Top-down Parser

Hwansoo Han

Parsing Techniques

- ***Top-down parsers***
  - *LL = **L**eft-to-right input scan, **L**eftmost derivation*
  - Start at the root of the parse tree and grow toward leaves
  - Pick a production & try to match the input
  - Bad "pick" $\Rightarrow$ may need to backtrack
  - Some grammars are backtrack-free *(predictive parsing)*

- ***Bottom-up parsers***
  - *LR = **L**eft-to-right input scan, **R**ightmost derivation*
  - Start at the leaves and grow toward root
  - As input is consumed, encode possibilities in an internal state
  - Start in a state valid for legal first tokens
  - Bottom-up parsers handle a large class of grammars

Top-down Parser

- **Problems in Top-down parser**
  - Backtrack $\Rightarrow$ predictive parser
  - Left-recursion $\Rightarrow$ may result in infinite loop

- **Predictive parser**
  - LL(1) property
  - Left factoring transforms some non-LL(1) to LL(1)

Remember the expression grammar?

- **Version with precedence derived last lecture**

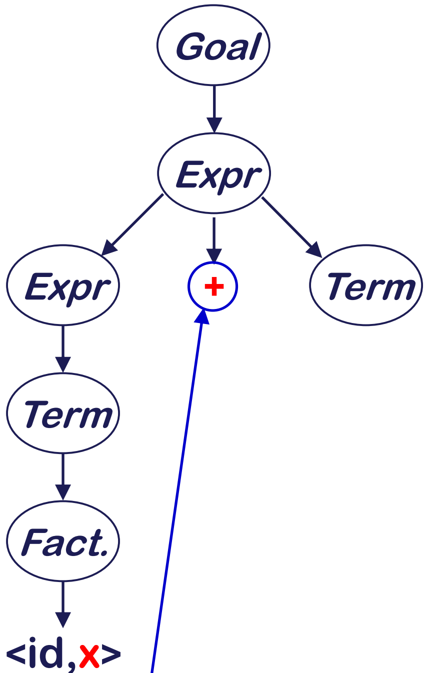
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	<u>number</u>
9		|	<u>id</u>

And the input <u>x</u> - <u>2</u> * <u>y</u>

Top-down Parser - backtrack (1)

- **Let's try x – 2 * y :** *Leftmost derivation, choose productions in an order that exposes problems*

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr + Term*	↑x - 2 * y
4	*Term + Term*	↑x - 2 * y
7	*Factor + Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
—	<id,x> + *Term*	x ↑- 2 * y

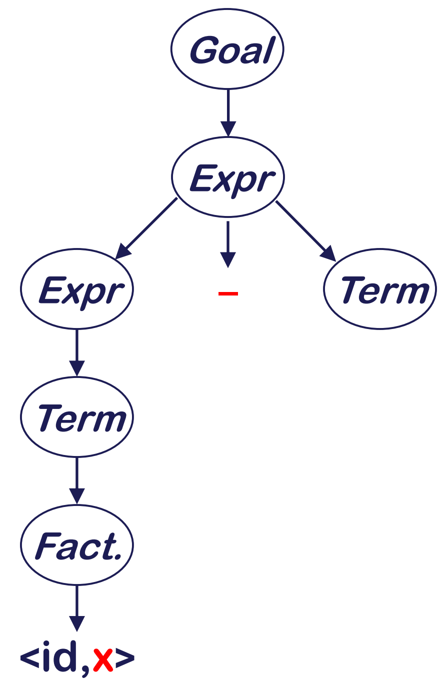


- This worked well, except that "–" doesn't match "+"
- The parser must backtrack to here

Top-down Parser – backtrack (2)

- **Continuing with x – 2 * y :**

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr – Term*	↑x – 2 * y
4	*Term – Term*	↑x – 2 * y
7	*Factor – Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y



This time, "–" and "–" matched

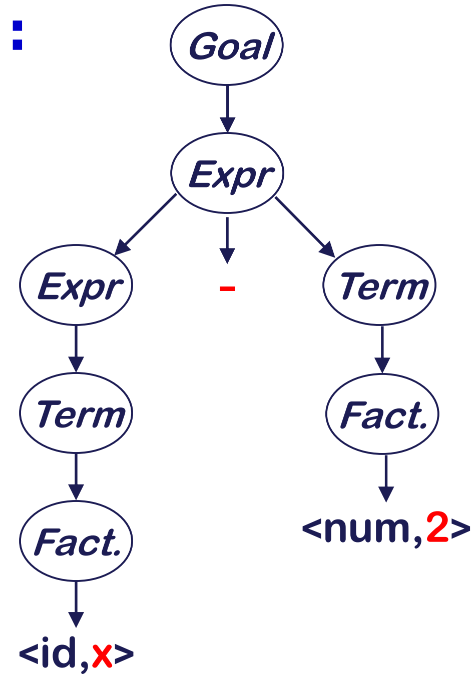
We can advance past "–" to look at "2"

⇒ **Now, we need to expand *Term* - the last *NT* on the fringe**

Top-down Parser – backtrack (3)

- **Trying to match the "2" in x – 2 * y :**

Rule	Sentential Form	Input
—	<id,x> – *Term*	x – ↑2 * y
7	<id,x> – *Factor*	x – ↑2 * y
9	<id,x> – <num,2>	x – ↑2 * y
—	<id,x> – <num,2>	x – 2 ↑* y



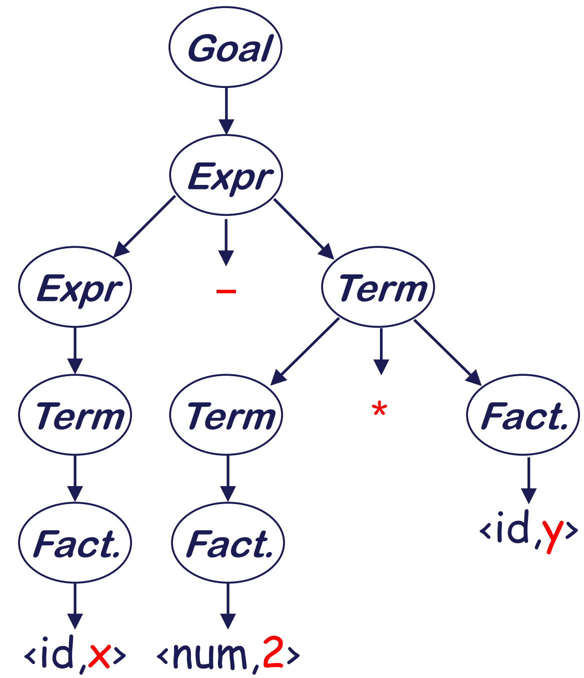
- **Where are we?**
  - "2" matches "2"
  - We have more input, but no *NT*s left to expand
  - The expansion terminated too soon

⇒ Need to backtrack

Top-down Parser - backtrack (4)

❖ **Trying again with "2" in x – 2 * y :**

Rule	Sentential Form	Input
—	<id,x> – *Term*	x - ↑2 * y
5	<id,x> – *Term* * *Factor*	x - ↑2 * y
7	<id,x> – *Factor* * *Factor*	x - ↑2 * y
8	<id,x> – <num,2> * *Factor*	x - ↑2 * y
—	<id,x> – <num,2> * *Factor*	x - 2 ↑* y
—	<id,x> – <num,2> * *Factor*	x - 2 * ↑y
9	<id,x> – <num,2> * <id,y>	x - 2 * ↑y
—	<id,x> – <num,2> * <id,y>	x - 2 * y↑



- This time, we matched & consumed all the input

⇒ Success!

Left Recursion

- ***Top-down parsers cannot handle left-recursive grammars***
  - Formally,

    A grammar is *left recursive* if $\exists\ A \in NT$ such that

    $\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

- **Our expression grammar is left recursive**
  - This can lead to non-termination in a top-down parser
  - We would like to convert the left recursion to right recursion

Eliminating Left Recursion

- **Remove left recursion**
  - Original grammar

    $F \rightarrow F\ \alpha$

    $\quad |\ \ \beta$

    where neither α nor β starts with F

  - Rewrite the above as

    $F \rightarrow \beta\ P$

    $P \rightarrow \alpha\ P$

    $\quad\quad |\ \ \varepsilon$

    where P is a new non-terminal

  - *Accepts the same language, but uses only right recursion*

Eliminating Left Recursion

- **The expression grammar contains two left recursions**

Expr	$\rightarrow$	*Expr* + *Term*	*Term*	$\rightarrow$	*Term* * *Factor*
	|	*Expr* - *Term*		|	*Term* / *Factor*
	|	*Term*		|	*Factor*

- **Applying the transformation yields**

Expr	$\rightarrow$	*Term* *Expr′*	*Term*	$\rightarrow$	*Factor* *Term′*
Expr′	$\rightarrow$	+ *Term* *Expr′*	*Term′*	$\rightarrow$	* *Factor* *Term′*
	|	- *Term* *Expr′*		|	/ *Factor* *Term′*
	|	ε		|	ε

  - These fragments use only right recursion
  - They retain the original left associativity (evaluate left to right)

Predictive Parsing

- **Basic idea**

  Given $A \rightarrow \alpha \mid \beta$*, the parser should be able to choose between* α & β

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

- **The LL(1) Property** (first version)
  - If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

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

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

Predictive Parsing - LL(1)

- **What about ε-productions?**

  $\Rightarrow$ They complicate the definition of LL(1)

  - If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(α), too
  - Define FIRST$^+$(α) for $A \rightarrow \alpha$ as

    FIRST(α) $\cup$ FOLLOW(A), if $\varepsilon \in$ FIRST(α)

    FIRST(α), otherwise
  - Then, a grammar is *LL(1)* iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

    FIRST$^+$(α) $\cap$ FIRST$^+$(β) = $\varnothing$

FOLLOW(α) is the set of all tokens in the grammar that can legally appear immediately after an α

FIRST$^+$(α) is meaningful, iff α is RHS of the rule $A \rightarrow \alpha$

The FIRST Set

- **Definition**
  - $\underline{x} \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ
- **Building FIRST(X)**
  - If X is a terminal (token), FIRST(X) = {X}
  - If $X \to \varepsilon$, then $\varepsilon \in$ FIRST(X)
  - <u>Iterate until</u> no more terminals or ε can be added to any FIRST(X)

    if $X \to y_1 y_2 \ldots y_k$ then

    $a \in$ FIRST(X) if $a \in$ FIRST(y_i) and $\varepsilon \in$ FIRST(y_h) for all $1 \le h < i$

    $\varepsilon \in$ FIRST(X) if $\varepsilon \in$ FIRST(y_i) for all $1 \le i \le k$

    <u>End iterate</u>
- **Note**
  - If $\varepsilon \notin$ FIRST(y_1), then FIRST(y_i) is irrelevant, for $i > 1$

The FOLLOW Set

- **Definition**
  - FOLLOW(A) is the set of terminals that can appear immediately to the right of A in some sentential form
- **Building FOLLOW(X) for all non-terminal X**
  - EOF $\in$ FOLLOW(S)
  - Iterate until no more terminals can be added to any FOLLOW(X)

    If $A \rightarrow \alpha B$, then put FOLLOW(A) in FOLLOW(B)

    If $A \rightarrow \alpha B \beta$, then put {FIRST($\beta$) - ε} in FOLLOW(B)

    If $A \rightarrow \alpha B \beta$ and $\varepsilon \in$ FIRST(β), then put FOLLOW(A) in FOLLOW(B)

    End iterate
- **Note**
  - FOLLOW is for non-terminals, no FOLLOW for terminals
  - No ε in FOLLOW(X) for any non-terminal X

Left Factoring

- **What if my grammar does not have the LL(1) property?**
  - $\Rightarrow$ Sometimes, we can transform the grammar

$\forall A \in NT$,
find the longest prefix α that occurs in two or more right-hand sides of A

if $\alpha \neq \varepsilon$ then replace all of the A productions,

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma\ ,$$

with

$$A \rightarrow \alpha\, Z \mid \gamma$$
$$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where Z is a new element of NT

Repeat until no common prefixes remain

$$\begin{aligned} A &\rightarrow \alpha\beta_1 \\ &\mid \alpha\beta_2 \\ &\mid \alpha\beta_3 \end{aligned}$$

$$\Downarrow$$

$$\begin{aligned} A &\rightarrow \alpha\, Z \\ Z &\rightarrow \beta_1 \\ &\mid \beta_2 \\ &\mid \beta_3 \end{aligned}$$

Left Factoring (An example)

- **Consider the following expression grammar**

Factor	$\rightarrow$	Identifier
	|	Identifier [*ExprList*]
	|	Identifier (*ExprList*)

FIRST$^+$(rhs_1) = { Identifier }
FIRST$^+$(rhs_2) = { Identifier }
FIRST$^+$(rhs_3) = { Identifier }

- **After left factoring, it becomes**

Factor	$\rightarrow$	**Identifier *Arguments***
Arguments	$\rightarrow$	**[*ExprList*]**
	|	**(*ExprList*)**
	|	ε

FIRST$^+$(rhs_1) = { [}
FIRST$^+$(rhs_2) = { (}
FIRST$^+$(rhs_3) = { ε } U FOLLOW(*Factor*)

$\Rightarrow$ It has the *LL(1)* property

- **This form has the same syntax, with the *LL(1)* property**

Left Recursion & Left Factoring *(Generality)*

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a CFG that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Example **that has no LL(1) grammar**

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

Language that Cannot Be LL(1)

Example

$\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

$$
\begin{aligned}
G &\rightarrow \underline{a}A\underline{b} \\
&\quad \mid \underline{a}B\underline{bb} \\
A &\rightarrow \underline{a}A\underline{b} \\
&\quad \mid \underline{0} \\
B &\rightarrow \underline{a}B\underline{bb} \\
&\quad \mid \underline{1}
\end{aligned}
$$

Problem: need an unbounded number of <u>a</u> characters before you can determine whether you are in the A group or the B group.

Automate Predictive Parsing

- **Given a grammar that has the *LL(1)* property**
  - Can write a simple routine to recognize each *lhs*
  - Code is both simple & fast
- **Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with**
  - $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \varnothing$ AND
  - $\text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$ AND
  - $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \varnothing$

```
/* find an A */
if (current_token ∈ FIRST+(β1))
    find a β1 and return true
else if (current_token ∈ FIRST+ (β2))
    find a β2 and return true
else if (current_token ∈ FIRST+ (β3))
    find a β3 and return true
else
    report an error and return false
```

Grammars with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the *LL(1)* property are called *predictive parsers*.

One kind of predictive parser is the *recursive descent* parser.

Predictive Parsing Example

- **Expression grammar, after transformation**

1	*Goal*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Term Expr′*
3	*Expr′*	$\rightarrow$	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	$\rightarrow$	*Factor Term′*
7	*Term′*	$\rightarrow$	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	$\rightarrow$	<u>id</u>
11		|	<u>number</u>

This produces a parser with six <u>*mutually recursive*</u> routines:

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *Factor*

Each recognizes one *NT* or *T*

The term <u>*descent*</u> refers to the direction in which the parse tree is built.

Recursive Descent Parser

A couple of routines from the expression parser

global variable

```
Goal( )
    token ← next_token( );
    if (Expr( ) = true & token = EOF)
        then next compilation step;
        else
            report syntax error;
            return false;

Expr( )
   if (Term( ) = false)
     then return false;
     else return Eprime( );
```

looking for EOF, found other token

```
Factor( )
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
      token ← next_token( );
      return true;
  else
     report syntax error;
     return false;
```

EPrime, *Term*, & *TPrime* follow the same basic lines

looking for Number or Identifier, found other token instead

Recursive Descent Parser (cont'd)

```
EPrime( )
  // Expr' → + Term Expr'
  // Expr' → - Term Expr'
  if (token = + or token = -) then
     token ← next_token( );
     if (Term()) then
         return EPrime();
     else return false; // Fail
  // Expr' → ε
  else if (token = EOF) then
      return true;
  else return false; // Fail
```

Prepare next token when current token is consumed

No next token is needed, when no input token is consumed, but ε

Expr' → α	**First$^+$**
+ Term Expr'	{ + }
- Term Expr'	{ - }
ε	{ε, EOF }

- ***Term*** **&** ***TPrime*** **follow the same basic lines**

Parse Tree - Recursive Descent Parser

- **To build a parse tree:**
  - Augment parsing routines to build nodes
  - Pass nodes between routines using a stack
  - Node for each symbol on *rhs*
  - Action is to pop *rhs* nodes, make them children of *lhs* node, and push this subtree
- **To build an abstract syntax tree**
  - Build fewer nodes
  - Put them together in a different order

Expr → Term Expr′

```
Expr( )
    result ← true;
    if (Term( ) = false) then
        return false;
    else if (EPrime( ) = false) then
        result ← false;
    else // successfully parsed!
        build an Expr node
        pop EPrime node
        pop Term node
        make EPrime & Term
              children of Expr
        push Expr node
    return result;
```

Success ⇒ build a piece of the parse tree

This is a preview of Chapter 4

Building Table-driven Parser

- **Strategy**
  - Encode knowledge in a table
  - Use a standard “skeleton” parser to interpret the table
- **Example**
  - The non-terminal *Factor* has two expansions

    Identifier or Number
  - Need a row for every *NT* & a column for every *T*
  - Table might look like:

Terminal Symbols

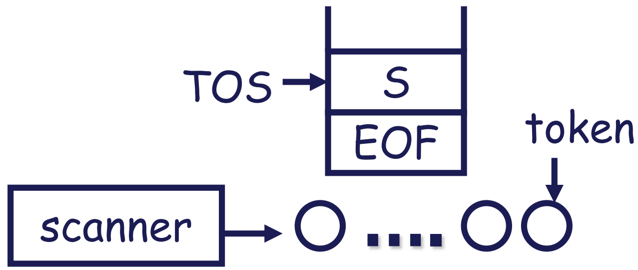
Non-terminal Symbols

	+	-	*	/	Id.	Num.	EOF
Factor	—	—	—	—	10	11	—

Error on ' + '

Reduce by rule 10 on ' x '

LL(1) *Skeleton Parser*



token ← next_token()
push EOF onto Stack
push the start symbol, *S*, onto Stack
TOS ← top of Stack

loop forever
    if TOS = EOF and token = EOF then
        break & report success   exit on success
    else if TOS is a terminal then
        if TOS matches token then
            pop Stack   // recognized TOS
            token ← next_token()
        else report error looking for TOS
    else   // TOS is a non-terminal
        if TABLE[TOS,token] is $A \rightarrow B_1B_2 \ldots B_k$ then
            pop Stack   // get rid of *A*
            push $B_k, B_{k-1}, \ldots, B_1$ // in that order
        else report error expanding TOS
    TOS ← top of Stack

LL(1) Skeleton Parser Example

			F	id				+		F	num					
		T	T’	T’	T’	ε		T	T	T’	T’	T’	ε			
G	E	E’	E’	E’	E’	E’	E’	E’	E’	E’	E’	E’	E’	E’	ε	
EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF	EOF
x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2	x+2
^	^	^	^	^	^	^	^	^	^	^	^	^	^	^	^	^

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	<u>id</u>
11		|	<u>number</u>

Building LL(1) table

- **Building the complete table for LL(1)**
  - Need a row for every *NT* & a column for every *T*
  - Need an algorithm to build the table

- **Filling in TABLE[X,y], X $\in$ *NT*, y $\in$ *T* $\cup$ {*EOF*}**
  1. entry is the rule $X \rightarrow \beta$, if y $\in$ $\text{FIRST}^+(\beta)$
  2. entry is error, otherwise

  - If any entry is defined multiple times, G is not *LL(1)*

Summary

- **Top-down parser**
  - Use leftmost derivation
  - Bad pick of rewrite rule results in *backtrack*
- **Left recursion removal**
  - Avoid non-terminating top-down parser
- **Predictive parsing**
  - LL(1) property ensures only one production rule is chosen by looking ahead one terminal symbol.
- **Left factoring**
  - Transform some non-LL(1) to LL(1)
- **Automatic top-down parser generation**
  - Recursive decent parser
  - Building LL(1) table: $f(X,y) \rightarrow P$ (where $X \in NT$, $y \in T$)