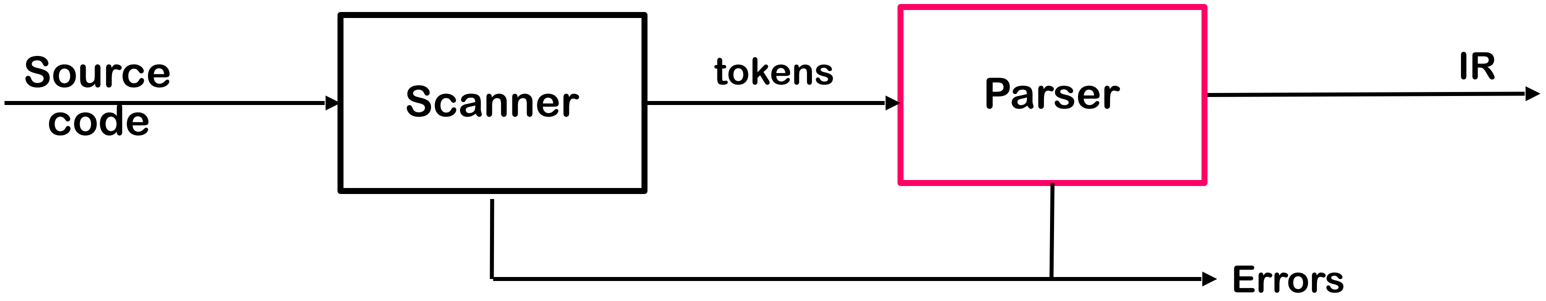
Introduction to Parsing

The Front End



Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

The Study of Parsing

The process of discovering a derivation for some sentence

- Need a mathematical model of syntax — a grammar G
- Need an algorithm for testing membership in $L(G)$

Roadmap for our study of parsing

1. Context-free grammars and derivations
2. Top-down parsing
   - Generated LL(1) parsers & hand-coded recursive descent parsers
3. Bottom-up parsing
   - Generated LR(1) parsers

Why Not Use Regular Languages & DFAs?

Not all languages are regular (RL's $\subset$ CFL's $\subset$ CSL's)

You cannot construct DFA's to recognize these languages

- L = $\{ p^k q^k \}$ (parenthesis languages)
- L = $\{ wcw^r \mid w \in \Sigma^* \}$

Neither of these is a regular language

To recognize these features requires an arbitrary amount of context (left or right …)

But, this issue is somewhat subtle. You can construct DFA's for

- Strings with alternating 0's and 1's

  $(\varepsilon \mid 1) (01)^* (\varepsilon \mid 0)$
- Strings with an even number of 0's and 1's

RE's can count bounded sets and bounded differences

$\Rightarrow$ Cannot add parenthesis, brackets, begin-end pairs, …

A More Useful Grammar Than Sheep Noise

To explore the uses of CFGs, we need a more complex grammar

0	Expr	→	Expr Op Expr
1		|	num
2		|	id
3	Op	→	+
4		|	-
5		|	*
6		|	/

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	<id,x> Op Expr
4	<id,x> - Expr
0	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
2	<id,x> - <num,2> * <id,y>

- Such a sequence of rewrites is called a derivation
- Process of discovering a derivation is called parsing

We denote this derivation: Expr ⇒ id – num * id

Derivations

The point of parsing is to construct a derivation

- At each step, we choose a nonterminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- Leftmost derivation — replace leftmost NT at each step
- Rightmost derivation — replace rightmost NT at each step

These are the two systematic derivations

(We don't care about randomly-ordered derivations!)

The example on the preceding slide was a leftmost derivation

- Of course, there is also a rightmost derivation
- Interestingly, it turns out to be different

Derivations

The point of parsing is to construct a derivation

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$$

- Each γ_i is a sentential form
  - If γ contains only terminal symbols, γ is a sentence in L(G)
  - If γ contains 1 or more non-terminals, γ is a sentential form
- To get γ_i from γ_{i-1}, expand some NT $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
  - Replace the occurrence of $A \in \gamma_{i-1}$ with β to get γ_i
  - In a leftmost derivation, it would be the first NT $A \in \gamma_{i-1}$

A left-sentential form occurs in a <u>leftmost</u> derivation

A right-sentential form occurs in a <u>rightmost</u> derivation

The Two Derivations for x - 2 * y

Leftmost derivation

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	<id,x> Op Expr
4	<id,x> - Expr
0	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
2	<id,x> - <num,2> * <id,y>

Rightmost derivation

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	Expr Op <id,y>
5	Expr * <id,y>
0	Expr Op Expr * <id,y>
1	Expr Op <num,2> * <id,y>
4	Expr - <num,2> * <id,y>
2	<id,x> - <num,2> * <id,y>

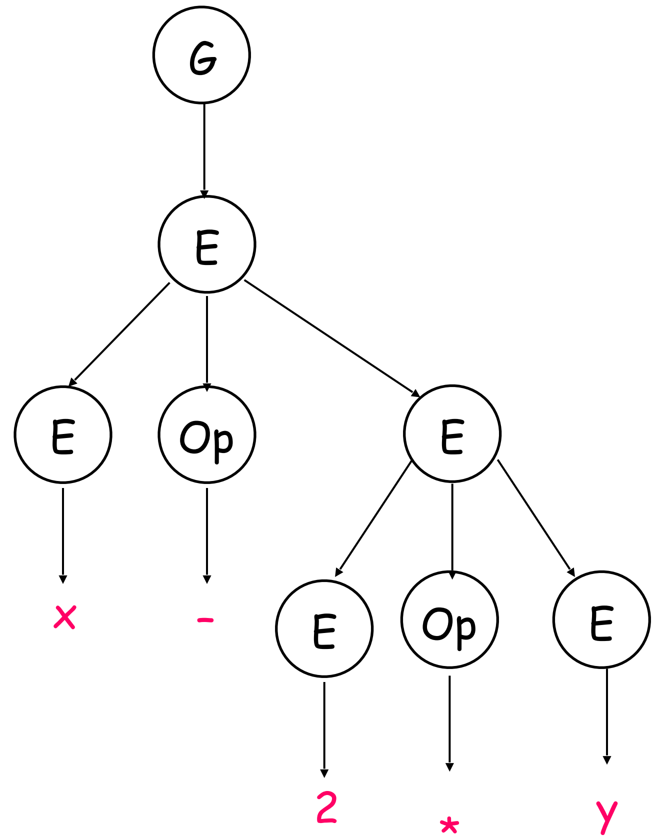
In both cases, Expr ⇒ id - num * id

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders!

Derivations and Parse Trees

Leftmost derivation

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	<id,x> Op Expr
4	<id,x> - Expr
0	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
2	<id,x> - <num,2> * <id,y>

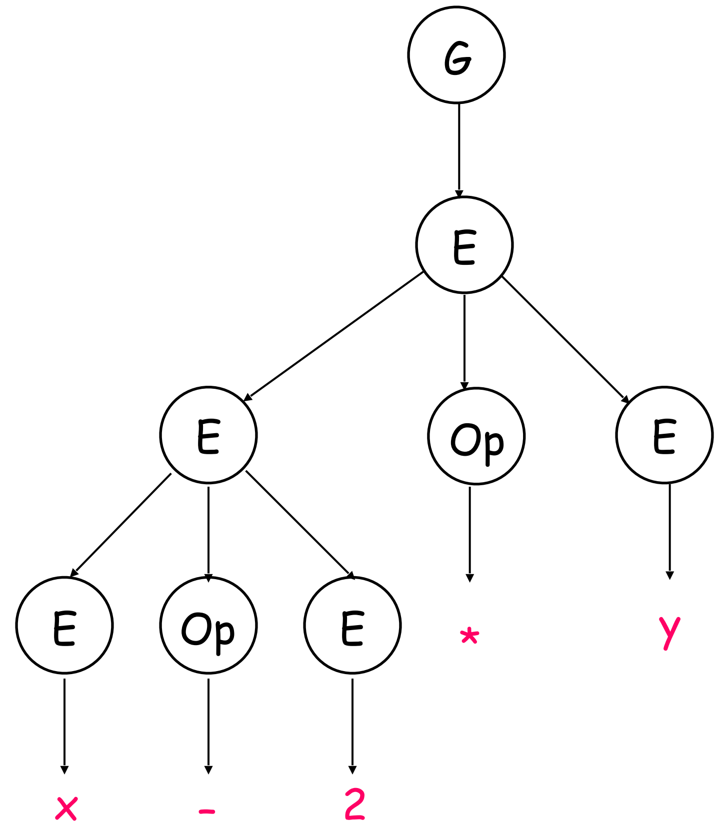


This evaluates as x - (2 * y)

Derivations and Parse Trees

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	Expr Op <id,y>
5	Expr * <id,y>
0	Expr Op Expr * <id,y>
1	Expr Op <num,2> * <id,y>
4	Expr - <num,2> * <id,y>
2	<id,x> - <num,2> * <id,y>

This evaluates as (x – 2) * y



This ambiguity is NOT good

Derivations and Precedence

These two derivations point out a problem with the grammar:

It has no notion of <u>precedence</u>, or implied order of evaluation

To add precedence

- Create a nonterminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions

- Parentheses first (level 1)
- Multiplication and division, next (level 2)
- Subtraction and addition, last (level 3)

Derivations and Precedence

Adding the standard algebraic precedence produces:

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

This grammar is slightly larger

- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations
- Correctness trumps the speed of the parser

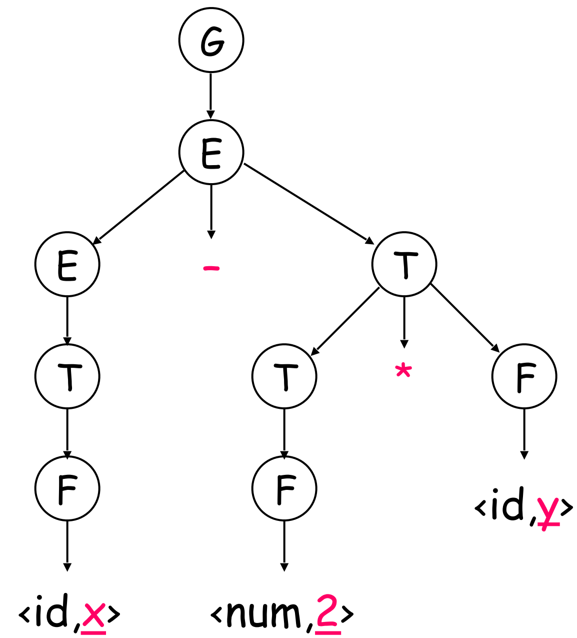
Let's see how it parses x - 2 * y

Cannot handle precedence in an RE for expressions

Introduced parentheses, too (beyond power of an RE)

Derivations and Precedence

Rule	Sentential Form
—	Goal
0	Expr
2	Expr - Term
4	Expr - Term * Factor
9	Expr - Term * <id,y>
6	Expr - Factor * <id,y>
8	Expr - <num,2> * <id,y>
3	Term - <num,2> * <id,y>
6	Factor - <num,2> * <id,y>
9	<id,x> - <num,2> * <id,y>



Its parse tree

It derives x – (2 * y), along with an appropriate parse tree.

Both the leftmost and rightmost derivations give the parse tree, because the grammar directly and explicitly encodes the desired precedence.

Ambiguous Grammars

Let's leap back to our original expression grammar.

It had other problems.

0	Expr	→	Expr Op Expr
1		|	number
2		|	id
3	Op	→	+
4		|	-
5		|	*
6		|	/

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	<id,x> Op Expr
4	<id,x> - Expr
0	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
2	<id,x> - <num,2> * <id,y>

we have alternatives here

- This grammar allows multiple leftmost derivations for x - 2 * y
- Hard to automate derivation if > 1 choice
- The grammar is ambiguous

Two Leftmost Derivations for x - 2 * y

The Difference:

- Different productions chosen on the second step

Original choice

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	<id,x> Op Expr
4	<id,x> - Expr
0	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
1	<id,x> - <num,2> * <id,y>

New choice

Rule	Sentential Form
—	Expr
0	Expr Op Expr
0	Expr Op Expr Op Expr
2	<id,x> Op Expr Op Expr
4	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
2	<id,x> - <num,2> * <id,y>

- Both derivations succeed in producing x - 2 * y

Two Leftmost Derivations for x - 2 * y

The Difference:

- Different productions chosen on the second step

Rule	Sentential Form
—	Expr
0	Expr Op Expr
2	<id,x> Op Expr
4	<id,x> - Expr
0	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
2	<id,x> - <num,2> * <id,y>

Rule	Sentential Form
—	Expr
0	Expr Op Expr
0	Expr Op Expr Op Expr
2	<id,x> Op Expr Op Expr
4	<id,x> - Expr Op Expr
1	<id,x> - <num,2> Op Expr
5	<id,x> - <num,2> * Expr
2	<id,x> - <num,2> * <id,y>

New choice

Different choice is possible, we are in the same situation

Ambiguous Grammars

Definitions

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar
  - However, they must have the same parse tree!

Classic example — the if-then-else problem

Stmt → if Expr then Stmt
| if Expr then Stmt else Stmt
| … other stmts …

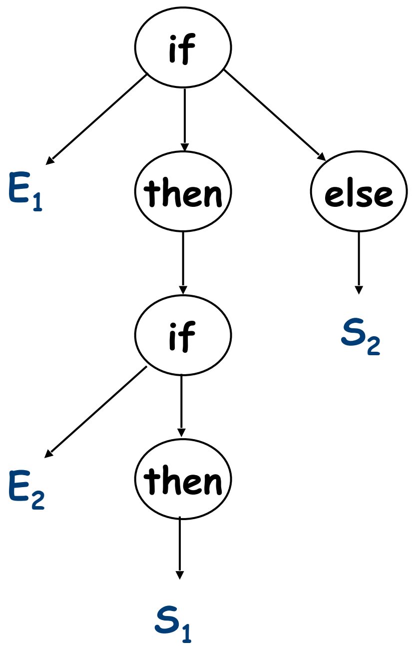
This ambiguity is inherent in the grammar

Stmt → if Expr then Stmt
| if Expr then Stmt else Stmt
| … other stmts …

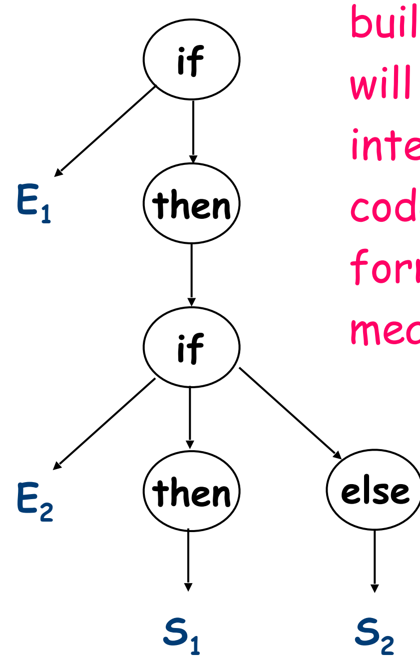
Ambiguity

This sentential form has two derivations

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$



production 2, then production 1



production 1, then production 2

Part of the problem is that the structure built by the parser will determine the interpretation of the code, and these two forms have different meanings!

Ambiguity

The grammar forces the structure to match the desired meaning.

Removing the ambiguity

- Must rewrite the grammar to avoid generating the problem
- Match each else to innermost unmatched if (common sense rule)

0	Stmt	→	if Expr then Stmt
1		|	if Expr then WithElse else Stmt
2		|	Other Statements
3	WithElse	→	if Expr then WithElse else WithElse
4		|	Other Statements

With this grammar, example has only one rightmost derivation

Intuition: once into WithElse, we cannot generate an unmatched else

… a final if without an else can only come through rule 2 …

Ambiguity

<u>if</u> $Expr_1$ <u>then</u> <u>if</u> $Expr_2$ <u>then</u> $Stmt_1$ <u>else</u> $Stmt_2$

Rule	Sentential Form
—	Stmt
0	<u>if</u> Expr <u>then</u> Stmt
1	<u>if</u> Expr <u>then</u> <u>if</u> Expr <u>then</u> WithElse <u>else</u> Stmt
2	<u>if</u> Expr <u>then</u> <u>if</u> Expr <u>then</u> WithElse <u>else</u> S_2
4	<u>if</u> Expr <u>then</u> <u>if</u> Expr <u>then</u> S_1 <u>else</u> S_2
?	<u>if</u> Expr <u>then</u> <u>if</u> E_2 <u>then</u> S_1 <u>else</u> S_2
?	<u>if</u> E_1 <u>then</u> <u>if</u> E_2 <u>then</u> S_1 <u>else</u> S_2

Other productions to derive Exprs

This grammar has only one rightmost derivation for the example

Deeper Ambiguity

Ambiguity usually refers to confusion in the CFG

Overloading can create deeper ambiguity

a = f(17)

In many Algol-like languages, f could be either a function or a subscripted variable

Disambiguating this one requires context

- Need values of declarations
- Really an issue of type, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - Step outside grammar rather than use a more complex grammar

Ambiguity - the Final Word

Ambiguity arises from two distinct sources

- Confusion in the context-free syntax (if-then-else)
- Confusion that requires context to resolve (overloading)

Resolving ambiguity

- To remove context-free ambiguity, rewrite the grammar
- To handle context-sensitive ambiguity takes cooperation
  - Knowledge of declarations, types, …
  - Accept a superset of L(G) & check it by other means (Context Sensitive analysis)
  - This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar

- Parsing techniques that "do the right thing"
- i.e., always select the same derivation

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" $\Rightarrow$ 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 lower 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 symbol is added to the border and it doesn't match the border, backtrack
3. Find the next node to be expanded (label $\in$ NT)

The key is picking the right production in step 1

— That choice should be guided by the input string

Remember the expression grammar?

We will call this version "the classic expression grammar"

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

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

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y

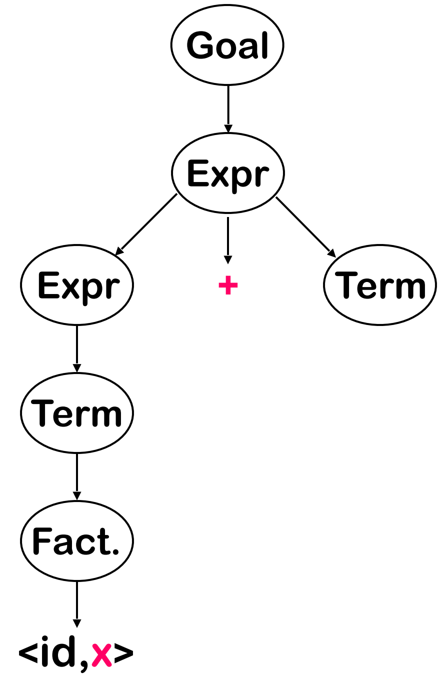
↑ is the position in the input buffer

Example

↑ is the position in the input buffer

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
0	Expr	↑x - 2 * y
1	Expr +Term	↑x - 2 * y
3	Term +Term	↑x - 2 * y
6	Factor +Term	↑x - 2 * y
9	<id,x> +Term	↑x - 2 * y
→	<id,x> +Term	x ↑- 2 * y



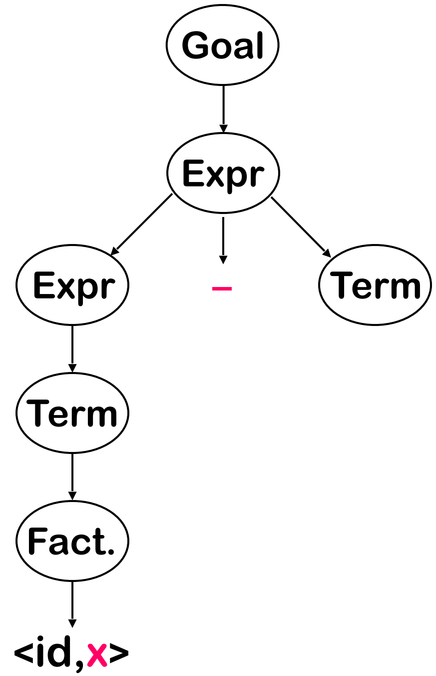
This worked well, except that "–" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
0	Expr	↑x - 2 * y
2	Expr -Term	↑x - 2 * y
3	Term -Term	↑x - 2 * y
6	Factor -Term	↑x - 2 * y
9	<id,x> - Term	↑x - 2 * y
→	<id,x> - Term	x ↑- 2 * y
→	<id,x> - Term	x - ↑2 * y



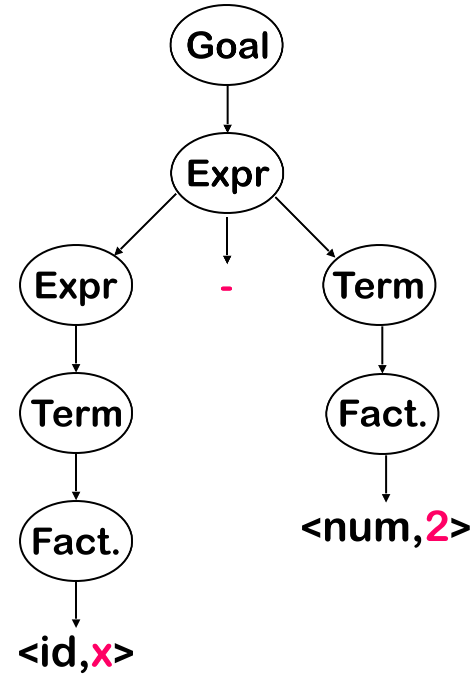
Now, “-” and “-” match

Now we can expand Term to match “2”

Example

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

Rule	Sentential Form	Input
→	<id,x> - Term	x - ↑2 * y
6	<id,x> - Factor	x - ↑2 * y
8	<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 NTs left to expand
- The expansion terminated too soon

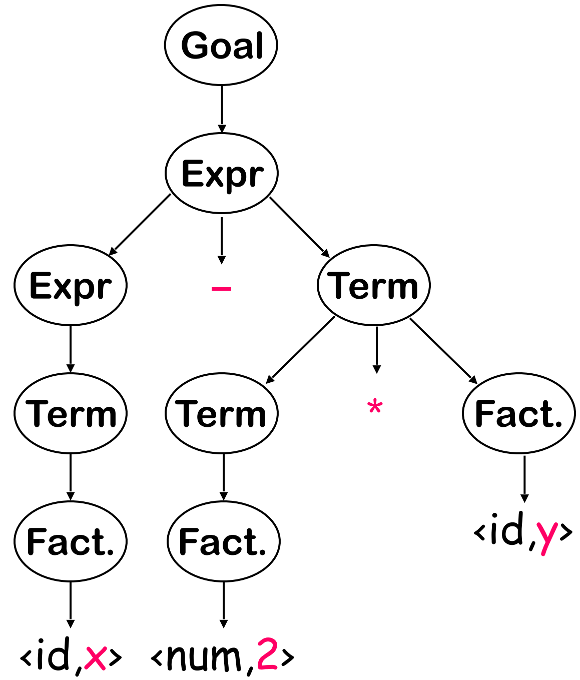
⟹ Need to backtrack

Example

The Point: The parser must make the right choice when it expands a NT. Wrong choices lead to wasted effort.

Trying again with "2" in x – 2 * y :

Rule	Sentential Form	Input
→	<id,x> - Term	x - ↑2 * y
4	<id,x> - Term * Factor	x - ↑2 * y
6	<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
0	Expr	↑x - 2 * y
1	Expr +Term	↑x - 2 * y
1	Expr + Term +Term	↑x - 2 * y
1	Expr + Term +Term + Term	↑x - 2 * y
1	And so on	↑x - 2 * y

Consumes no input!

This expansion doesn't terminate

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

Our classic expression grammar is left recursive

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

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

Non-termination is <u>always</u> a bad property in a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$\begin{aligned} Fee &\rightarrow Fee\ \alpha \\ &\mid\ \beta \end{aligned}$$

where neither α nor β start with Fee

We can rewrite this fragment as

$$\begin{aligned} Fee &\rightarrow \beta\ Fie \\ Fie &\rightarrow \alpha\ Fie \\ &\mid\ \varepsilon \end{aligned}$$

where Fie is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string

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

Right recursion often means right associativity. In this case, the grammar does not display any particular associative bias.

Eliminating Left Recursion

Substituting them back into the grammar yields

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

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
  - ⇒ The naïve transformation yields a right recursive grammar, which changes the implicit associativity
- 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

G → E

E → E + T

E → T

T → E * T

T → <u>id</u>

2. A_i = E

G → E

E → T E'

E' → + T E'

E' → ε

T → E * T

T → <u>id</u>

3. A_i = T, A_s = E

G → E

E → T E'

E' → + T E'

E' → ε

T → T E' * T

T → <u>id</u>

4. A_i = T

G → E

E → T E'

E' → + T E'

E' → ε

T → <u>id</u> T'

T' → E ' * T T'

T' → ε

Go to Algorithm

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
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

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

We will focus, for now, on LL(1) grammars & predictive parsing

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

We will defer the problem of how to compute FIRST sets for the moment.

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, $\underline{x} \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

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 !

This is almost correct
See the next slide

Predictive Parsing

What about ε-productions?

⇒ 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(A), too, where

FOLLOW(A) = the set of terminal symbols that can immediately follow A in a sentential form

Define FIRST$^+$($A \rightarrow \alpha$) as

- FIRST(α) ∪ 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$^+$($A \rightarrow \alpha$) ∩ FIRST$^+$($A \rightarrow \beta$) = ∅

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

$$FIRST^+(A \rightarrow \beta_i) \cap FIRST^+(A \rightarrow \beta_j) = \emptyset \text{ if } i \neq j$$

```
/* find an A */
if (current_word ∈ FIRST(A→β1))
    find a β1 and return true
else if (current_word ∈ FIRST(A→β2))
    find a β2 and return true
else if (current_word ∈ FIRST(A→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" a procedure for each nonterminal

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

Parsers that capitalize on the LL(1) property are called <u>predictive parsers</u>.

One kind of predictive parser is the <u>recursive descent</u> parser.

Recursive Descent Parsing

Recall the expression grammar, after transformation

0	Goal	→	Expr
1	Expr	→	Term Expr'
2	Expr'	→	+ Term Expr'
3		|	- Term Expr'
4		|	ε
5	Term	→	Factor Term'
6	Term'	→	* Factor Term'
7		|	/ Factor Term'
8		|	ε
9	Factor	→	(Expr)
10		|	number
11		|	id

This produces a parser with six mutually recursive routines:

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

Each recognizes one NT or T

The term descent refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
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 Number, Identifier, or "(", found token instead, or failed to find Expr or ")" after "("

```
Factor( )
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
     token ← next_token( );
     return true;
  else if (token = Lparen)
     token ← next_token( );
     if (Expr( ) = true & token = Rparen) then
        token ← next_token( );
        return true;
  // fall out of if statement
  report syntax error;
     return false;
```

EPrime, Term, & TPrime follow the same basic lines

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  – Context-free grammars ✓
- Top-down parsers
  – Algorithm & its problem with left recursion ✓
  – Ambiguity ✓
  – Left-recursion removal ✓
- Predictive top-down parsing
  – The LL(1) condition ✓
  – Simple recursive descent parsers
  – First and Follow sets
  – Table-driven LL(1) parsers

What If My Grammar Is Not LL(1) ?

Can we transform a non-LL(1) grammar into an LL(1) gramar?

- In general, the answer is no
- In some cases, however, the answer is yes

Assume a grammar G with productions $A \rightarrow \alpha\, \beta_1$ and $A \rightarrow \alpha\, \beta_2$

- If α derives anything other than ε, then

$$\text{FIRST}^+(A \rightarrow \alpha\, \beta_1) \cap \text{FIRST}^+(A \rightarrow \alpha\, \beta_2) \neq \varnothing$$

- And the grammar is not LL(1)

If we pull the common prefix, α, into a separate production, we may make the grammar LL(1).

$$A \rightarrow \alpha\, A',\ A' \rightarrow \beta_1 \text{ and } A' \rightarrow \beta_2$$

Now, if $\text{FIRST}^+(A' \rightarrow \beta_1) \cap \text{FIRST}^+(A' \rightarrow \beta_2) = \varnothing$, G may be LL(1)

What If My Grammar Is Not LL(1) ?

Left Factoring

For each nonterminal A

find the longest prefix α common to 2 or more alternatives for A

if $\alpha \neq \varepsilon$ then

replace all of the productions

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \alpha\beta_n \mid \gamma$

with

$A \rightarrow \alpha A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

Repeat until no nonterminal has alternative rhs' with a common prefix

This transformation makes some grammars into LL(1) grammars

There are languages for which no LL(1) grammar exists

Left Factoring Example

Consider a simple right-recursive expression grammar

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

To choose between 1, 2, & 3, an LL(1) parser must look past the <u>number</u> or <u>id</u> to see the operator.

FIRST⁺(1) = FIRST⁺(2) = FIRST⁺(3)

and

FIRST⁺(4) = FIRST⁺(5) = FIRST⁺(6)

Let's left factor this grammar.

Left Factoring Example

After Left Factoring, we have

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

Clearly,

FIRST$^+$(2), FIRST$^+$(3), & FIRST$^+$(4)

are disjoint, as are

FIRST$^+$(6), FIRST$^+$(7), & FIRST$^+$(8)

The grammar now has the LL(1) property

This transformation makes some grammars into LL(1) grammars.

There are languages for which no LL(1) grammar exists.

First and Follow Sets

First(α)

For some $\alpha \in (T \cup NT)^*$, 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

Follow(A)

For some $A \in NT$, define Follow(A) as the set of symbols that can occur immediately after A in a valid sentential form

Follow(S) = {EOF}, where S is the start symbol

To build Follow sets, we need First sets …

Computing FIRST Sets

For a grammar symbol X, FIRST(X) is defined as follows.

- For every terminal X, FIRST(X) = {X}.
- For every nonterminal X, if $X \rightarrow Y_1Y_2 \ldots Y_n$ is a production, then
  - $\text{FIRST}(Y_1) \subseteq \text{FIRST}(X)$.
  - Furthermore, if $Y_1, Y_2, \ldots, Y_k$ are nullable ($Y_i \xrightarrow{*} \varepsilon$) then
    $$\text{FIRST}(Y_{k+1}) \subseteq \text{FIRST}(X).$$

FIRST

- We are concerned with FIRST(X) only for the nonterminals of the grammar.
- FIRST(X) for terminals is trivial.
- According to the definition, to determine FIRST(*A*), we must inspect all productions that have *A* on the left.

FIRST Example

Let the grammar be

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E' \mid \varepsilon.$

$T \rightarrow F\,T'$

$T' \rightarrow *\,F\,T' \mid \varepsilon.$

$F \rightarrow (E) \mid$ **id** $\mid$ **num**

- Find FIRST(E).
  - E occurs on the left in only one production

    $$E \rightarrow T\,E'.$$

  - Therefore, FIRST(T) $\subseteq$ FIRST(E).
  - Furthermore, T is not nullable.
  - Therefore, FIRST(E) = FIRST(T).
  - We have yet to determine FIRST(T)

FIRST Example

Let the grammar be

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'\ |\ \varepsilon.$$

$$T \rightarrow F\ T'$$

$$T' \rightarrow *\ F\ T'\ |\ \varepsilon.$$

$$F \rightarrow (E)\ |\ \textbf{id}\ |\ \textbf{num}$$

- Find FIRST(T).
  - T occurs on the left in only
  - one production

$$T \rightarrow F\ T'.$$

  - Therefore, FIRST(F) $\subseteq$ FIRST(T).
  - Furthermore, F is not nullable.
  - Therefore, FIRST(T) = FIRST(F).
  - We have yet to determine FIRST(F).

FIRST Example

Let the grammar be

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'\ |\ \varepsilon.$$

$$T \rightarrow F\ T'$$

$$T' \rightarrow *\ F\ T'\ |\ \varepsilon.$$

$$F \rightarrow (E)\ |\ \mathbf{id}\ |\ \mathbf{num}$$

- Find FIRST(F).
  - FIRST(F) = {(, **id**, **num**}.
- Therefore,
  - FIRST(E) = {(, **id**, **num**}.
  - FIRST(T) = {(, **id**, **num**}.
- Find FIRST(E').
  - FIRST(E') = {+}.
- Find FIRST(T').
  - FIRST(T') = {*}.

Computing FOLLOW Sets

- For a grammar symbol X, FOLLOW(X) is defined as follows.
  - If S is the start symbol, then $\$ \in$ FOLLOW(S).
  - If $A \rightarrow \alpha B \beta$ is a production, then FIRST(β) $\subseteq$ FOLLOW(B).
  - If $A \rightarrow \alpha B$ is a production, or $A \rightarrow \alpha B \beta$ is a production and β is nullable, then FOLLOW(A) $\subseteq$ FOLLOW(B).

FOLLOW

- We are concerned about FOLLOW(X) only for the nonterminals of the grammar.
- According to the definition, to determine FOLLOW(A), we must inspect all productions that have *A* on the right.

FOLLOW Example

Let the grammar be

$$E \rightarrow T\,E'$$

$$E' \rightarrow +\,T\,E' \mid \varepsilon.$$

$$T \rightarrow F\,T'$$

$$T' \rightarrow *\,F\,T' \mid \varepsilon.$$

$$F \rightarrow (E) \mid \mathbf{id} \mid \mathbf{num}$$

- Find FOLLOW(E).
  - E is the start symbol, therefore $\$ \in$ FOLLOW(E).
  - E occurs on the right in only one production.

    $$F \rightarrow (E).$$

  - Therefore FOLLOW(E) = {$,)}.

FOLLOW Example

Let the grammar be

$$E \rightarrow T\,E'$$

$$E' \rightarrow +\,T\,E' \mid \varepsilon.$$

$$T \rightarrow F\,T'$$

$$T' \rightarrow *\,F\,T' \mid \varepsilon.$$

$$F \rightarrow (E) \mid \mathbf{id} \mid \mathbf{num}$$

- Find FOLLOW(E').
  - E' occurs on the right in two productions.

    $$E \rightarrow T\,E'$$

    $$E' \rightarrow +\,T\,E'.$$

  - Therefore, FOLLOW(E') = FOLLOW(E) = {$,)}.

FOLLOW Example

Let the grammar be

$$E \rightarrow T\,E'$$

$$E' \rightarrow +\,T\,E' \mid \varepsilon.$$

$$T \rightarrow F\,T'$$

$$T' \rightarrow *\,F\,T' \mid \varepsilon.$$

$$F \rightarrow (E) \mid \textbf{id} \mid \textbf{num}$$

- Find FOLLOW(T).
  - T occurs on the right in two productions.

    $$E \rightarrow T\,E'$$

    $$E' \rightarrow +\,T\,E'.$$

  - Therefore, FOLLOW(T) contains FIRST(E') = {+}.
  - However, E' is nullable, therefore it also contains FOLLOW(E) = {\$,)} and FOLLOW(E') = {\$,)}.
  - Therefore, FOLLOW(T) = {+, \$,)}.

FOLLOW Example

Let the grammar be

$$E \rightarrow T\,E'$$

$$E' \rightarrow +\,T\,E' \mid \varepsilon.$$

$$T \rightarrow F\,T'$$

$$T' \rightarrow *\,F\,T' \mid \varepsilon.$$

$$F \rightarrow (E) \mid \textbf{id} \mid \textbf{num}$$

- Find FOLLOW(T').
  - T' occurs on the right in two productions.

$$T \rightarrow F\,T'$$

$$T' \rightarrow *\,F\,T'.$$

  - Therefore, FOLLOW(T') = FOLLOW(T) = {$,), +}.

FOLLOW Example

Let the grammar be

$$E \rightarrow T\,E'$$

$$E' \rightarrow +\,T\,E' \mid \varepsilon.$$

$$T \rightarrow F\,T'$$

$$T' \rightarrow *\,F\,T' \mid \varepsilon.$$

$$F \rightarrow (E) \mid \mathbf{id} \mid \mathbf{num}$$

- Find FOLLOW(F).
  - F occurs on the right in two productions.

    $$T \rightarrow F\,T'$$

    $$T' \rightarrow *\,F\,T'.$$

  - Therefore, FOLLOW(F) contains FIRST(T') = {*}.
  - However, T' is nullable, therefore it also contains FOLLOW(T) = {+, \$,)} and FOLLOW(T') = {\$,), +}.
  - Therefore, FOLLOW(F) = {*, \$,), +}.

Classic Expression Grammar

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

Symbol	FIRST	FOLLOW
<u>num</u>	<u>num</u>	Ø
<u>id</u>	<u>id</u>	Ø
+	+	Ø
-	-	Ø
*	*	Ø
/	/	Ø
<u>(</u>	<u>(</u>	Ø
<u>)</u>	<u>)</u>	Ø
<u>eof</u>	<u>eof</u>	Ø
ε	ε	Ø
Goal	<u>(</u>,<u>id</u>,<u>num</u>	eof
Expr	<u>(</u>,<u>id</u>,<u>num</u>	<u>)</u>, eof
Expr’	+, -, ε	<u>)</u>, eof
Term	<u>(</u>,<u>id</u>,<u>num</u>	+, -, <u>)</u>, eof
Term’	*, /, ε	+,-, <u>)</u>, eof
Factor	<u>(</u>,<u>id</u>,<u>num</u>	+,-,*,/,<u>)</u>,eof

Classic Expression Grammar

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

Prod'n	FIRST+
0	<u>(</u>,<u>id</u>,<u>num</u>
1	<u>(</u>,<u>id</u>,<u>num</u>
2	+
3	-
4	ε,<u>)</u>, eof
5	<u>(</u>,<u>id</u>,<u>num</u>
6	*
7	/
8	ε,+,-, <u>)</u>, eof
9	<u>number</u>
10	<u>id</u>
11	<u>(</u>

Building Top-down Parsers for LL(1) Grammars

Given an LL(1) grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs's
  - Each returns true on success and throws an error on false
  - Simple, working (perhaps ugly) code
- This automatically constructs a recursive-descent parser

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

Building Top-down Parsers

Strategy

- Encode knowledge in a table
- Use a standard "skeleton" parser to interpret the table

Example

- The non-terminal Factor has 3 expansions
  - (Expr) or Identifier or Number
- Table might look like:

0	Goal	→ Expr
1	Expr	→ Term Expr'
2	Expr'	→ + Term Expr'
3		| - Term Expr'
4		| ε
5	Term	→ Factor Term'
6	Term'	→ * Factor Term'
7		| / Factor Term'
8		| ε
9	Factor	→ number
10		| id
11		| (Expr)

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Id.	Num.	(	)	EOF
Factor	—	—	—	—	10	9	11	—	—

Cannot expand Factor into an operator ⇒ error

Expand Factor by rule 9 with input "number"

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr′	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term′	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Row we built earlier

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need an interpreter for the table (skeleton parser)

LL(1) Skeleton Parser

```
word ← NextWord()               // Initial conditions, including
push EOF onto Stack             // a stack to track local goals
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and word = EOF then
     break & report success   // exit on success
  else if TOS is a terminal then
     if TOS matches word then
        pop Stack           // recognized TOS
        word ← NextWord()
     else report error looking for TOS  // error exit
  else                        // TOS is a non-terminal
     if TABLE[TOS,word] is A→ B1B2…Bk then
        pop Stack                 // get rid of A
        push Bk, Bk-1, …, B1      // in that order
     else break & report error expanding TOS
  TOS ← top of Stack
```

Building Top-down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need a table-driven interpreter for the table
- Need an algorithm to build the table

Filling in TABLE[X,y], $X \in NT$, $y \in T$

1. entry is the rule $X \rightarrow \beta$, if $y \in \text{FIRST}^{+}(X \rightarrow \beta)$
2. entry is error if rule 1 does not define

If any entry has more than one rule, *G* is not LL(1)

We call this algorithm the LL(1) table construction algorithm