Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), operator precedence)*

- 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 Parsing

A top-down parser starts with the root of the parse tree

The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until the leaves 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 symbol is added to the fringe and it doesn't match the fringe, backtrack*
3. *Find the next node to be expanded* *(label ∈ NT)*

The key is picking the right production in step 1

> *That choice should be guided by the input string*

Remember the expression grammar?

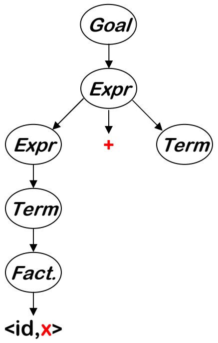
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
9		|	(*Expr*)

And the input x – 2 * y

Example

Let's try x – 2 * y :

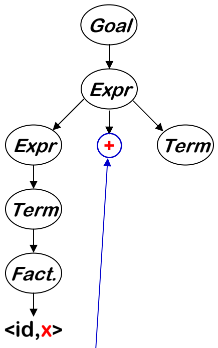
Rule	*Sentential Form*	*Input*
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr* + *Term*	↑x – 2 * y
4	*Term* + *Term*	↑x – 2 * y
7	*Factor* + *Term*	↑x – 2 * y
9	<id,x> + *Term*	↑x – 2 * y
9	<id,x> + *Term*	x ↑– 2 * y



Example

Let's try x – 2 * y :

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
9	<id,x> + *Term*	x ↑– 2 * y



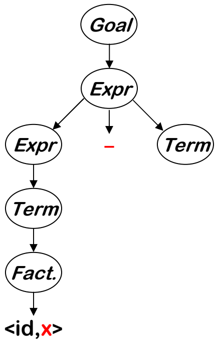
This worked well, except that "–" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x – 2 * y :

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
~~2~~ 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

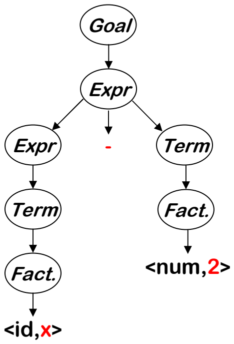
We can advance past "–" to look at "2"

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

Example

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?

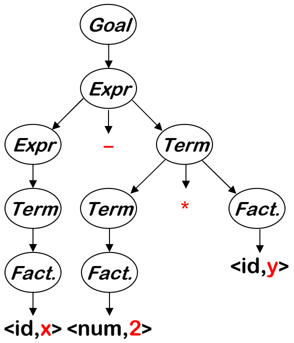
- **"2" matches "2"**
- **We have more input, but no *NT*s left to expand**
- **The expansion terminated too soon**

⇒ **Need to backtrack**

Example

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!**

Another possible parse

Other choices for expansion are possible

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr* + *Term*	↑x – 2 * y
2	*Expr* + *Term* + *Term*	↑x – 2 * y
2	*Expr* + *Term* + *Term* + *Term*	↑x – 2 * y
2	*Expr* + *Term* + *Term* + …+ *Term*	↑x – 2 * y

consuming no input !

This doesn't terminate *(obviously)*

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is *left recursive* if ∃ $A \in N$ such that
∃ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (N \cup T)^{+}$

Our expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

Fee $\rightarrow$ *Fee* α
    | β

where neither α nor β start with *Fee*

Note that*: Fee* $\Rightarrow \beta\ \alpha^*$

We can rewrite this to generate β first, as

Fee $\rightarrow$ β *Fie*
Fie $\rightarrow$ α *Fie*
    | ε

where *Fie* is a new non-terminal

This accepts the same language, but uses only right recursion

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
	|	*Expr* – *Term*		|	*Term* / Factor
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	→	*Term Expr′*	*Term*	→	*Factor Term′*
Expr′	|	*+ Term Expr′*	*Term′*	|	* *Factor Term′*
	|	*– Term Expr′*		|	*/ Factor Term′*
	|	ε		|	ε

These fragments use only right recursion

They retains the original left associativity

Eliminating Left Recursion

Substituting back into the grammar yields

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*	→	number
11		|	id
12		|	(*Expr*)

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.

Eliminating Left Recursion

The transformation eliminates immediate left recursion

What about more general, indirect left recursion ?

The general algorithm:

```
arrange the NTs into some order $A_1, A_2, \ldots, A_n$
for $i \leftarrow 1$ to $n$
    for $s \leftarrow 1$ to $i - 1$
        replace each production $A_i \rightarrow A_s\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
            where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current productions for $A_s$
        eliminate any immediate left recursion on $A_i$
            using the direct transformation
```

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$), and no epsilon productions

Eliminating Left Recursion

How does this algorithm work?

1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through NT in order
3. Inner loop ensures that a production expanding A_i has no non-terminal A_s in its *rhs,* for $s < i$
4. Last step in outer loop converts any direct recursion on A_i to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion

At the start of the i^{th} outer loop iteration

For all $k < i$, no production that expands A_k contains a non-terminal A_s in its rhs, for $s < k$

Example

- Order of symbols: *G, E, T*

1. $A_i = G$	*2. $A_i = E$*	*3. $A_i = T, A_s = E$*	*4. $A_i = T$*
$G \rightarrow E$	$G \rightarrow E$	$G \rightarrow E$	$G \rightarrow E$
$E \rightarrow E + T$	$E \rightarrow T E'$	$E \rightarrow T E'$	$E \rightarrow T E'$
$E \rightarrow T$	$E' \rightarrow + T E'$	$E' \rightarrow + T E'$	$E' \rightarrow + T E'$
$T \rightarrow E \sim T$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
$T \rightarrow$ id	$T \rightarrow E \sim T$	$T \rightarrow T E' \sim T$	$T \rightarrow$ id T'
	$T \rightarrow$ id	$T \rightarrow$ id	$T' \rightarrow E \sim T T'$
			$T' \rightarrow \varepsilon$

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - Context-free grammars
  - Ambiguity
- Top-down parsers
  - Algorithm & its problem with left recursion
  - Left-recursion removal
- Predictive top-down parsing – When can we make the right decision without backtracking?
  - The LL(1) condition
  - Simple recursive descent parsers

Picking the "Right" Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount
- E.g., the Cocke-Younger-Kasami algorithm or Earley's algorithm
  - $O(n^3)$ on size of input.

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

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

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, $x \in$ FIRST(α) *iff* $\alpha \Rightarrow^* x\,\gamma$, for some γ

When is First(α) useful?

- When there is no choice in what production to choose.

The LL(1) Property

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 !

(*Pursuing this idea leads to LL(1) parser generators...)*

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 pairwise emptiness, i.e., $i \neq j$

$$\text{FIRST}(\beta_i) \cap \text{FIRST}(\beta_j) = \varnothing$$

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

Of course, there is more detail to "find a β_i"

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.

LL(1) process

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* – *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id
9		|	(*Expr*)

Given X → α, compute FIRST(α)

NT	FIRST(α)
G	E T F num id (
E	E T F num id (
T	T F num id (
F	Num id (

	+	-	(	)	*	/	Id	num
G			G→E				G→E	G→E
E			E→E+T E→E-T E→T				E→E+T E→E-T E→T	E→E+T E→E-T E→T
T			T→T*F T→T/F T→F				T→T*F T→T/F T→F	T→T*F T→T/F T→F
F			F→(E)				F→ id	F→num

- **We know since grammar is left recursive that it can't be LL(1).**
- **Multiple entries in table prove it – no explicit rule to use.**

But recall grammar after left recursion eliminated

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*	→	number
11		|	id
12		|	(*Expr*)

Given X → α, compute FIRST(α)

NT	FIRST(α)	FOLLOW(ε)
G	E T F num id (	
E	T F num id (	
E'	+ -	⊥ (end string))
T	F num id (	
T'	* /	⊥) + -
F	Num id (	

What do we do with ε rules?

FIRST(ε):

if X → ε then FIRST(ε) = FIRST(FOLLOW(X))

But recall grammar after left recursion eliminated

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*	→	number
11		|	id
12		|	(*Expr*)

NT	First(α)	FOLLOW(ε)
G	E T F num id (	
E	T F num id (	
E'	+ -	⊥)
T	F num id (	
T'	* /	⊥) + -
F	Num id (	

	+	-	(	)	*	/	Id	num	⊥
G			G→E				G→E	G→E	
E			E→TE'				E→TE'	E→TE'	
E'	E'→+TE'	E'→-TE'		E'→ ε					E'→ ε
T			T→FT'				T→FT'	T→FT'	
T'	T'→ ε	T'→ ε		T'→ ε	T'→*FT'	T'→/FT'			T'→ ε
F			F→(E)				F→ id	F→num	

Recursive Descent Parsing

Recall the expression grammar, after transformation

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*	→	number
11		|	id
12		|	(*Expr*)

This produces a parser with six *mutually recursive* routines:

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

Each recognizes one *NT*

The term *descent* refers to the direction in which the parse tree is traversed (or built).

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
    token ← next_token( );
    if (Expr( ) = true)
        then next compilation step;
    else
        return false;

Expr( )
    result ← true;
    if (Term( ) = false)
        then result ← false;
        else if (EPrime( ) = false)
            then result ← true; // term found
    return result;
```

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

EPrime, Term, & TPrime follow along the same basic lines (Figure 3.4, EAC)

Recursive Descent Parsing

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

This is a preview of Chapter 4

Recursive Descent in Object-Oriented Languages

- Shortcomings of Recursive Descent
  - Procedural
  - Parse tree construction is a side activity
- Solution
  - Associate a class with each non-terminal symbol
    - → Allocated object contains pointer to the parse tree

```
abstract class NonTerminal {

        protected Scanner s;
        protected TreeNode tree;

        public NonTerminal(Scanner scnr) { s = scnr; tree = null; }

        public abstract boolean isPresent();

        public TreeNode abSynTree() { return tree; }
}
```

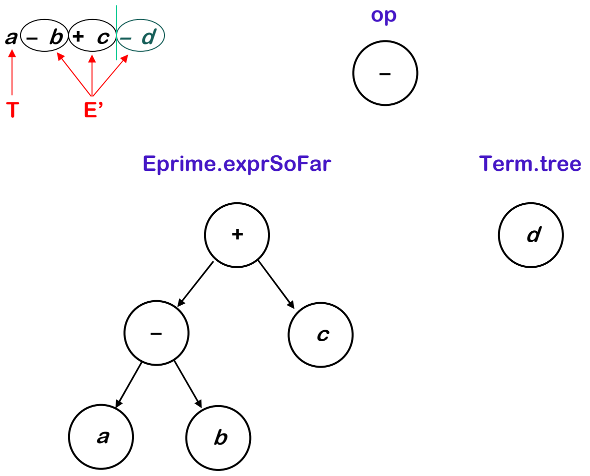
Implementation of Expr

```
class Expr extends NonTerminal {

        public Expr(Scanner scnr) {super(scnr);}

        public boolean isPresent() { // construct AST too

                Term operand1 = new Term(s);
                if (!operand1.isPresent()) return false;
                tree = operand1.abSynTree();

                EPrime operand2 = new EPrime(s, tree);
                if (operand2.isPresent())
                        tree = operand2.absSynTree();

                // here tree is either the tree for the Term
                //              or the tree for Term followed by EPrime
                return true;
        }
}
```

Implementation of EPrime

```
class EPrime extends NonTerminal {

        protected TreeNode exprSofar;

        public EPrime(Scanner scnr, TreeNode p)
                { super(scnr); exprSofar = p; }

        public boolean isPresent() { // construct AST too
                TokenType op = s.nextToken();
                if (op == PLUS | op == MINUS) {
                        s.advance();
                        Term operand2 = new Term(s);
                        if (!operand2.isPresent()) throw new SyntaxError(s);

                        tree = new TreeNode(op, exprSofar, operand2.absSynTree());

                        Eprime operand3 = new Eprime(s, tree);
                        if (operand3.isPresent()) tree = operand3.absSynTree();
                        return true;
                }
                else return false;
        }
}
```

Tree Building in EPrime



Implementation of Factor

```
class Factor extends NonTerminal {

        public Factor(Scanner scnr) {super(scnr);}

        public boolean isPresent() { // with semantic processing
                TokenType op = s.nextToken();
                if (op == IDENTIFIER | op == NUMBER) {
                        tree = new TreeNode(op, s.tokenValue());
                        s.advance();
                        return true;
                }
                else if (op == LPAREN) {
                        s.advance();
                        Expr operand = new Expr(s);
                        if (!operand.isPresent()) throw new SyntaxError(s);
                        if (s.nextToken() != RPAREN) throw new SyntaxError(s);
                        s.advance();
                        tree = operand.absSynTree();
                        return true;
                }
                else return false;
        }
}
```

Left Factoring

What if my grammar does not have the LL(1) property?

⇒ **Sometimes, we can transform the grammar**

The Algorithm

```
∀ $A \in NT$,
  find the longest prefix $\alpha$ that occurs in two
      or more right-hand sides of A
  if $\alpha \neq \varepsilon$ then replace all of the A productions,
      $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$ ,
  with
      $A \rightarrow \alpha Z \mid \gamma$
      $Z \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
  where Z is a new element of NT

Repeat until no common prefixes remain
```

Left Factoring *(An example)*

Consider the following fragment of the expression grammar

Factor → Identifier
| Identifier [*ExprList*]
| Identifier (*ExprList*)

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { Identifier }
FIRST(rhs_3) = { Identifier }

After left factoring, it becomes

Factor → Identifier *Arguments*
Arguments → [*ExprList*]
| (*ExprList*)
| ε

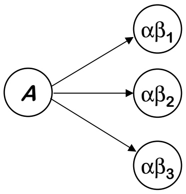
FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = FOLLOW(*Factor*)
⇒ It has the *LL(1)* property

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

Left Factoring

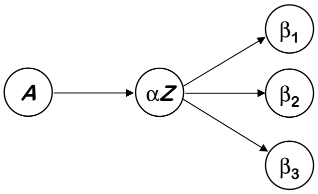
A graphical explanation for the same idea

$A \rightarrow \alpha\beta_1$
$\mid \alpha\beta_2$
$\mid \alpha\beta 3$



becomes …

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



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

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

---

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

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

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