Types of parsing

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 tokens
- 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 the grammar.

To build a parse, 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.

$\Rightarrow$ should be guided by input string

Example grammar

This is a grammar for simple expressions:

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`

Consider parsing the input string `x - 2 * y`

Backtracking parse example

One possible parse for `x - 2 * y`

Prod'n	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> - <term>	↑x - 2 * y
–	<id> - <term>	x ↑- 2 * y
–	<id> - <term>	x - ↑2 * y
7	<id> - <factor>	x - ↑2 * y
9	<id> - <num>	x - ↑2 * y
–	<id> - <num>	x - 2 ↑* y
–	<id> - <term>	x - ↑2 * y
5	<id> - <term> * <factor>	x - ↑2 * y
7	<id> - <factor> * <factor>	x - ↑2 * y
9	<id> - <num> * <factor>	x - ↑2 * y
–	<id> - <num> * <factor>	x - 2 ↑* y
–	<id> - <num> * <factor>	x - 2 * ↑y
9	<id> - <num> * <id>	x - 2 * ↑y
–	<id> - <num> * <id>	x - 2 * y↑

Example

Another possible parse for `x - 2 * y`

Prod'n	Sentential form	Input
–	<goal>	$\uparrow$`x - 2 * y`
1	<expr>	$\uparrow$`x - 2 * y`
2	<expr> + <term>	$\uparrow$`x - 2 * y`
2	<expr> + <term> + <term>	$\uparrow$`x - 2 * y`
2	<expr> + <term> + <term> + <term>	$\uparrow$`x - 2 * y`
2	<expr> + <term> + <term> + <term> + $\cdots$	$\uparrow$`x - 2 * y`
2	$\cdots$	$\uparrow$`x - 2 * y`

If the parser makes the wrong choices, the expansion doesn't terminate.

This isn't a good property for a parser to have.

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 $A \Rightarrow^{+} A\alpha$ for some string α.

Our simple expression grammar is left recursive.

Eliminating left recursion

To remove left recursion, we can transform the grammar.

Consider the grammar fragment:

$$\begin{array}{rcl} \text{<foo>} & ::= & \text{<foo>}\ \alpha \\ & | & \beta \end{array}$$

where α and β do not start with <foo>.

We can rewrite this as:

$$\begin{array}{rcl} \text{<foo>} & ::= & \beta\ \text{<bar>} \\ \text{<bar>} & ::= & \alpha\ \text{<bar>} \\ & | & \epsilon \end{array}$$

where <bar> is a new non-terminal.

This fragment contains no left recursion.

Example

Our expression grammar contains two cases of left recursion

$$
\begin{array}{lcl}
<\text{expr}> & ::= & <\text{expr}> \texttt{+} <\text{term}> \\
 & | & <\text{expr}> \texttt{-} <\text{term}> \\
 & | & <\text{term}> \\
<\text{term}> & ::= & <\text{term}> \texttt{*} <\text{factor}> \\
 & | & <\text{term}> \texttt{/} <\text{factor}> \\
 & | & <\text{factor}>
\end{array}
$$

Applying the transformation gives

$$
\begin{array}{lcl}
<\text{expr}> & ::= & <\text{term}> <\text{expr}'> \\
<\text{expr}'> & ::= & \texttt{+} <\text{term}> <\text{expr}'> \\
 & | & \texttt{-} <\text{term}> <\text{expr}'> \\
 & | & \epsilon \\
<\text{term}> & ::= & <\text{factor}> <\text{term}'> \\
<\text{term}'> & ::= & \texttt{*} <\text{factor}> <\text{term}'> \\
 & | & \texttt{/} <\text{factor}> <\text{term}'> \\
 & | & \epsilon
\end{array}
$$

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 $\leftarrow$ 1 to n
    for j $\leftarrow$ 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 ($A ::= \epsilon$).

How does this algorithm work?

1. impose an 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_j with $j < i$
4. It forward substitutes those away
5. last step in the outer loop converts any direct recursion on A_i to right recursion using the simple transformation showed earlier
6. new non-terminals are added at the end of the order and only involve right recursion

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

for all $k < i$, $\nexists$ a production expanding A_k
that has A_l in its *rhs*, for $l < k$.

At the end of the process ($n < i$), the grammar has no remaining left recursion.

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

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).

Recursive Descent Parsing

Properties

- top-down parsing algorithm
- parser built on procedure calls
- procedures may be (mutually) recursive

Algorithm

- write procedure for each non-terminal
- turn each production into clause
- insert call
  - to procedure A() for non-terminal A
  - to match(`x`) for terminal `x`
- start by invoking procedure for start symbol S

Example

`A ::= a B c` $\Rightarrow$ `A() { match(a); B(); match(c); }`

Recursive Descent Parsing

Example grammar

1	S	::=	a A
2		|	b
3	A	::=	S c

Helpers

```
tok; // current token

match(x) {
  if (tok != x)
    error();
  tok = getToken();
}
```

Parser

```
S() {
  if (tok == a)
    match(a); A();
  else if (tok == b)
    match(b);
  else error();
}

A() {
  S(); match(c);
}
```

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.

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 derived from α.

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

LL(1) property

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

$$FIRST(\alpha) \cap FIRST(\beta) = \epsilon$$

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

Pursuing this idea leads to *predictive* LL(1) parsers.

Left Factoring

What if a grammar does not have the LL(1) 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 $\alpha \neq \epsilon$, then replace all of the A productions
$A ::= \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$
with

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

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 `/`.

$$FIRST(2) \cap FIRST(3) \cap FIRST(4) \neq \emptyset$$

This grammar *fails* the test.

Note: *This grammar is right-associative.*

Example

There are two nonterminals 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

1	<goal>	::=	<expr>
2	<expr>	::=	<term> <expr′ >
3	<expr′ >	::=	`+` <expr>
4		|	`-` <expr>
5		|	ϵ
6	<term>	::=	<factor> <term′ >
7	<term′ >	::=	`*` <term>
8		|	`/` <term>
9		|	ϵ
10	<factor>	::=	**`number`**
11		|	**`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
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> <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 *context free languages* do not have such a grammar.

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

The FIRST set

For a string of grammar symbols α, define $\text{FIRST}(\alpha)$ as

- the set of terminal symbols that begin strings derived from α
- if $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$ contains the set of tokens valid in the first position of α

To build $\text{FIRST}(X)$:

1. if X is a terminal, $\text{FIRST}(X)$ is $\{X\}$
2. if $X ::= \epsilon$, then $\epsilon \in \text{FIRST}(X)$
3. if $X ::= Y_1 Y_2 \cdots Y_k$ then put $\text{FIRST}(Y_1)$ in $\text{FIRST}(X)$
4. if X is a non-terminal and $X ::= Y_1 Y_2 \cdots Y_k$, then $a \in \text{FIRST}(X)$ if $a \in \text{FIRST}(Y_i)$ and $\epsilon \in \text{FIRST}(Y_j)$ for all $1 \leq j < i$
   (If $\epsilon \notin \text{FIRST}(Y_1)$, then $\text{FIRST}(Y_i)$ is irrelevant, for $1 < i$)

Our example grammar

1	⟨goal⟩	::=	⟨expr⟩
2	⟨expr⟩	::=	⟨term⟩ ⟨expr′⟩
3	⟨expr′⟩	::=	`+` ⟨expr⟩
4		|	`-` ⟨expr⟩
5		|	ϵ
6	⟨term⟩	::=	⟨factor⟩ ⟨term′⟩
7	⟨term′⟩	::=	`*` ⟨term⟩
8		|	`/` ⟨term⟩
9		|	ϵ
10	⟨factor⟩	::=	`num`
11		|	`id`

rule	1	2	3	4	*FIRST*
goal	–	–	`num,id`	–	{`num,id`}
expr	–	–	`num,id`	–	{`num,id`}
expr′	–	ϵ	`+,-`	–	{ϵ,`+,-`}
term	–	–	`num,id`	–	{`num,id`}
term′	–	ϵ	`*,/`	–	{ϵ,`*,/`}
factor	–	–	`num,id`	–	{`num,id`}
`num`	`num`	–	–	–	{`num`}
`id`	`id`	–	–	–	{`id`}
`+`	`+`	–	–	–	{`+`}
`-`	`-`	–	–	–	{`-`}
`*`	`*`	–	–	–	{`*`}
`/`	`/`	–	–	–	{`/`}

For a non-terminal A, define FOLLOW(A) as

> the set of terminals that can appear immediately to the right of A in some sentential form

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it

A terminal symbol has no FOLLOW set

To build FOLLOW(X):

1. place `eof` in FOLLOW($\langle$goal$\rangle$)
2. if $A ::= \alpha B \beta$, then put $\{\text{FIRST}(\beta) - \epsilon\}$ in FOLLOW(B)
3. if $A ::= \alpha B$ then put FOLLOW(A) in FOLLOW(B)
4. if $A ::= \alpha B \beta$ and $\epsilon \in \text{FIRST}(\beta)$, then put FOLLOW($A$) in FOLLOW($B$)

The FOLLOW construction

rule	1	2	3	4	*FOLLOW*
goal	`eof`	–	–	–	{`eof`}
expr	–	–	`eof`	–	{`eof`}
expr′	–	–	`eof`	–	{`eof`}
term	–	`+,-`	–	`eof`	{`eof,+,-`}
term′	–	–	`eof,+,-`	–	{`eof,+,-`}
factor	–	`*,/`	–	`eof,+,-`	{`eof,+,-,*,/`}

Using FIRST and FOLLOW

To build a predicative recursive-descent parser:

For each production $A ::= \alpha$ and lookahead *token*

- expand A using production if $token \in \text{FIRST}(\alpha)$
- if $\epsilon \in \text{FIRST}(\alpha)$ expand A using production if $token \in \text{FOLLOW}(A)$
- all other tokens return *error*

If multiple choices, the grammar is not *LL(1)* (predicative).

	`id`	`num`	`+`	`-`	`*`	`/`	`eof`
⟨goal⟩	$g \rightarrow e$	$g \rightarrow e$	–	–	–	–	–
⟨expr⟩	$e \rightarrow te'$	$e \rightarrow te'$	–	–	–	–	–
⟨expr′⟩	–	–	$e' \rightarrow \texttt{+}e$	$e' \rightarrow \texttt{-}e$	–	–	$e' \rightarrow \epsilon$
⟨term⟩	$t \rightarrow ft'$	$t \rightarrow ft'$	–	–	–	–	–
⟨term′⟩	–	–	$t' \rightarrow \epsilon$	$t' \rightarrow \epsilon$	$t' \rightarrow \texttt{*}t$	$t' \rightarrow \texttt{/}t$	$t' \rightarrow \epsilon$
⟨factor⟩	$f \rightarrow \texttt{id}$	$f \rightarrow \texttt{num}$	–	–	–	–	–

Features

- input parsed from left to right
- leftmost derivation
- one token lookahead

Definition

A grammar G is *LL(1)* if and only if, for all non-terminals A, each distinct pair of productions $A ::= \beta$ and $A ::= \gamma$ satisfy the condition $\text{FIRST}(\beta) \cap \text{FIRST}(\gamma) = \emptyset$

A grammar G is $LL(1)$ if and only if for each set of productions $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \cdots, \text{FIRST}(\alpha_n)$ are all pairwise disjoint
2. if $\alpha_i \Rightarrow^* \epsilon$, then $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$, for all $1 \leq j \leq n, i \neq j$.

If G is ϵ–free, condition 1 is sufficient.

LL(1) grammars

Provable facts about *LL(1)* grammars:

- no left recursive grammar is *LL(1)*
- no ambiguous grammar is *LL(1)*
- *LL(1)* parsers operate in linear time
- an ϵ–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* *LL(1)* grammar

Not all grammars are *LL(1)*

- $S ::= aS \mid a$

  is not *LL(1)*

  $\text{FIRST}(aS) = \text{FIRST}(a) = \{a\}$

- $$S ::= aS'$$
  $$S' ::= aS' \mid \epsilon$$

  accepts the same language and is *LL(1)*

LL grammars

LL(1) grammars

- may need to rewrite grammar (left recursion, left factoring)
- resulting grammar larger, less maintainable

LL(k) grammars

- k-token lookahead, more powerful than *LL(1)* grammars
- example:
  $S ::= ac \mid abc$ is *LL(2)*

Not all grammars are *LL(k)*

- example:
  $S ::= a^i b^j \qquad \text{where } i \geq j$
- equivalent to dangling else problem
- problem - must choose production after k tokens of lookahead

Bottom-up parsers avoid this problem