Examples: Recursive-descent predictive-parsing

Task: Determine if grammar is suitable for LL(1) parsing. If so write PDA parser code.

Recall: LL(1) is shorthand for
- consuming tokens "L"eft-to-right (the first L)
- making a "L"eftmost derivation (the second L), and
- Looking at the next "1" input token to make decisions.

Simulating PDA stack to match string to CFG is LL(1).

Suitable?

For LL(1) parsing, a grammar must be:
- Unambiguous
- Not left-recursive
- No conflicting prediction tokens

For example, the following are not LL(1)...

Ambiguous: $S \to SS \mid (S) \mid \lambda$
Left-Recursive: $S \to Sa \mid \lambda$
Prediction conflict: $S \to aaS \mid abS \mid \lambda$

Example: a*b*c*

$$S \to aS \mid T$$
$$T \to bT \mid R$$
$$R \to cR \mid \lambda$$

Not ambiguous

No left recursion

Conflicting prediction tokens?
Let's calculate them to see.

Step 1: Find Nullable

Often can tell if production is nullable by inspection.
Can also use fixed-point algorithm.
Use prior column information to fill next column.

Prod	Init	Iter 1	Iter 2	Iter 3
$S \to aS$	false	false	false	false
$S \to T$	false	false	**true**	**true**
$T \to bT$	false	false	false	false
$T \to R$	false	**true**	**true**	**true**
$R \to cR$	false	false	false	false
$R \to \lambda$	**true**	**true**	**true**	**true**

All non-terminals are nullable.

Step 2: Find First and Follow set relations

From $S \to aS$:

$$
\begin{gathered}
a \in \mathit{First}(S) \\
\mathit{Follow}(S) \subseteq \mathit{Follow}(S) \quad <== \text{Does nothing; can ignore}
\end{gathered}
$$

From $S \to T$:

$$
\begin{gathered}
\mathit{First}(T) \subseteq \mathit{First}(S) \\
\mathit{Follow}(S) \subseteq \mathit{Follow}(T)
\end{gathered}
$$

Step 2: Find First and Follow set relations

From $T \to bT$:

$$b \in First(T)$$
$$Follow(T) \subseteq Follow(T)$$

From $T \to R$:

$$First(R) \subseteq First(T)$$
$$Follow(T) \subseteq Follow(R)$$

Step 2: Find First and Follow set relations

From $R \to cR$:

$$c \in First(R)$$
$$Follow(R) \subseteq Follow(R)$$

From $R \to \lambda$: $Nothing$

From $S' \to S\$$: $\$ \in Follow(S)$

Step 2: Find First and Follow set relations

All the relations (excluding ones that do nothing):

$$
\begin{aligned}
a &\in First(S) \\
b &\in First(T) \\
c &\in First(R) \\
First(T) &\subseteq First(S) \\
First(R) &\subseteq First(T) \\
\$ &\in Follow(S) \\
Follow(S) &\subseteq Follow(T) \\
Follow(T) &\subseteq Follow(R)
\end{aligned}
$$

Step 3: Find non-terminal First and Follow

Init sets with $\in$ relations.

Set	Init
$First(S)$	$\{a\}$
$First(T)$	$\{b\}$
$First(R)$	$\{c\}$
$Follow(S)$	$\{\$\}$
$Follow(T)$	$\{\}$
$Follow(R)$	$\{\}$

Step 3: Find non-terminal First and Follow

Use prior column and to update next column

Set	Init	Iter 1
$First(S)$	$\{a\}$	$\{a, b\}$
$First(T)$	$\{b\}$	$\{b, c\}$
$First(R)$	$\{c\}$	$\{c\}$
$Follow(S)$	$\{\$\}$	$\{\$\}$
$Follow(T)$	$\{\}$	$\{\$\}$
$Follow(R)$	$\{\}$	$\{\}$

Step 3: Find non-terminal First and Follow

Use prior column and to update next column

Set	Init	Iter 1	Iter 2
$First(S)$	$\{a\}$	$\{a, b\}$	$\{a, b, c\}$
$First(T)$	$\{b\}$	$\{b, c\}$	$\{b, c\}$
$First(R)$	$\{c\}$	$\{c\}$	$\{c\}$
$Follow(S)$	$\{\$\}$	$\{\$\}$	$\{\$\}$
$Follow(T)$	$\{\}$	$\{\$\}$	$\{\$\}$
$Follow(R)$	$\{\}$	$\{\}$	$\{\$\}$

Step 3: Find non-terminal First and Follow

Use prior column and to update next column.
Because nothing changes, we're done.

Set	Init	Iter 1	Iter 2	Iter 3
$First(S)$	$\{a\}$	$\{a, b\}$	$\{a, b, c\}$	$\{a, b, c\}$
$First(T)$	$\{b\}$	$\{b, c\}$	$\{b, c\}$	$\{b, c\}$
$First(R)$	$\{c\}$	$\{c\}$	$\{c\}$	$\{c\}$
$Follow(S)$	$\{\$\}$	$\{\$\}$	$\{\$\}$	$\{\$\}$
$Follow(T)$	$\{\}$	$\{\$\}$	$\{\$\}$	$\{\$\}$
$Follow(R)$	$\{\}$	$\{\}$	$\{\$\}$	$\{\$\}$

Step 4: Determine predictors for each non-terminal

Prod	First RHS	RHS Nullable?	If so, Follow LHS	Predictor
$S \to aS$	$\{a\}$	No	—	a
$S \to T$	$\{b, c\}$	Yes	$\{\$\}$	$b, c, \$$
$T \to bT$	$\{b\}$	No	—	b
$T \to R$	$\{c\}$	Yes	$\{\$\}$	$c, \$$
$R \to cR$	$\{c\}$	No	—	c
$R \to \lambda$	$\{\}$	Yes	$\{\$\}$	$\$$

Step 4: Determine predictors for each non-terminal

Predictors for the two S productions do not conflict.

Predictors for the two T productions do not conflict.

Predictors for the two R productions do not conflict.

We have non-conflicting prediction tokens!

Grammar is suitable for LL(1) parsing.

Handling stack top and next token

```
if top in ('a', 'b', 'c'):  # try input/stack match
    toks.match(top)
elif top == 'S' and tok == 'a':
    stack.append('S')
    stack.append('a')
elif top == 'S' and (tok == None or tok in ('b', 'c')):
    stack.append('T')
elif top == 'T' and tok == 'b':
    stack.append('T')
    stack.append('b')
elif top == 'T' and (tok == None or tok == 'c'):
    stack.append('R')
elif top == 'R' and tok == 'c':
    stack.append('R')
    stack.append('c')
elif top == 'R' and tok == None:
    pass    # Push nothing
else:
    raise Exception    # Unrecognized top/tok combination
```

Recursive descent version

Prod	First RHS	RHS Nullable?	If so, Follow LHS	Predictor
$S \to aS$	$\{a\}$	No	—	a
$S \to T$	$\{b, c\}$	Yes	$\{\$\}$	$b, c, \$$

```
def parseS(toks):
    tok = toks.next()
    if tok == 'a':
        toks.match('a')
        parseS(toks)
    elif tok == None or tok in ('b', 'c'):
        parseT(toks)
    else:
        raise Exception
```

Recursive descent version

Prod	First RHS	RHS Nullable?	If so, Follow LHS	Predictor
$T \to bT$	$\{b\}$	No	—	b
$T \to R$	$\{c\}$	Yes	$\{\$\}$	$c, \$$

```
def parseT(toks):
    tok = toks.next()
    if tok == 'b':
        toks.match('b')
        parseT(toks)
    elif tok == None or tok == 'c':
        parseR(toks)
    else:
        raise Exception
```

Recursive descent version

Prod	First RHS	RHS Nullable?	If so, Follow LHS	Predictor
$R \to cR$	$\{c\}$	No	–	c
$R \to \lambda$	$\{\}$	Yes	$\{\$\}$	$\$$

```
def parseR(toks):
    tok = toks.next()
    if tok == 'c':
        toks.match('c')
        parseR(toks)
    elif tok == None:
        pass
    else:
        raise Exception
```

Recursive descent version

Call `parseS` and check that input all consumed

```
def recursive_parse(input):
    toks = scanner(input)
    parseS(toks)
    if toks.next() != None:
        raise Exception
```