G52MAL 2012/13: Lecture 13
Recursive-Descent Parsing: Introduction

Henrik Nilsson
University of Nottingham, UK

This Lecture

• What is Parsing?
• Recursive-Descent Parsing Fundamentals
• Handling Choice

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 \( G \).

What is Parsing? (2)

• A **parser** is a program that carries out parsing; i.e., essentially (for CFGs) a realization of a PDA.
• For most practical applications, a parser will also return a structured representation of a word \( w \in L(G) \): its **derivation** or **parse tree** (although usually a simplified version, an **Abstract Syntax Tree**).
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:

\[ w \in L(G)? \]

This suggests the following type for the parser:

\[
\text{parser} :: \left[ \text{Token} \right] \rightarrow \text{Bool}
\]

Token is “compiler speak” for (input) symbol.

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

\[
w \in L(S) \iff \exists w_1, w_2 . \ w = w_1w_2 \\
\quad \land w_1 \in L(A) \\
\quad \land w_2 \in L(B)
\]

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:

\[
\text{parseX} :: \left[ \text{Token} \right] \rightarrow \text{Maybe} \left[ \text{Token} \right]
\]

(Recall: \( \text{data Maybe a} = \text{Nothing} | \text{Just a} \))
Recursive-Descent Parsing (4)

Now we can construct a parser for $L(S)$

$$S \rightarrow AB$$

in terms of parsers for $L(A)$ and $L(B)$:

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

Exercise

Suppose `type Token = Char` and

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

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

• Evaluate `parseA`, `parseB`, and `parseS` on “abcd”.

• What are the productions for $A$ and $B$?

Recursive-Descent Parsing (5)

Or we can simplify to just

```hs
parseS :: [Token] -> Maybe [Token]
p 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.

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 datastructure.

• Thus, a recursive-descent parser is a kind of PDA.
Recursive-Descent Parsing (6)

We also need a way to handle choice, as in

\[ S \rightarrow AB | CD \]

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

\[ S \rightarrow aAB | cCD \]

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:

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

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

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

A Simple Recursive-Descent Parser (2)

Production: \( S \rightarrow aA | bBA \)

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

A Simple Recursive-Descent Parser (3)

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

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

Production: \( B \rightarrow bB | \epsilon \)

```haskell
parseB :: [Token] -> Maybe [Token]
parseB ('b' : ts) = parseB ts
parseB ts = Just ts
```

Note: Since \( A \Rightarrow \epsilon \) and \( B \Rightarrow \epsilon \), it is **not** a syntax error if the next token is not, respectively, \( a \) and \( b \).
Choice (1)

Now consider:

```
S \rightarrow aA \mid aBA
A \rightarrow aA \mid \epsilon
B \rightarrow bB \mid \epsilon
```

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

Choice (2)

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

Production: \( S \rightarrow aA \mid aBA \)

```
parseS ('a' : ts) =
  case parseA ts of
    Just ts' -> Just ts'
    Nothing ->
      case parseB ts of
        Nothing -> Nothing
        Just ts' -> parseA ts'
```

Choice (3)

Similarly, to handle \( \epsilon \)-productions:

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

```
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.

Choice (4)

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

Consider:

```
S \rightarrow AB
A \rightarrow aA \mid \epsilon
B \rightarrow ab
```
**Choice (5)**

Parsing functions:

```haskell
parseA ('a' : ts) = parseA ts
parseA ts = Just ts

parseB ('a' : 'b' : ts) = Just ts
parseB ts = Nothing

parseS ts =
  case parseA ts of
    Nothing -> Nothing
    Just ts' -> parseB ts'
```

**Choice (6)**

Will it work? Consider parsing `ab`. Clearly derivable from the grammar!

But:

```haskell
parseS "ab" = Nothing
```

Why? Because

```haskell
parseA "ab" = Just "b"
```

I.e., committed to the choice \( A \rightarrow a \), and will never try \( A \rightarrow \epsilon \): a “blind alley”.

Changing order may solve this, but will cause other problems.

**Choice (7)**

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.

**Choice (8)**

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.)