack is  $\sigma b$ ), or
- Reduce a production  $X \rightarrow \gamma$ , assuming that  $\sigma = \alpha\gamma$  (new stack is  $\alpha X$ ) ?

The main idea to solve this problem is to decide based on a prefix  $\alpha$  of the stack plus the lookahead. The prefix  $\alpha$  is different for different possible reductions since in productions  $X \rightarrow \gamma$  and  $Y \rightarrow \beta, \gamma$  and  $\beta$  might have different lengths.

### 8.3.4 LR(0) States

A **LR(0) state** consists of items to track progress on possible upcoming reductions. A **LR(0) item** is a production with an extra separator  $\cdot$  in the RHS. Example items could be:  $S \rightarrow \cdot (L)$  or  $S \rightarrow (\cdot L)$  or  $L \rightarrow S$ .

The intuition for the meaning of the dot is:

- Stuff before the  $\cdot$  is already on the stack
- Stuff after the  $\cdot$  is what might be seen next

### 8.3.5 Constructing The DFA

We will consider the following grammar:

$$\begin{aligned} S' &\rightarrow S\$ \\ S &\rightarrow (L) \mid \text{id} \\ L &\rightarrow S \mid L, S \end{aligned}$$

The first step when creating the DFA is to add a new production  $S' \rightarrow S$  to the grammar. The *start state* of the DFA is the empty stack, so it contains the item  $S' \rightarrow \cdot S$ . We then proceed to add the **closure of the state** to our DFA:

- Add items for all productions whose LHS non-terminal occurs in an item in the state just after the  $\cdot$  (e.g.  $S$  in  $S' \rightarrow \cdot S$ )
- The added items have the  $\cdot$  located at the beginning
- Note that newly added items may cause yet more items to be added to the state, we keep iterating until a *fixed point* is reached

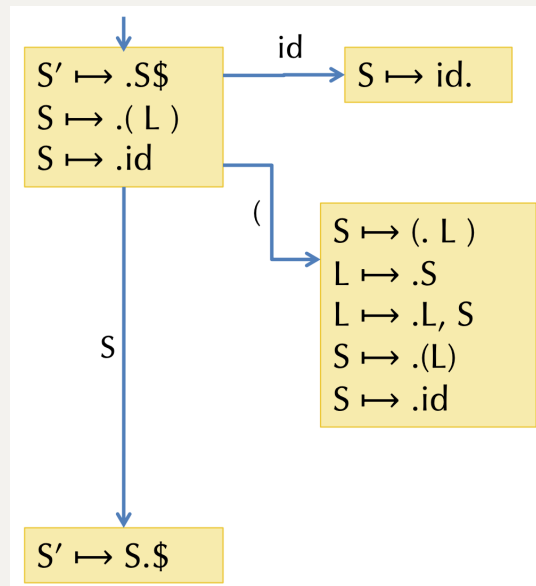
*Example:*

$$\text{Closure}(\{S' \rightarrow \cdot S\}) = \{S' \rightarrow \cdot S, S \rightarrow \cdot (L), S \rightarrow \cdot \text{id}\}$$

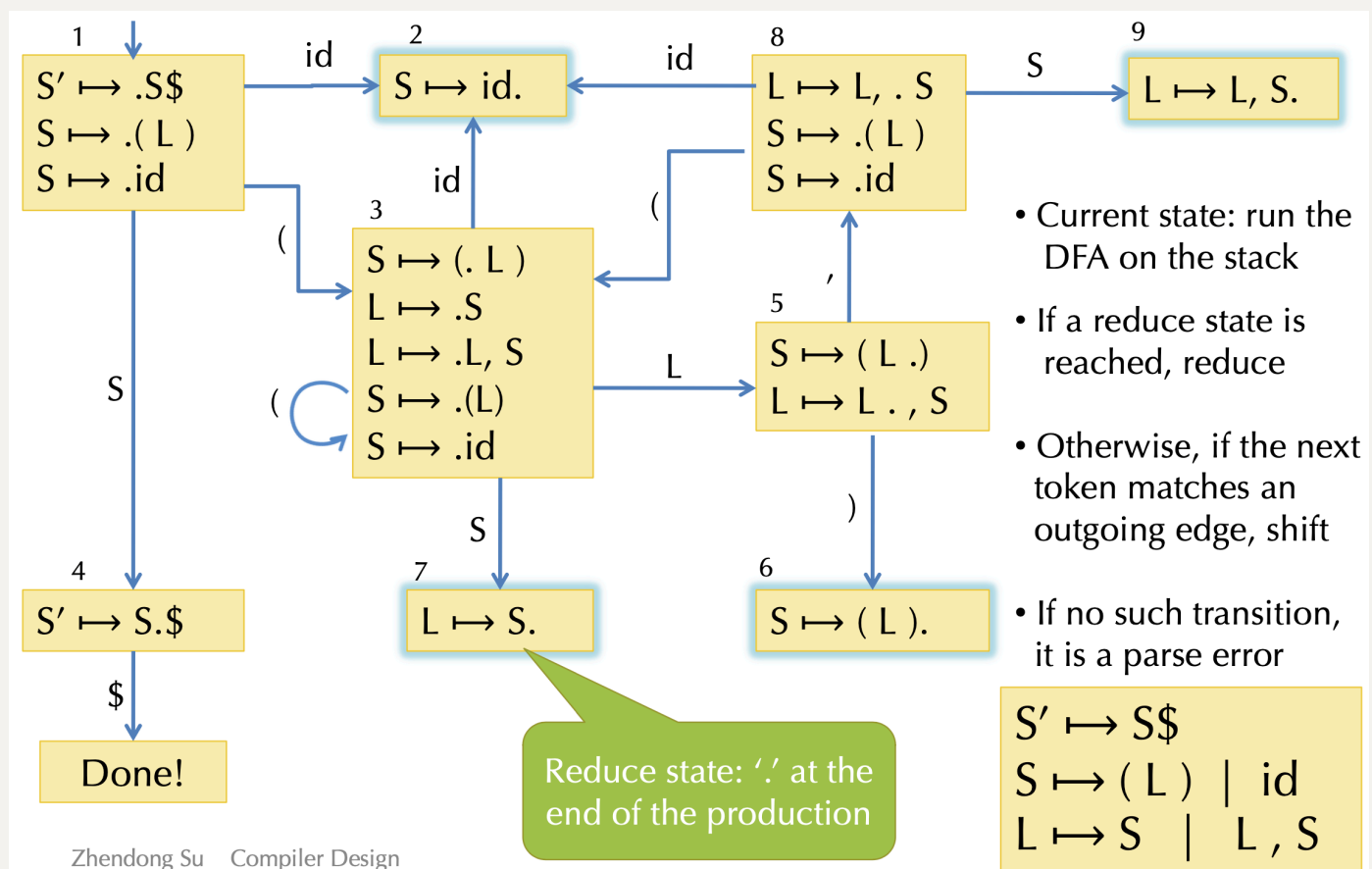
Next we need to add the *transitions*:

1. First, we see what terminals and non-terminals can appear after the  $\cdot$  in the source state.
2. The target state initially includes all items from the source state that have the edge-label symbol after the  $\cdot$ , but we advance the  $\cdot$  to simulate shifting the item onto the stack.
3. Finally, for each new state, we again take the closure of it.





By continuing the above approach, we will reach the following **full DFA** for our example:



### 8.3.6 Using The DFA

To use our DFA now, we run the parser stack  $\sigma$  through the DFA. The resulting state tells us what productions may be reduced next:

- If not in a reduce state, we shift the next symbol and transition w.r.t. the DFA
- If in a reduce state,  $X \rightarrow \gamma$  with stack  $\alpha\gamma$ , we **pop gamma** and **push x**

**Optimization:** There is no need to rerun the DFA from the beginning at each step. We might simply store the state with each symbol on the stack, e.g.  ${}_1(3)_3L_5)_6$ . Then:

- On a reduction  $X \rightarrow \gamma$ , we **pop** the stack to reveal the state too, e.g. from stack  ${}_1(3)_3L_5)_6$  we reduce  $S \rightarrow (L)$  to reach stack  ${}_1(3$
- Next, we push the reduction symbol, e.g. to reach the stack  ${}_1(3S$
- Then we take just one step in the DFA to find the next state  ${}_1(3S_7$

### 8.3.7 Implementing The Parsing Table

We represent the DFA as a table of shape `state * (terminals + nonterminals)`.

Entries for the **action table** specify two kinds of actions:

- Shift and go to state  $n$
- Reduce using the reduction  $X \rightarrow \gamma$ : First, **pop gamma** of the stack to reveal the state, second, look up  $X$  in the **goto table** and go to that state

*Example:*

	(	)	id	,	\$	S	L
1	s3		s2			g4	
2	$S \mapsto id$	$S \mapsto id$	$S \mapsto id$	$S \mapsto id$	$S \mapsto id$		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	$S \mapsto (L)$	$S \mapsto (L)$	$S \mapsto (L)$	$S \mapsto (L)$	$S \mapsto (L)$		
7	$L \mapsto S$	$L \mapsto S$	$L \mapsto S$	$L \mapsto S$	$L \mapsto S$		
8	s3		s2			g9	
9	$L \mapsto L,S$	$L \mapsto L,S$	$L \mapsto L,S$	$L \mapsto L,S$	$L \mapsto L,S$		

**sx** = shift and go to state x

**gx** = go to state x

### 8.3.8 LR(0) Limitations

An LR(0) machine only works if states with reduce actions have a *single* reduce action. In such states, the machine always reduces, ignoring lookahead.

With more complex grammars, the DFA construction will yield states with *shift/reduce* and *reduce/reduce* problems:

OK	shift/reduce	reduce/reduce
$S \mapsto ( L ).$	$S \mapsto ( L ).$ $L \mapsto .L , S$	$S \mapsto L , S.$ $S \mapsto , S.$

## 8.4 LR(1) Parsing

The algorithm for **LR(1) parsing** is similar to LR(0) DFA construction:

- LR(1) state is the set of all LR(1) items
- An LR(1) item is an LR(0) item plus a set of lookahead symbols, i.e.  $A \rightarrow \alpha. \beta, \mathcal{L}$

However, the **LR(1) closure** is a little more complex:

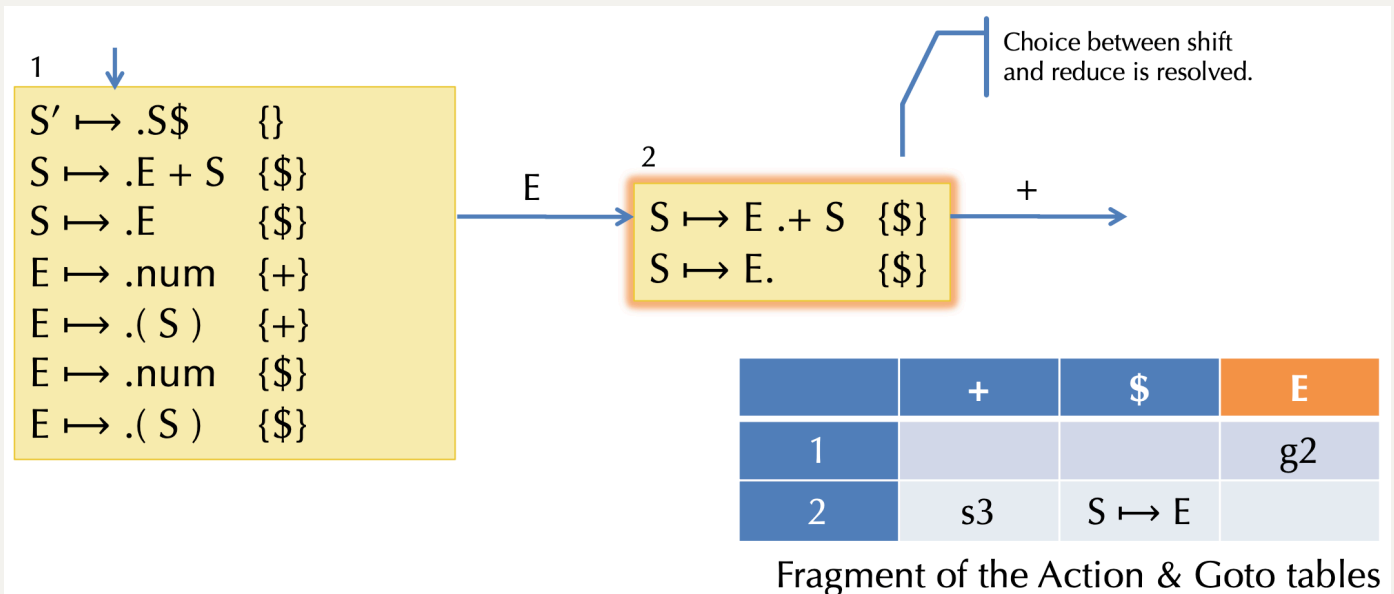
1. We first form the set of items just as we did in the LR(0) algorithm
2. Whenever a new item  $C \rightarrow \gamma$  is added, because the item  $A \rightarrow \beta. C\delta, \mathcal{L}$  is already in the set, we need to compute its lookahead set  $\mathcal{M}$ :
  - a. The lookahead set  $\mathcal{M}$  includes  $\text{FIRST}(\delta)$ , i.e. the set of terminals that may start strings derived from  $\delta$
  - b. If  $\delta$  is or can derive  $\epsilon$ , then the lookahead  $\mathcal{M}$  also contains  $\mathcal{L}$

### 8.4.1 Example Closure in LR(1)

$$\begin{aligned}
 S' &\mapsto S\$ \\
 S &\mapsto E + S \mid E \\
 E &\mapsto \text{number} \mid ( S )
 \end{aligned}$$

- Start item:  $S' \mapsto .S\$$  ,  $\{\}$
- Since  $S$  is to the right of a '.', add
 
$$\begin{aligned}
 S &\mapsto .E + S \quad , \quad \{\$\} \\
 S &\mapsto .E \quad , \quad \{\$\}
 \end{aligned}$$
 Note:  $\{\$\}$  is  $\text{FIRST}(\$)$
- Need to keep closing, since  $E$  appears to the right of a '.' in ' $.E + S$ '
 
$$\begin{aligned}
 E &\mapsto .\text{number} \quad , \quad \{+\} \\
 E &\mapsto .( S ) \quad , \quad \{+\}
 \end{aligned}$$
 Note:  $+$  added for reason 1  
 $\text{FIRST}(+ S) = \{+\}$
- Because  $E$  also appears to the right of '.' in ' $.E$ ' we get:
 
$$\begin{aligned}
 E &\mapsto .\text{number} \quad , \quad \{\$\} \\
 E &\mapsto .( S ) \quad , \quad \{\$\}
 \end{aligned}$$
 Note:  $\$$  added for reason 2  
 $\delta$  is  $\epsilon$
- All items are distinct, so we're done

## 8.4.2 Using The DFA



The behavior is determined if:

- There is no overlap among the lookahead sets for each reduce item, and
- None of the lookahead symbols appear to the right a  $.$

### 8.4.3 LR Variant: LALR(1)

Consider for example the following two LR(1) states:

$$S_1 : \quad \{[X \rightarrow \alpha. , a], [Y \rightarrow \beta. , c]\}$$

$$S_2 : \quad \{[X \rightarrow \alpha. , b], [Y \rightarrow \beta. , d]\}$$

They have the same core and can therefore be *merged*. The merged state contains:

$$\{[X \rightarrow \alpha. , a/b], [Y \rightarrow \beta. , c/d]\}$$

These are so-called **LALR(1)** states. Typically there are 10 times fewer LALR(1) states than LR(1). However, LALR(1) may introduce new reduce/reduce conflicts (but not new shift/reduce conflicts).