

# Programming Language Implementation VI

In this lecture we will look at **bottom-up parsing**.

- **Table driven bottom-up parsing.**
- **LR parsing.**

The material is (loosely) based on Aho et al Chapter 4.

## Bottom-Up Parsing

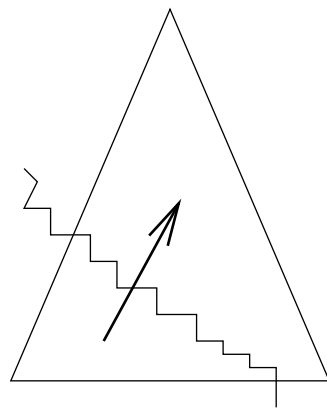
In **bottom-up parsing** the parse tree is built from the leaves upwards interpreting productions from right to left (ie. the input is reduced to the start symbol of  $G$ ).

Example

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow bC \\ B &\rightarrow d \\ C &\rightarrow bcC \mid f \end{aligned}$$

inverted rightmost derivation:

$$abbcfde \rightarrow abbcCde \rightarrow abCde \rightarrow aAde \rightarrow aABe \rightarrow S$$



bottom-down  
parsing

In the next two lectures we will look at **table-driven** bottom-up parsers and a generic **dynamic programming** approach.

## Why Bottom-Up Parsing

- Left Recursion
  - Top-down parsers loop infinitely for left-recursive grammars.
  - Bottom-up parser can process left-recursion.
- Expressiveness
  - The class LL(1) of grammars that can be parsed deterministically using a top-down parser is a proper subset of the class LR(1) of grammars that can be parsed deterministically using bottom-up techniques with a lookahead of 1.
  - LR parsing, the most general known non-backtracking shift-reduce method, works for almost all the known programming language constructs (exception e.g. Haskell).
- Error Handling
  - Several well-known error handling techniques are applicable for bottom-up parsing. LR parsing can detect errors as soon as it is possible on a left to right scan.
- **Disadvantage:** Table construction is expensive.

## Shift Reduce Parsing

Most efficient bottom-up parsing algorithms are based on **shift-reduce** parsing. This

- processes symbols left-to-right
- has limited lookahead
- no backtracking
- constructs the derivation tree bottom-up.

Operator-precedence parsing and LR parsing are well-known examples of shift-reduce based parsing.

Shift-reduce parsing constructs a **rightmost** derivation in reverse, reducing a **handle** at each step.

A **handle**  $h$  of a string of grammar symbols  $s$  is a substring that matches the RHS of some production  $P : A \rightarrow h$ , and whose reduction to  $A$ , the LHS of  $P$ , is a step along a reverse rightmost derivation of  $s$ .

$$S \xleftarrow{*}_{rrm} aAw \xleftarrow{rrm} ahw = s$$

Obviously not every RHS of a production is a handle.

## Handles

Consider the grammar

$$exp \rightarrow exp + exp \mid exp * exp \mid \mathbf{int} \mid (exp)$$

This has the **rightmost derivation**

$$\begin{aligned} exp &\rightarrow_{rm} \underline{exp + exp} \\ &\rightarrow_{rm} exp + \underline{exp * exp} \\ &\rightarrow_{rm} exp + exp * \underline{int_3} \\ &\rightarrow_{rm} exp + \underline{int_2} * int_3 \\ &\rightarrow_{rm} \underline{int_1} + int_2 * int_3 \end{aligned}$$

Shift-reduce parsing works as follows:

Right-sentential form	Production
$\underline{int_1} + int_2 * int_3$	$exp \rightarrow \mathbf{int}$
$exp + \underline{int_2} * int_3$	$exp \rightarrow \mathbf{int}$
$exp + exp * \underline{int_3}$	$exp \rightarrow \mathbf{int}$
$exp + \underline{exp * exp}$	$exp \rightarrow exp * exp$
$\underline{exp + exp}$	$exp \rightarrow exp + exp$
$exp$	

where the underlined symbols are called **handles**.

Note that this grammar is ambiguous and  $exp + exp * exp$  has two possible handles.

## Stack Implementatation of Shift-Reduce Parsing

The idea is to use a **stack** to hold the grammar symbols (terminals and non-terminals).

The parser starts with

STACK	INPUT
\$	$a_1a_2\dots a_n$ \$

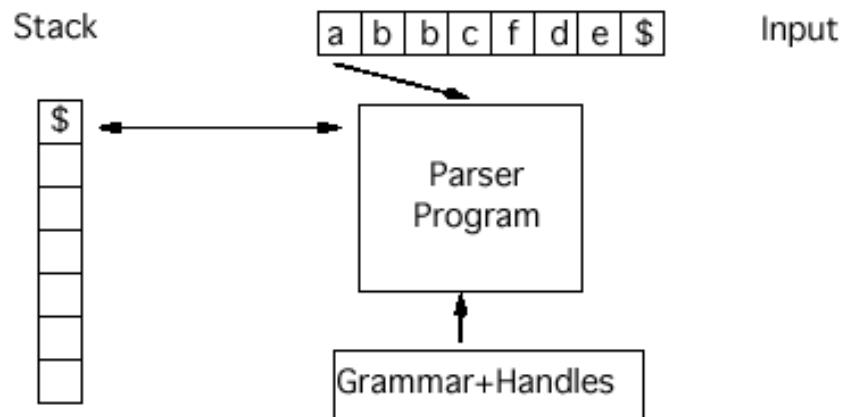
and wants to reach

STACK	INPUT
\$ S	\$

The parser repeatedly performs one of the following actions

- A **shift** action which pushes the next input symbol on top of the stack.
- A **reduce** action which pops a **handle** from the stack chooses a production and pushes the production's LHS symbol onto the stack.
- An **accept** action when the string is parsed.
- An **error** action when the parser discovers a syntax error.

## Shift Reduce Algorithm



```
repeat
  repeat
    shift current input symbol on stack;
    advance input pointer
  until a handle is on top of the stack or input is empty
  reduce handle (if present) to its corresponding LHS
  (= pop handle; push LHS)
until stack=$S or input=$
if stack=$S and input=$ accept.
```

A stack is an appropriate data structure, because in SR parsing a handle will always appear on top of the stack, never inside.

## Example of Shift-Reduce Parsing

STACK	INPUT	ACTION
\$	$int_1 + int_2 * int_3$ \$	shift
\$ $int_1$	$+int_2 * int_3$ \$	reduce by $exp \rightarrow \mathbf{int}$
\$ $exp$	$+int_2 * int_3$ \$	shift
\$ $exp +$	$int_2 * int_3$ \$	shift
\$ $exp + int_2$	$*int_3$ \$	reduce by $exp \rightarrow \mathbf{int}$
\$ $exp + exp$	$*int_3$ \$	shift
\$ $exp + exp *$	$int_3$ \$	shift
\$ $exp + exp * int_3$	\$	reduce by $exp \rightarrow \mathbf{int}$
\$ $exp + exp * exp$	\$	reduce by $exp \rightarrow exp * exp$
\$ $exp + exp$	\$	reduce by $exp \rightarrow exp + exp$
\$ $exp$	\$	accept

Shift-reducing parsing is based on the fact that a handle must always appear on **top** of the stack.

## Conflicts in Shift-Reduce Parsing

When backtracking is not used, the parser has to decide deterministically at each step which action to apply.

Some grammars produce conflicts that render the parser unable to decide deterministically. These cannot be used for deterministic shift-reduce parsing (without backtracking).

- shift/reduce conflicts: It cannot be decided whether to shift or to apply a production.
- reduce/reduce conflicts: It cannot be decided which of several productions to apply.

Example

$$\begin{aligned} stmt &\rightarrow \mathbf{if} \text{ expr } \mathbf{then} \text{ stmt} \\ stmt &\rightarrow \mathbf{if} \text{ expr } \mathbf{then} \text{ stmt } \mathbf{else} \text{ stmt} \\ stmt &\rightarrow \text{other forms} \dots \end{aligned}$$

When `if expr then stmt` is on top of the stack and `else` is the current input symbol, a shift/reduce conflict occurs.

## Operator Grammars

We defer the discussion of how shift/reduce parser for a very general class of languages can be built (in a rather complex way) and first illustrate how this can be done by hand in a special case.

**Operator grammars** are grammars

- without  $\epsilon$  production
- in which no production RHS has two adjacent non-terminals

This class of grammars is interesting because it captures arithmetic expressions (which are difficult to handle with other simpler parsing techniques).

Example

$$\begin{aligned} E &\rightarrow EAE \mid (E) \mid id \\ A &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

is not an operator grammar. However, we can easily transform it into one:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$$

Historically, many parsers have been built on recursive descent techniques employing operator descent techniques in sub-parser for expressions.

## Operator Precedence Relations

To implement an operator precedence (shift-reduce) parser we introduce so-called **precedence relations** that help us to keep track of handles.

Precedence relations are defined on pairs of terminals

Relation	Interpretation
$a \triangleleft b$	$a$ yields precedence to $b$
$a \doteq b$	$a$ and $b$ have same precedence
$a \triangleright b$	$a$ takes precedence over $b$

To decide when to shift and when to reduce in operator precedence parsing we remove all non-terminal symbols from the sentential form that we want to reduce and insert the proper precedence relations.

We then shift/reduce using the following steps:

1. Scan the string from the left until the first  $\triangleright$  is encountered.
2. Scan backwards skipping all  $\doteq$  relations until the first  $\triangleleft$  is encountered.
3. Everything between the two relation symbols found in this way is the handle to be reduced. This includes, of course, all the non-terminals between the symbols as well as potentially the surrounding non-terminals.

Intuitively we could say that the precedence relations recover the proper parentheses for the expression so that it can properly be parsed.

## Operator Precedence Example

Consider the grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

A useful precedence table is

	<i>id</i>	+	*	\$
<i>id</i>		▷	▷	▷
+	◁	▷	◁	▷
	◁	▷	▷	▷
\$	◁	◁	◁	

Note: We need to use \$ as the delimiter of the sentential form.

The sentential form

$id + id * id$  will be transformed into

$$\$ \triangleleft id \triangleright + \triangleleft id \triangleright * \triangleleft id \triangleright \$$$

So the leftmost  $id$  will be reduced first, followed by the other  $ids$ .  
After this we obtain

$E + E * E$  from which we drop the non-terminals and insert relations.  
This yields

$$\$ \triangleleft + \triangleleft * \triangleright \$ \text{ which gives us the handle } E * E.$$

## Constructing Precedence Tables

Obviously, the magic lies in the construction of the operator precedence table.

The following heuristics helps if we have an arithmetic expression language with well-defined precedence and associativity rules.

1. for two operators  $\phi_1, \phi_2$  if  $\phi_1$  has higher precedence than  $\phi_2$  make  $\phi_1 \triangleright \phi_2$  and  $\phi_2 \triangleleft \phi_1$ . for example  $* \triangleright +, + \triangleleft *$ . In this way the handle belonging to the innermost level of “parentheses” will be selected first.
2. for two operators  $\phi_1, \phi_2$  if  $\phi_1$  has the same precedence as  $\phi_2$ :
  - if  $\phi_1, \phi_2$  are left-associative, make  $\phi_1 \triangleright \phi_2, \phi_2 \triangleright \phi_1$ . For example,  $+ \triangleright +, + \triangleright -, - \triangleright -, - \triangleright +$ .
  - if  $\phi_1, \phi_2$  are right-associative, make  $\phi_1 \triangleleft \phi_2, \phi_2 \triangleleft \phi_1$ .

In this way the leftmost/rightmost subexpression of equal precedence will be selected first.

3. for all operators  $\phi$  make  $\phi \triangleleft id, id \triangleright \phi, \phi \triangleleft (, ( \triangleleft \phi, ) \triangleright \phi, \phi \triangleright ), \phi \triangleright \$, \$ \triangleleft \phi$ . This forces the handle between the \$ end markers (and makes sure that identifier and expressions in parentheses are reduced first).

## Operator Precedence Parsing Algorithm

```
initialize the stack to $;
initialize the input buffer to w$;

repeat forever
    if ($ is on top of the stack and ip point to $)
        then return;
    else begin
        let a be the topmost terminal on the stack
let b be the symbol that ip points to;

if a < b or a = b then begin (* shift *)
    push b onto the stack;
    advance ip to next input symbol;
end;
else if a > b then (* reduce *)
    repeat
        x:= pop();
        until top() < x
    else error();
end;
```

Notation: in the algorithm we have used  $<$  for  $\triangleleft$ ,  $>$  for  $\triangleright$ ,  $=$  for  $\doteq$ .

## Unary Operators in Precedence Parsing

Unary operators usually present a little bit more of a challenge.

For a “special” unary operator (for example, the logical negation  $\neg$ ) we can define the precedence relatively easily:

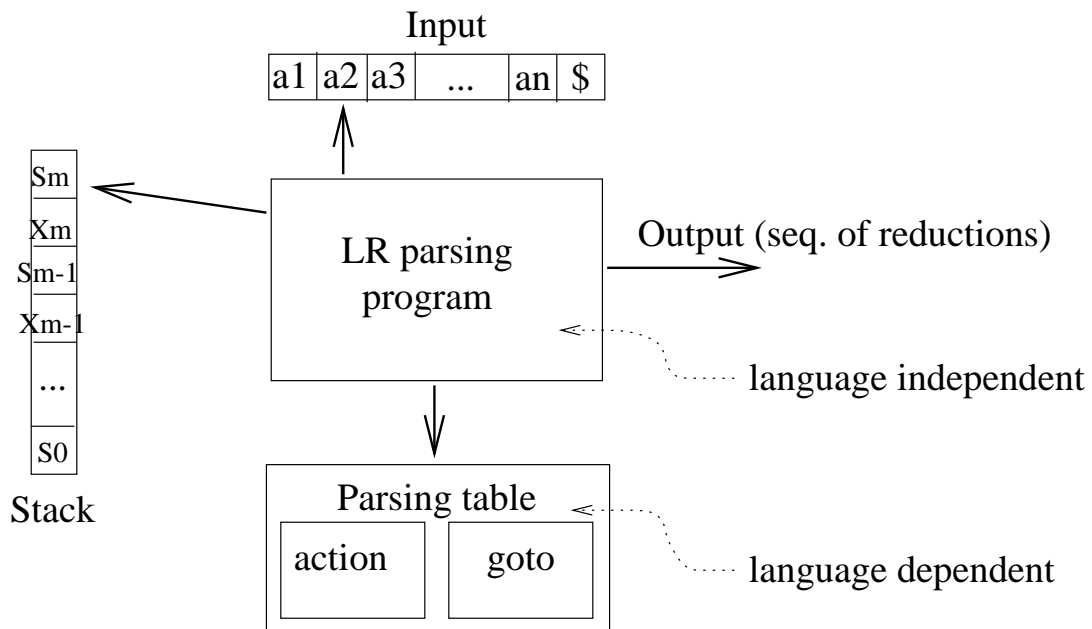
- for any operator  $\phi$  make  $\phi \triangleleft \neg$
- for operators of higher precedence than  $\neg$  make  $\neg \triangleleft \phi$
- for operators of lower precedence than  $\neg$  make  $\neg \triangleright \phi$

Note that the situation becomes more challenging if we have operators that are used both as unary and binary operators, for example “-” or “not”.

# LR Parsers

## LR parsers

- work for most context-free languages,
- can be implemented efficiently,
- allow good error handling,
- are very complex but **parser generators** help.



## LR Parsing Algorithm

**repeat forever**

$s := \text{top}(\text{stack})$

$a :=$  current input symbol

**if**  $\text{action}[s, a]$  is *shift*  $s'$  **then**

    push  $a$  then  $s'$  on to *stack*

    advance to next input symbol

**else if**  $\text{action}[s, a]$  is *reduce*  $A \rightarrow \beta$  **then**

    pop  $2 \times |\beta|$  symbols off *stack*

$s' := \text{top}(\text{stack})$

    push  $A$  then  $\text{goto}[s', A]$  on to *stack*

    output " $A \rightarrow \beta$ "

**else if**  $\text{action}[s, a]$  is *accept* **then**

**return**

**else** *error*()

## Example of LR Parsing

Recall the grammar

$$exp \rightarrow exp + term \quad (1)$$

$$exp \rightarrow term \quad (2)$$

$$term \rightarrow term * factor \quad (3)$$

$$term \rightarrow factor \quad (4)$$

$$factor \rightarrow (exp) \quad (5)$$

$$factor \rightarrow \mathbf{int} \quad (6)$$

The parsing table for this grammar is

<i>STATE</i>	<i>ACTION</i>					<i>GOTO</i>		
	<b>int</b>	+	*	( )	\$	exp	term	factor
0	<i>s</i> 5			<i>s</i> 4		1	2	3
1		<i>s</i> 6			<i>acc</i>			
2		<i>r</i> 2	<i>s</i> 7		<i>r</i> 2	<i>r</i> 2		
3		<i>r</i> 4	<i>r</i> 4		<i>r</i> 4	<i>r</i> 4		
4	<i>s</i> 5			<i>s</i> 4		8	2	3
5		<i>r</i> 6	<i>r</i> 6		<i>r</i> 6	<i>r</i> 6		
6	<i>s</i> 5			<i>s</i> 4			9	3
7	<i>s</i> 5			<i>s</i> 4				10
8		<i>s</i> 6			<i>s</i> 11			
9		<i>r</i> 1	<i>s</i> 7		<i>r</i> 1	<i>r</i> 1		
10		<i>r</i> 3	<i>r</i> 3		<i>r</i> 3	<i>r</i> 3		
11		<i>r</i> 5	<i>r</i> 5		<i>r</i> 5	<i>r</i> 5		

where

*si* is shift and stack state *i*

*rj* is reduce using production *j*

*acc* is accept.

## Example of LR Parsing (Cont.)

STACK	INPUT	ACTION
0	$int_1 + int_2 \$$	

## Classes of LR Parsers

There are several different classes of LR grammars

- Canonical LR
  - largest class of LR grammars
  - large number of states, expensive construction
  - stops immediately at error position without further reduction
- SLR (simple LR)
  - smallest class of LR grammars
  - small number of states, simple construction
  - makes unnecessary reduce moves when error is encountered
- LALR (lookahead LR)
  - more powerful than SLR, but less than canonical LR
  - approximately same table size as SLR
  - intermediate construction complexity
  - employed in real parser generators like YACC.

The basic idea for all these methods is to construct a goto table that models the transition function of a DFA recognizing the viable prefixes of the grammar.

## Viable Prefixes

**Handles** can only appear on top of the stack in SR parsing.

A prefix of a right sentential form that can appear on top of the stack of an SR parser is called a **viable prefix**.

This is equivalent to the following definitions:

- A prefix of a right sentential form  $w$  is called a viable prefix, if it does not extend past the right hand of the rightmost handle of  $w$ .
- If  $S \xleftarrow{*}_{rrm} uXw \xleftarrow{rrm} uvw$  then  $v$  is a handle of  $uvw$  and each prefix of  $uv$  is a viable prefix.

Therefore, as long as the input seen so far can be reduced to a viable prefix, no error has occurred.

## Summary

We have looked at bottom-up parsing:

- Table driven bottom-up parsing.
- Operator-precedence parsing.
- LR parsing.

## Homework

- Read Chapter 4 of Aho et al.