Technical Aside

Viable prefix -- automaton

References:

- https://stackoverflow.com/questions/4202181/explanation-about-viable-prefix
- Lecture notes at http://www.cs.williams.edu/~tom/courses/434/outlines/lect14_2.html
- Parsing theory vol II Book:

  Sippu, Seppo, and Eljas Soisalon-Soininen. *Parsing Theory: Volume II LR (k) and LL (k) Parsing*. Vol. 20. Springer Science & Business Media, 2013.

Right-sentential form

- A **right-sentential form** is a sentential form which can be reached by rightmost derivation,
  – which is another way to describe repeated expansion of only the rightmost non-terminal when proceeding from the start symbol.
  – This is a rightmost derivation, and all the forms in it are therefore right-sentential forms:

Example: right-sentential form

- For the followng grammar, Terminals = { **NUMBER**, +, *, (,) }

```
expr -> expr + term | term
term -> term * factor | factor
factor -> NUMBER | ( expr )
```

- Each *right hand side* is a right-sentential form

```
expr -> expr + term
     -> expr + term * factor
     -> expr + term * NUMBER
     -> expr + factor * NUMBER
     -> expr + NUMBER * NUMBER
     -> expr + term + NUMBER * NUMBER
     -> expr + NUMBER + NUMBER * NUMBER
     -> term + NUMBER + NUMBER * NUMBER
     -> NUMBER + NUMBER + NUMBER * NUMBER
```

Last one is a right-most sentential form and a sentence.

Prefix of a sentential form

- A **prefix** of a sentential form (whether right or otherwise) is a sequence of input symbols that consists of zero or more leading symbols of that sentential form.
  - The empty sequence is trivially a prefix of every sentential form, and
  - the complete sequence of symbols making up a sentential form is also trivially a prefix of it.
- Can be generalized to prefix of a right-sentential form.

```
expr -> expr + term | term
term -> term * factor | factor
factor -> NUMBER | ( expr )
```

```
expr -> expr + term
     -> expr + term * factor
     -> expr + term * NUMBER
     -> expr + factor * NUMBER
     -> expr + NUMBER * NUMBER
     -> expr + term + NUMBER * NUMBER
     -> expr + NUMBER + NUMBER * NUMBER
     -> term + NUMBER + NUMBER * NUMBER
     -> NUMBER + NUMBER + NUMBER * NUMBER
```

- Some prefixes of right-sentential forms:

ϵ

`expr`

`expr + term`

`expr + term *`

`expr +`

`expr + term * factor`

`expr + term * NUMBER`

`expr + factor *`

`term + NUMBER +`

`NUMBER + NUMBER +`

`NUMBER + NUMBER + NUMBER *`

```
expr -> expr + term | term
term -> term * factor | factor
factor -> NUMBER | ( expr )
```

```
expr -> term
     -> term * factor
     -> . .
```

- These are also prefixes of right-sentential forms:

```
term
```

```
term *
```

```
term * factor
```

Simple phrase

- A **simple phrase** is the expansion of a single non-terminal symbol that holds a place in a sentential form.
  - In the given example, **term * factor** is a simple phrase because it is an expansion of **term.**
  - **term, factor, NUMBER, term * factor, (expr)** are also simple phrases

```
expr -> expr + term | term
term -> term * factor | factor
factor -> NUMBER | ( expr )
```

Body of a production is a simple phrase.

The handle

- The **handle:** In a right-sentential form, it is a *simple phrase* that is included because of expansion of a non-terminal most recently.
  - In a rightmost derivation, the handle is easy to identify, since
    - it's the sequence of symbols that resulted from the most recently expanded non-terminal.
  - In LR parsing, **the handle** is the *simple phrase* that needs to be *reduced* at that point of parsing.

```
expr -> expr + term | term
term -> term * factor | factor
factor -> NUMBER | ( expr )
```

```
expr -> expr + term
     -> expr + term * factor
     -> expr + term * NUMBER
     -> expr + factor * NUMBER
     -> expr + NUMBER * NUMBER
     -> expr + term + NUMBER * NUMBER
     -> expr + NUMBER + NUMBER * NUMBER
     -> term + NUMBER + NUMBER * NUMBER
     -> NUMBER + NUMBER + NUMBER * NUMBER
```

```
expr -> expr + term
     -> expr + term * factor
     -> expr + term * NUMBER
     -> expr + factor * NUMBER
     -> expr + NUMBER * NUMBER
     -> expr + term + NUMBER * NUMBER
     -> expr + NUMBER + NUMBER * NUMBER
     -> term + NUMBER + NUMBER * NUMBER
     -> NUMBER + NUMBER + NUMBER * NUMBER
```

handle

Viable prefix

- A **viable prefix** of a right-sentential form is a prefix which does not extend beyond that form's handle –
  - in other words, that prefix which contains no reducible simple phrases, except possibly the handle (in this case, the prefix extends exactly to the end of the handle).

```
expr -> expr + term
     -> expr + term * factor
     -> expr + term * NUMBER
     -> expr + factor * NUMBER
     -> expr + NUMBER * NUMBER
     -> expr + term + NUMBER * NUMBER
     -> expr + NUMBER + NUMBER * NUMBER
     -> term + NUMBER + NUMBER * NUMBER
     -> NUMBER + NUMBER + NUMBER * NUMBER
```

```
expr -> expr + term | term
term -> term * factor | factor
factor -> NUMBER | ( expr )
```

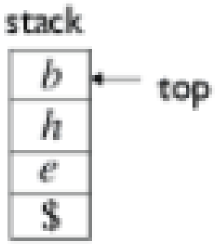
- Some viable prefixes:

  ε, expr, expr +, expr + tem, expr + tem *,
  term, NUMBER, term * NUMBER

  expr -> term
  -> term * factor
  -> term * NUMBER

- Below are not viable prefixes:

  expr + factor *, term +, term + NUMBER,
  NUMBER +, NUMBER + NUMBER



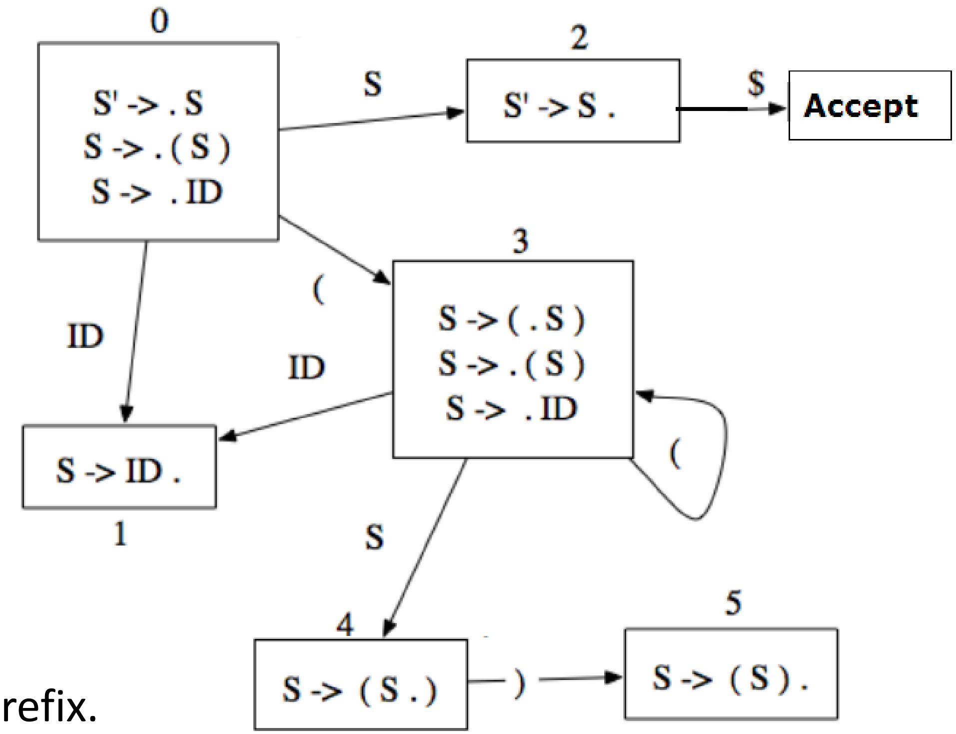
Viable prefix on stack makes you happy

- From a shift-reduce parser's point of view, as long as you have a viable prefix on the stack, you can expect for a successful parse.
  - if handle appears on top of the stack, you will reduce.
  - Else you will shift the next token
    - if this makes the contents of stack in to a **non-viable prefix**, you will declare *fail*.

WHY VIABLE PREFIX?

- **The set of all viable prefixes of a context-free language is itself a regular language!**
- You can therefore build a finite automaton that recognizes the regular language of viable prefixes, and use it to determine when to shift and when to reduce.
- This combination of a stack and a finite state machine is essentially a push-down automaton, which is exactly the class of automaton needed to recognize a context-free language.

$$
\begin{array}{l}
(0)\ S \rightarrow (S) \\
(1)\ \ S' \rightarrow S \\
(2)\ S \rightarrow id
\end{array}
$$



By traversing the LR(0) automaton,
the string you get is always a viable prefix.
Eg: Few viable prefixes:

(
((
((S
(((id

Few non-viable prefixes:

id id
()
)

Stack containing a viable prefix

- Stack containing a viable prefix α says that
  - There is a sequence of tokens w, such that αw is a right-most sentential form.
  - That is, we can expect a successful parse.
  - So, if the stack, somehow contains a non-viable prefix, then immediately one can say that the parse is a failure (the given string is not in the language).
  - **Then, why is that even when the input string is in that language, an LR parsing can fail?**

Why different LR parsers?

- From one viable prefix, we go to the next viable prefix.
  - We do shift or we do reduce, for this.
  - With one action, the stack content with remaining tokens is a right-sentential form.
    - if you come to this stage, you can expect a successful parse.
  - But, with some other action, the stack with remaining tokens is not a right-sentential form.
    - if you come to this stage, you will fail.
  - The parsing technique, if is unable to decide which action to follow, it may choose a wrong choice and fail.
    - The weak the parser, this happens frequently.
    - The strong the parser, this happens less frequently.

We can go for stronger parsers (ultimately, wrong choices can be totally avoided), but these are costly (need to see more look-ahead, need to maintain more information).

- In LR parsing techniques the stack always contains a viable prefix.
- But the stack content + remaining input string may fail to be a valid right-sentential form.
  - For this you need to see all the tokens that are in the input buffer.
  - This is the reason why see more on the input buffer makes the parser stronger.
    - But, this increases the cost of parsing !!

Hierarchy of languages that can be parsed by various parsers