What is Parsing? (1)

- According to Merriam-Webster OnLine (www.webster.com), parse means:
  to resolve (as a sentence) into component parts of speech and describe them grammatically
- In CS, we take this to mean answering
  \( w \in L(G)? \)
  for a CFG \( G \) by analysing the structure of \( w \) according to \( G \); i.e., to recognize the language generated by a grammar \( G \).

Parsing Strategies

There are two basic strategies for parsing: top-down and bottom up.
- A top-down parser attempts to carry out a derivation matching the input starting from the start symbol; i.e., it constructs the parse tree for the input from the root downwards in preorder.
- A bottom-up parser tries to construct the parse tree from the leaves upwards by using the productions “backwards”.

Recursive-Descent Parsing (1)

Recursive-descent parsing is a way to implement top-down parsing.
We are just going to focus on the language recognition problem:
\[
\begin{align*}
  \text{parse} :: & \text{[Token]} \rightarrow \text{Bool} \\
  \text{Token} & \text{is “compiler speak” for (input) symbol.}
\end{align*}
\]

Recursive-Descent Parsing (2)

Consider a typical production in some grammar \( G \):
\[
S \rightarrow AB
\]
Let \( L(X) \) be the language \( \{ w \in T^* | X \xrightarrow{G} w \} \).
Note that
\[
\begin{align*}
  w & \in L(S) \iff \exists w_1, w_2. \ w = w_1 w_2 \\
  & \land w_1 \in L(A) \\
  & \land w_2 \in L(B)
\end{align*}
\]
I.e., given a parser for \( L(A) \) and a parser for \( L(B) \), we can construct a parser for \( L(S) \).

Recursive-Descent Parsing (3)

But we need a way to divide the input word \( w \)!

**Idea!**
Each parser
- tries to derive a prefix of the input according to the productions for the nonterminal
- returns the remaining suffix if successful.

New type:
\[
\begin{align*}
  \text{parseX :: } & \text{[Token]} \rightarrow \text{Maybe [Token]} \\
  \text{Token} & \text{is “compiler speak” for (input) symbol.}
\end{align*}
\]

Recursive-Descent Parsing (4)

Now we can construct a parser for \( L(S) \)
\[
S \rightarrow AB
\]
in terms of parsers for \( L(A) \) and \( L(B) \):
\[
\begin{align*}
  \text{parseS :: } & \text{[Token]} \rightarrow \text{Maybe [Token]} \\
  \text{parseS ts} & = \\
  \text{case parseA ts of} \\
  \text{Nothing} & \rightarrow \text{Nothing} \\
  \text{Just ts'} & \rightarrow \\
  \text{case parseB ts' of} \\
  \text{Nothing} & \rightarrow \text{Nothing} \\
  \text{Just ts''} & \rightarrow \text{Just ts''}
\end{align*}
\]
Recursive-Descent Parsing (5)

Or we can simplify to just

```haskell
parseS :: [Token] -> Maybe [Token]
parseS ts =
  case parseA ts of
    Nothing -> Nothing
    Just ts' -> parseB ts'
```

This is called recursive-descent parsing because the parse functions (usually) end up being (mutually) recursive.

Exercise

Suppose type Token = Char and

```haskell
parseA :: [Token] -> Maybe [Token]
parsersA ('a' : ts) = Just ts
parseA _ = Nothing

parseB :: [Token] -> Maybe [Token]
parsersB ('b' : ts) = Just ts
parseB _ = Nothing
```

- Evaluate `parseA`, `parseB`, and `parseS` on “abcd”.
- What are the productions for `A` and `B`?

Recursive-Descent Parsers and PDAs

- Fundamental to the implementation of a recursive computation is a stack that
  - keeps track of the state of the computation
  - allows for subcomputations (to any depth).
- In a language that supports recursive functions and procedures, the stack isn’t explicitly visible. But internally, it is the central data structure.
- Thus, a recursive-descent parser is a kind of PDA.

Recursive-Descent Parsing (6)

We also need a way to handle choice, as in

```
S → AB | CD
```

We are first going to consider the case when the choice is obvious, as in

```
S → aAB | cCD
```

I.e. we assume it is manifest from the grammar that we can choose between productions with a one-symbol lookahead.

A Simple Recursive-Descent Parser (1)

Consider:

```
S → aA | bBA
A → aA | ε
B → bB | ε
```

We are going to need one parsing function for each non-terminal:

- `parseS :: [Token] -> Maybe [Token]`
- `parseA :: [Token] -> Maybe [Token]`
- `parseB :: [Token] -> Maybe [Token]`

Choice (1)

Now consider:

```
S → aA | bBA
    A → aA | ε
    B → bB | ε
```

In `parseS`, should `parseA` or `parseB` be called once `a` has been read?

A Simple Recursive-Descent Parser (2)

Production: `S → aA | bBA`

```haskell
parseS :: [Token] -> Maybe [Token]
parsersS ('a' : ts) =
  case parseA ts of
    Nothing -> Nothing
    Just ts' -> parseB ts'
parsersS ('b' : ts) =
  case parseB ts of
    Nothing -> Nothing
    Just ts' -> parseS ts'
parsersS _ = Nothing
```

Choice (2)

We could try the alternatives in order; i.e., a limited form of backtracking:

Production: `S → aA | bBA`

```haskell
parseS ('a' : ts) =
  case parseA ts of
    Nothing -> Nothing
    Just ts' -> parseS ts'
parseS ('b' : ts) =
  case parseB ts of
    Nothing -> Nothing
    Just ts' -> parseS ts'
```
Similarly, to handle \( \epsilon \)-productions:

Production: \( A \rightarrow aA | \epsilon \)

```haskell
parseA :: [Token] -> Maybe [Token]
parseA ('a' : ts) = parseA ts
parseA ts = Just ts
```

If the present input starts with an \( a \), consume it and continue. Only if this fails will the always successful \( \epsilon \)-rule be used! The opposite order would not be very useful.

Limited backtracking is *not* an exhaustive search: liable to get stuck in "blind alleys".

Consider:

\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow aA | \epsilon \\
B & \rightarrow ab
\end{align*}
\]

One principled approach is to try *all* alternatives; i.e., *full backtracking* (aka *list of successes*):

- Each parsing function returns a list of all possible suffixes. Type:
  ```haskell
  parseX :: [Token] -> [[Token]]
  ```
- Translate \( A \rightarrow \alpha | \beta \) into
  ```haskell
  parseA ts = parseAlpha ts ++ parseBeta ts
  ```
- An empty list indicates no possible parsing.

However:

- backtracking is computationally expensive
- issues with error reporting: where exactly lies the problem if it only *after* an exhaustive search becomes apparent that there is no possible way to parse a word?

We are going to look at another principled approach that avoids backtracking: *predictive parsing*. (But the grammar must satisfy certain conditions.)