Bottom-Up Parsing

- So far we have seen parsers that try to find a derivation from the starting grammar symbol to the input sting.
- In this section we will be considering the essentially equivalent approach of finding a path from the input sentence to the starting symbol (bottom-up).
- We will be discussing a general technique for bottom-up parsing called shift-reduce parsing and an implementation of it called LR parsing.
- This method is used in automatic parser generators such as BISON, which we will cover in the next lessons.

Notes on Shift-Reduce Parsing

- Bottom-Up parsing can be thought of **reducing** the input string to the starting symbol of the grammar.
- Each reduction step involves a sequence of symbols from the input being replaced by a left hand side non-terminal according to the grammar rules.
- Just as in producing a Top-Down derivation there may be several non-terminals that may be expanded at any one step, in bottom-up parsing, there may be several sequences of symbols in the input string that may be reduced in a step.

LL and LR Parsing

- It is important to choose a parsing methodology to apply to each derivation or reduction and apply it consistently.
- In in top-down parsing we always expand the leftmost non-terminal in a sentential form then we obtain a **leftmost** derivation. If we always expand the rightmost non-terminal then the derivation is **rightmost**.
- The rule that we will apply to bottom-up parsing is to always reduce the sequence of symbols which would trace a rightmost derivation in reverse. So, the input is scanned from left to right (hence LR).
- LL parsers scan the input from left to right to produce a leftmost derivation.

Example (1)

- Consider the following grammar

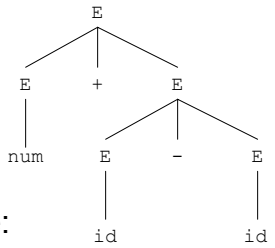
```
E ::= E + E
E ::= E - E
E ::= id
E ::= num
```

- And the input string '`1 + x - y`'. A leftmost derivation would be:

```
E → E + E
  → num + E
  → num + E - E
  → num + id - E
  → num + id - id
```

Example (2)

- The parse tree would be:



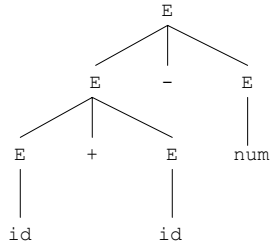
- A rightmost derivation would be:

E → E + E
→ E + E – E
→ E + E – id
→ E + id – id
→ num + id – id

- Which gives the same parse tree.

Example (3)

One should note however that the grammar is ambiguous. We could have the following derivations:



Leftmost	Rightmost
E → E – E	E → E – E
→ E + E – E	→ E – id
→ num + E – E	→ E + E – id
→ num + id – E	→ E + id – id
→ num + id – id	→ num + id – id

Rightmost Derivations in Reverse (1)

- The first rightmost derivation we had was:

```
E → E + E
  → E + E – E
  → E + E – id
  → E + id – id
  → num + id – id
```

- If applied in reverse we get a bottom-up parse:

```
num + id – id → E + id – id
              → E + E – id
              → E + E – E
              → E + E
              → E
```


Rightmost Derivations in Reverse (2)

- The second rightmost derivation we had was:

```
E → E – E
  → E – id
  → E + E – id
  → E + id – id
  → num + id – id
```

- If applied in reverse we get a bottom-up parse:

```
num + id – id → E + id – id
              → E + E – id
              → E - id
              → E - E
              → E
```

Handles (1)

- The term **input string** will be used to refer to any sentential form in the reduction of a sentence to the starting symbol.
- The term **handle** is a sequence of symbols in the input string which, if replaced by a matching left hand side non-terminal, leads to the tracing out of the reversed rightmost derivation of the original sentence.
- Consider the example we had before in reducing `num + id – id` to `E`.

Handles (2)

Right Sentential Form	Handle	Reducing Production
num + id1 – id2	num	E → num
E + id1 – id2	id1	E → id
E + E – id2	id2	E → id
E + E – E	E – E	E → E – E
E + E	E + E	E → E + E
E		

Handles (3)

Right Sentential Form	Handle	Reducing Production
num + id1 – id2	num	E → num
E + id1 – id2	Id1	E → id
E + E – id2	E + E	E → E + E
E – id2	id2	E → id
E – E	E – E	E → E – E
E		

Handles (4)

- Note that upon having reached the input sting '`E + E – id`', there are two possible handles '`id`' or '`E + E`'. This reflects the ambiguity in the grammar. If a grammar is unambiguous, then for any input string there would be only one handle for any stage in the reduction.
- An issue in designing a bottom-up parser is to
  - Decide how to locate handles in the input string.
  - How to choose which left hand side to replace with assuming that there may be more than one left hand side for a handle.

Stack Implementation of Shift Reduce Parsing (1)

- Shift-reduce parsers are usually implemented using a stack to hold grammar symbols and an input buffer to hold the string X to be parsed.
- We shall use the dollar symbol, $, to denote the bottom of the stack and the end of the input string.
- Initially we would have:

Stack	Input String
$	X$

Stack Implementation of Shift Reduce Parsing (2)

- The parse works by shifting symbols from the input string to the stack until a handle appears on the top of the stack.
- The parser reduces the handle on top of the stack to the left hand side of the appropriate production.
- The cycle is repeated until we find an error or the stack consists of the starting symbol and the input is empty.
- At this point, the parser stops and reports success.

Stack	Input String
$	X$

Example (1)

- Consider the following unambiguous grammar…

```
E → E + T
E → E – T
E → T
T → id
T → num
```

- …to parse '`num + id + id`'

Example (2)

Stack	Input	Action
$	num + id – id $	Shift num
$ num	+ id – id $	Reduce T → num
$ T	+ id – id $	Reduce E → T
$ E	+ id – id $	Shift +
$ E +	id – id $	Shift id
$ E + id	– id $	Reduce T → id
$ E + T	– id $	Reduce E → E + T
$ E	– id $	Shift –
$ E –	id $	Shift id
$ E – id	$	Reduce T → id
$ E – T	$	Reduce E → E – T
$ E	$	Accept

Parsing Actions

- Shift:
  - In a shift operation, the next input symbol is put on the top of the stack.
- Reduce:
  - The handle which is on top of the stack is replaced by the appropriate non-terminal symbol.
- Accept:
  - The parser accepts when all the input symbols are consumed and there is the sentence symbol on the stack.
- Error:
  - The parser encounters an error an calls any error recovery routine.

Shift-Reduce Conflicts (1)

- There are grammars that cannot be parsed by a shift reduce parser. In such cases, the parser can get into a state in which it cannot decide whether to shift or reduce.
- Ambiguous grammars are of such a type because there may be more than one handle at a time under certain circumstances.
- Consider the dangling else grammar:

  `S → if E then S | if E then S else S`

Shift-Reduce Conflicts (2)

- The shift-reduce parser may find itself in the following situation:

Stack	Input String
`$ if E then if E then S`	`Else S $`

- At this point the parser wouldn't know whether to shift the else onto the stack or reduce the first production for s to get:

Stack	Input String
`$ if E then S`	`Else S $`

Good News

- Shift-reduce parsers may be easily modified to handle such grammars in a consistent way. For example, such shift-reduce conflicts may be resolved by forcing the parser to shift.
- Shift-reduce conflicts are not very common and are often an indication that there is a problem in the definition of the language.