Bottom-Up (Shift/Reduce) Parsing

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

©Hans de Nivelle, 2025

On previous slides we have seen top-down parsing.

It is usually implemented by hand: One has to write a parse function for each non-terminal symbol.

One usually has to merge grammar rules, and allow more general forms, in order to make it work.

We now discuss another approach to parsing, called bottom-up parsing.

It has the following advantages over top-down parsing:

- Attribute computation is easy.
- Since choices are made only when the complete right-hand side is read, common prefixes are unproblematic. Because of this, there is usually no need to modify grammar rules.
- The parser can be generated automatically.

One big disadvantage is the fact that bottom-up parsing does not suppport left/right information flow. (For example, checking declarations of variables) This can be handled by global variables.

Bottom-Up Parsing

The parser uses two variables: The parsestack (stack of symbols with their attributes), and a lookahead (optional symbol).

shift (Lookahead must be defined): Push the current lookahead to the stack. Make the lookahead undefined.

reduce (The top of the stack must contain the right hand side of a rule): Remove the right hand side from the stack, and replace it by the left hand side of the rule. Compute the attribute of the new symbol.

read (Lookahead must be undefined): Read a symbol from the input source and put it in lookahead.

Because people in the 70-ies didn't like undefinedness, this style of parsing is also called shift/reduce parsing.

Reductions are always made on the top of the stack, never in the middle.

Example

The same grammar that we used for top-down parsing:

$$
\begin{array}{lcllll}
E & \rightarrow & E + E_1 & | & E - E_1 & | \quad E_1 \\
E_1 & \rightarrow & E_1 \times E_2 & | & E_1 \,/\, E_2 & | \quad E_2 \\
E_2 & \rightarrow & -E_2 & | & E_3 & \\
E_3 & \rightarrow & \textbf{int} & | & \textbf{double} & | \quad \textbf{ident} \quad | \\
 & & \textbf{ident}\,(\,A\,) & | & (\,E\,) & \\
A & \rightarrow & E & | & A\,,\ E &
\end{array}
$$

$$\Sigma = \{E, E_1, E_2, E_3, A\} \cup$$

$$= \{\ +,\ -,\ \times,\ /,\ \textbf{int},\ \textbf{double},\ \textbf{ident},\ (,\ ),\ ',' \ \}.$$

Start symbol is E.

Assume that we want to parse $f(a, b + 3)$.

The parser starts with empty stack, and empty lookahead. It will first read, after which **ident** with attribute f is the lookahead.

The parser decides to shift. Now the stack contains **ident** with attribute f and the lookahead is empty again.

We can either reduce $E_3 \rightarrow$ **ident**, or try to read the next symbol. Since the next symbol could be '(', we have to read.

Now we have **ident** with attribute 3 on the stack, and $'('$ in the lookahead.

We could reduce $E_3 \rightarrow$ **ident**, but if we do that, we will get stuck, because there cannot be a '(' after an E_3.

So we decide to shift '('. Now the stack contains **ident**, '(' and lookahead is empty again.

The rest of the parse continues as below. (I left out the attributes for simplicity.)

stack	lookahead	input	decision
ident (		a, b + 3)	read
string (	**string**	, b + 3)	shift
string (**string**		, b + 3)	reduce
string (E_3		, b + 3)	reduce
string (E_2		, b + 3)	reduce
string (E_1		, b + 3)	reduce
string (E		, b + 3)	reduce
string (A		, b + 3)	read
string (A	,	b + 3)	shift
string (A ,		b + 3)	read

stack	lookahead	input	decision
string (A ,	**string**	+ 3)	shift
string (A , **string**		+ 3)	reduce
string (A , E_3		+ 3)	reduce 3 times
string (A , E		+ 3)	read
string (A , E	+	3)	shift
string (A , E +		3)	read
string (A , E +	3	)	shift
string (A , E + **double**		)	reduce
string (A , E + E_3		)	reduce 2 times

stack	lookahead	input	decision
string (A , E + E_1		)	reduce
string (A , E		)	reduce
string (A		)	read
string (A	)		shift
string (A)			reduce
E_3			reduce 3 times
E			input accepted

Making the Decisions

The parser has the following options:

- If the top of the stack matches the right hand side of a rule, it can reduce.
- If it has a lookahead, it can shift this lookahead.
- If there is no lookahead and there is input left, it can read.

If exactly one reduction is possible, and it is certain that this reduction has to be made, then the parser reduces.

Otherwise, if there is no lookahead, the parser reads.

The hard part is choosing between shifting and reducing, or choosing the right reduction, when more than one reduction is possible. For this, the parser uses the lookahead.

The underlying theory will be shown in another set of slides.

Comparing Top-Down with Bottom-Up

When function **parseE**( ) is called, it has to decide immediately, if it calls **parseE** again, or **parseE$_1$**. The choice depends on a '+' or '-', which it cannot see yet. Because of this, we have to change the grammar.

The bottom-up parser chooses between the rules
$E \to E_1$, $E \to E_1 \times E_2$, $E \to E_1 \,/\, E_2$, when E_1 is already on the stack. At that moment, much more of the input has been read, and the choice is easier.

Because of this, it is usually not necessary to change the grammar with a bottom-up parser.

A similar situation occurs when the grammar contains:

Stat	$\to$	**if** E **then** Stat
		if E **then** Stat **else** Stat

The choice needs to be made only after Stat.

Parser Generation Tools/Practical Aspects

There exist many parser generation tools that support attribute grammars. (Yacc, Bison, CUP, Maphoon).

In Maphoon, the attribute functions are represented by C^{++} code.

A rule of form $A \to A + B$ with atttribute function
$f(x, y, z) = x + z$ is represented by:

```
expr => expr:e1 PLUS expr1:e2 { return e1 + e2; }
```

LALR parsing

LALR stands for look ahead left right. It is a technique for deciding when reductions have to be made in shift/reduce parsing.

Often, it can make the decisions without using a look ahead. Sometimes, a look ahead of length one is required.

Most parser generators (and in particular Bison, Yacc and CUP) construct LALR parsers.

In LALR parsing, a deterministic finite automaton is used for determining when reductions have to be made. The deterministic finite automaton is usually called prefix automaton.

I explain how it is constructed in another set of slides.