Top-down versus bottom-up

Top-down parsers

- Start at the root of derivation tree and fill in
- Picks a production and tries to match the input
- May require backtracking
- Some grammars are backtrack-free(predictive)

Bottom-up parsers

- Start at the leaves and fill in
- Start in a state valid for legal first token
- As input is consumed, change state to encode possibilities (recognize valid prefixes)
- Use a stack to store both state and sentential forms

Top-down parsing

A top-down parser starts with the root of the parse tree. It is labeled with the start symbol or goal symbol of grammar.

To build a parse tree, it repeats the following steps until the fringe of the parse tree matches the input string.

1. At a node labeled A, select a production with A on its lhs and for each symbol on its rhs, construct the appropriate child.
2. When a terminal is added to the fringe that doesn't match the input string, backtrack.
3. Find the next node to be expanded. (Must have a label in NT)

The key is selecting the right production in step 1.

⇨ should be guided by input string

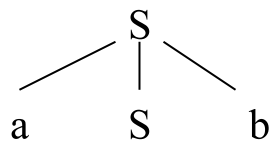
Example

Consider the following grammar

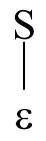
e.g.) S $\rightarrow$ aSb | ε

to parse an input string *aaabbb*

(1)

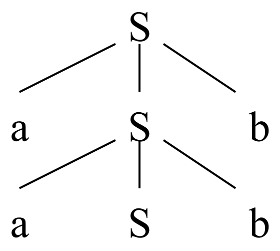


(2)



After reading the first character(*a*) of the input string, select (1).

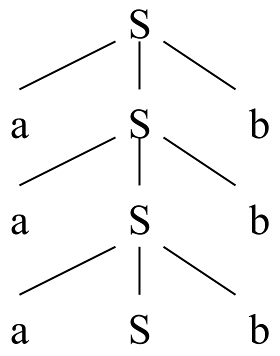
(3)



After reading the second character(*a*) of the input string, select (3).

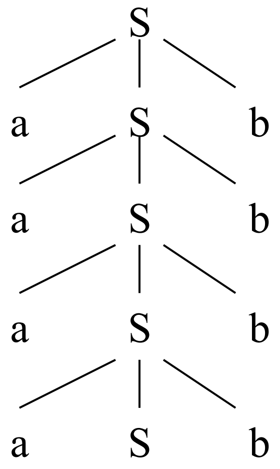
Example

(4)



After reading the third character(*a*) of the input string, select (4).

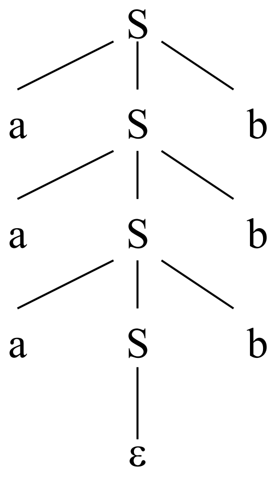
(5)



After reading the forth character(*b*) of the input string, Parser finds the mismatch between the input string and the parsing tree.
Production rule S $\rightarrow$ aSb is not applicable. So backtrack and try another production rule S $\rightarrow$ ε. Then we get (6)

Example

(6)



Lest of characters(*bbb*) are matched and the parsing tree building is over.

Left Recursion

Top-down parsers cannot handle left-recursion in a grammar

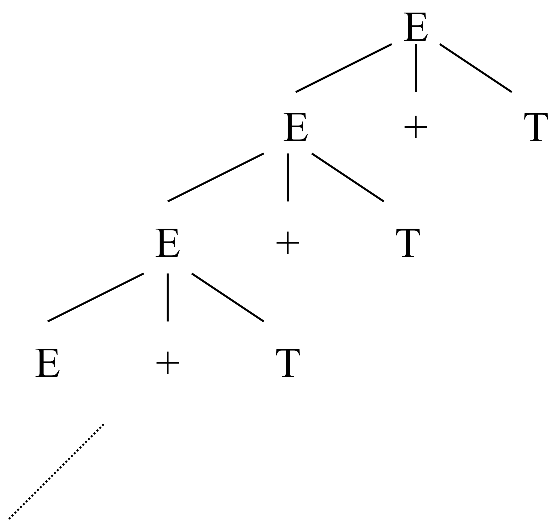
Formally,

A grammar is left recursive if $\exists$ A $\in$ NT such that
$\exists$ a derivation A $=>^+$ A α for some string α.

What happens if a grammar is left recursive

⇨ parsing does not terminate

e.g.) E $\rightarrow$ E + T



Eliminating Left Recursion

To remove left recursion, we can transform the grammar.

Consider the grammar fragment:

```
    <foo> ::= <foo> α
            | β
```

where α and β do not start with <foo>

We can write this as:

```
    <foo> ::= β <bar>
    <bar> ::= α <bar>
            | ε
```

where <bar> is a new non-terminal.

This fragment contains no left recursion.

Example

Our expression grammar contains two cases of left recursion.

```
<expr> ::= <expr> + <term>
           | <expr> - <term>
           | <term>
<term> ::= <term> * <factor>
           | <term> / <factor>
           | <factor>
```

Applying the transformation gives

```
<expr> ::= <term> <expr'>
<expr'> ::= + <term> <expr'>
           | ε
           | – <term> <expr'>
<term> ::= <factor> <term'>
<term'> ::= * <factor> < term'>
           | ε
           | / <factor> <term'>
```

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

A temptation is to clean up the grammar like this instead:

```
<goal> ::= <expr>
<expr> ::= <term> + <expr>
           | <term> – <expr>
           | <term>
<term> ::= <factor> * <term>
           | <factor> / <term>
           | <factor>
<factor> ::= number
           | id
```

This grammar

- accepts the same language
- uses right recursion
- has no ε productions

Unfortunately, it generates different associativity
Same syntax, different meaning

Eliminating left recursion

A general technique for removing left recursion

```
arrange the non-terminals in some order
    $A_1, A_2, \ldots, A_n$
for i ← 1 to n
    for j ← 1 to i – 1
        replace each production of the form
          $A_i ::= A_j\gamma$ with the productions
          $A_i ::= \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
          where $A_j ::= \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
          are all the current $A_j$ productions.
    eliminate any immediate left recursion on $A_i$
      using the direct transformation
```

This assumes that the grammar has no cycles
($A \Rightarrow^+ A$) or ε productions ($A ::= \varepsilon$).

Example) If we apply above algorithm to the below grammar:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

is converted into

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

in the second iteration(for i = 2), then we get

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid eA'$

$A' \rightarrow c' \mid ad'A' \mid \varepsilon$

by eliminating any immediate left recursion on A

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production

Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms

Fortunately

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

Among the interesting subclasses are LL(1) and LR(1)

Predictive Parsing

Basic idea:

For any two productions $A ::= \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.

For some rhs $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string derived from α.
That is, $x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x\gamma$ for some γ.

Key Property:
Whenever two productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

This would allow the parser to make a correct choice with a lookahead of only one symbol!

The example grammar has this property!

Left Factoring

What if a grammar does not have this property?

Sometimes, we can transform a grammar to have this property.

For each non-terminal A find the longest prefix α common to two or more of its alternatives.

If $\alpha \neq \varepsilon$, then replace all of the A productions
$A ::= \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$
with

$A ::= \alpha L \mid \gamma$
$L ::= \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

where L is a new non-terminal

Repeat until no two alternatives for a single non-terminal have a common prefix.

Example

Consider a right-recursive version of the expression grammar:

1. <goal> ::= <expr>
2. <expr> ::= <term> + <expr>
3. | <term> – <expr>
4. | <term>
5. <term> ::= <factor> * <term>
6. | <factor> / <term>
7. | <factor>
8. <factor> ::= number
9. | id

To choose between productions 2, 3, & 4, the parser must see past the number or id and look at the +, –, *, or /.

$$\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \varnothing$$

This grammar fails the test.

Note: This grammar is right-associative

Example

There are two non-terminals that must be left factored:

```
<expr> ::= <term> + <expr>
         | <term> – <expr>
         | <term>
<term> ::= <factor> * <term>
         | <factor> / <term>
         | <factor>
```

Applying the transformation gives us:

```
<expr> ::= <term> <expr’>
<expr’> ::= + <expr>
          | – <expr>
          | ε
<term> ::= <factor> <term’>
<term’> ::= * <term>
          | / <term>
          | ε
```

Example

Substituting back into the grammar yields

```
<goal> ::= <expr>
<expr> ::= <term> <expr'>
<expr'> ::= + <term> <expr'>
          | – <term> <expr'>
          | ε
<term> ::= <factor> <term'>
<term'> ::= * <term>
          | / <term>
          | ε
<factor> ::= number
          | id
```

Now, selection requires only a single token lookahead.

Note: this grammar is still right-associative.

Example

	Sentential form	Input
	<goal>	↑x – 2 * y
1	<expr>	↑x – 2 * y
2	<term> <expr’>	↑x – 2 * y
6	<factor> <term’> <expr’>	↑x – 2 * y
11	<id> <term’> <expr’>	↑x – 2 * y
-	<id> <term’> <expr’>	x ↑– 2 * y
9	<id> ε <expr’>	x ↑– 2 * y
4	<id> – <expr>	x ↑– 2 * y
-	<id> – <expr>	x – ↑2 * y
2	<id> – <term> <expr’>	x – ↑2 * y
6	<id> – <factor> <term’> <expr’>	x – ↑2 * y
10	<id> – <num> <term’> <expr’>	x – ↑2 * y
-	<id> – <num> <term’> <expr’>	x – 2 ↑* y
7	<id> – < num > * <term> <expr’>	x – 2 ↑* y
-	<id> – < num > * <term> <expr’>	x – 2 * ↑y
6	<id> – < num > * <factor> <term’> <expr’>	x – 2 * ↑y
11	<id> – < num > * <id> <term’> <expr’>	x – 2 * ↑y
-	<id> – < num > * <id> <term’> <expr’>	x – 2 * y ↑
9	<id> – < num > * <id> <expr’>	x – 2 * y ↑
5	<id> – < num > * <id>	x – 2 * y ↑

The next symbol determined each choice correctly.

Generality

Question:

By eliminating left recursion and left factoring, can we transform an arbitrary context free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer:

Given a context free grammar that doesn't meet our conditions, it is undecidable whether an equivalent grammar exists that does meet our conditions.

Many contex free languages do not have such a grammar.

$\{a^n 0 b^n \mid n \geq 1\} \cup \{ a^n 0 b^{2n} \mid n \geq 1\}$

Designing a Predictive Parser

A predictive parser is a program consists of a procedure for every nonterminal. Each procedure does two things.

1. It decides which production to use by looking at the lookahead symbol
2. The procedure uses a production by mimicking the right side. A nonterminal results in a call to the procedure for the nonterminal, and a token matching results in the next input token being read.

Recursive Descent Parser

Now we can produce a simple recursive descent parser from this grammar.

```
goal:
        token ← next_token();
        if (expr() = ERROR | token ≠ EOF) then
                return ERROR;
expr:
        if (term() = ERROR) then
                return ERROR;
        else return expr_prime();
expr_prime:
        if (token = PLUS) then
                token ← next_token();
                return expr();
        else if (token = MINUS) then
                token ← next_token();
                return expr();
        else return OK;
```

Recursive Descent Parser

```
term:
    if (factor() = ERROR) then
        return ERROR;
    else return term_prime();
term_prime:
    if (token = MULT) then
        token ← next_token();
        return term();
    else if (token = DIV) then
        token ← next_token();
        return term();
    else return OK;
factor:
    if (token = NUM) then
        token ← next_token();
        return OK;
    else if (token = ID) then
        token ← next_token();
        return OK;
    else return ERROR;
```

Building the Tree

One of the key jobs of the parser is to build an intermediate representation of the source code.

To build an abstract syntax tree, we can simply insert code at the appropriate points:

- factor() can stack nodes id, num
- term_prime() can stack nodes *, /
- term() can pop 3, build and push subtree
- expr_prime() can stack nodes +, –
- expr() can pop 3, build and push subtree
- goal() can pop and return tree

Homework #1
Implement a predictive parser for the example expression grammar.
You need to construct an abstract syntax tree for input expressions.
Emit the abstract syntax tree for an input expression in preorder.

Due: Sep. 27, 2018