Compiler Construction

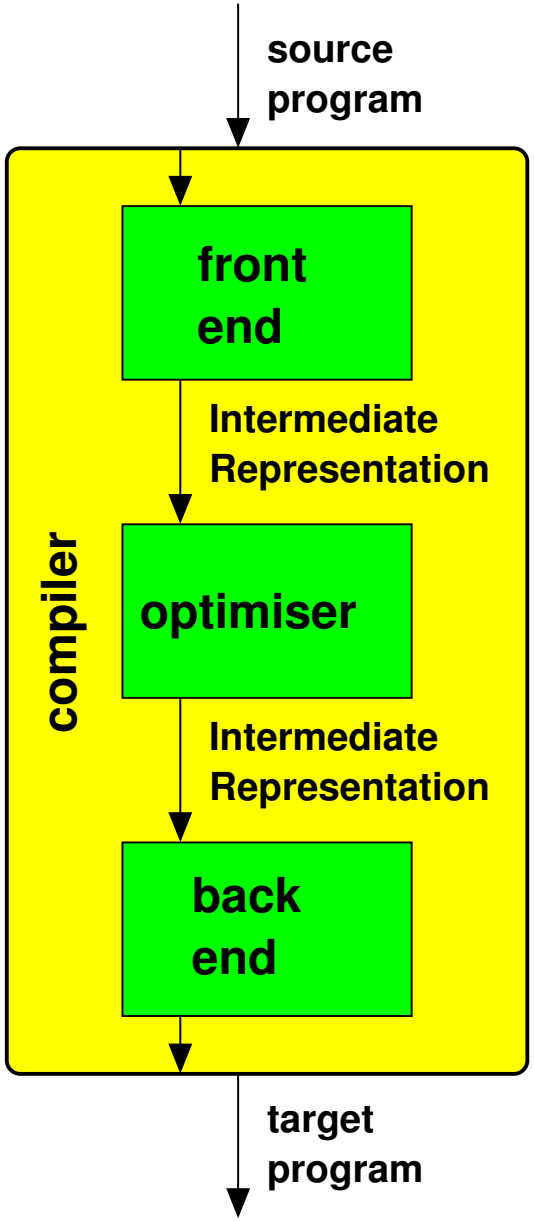
Chapter 4

Syntactic Analysis

Clemens Grelck

System and Network Engineering Lab

Advanced Compiler Structure



Front end:

- Analysis of source program

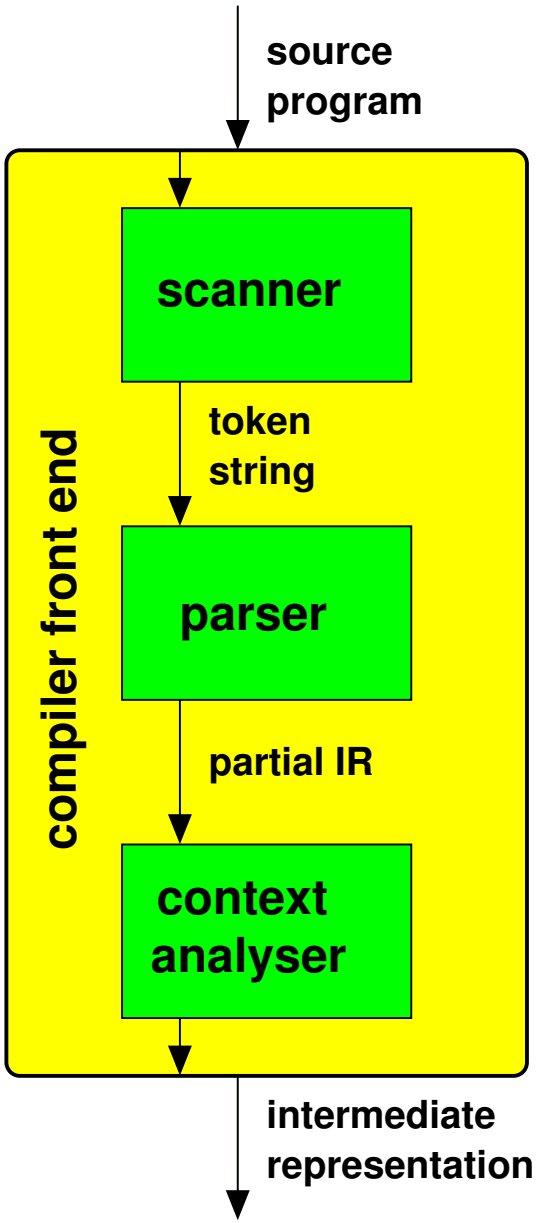
Optimiser (Middle end):

- Code transformation within intermediate representation

Back end:

- Synthesis of target program

Compiler Frontend



Scanner:

- Lexicographic analysis
- Input: finite stream of characters
- Output: finite stream of tokens

Parser:

- Syntactic analysis
- Input: finite stream of tokens
- Output: abstract syntax tree

Context analyser:

- Semantic analysis
- Input: abstract syntax tree (AST)
- Output: abstract syntax tree with symbol tables

Parser

Syntactic analysis:

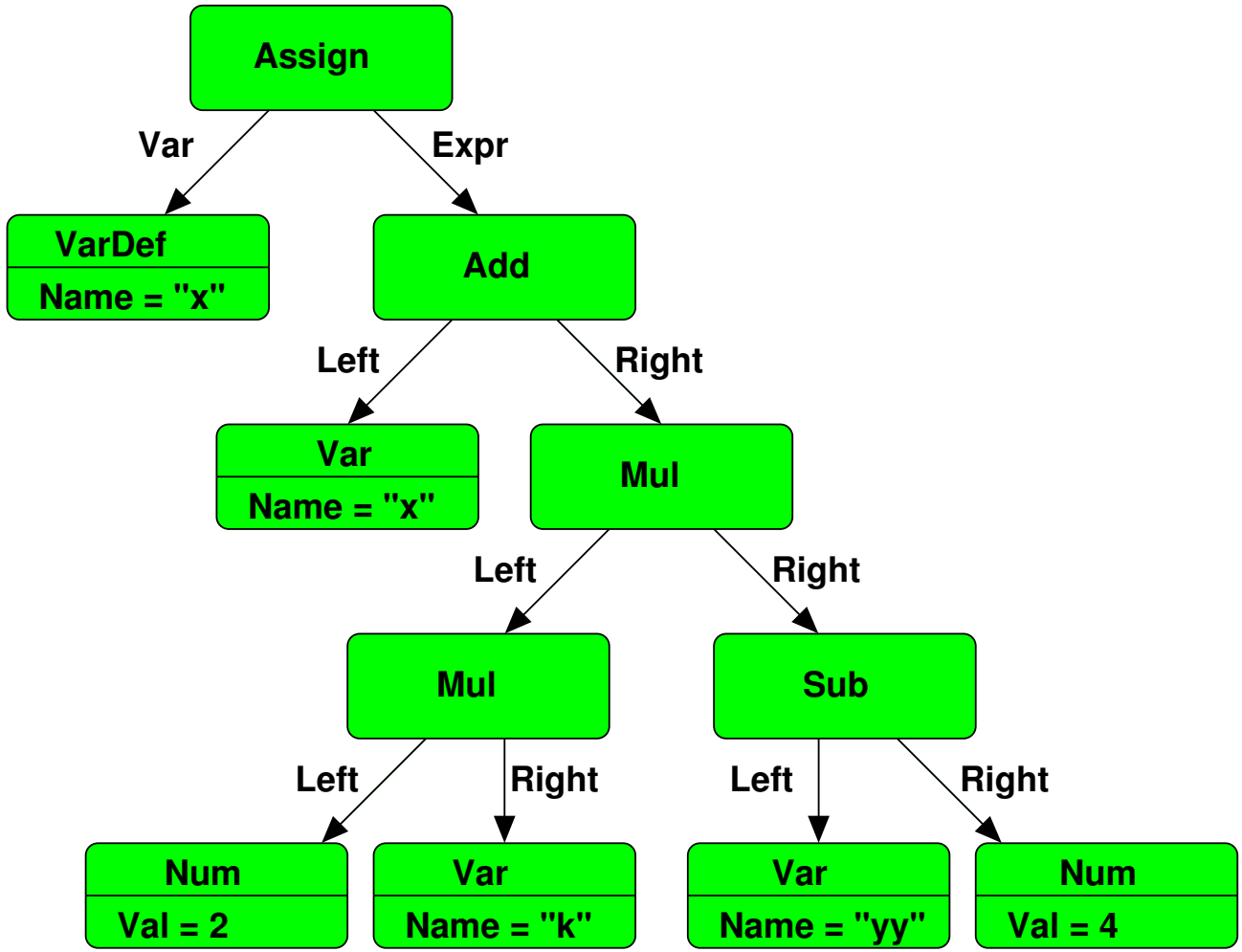
- Input: finite stream of tokens
- Output: abstract syntax tree (AST)
- Tree structure exposed explicitly
- Structure-defining tokens vanish
  - semicolons
  - brackets
- Operator precedence and associativity resolved
- Detection of syntactic errors
- Passed programs called well-formed

Parser Example: x = x + 2 * k * (yy-4);

Running example:

```
ID('x')
ASSIGN
ID('x')
ADD
NUM(2)
MUL
ID('k')
MUL
BRACKET_L
ID('yy')
SUB
NUM(4)
BRACKET_R
SEMICOLON
```

After syntactic analysis:



What is Parsing ?

- Scanning was about recognising words in a stream of characters

What is Parsing ?

- Scanning was about recognising words in a stream of characters
- Not all words exist in a given language

What is Parsing ?

- Scanning was about recognising words in a stream of characters
- Not all words exist in a given language
- But: a language is more than a collection of words

What is Parsing ?

- Scanning was about recognising words in a stream of characters
- Not all words exist in a given language
- But: a language is more than a collection of words
- A language is made up of sentences

What is Parsing ?

- Scanning was about recognising words in a stream of characters
- Not all words exist in a given language
- But: a language is more than a collection of words
- A language is made up of sentences
- A sentence is a sequence of words

What is Parsing ?

- Scanning was about recognising words in a stream of characters
- Not all words exist in a given language
- But: a language is more than a collection of words
- A language is made up of sentences
- A sentence is a sequence of words
- Not all sequences of words form proper sentences

What is Parsing ?

- Scanning was about recognising words in a stream of characters
- Not all words exist in a given language
- But: a language is more than a collection of words
- A language is made up of sentences
- A sentence is a sequence of words
- Not all sequences of words form proper sentences
- Parsing is about identifying legal sentences in a stream of words

Describing Syntax

Question:

- What are the legal sentences of a language ?

Describing Syntax

Question:

- What are the legal sentences of a language ?

Answer:

- Syntax is described using grammars !

Describing Syntax

Question:

- What are the legal sentences of a language ?

Answer:

- Syntax is described using grammars !

Example:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
 & | & \textbf{num} \\
\mathit{Op} & \Rightarrow & + \mid - \mid * \mid /
\end{array}
$$

Describing Syntax

Question:

- What are the legal sentences of a language ?

Answer:

- Syntax is described using grammars !

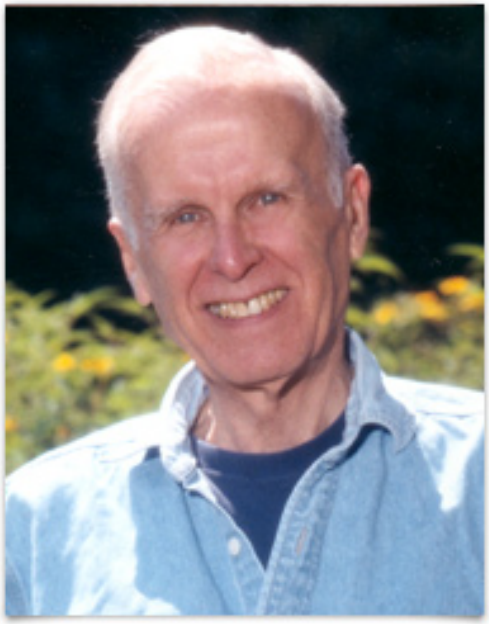
Example:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
 & | & \textbf{num} \\
\mathit{Op} & \Rightarrow & + \mid - \mid * \mid /
\end{array}
$$

This notation is called Backus-Naur Form

John Backus and Peter Naur

John Backus



- American computer scientist
- 1924–2007
- Columbia University
- (Co-)Inventor of FORTRAN
- 1977 Turing Award

Peter Naur

- Danish computer scientist
- 1928–2016
- University of Copenhagen
- (Co-)Inventor of ALGOL
- 2005 Turing Award

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr}\ \textit{Op}\ \textit{Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr}\ \textit{Op}\ \textit{Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

Expr

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr Op Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

Expr $\rightarrow$ Expr Op Expr

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr Op Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

Expr → Expr Op Expr
→ Expr Op Expr Op Expr

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr Op Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

$$\begin{aligned} \text{Expr} &\rightarrow \text{Expr Op Expr} \\ &\rightarrow \text{Expr Op Expr Op Expr} \\ &\rightarrow \text{num(2) Op Expr Op Expr} \end{aligned}$$

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr Op Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

$$\begin{aligned}
\text{Expr} &\rightarrow \text{Expr Op Expr} \\
&\rightarrow \text{Expr Op Expr Op Expr} \\
&\rightarrow \text{num(2) Op Expr Op Expr} \\
&\rightarrow \text{num(2) add Expr Op Expr}
\end{aligned}$$

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr Op Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

$$\begin{aligned}
\text{Expr} &\rightarrow \text{Expr Op Expr} \\
&\rightarrow \text{Expr Op Expr Op Expr} \\
&\rightarrow \text{num(2) Op Expr Op Expr} \\
&\rightarrow \text{num(2) add Expr Op Expr} \\
&\rightarrow \text{num(2) add num(3) Op Expr}
\end{aligned}$$

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr Op Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

$$\begin{aligned}
\text{Expr} &\rightarrow \text{Expr Op Expr} \\
&\rightarrow \text{Expr Op Expr Op Expr} \\
&\rightarrow \text{num(2) Op Expr Op Expr} \\
&\rightarrow \text{num(2) add Expr Op Expr} \\
&\rightarrow \text{num(2) add num(3) Op Expr} \\
&\rightarrow \text{num(2) add num(3) mul Expr}
\end{aligned}$$

Backus-Naur Form

Production rules describe possible derivations of legal sentences from a dedicated start symbol:

$$\textit{Expr} \Rightarrow \textit{Expr Op Expr} \mid \textbf{num}$$

$$\textit{Op} \Rightarrow + \mid - \mid * \mid /$$

Example: 2+3*4:

Expr → Expr Op Expr
→ Expr Op Expr Op Expr
→ num(2) Op Expr Op Expr
→ num(2) add Expr Op Expr
→ num(2) add num(3) Op Expr
→ num(2) add num(3) mul Expr
→ num(2) add num(3) mul num(4)

Original Backus-Naur Form

Problem:

- Origins of Backus-Naur form in the 50s and 60s
- No “nice fonts” available

Original Backus-Naur Form

Problem:

- Origins of Backus-Naur form in the 50s and 60s
- No "nice fonts" available

Example reloaded:

```
<Expr> ::= <Expr> <Op> <Expr> | num
<Op>   ::= + | - | * | /
```

- Non-terminals marked by angular brackets
- Terminals as plain text
- "::=" instead of arrow
- Bar does exist on US keyboards
- Still frequently found notations

Recap: Context-Free Grammars

Definition:

A context-free grammar is a 4-tuple (T, N, S, P) where

Recap: Context-Free Grammars

Definition:

A context-free grammar is a 4-tuple (T, N, S, P) where

- T is a non-empty set of terminal symbols
  - words of the language
  - recognised by scanner

Recap: Context-Free Grammars

Definition:

A context-free grammar is a 4-tuple (T, N, S, P) where

- T is a non-empty set of terminal symbols
  - words of the language
  - recognised by scanner
- N is a non-empty set of non-terminal symbols
  - disjoint from T
  - syntactic variables used to structure the grammar

Recap: Context-Free Grammars

Definition:

A context-free grammar is a 4-tuple (T, N, S, P) where

- T is a non-empty set of terminal symbols
  - words of the language
  - recognised by scanner
- N is a non-empty set of non-terminal symbols
  - disjoint from T
  - syntactic variables used to structure the grammar
- $S \in N$ is a designated start symbol
  - Language contains all sentences derivable from S

Recap: Context-Free Grammars

Definition:

A context-free grammar is a 4-tuple (T, N, S, P) where

- T is a non-empty set of terminal symbols
  - words of the language
  - recognised by scanner
- N is a non-empty set of non-terminal symbols
  - disjoint from T
  - syntactic variables used to structure the grammar
- $S \in N$ is a designated start symbol
  - Language contains all sentences derivable from S
- $P : N \to (T \cup N)^*$ is a set of productions
  - Derivation or rewrite rules
  - Single non-terminal symbol on left-hand side ensures context-freedom
  - But: multiple derivations of same non-terminal symbol allowed

Context-Free Grammars vs Backus-Naur Form

Relationship:

Backus-Naur form is the computer scientist's way to describe context-free grammars:

- Terminals and non-terminals distinguished by notation
- Start symbol implicitly taken as left hand side of first production rule
- Productions as in context-free grammars

From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

Expr	$\Rightarrow$	*Expr Op Expr* | **num**
Op	$\Rightarrow$	+ | - | * | /

From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

Expr	$\Rightarrow$	*Expr Op Expr* $\mid$ **num**
Op	$\Rightarrow$	+ $\mid$ - $\mid$ * $\mid$ /

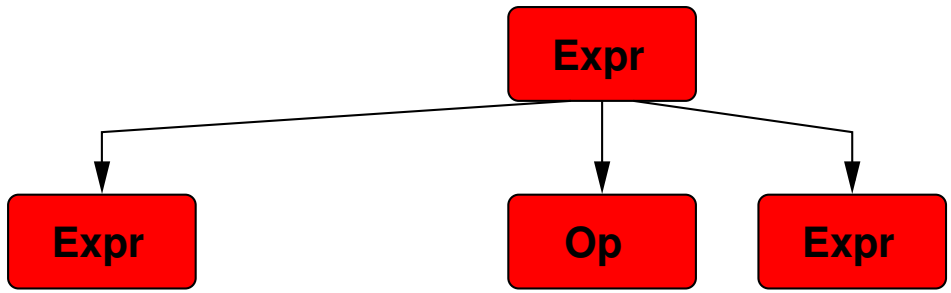
From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



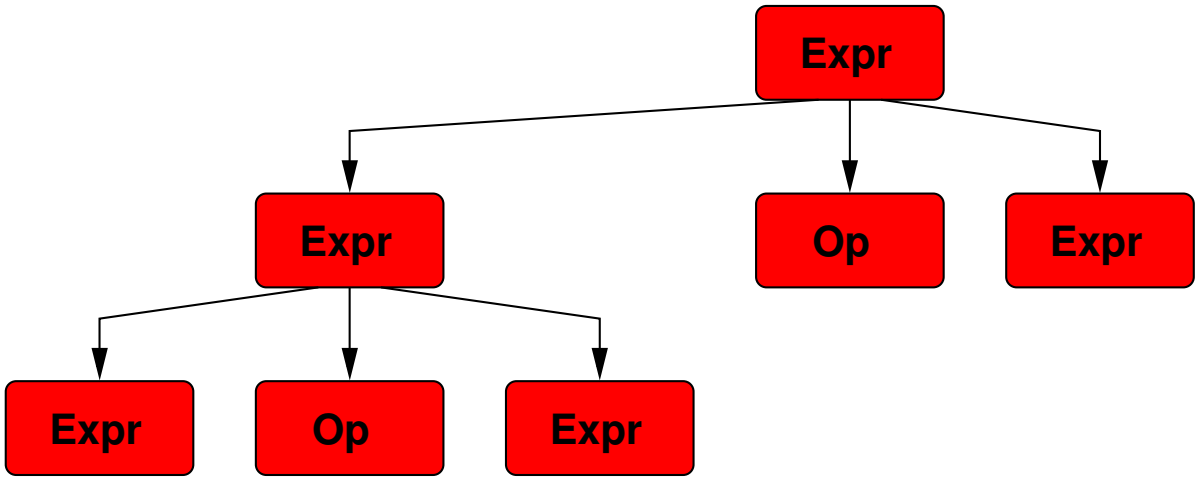
From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



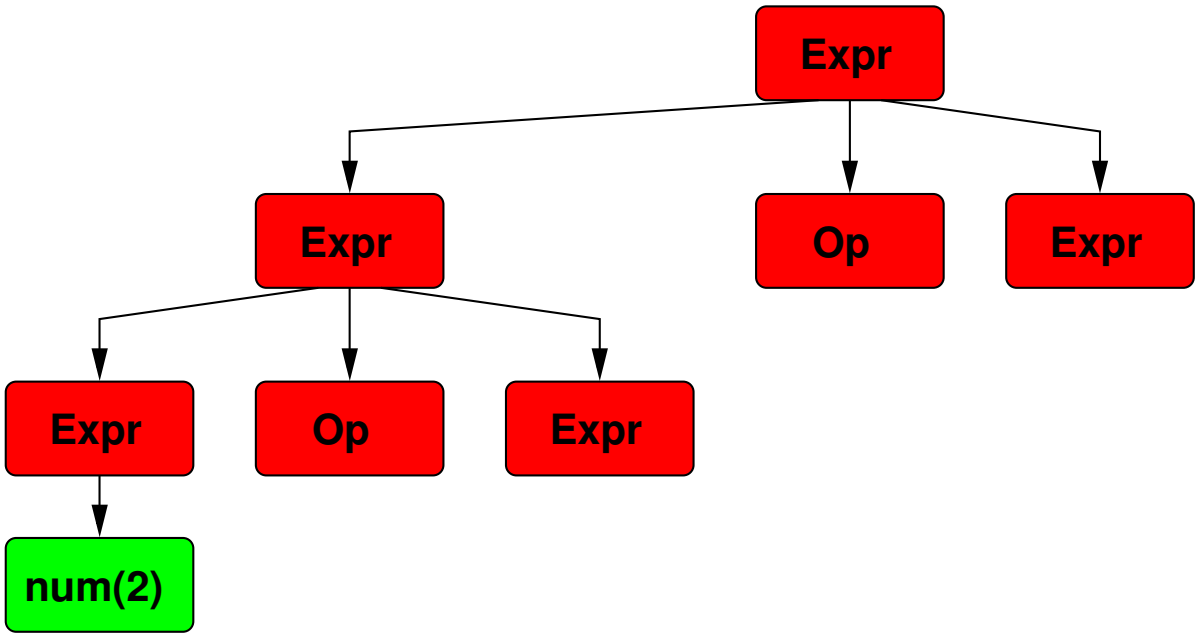
From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



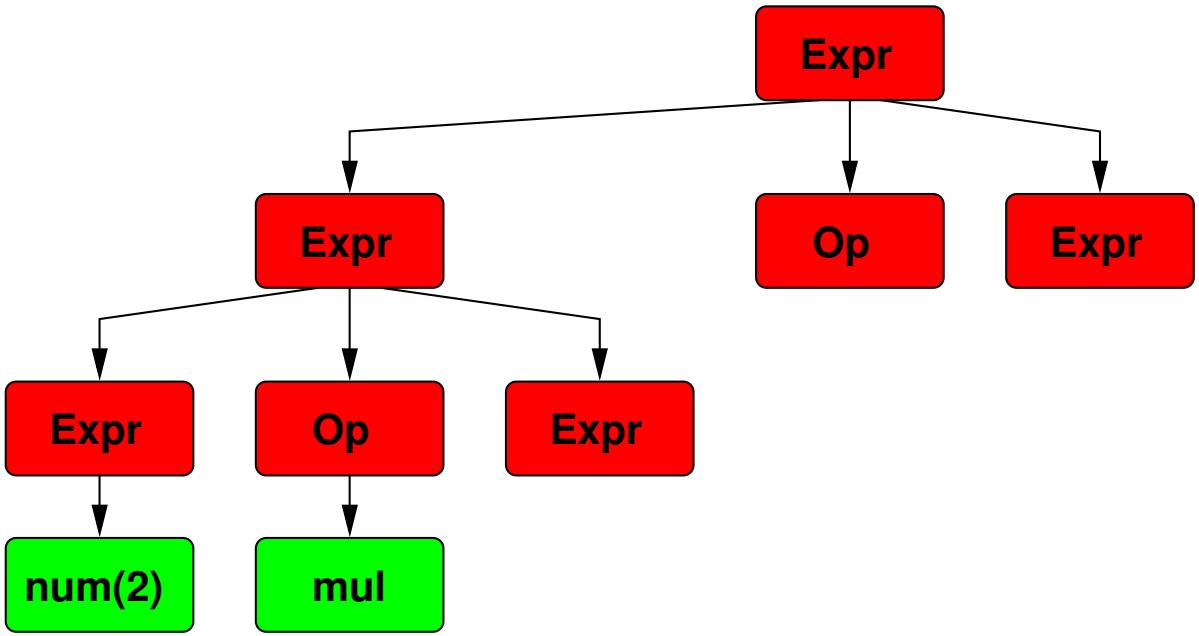
From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



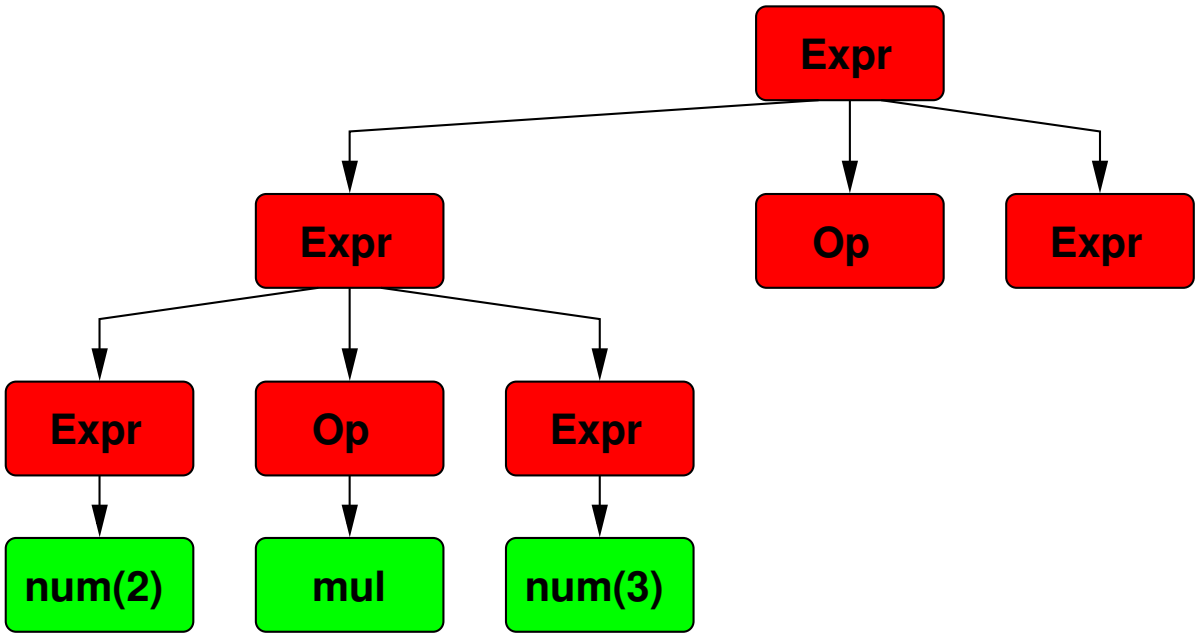
From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



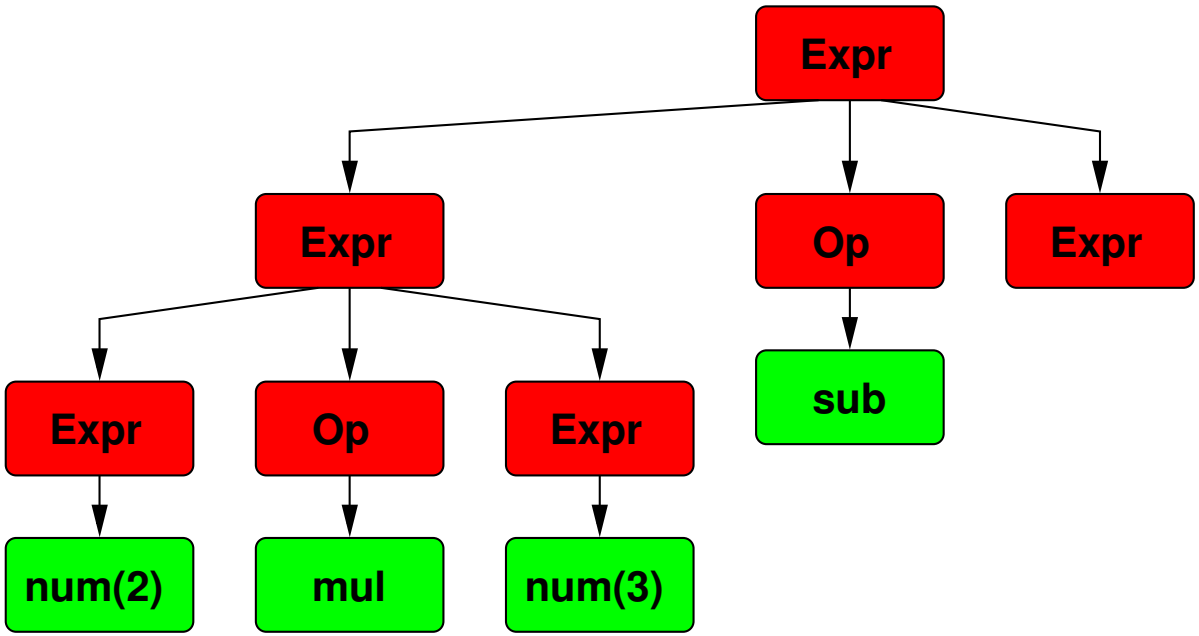
From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



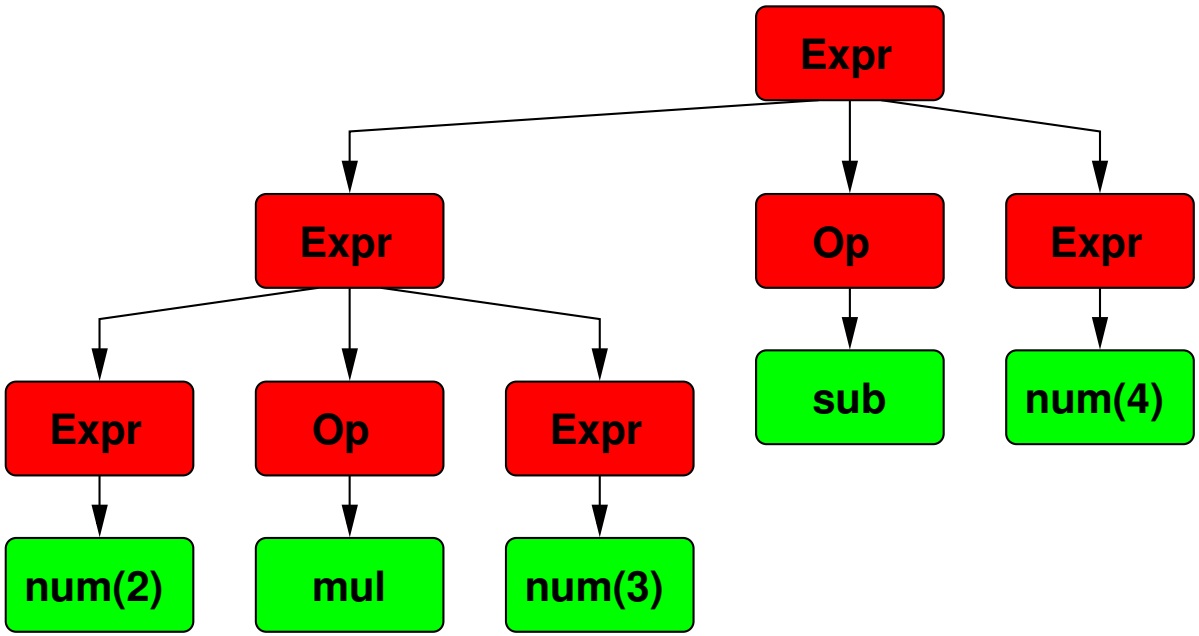
From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \textbf{num} \\
\mathit{Op} & \Rightarrow & + \mid - \mid * \mid /
\end{array}
$$

Exploitation:

- Each derivation yields a new node in the abstract syntax tree.
- The choice of derivation rule determines the kind of node.

From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \textbf{num} \\
\mathit{Op} & \Rightarrow & + \mid - \mid * \mid /
\end{array}
$$

Exploitation:

- Each derivation yields a new node in the abstract syntax tree.
- The choice of derivation rule determines the kind of node.

Derivation order:

- We always derived the left-most non-terminal symbol !

From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \textbf{num} \\ \mathit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Exploitation:

- Each derivation yields a new node in the abstract syntax tree.
- The choice of derivation rule determines the kind of node.

Derivation order:

- We always derived the left-most non-terminal symbol !
- What if we choose the right-most non-terminal instead ?

From Token Sequence to Syntax Tree

Observation:

- Recursive derivation of productions creates a tree !

Example: 2*3-4:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Exploitation:

- Each derivation yields a new node in the abstract syntax tree.
- The choice of derivation rule determines the kind of node.

Derivation order:

- We always derived the left-most non-terminal symbol !
- What if we choose the right-most non-terminal instead ?
- **Can the derivation strategy affect the parse tree ?**

Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

Expr	$\Rightarrow$	*Expr Op Expr* $\mid$ **num**
Op	$\Rightarrow$	+ $\mid$ - $\mid$ * $\mid$ /

Can the derivation strategy affect the parse tree ?

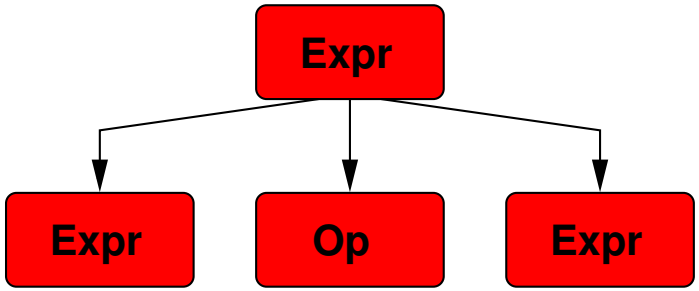
Example: 2*3-4:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \mathbf{num} \\
\mathit{Op} & \Rightarrow & + \mid - \mid * \mid /
\end{array}
$$

Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

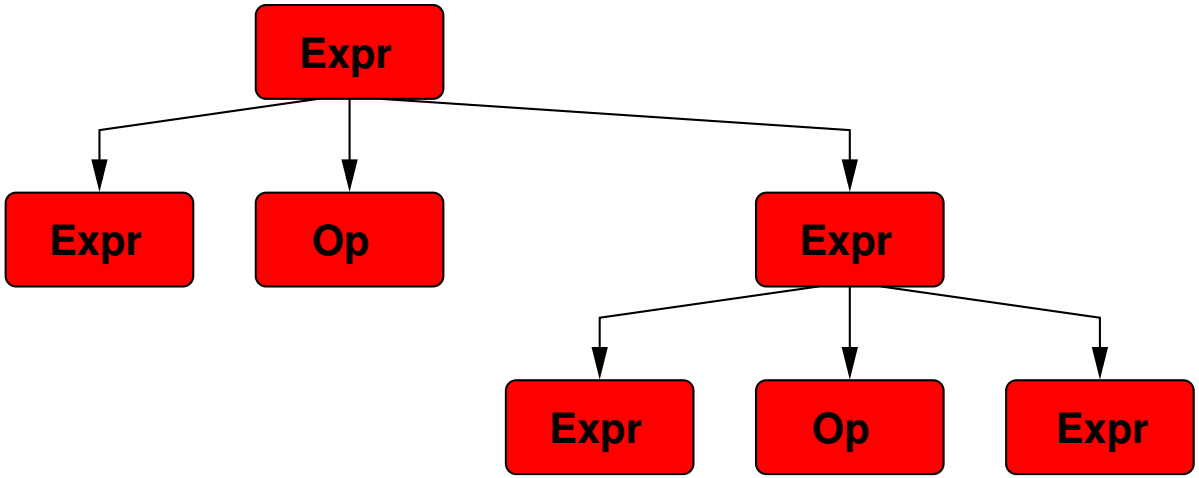
$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

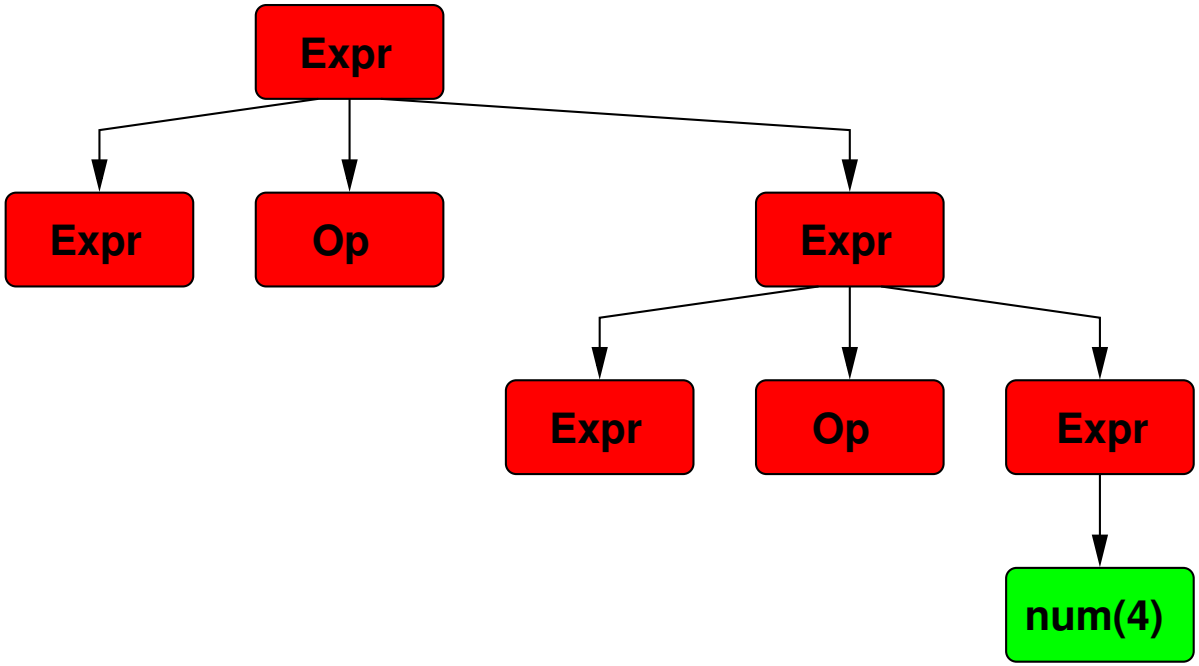
$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

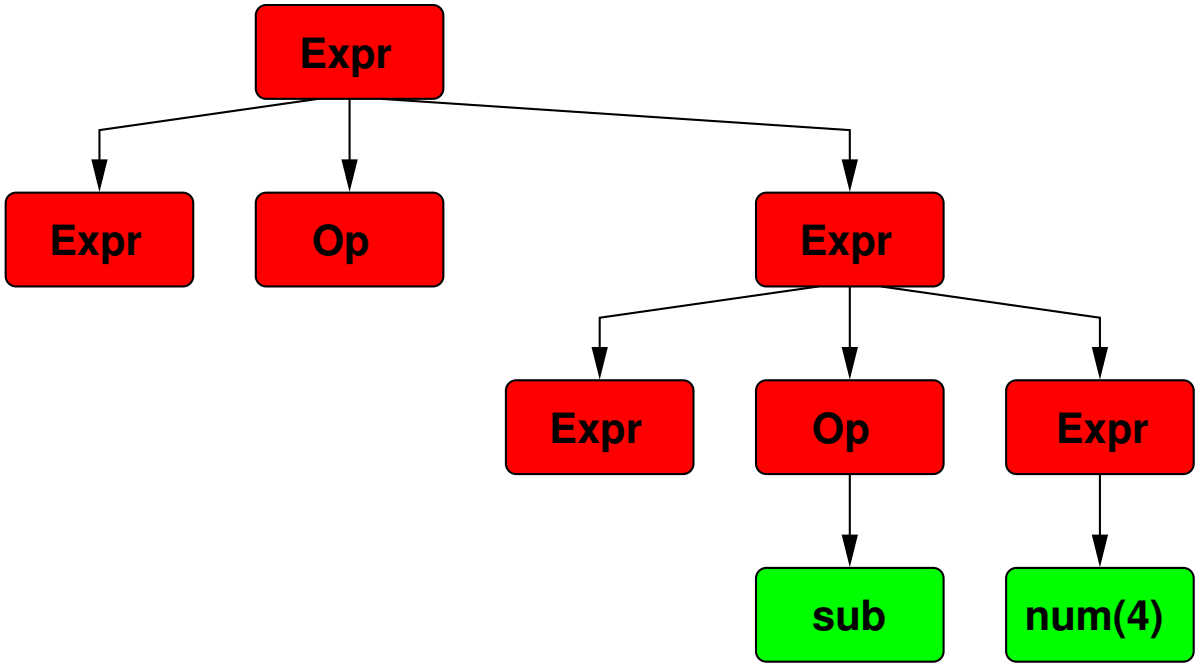
$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

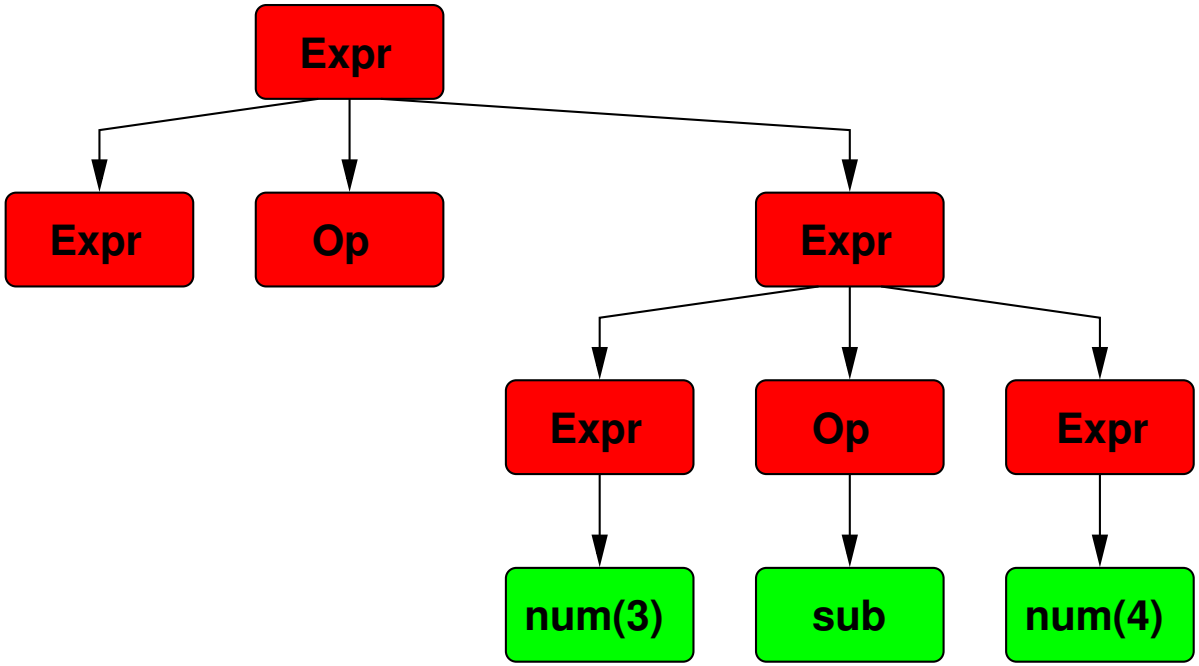
$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

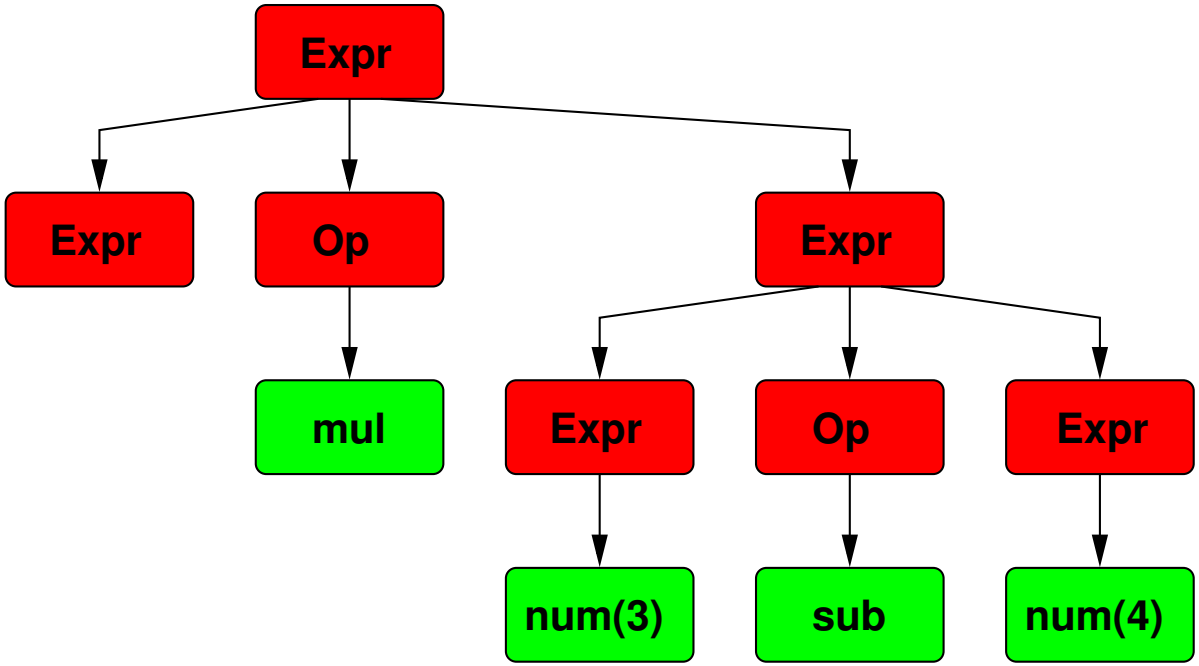
$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

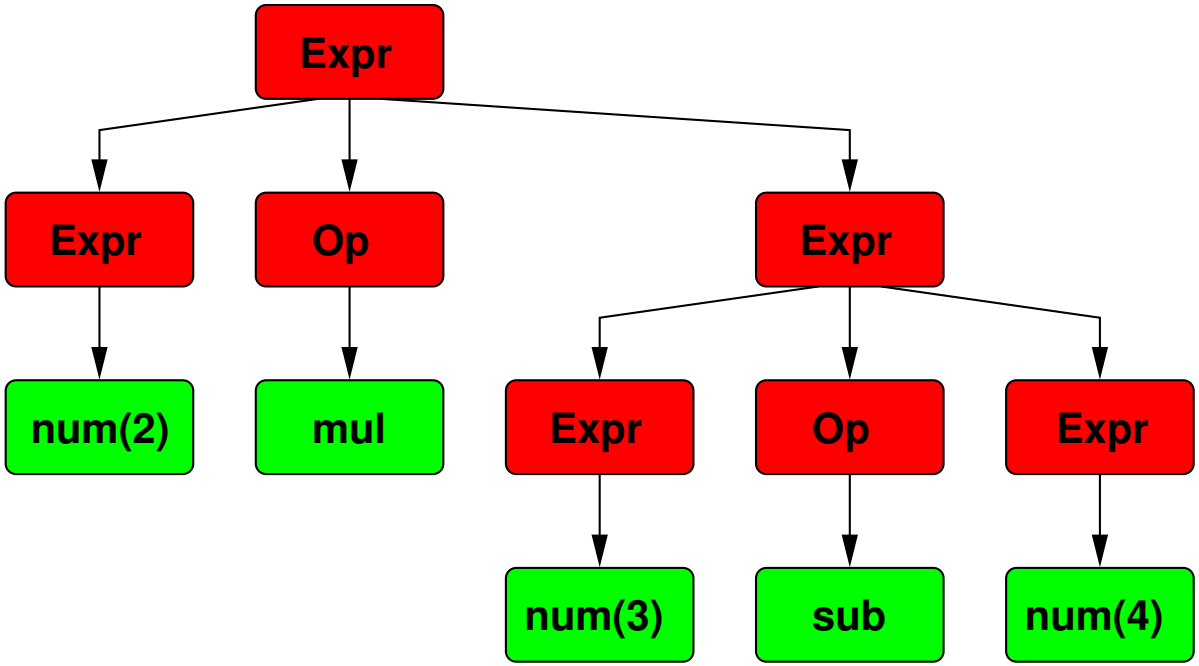
$$\begin{array}{lcl} \mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \mathbf{num} \\ \mathit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



Can the derivation strategy affect the parse tree ?

Example: 2*3-4:

$$\begin{array}{lcl} \mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \mathbf{num} \\ \mathit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$



Derivation Order and Ambiguity

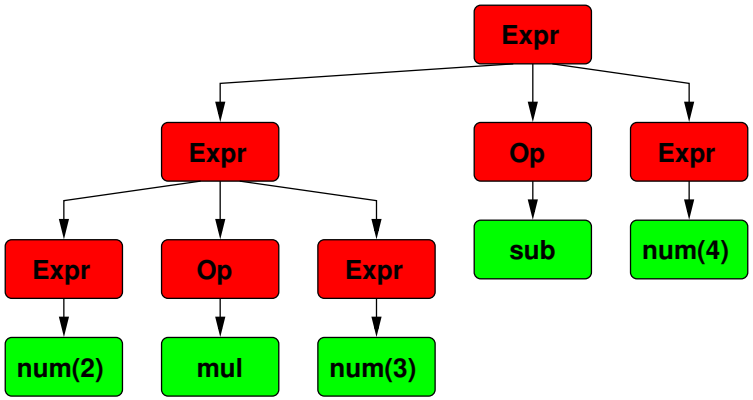
Observation:

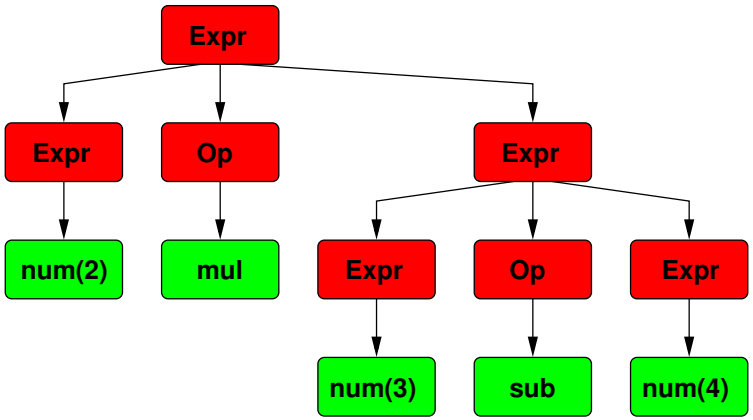
- Different derivations may indeed lead to different parse trees !!

Example:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Derivation of 2*3-4: leftmost / rightmost:



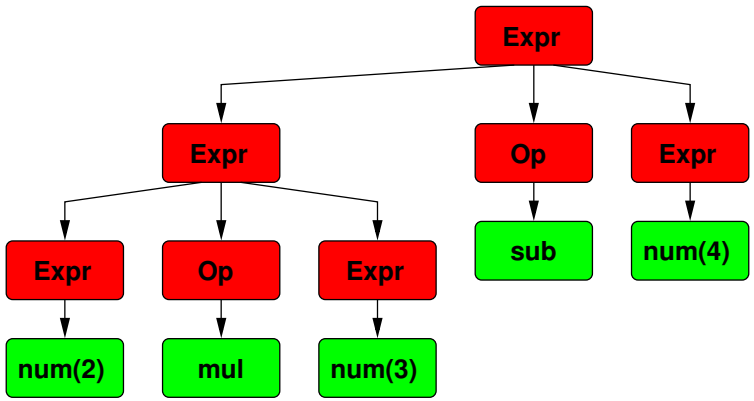


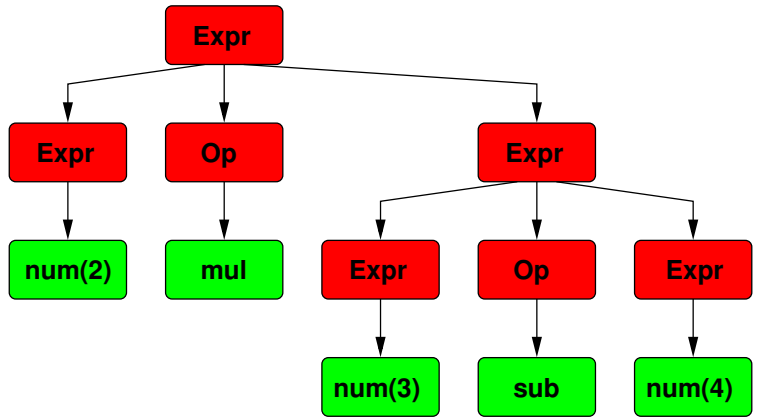
Derivation Order and Ambiguity

Example:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \mathbf{num} \\
\mathit{Op} & \Rightarrow & + \mid - \mid * \mid /
\end{array}
$$

Derivation of 2*3-4: leftmost / rightmost:





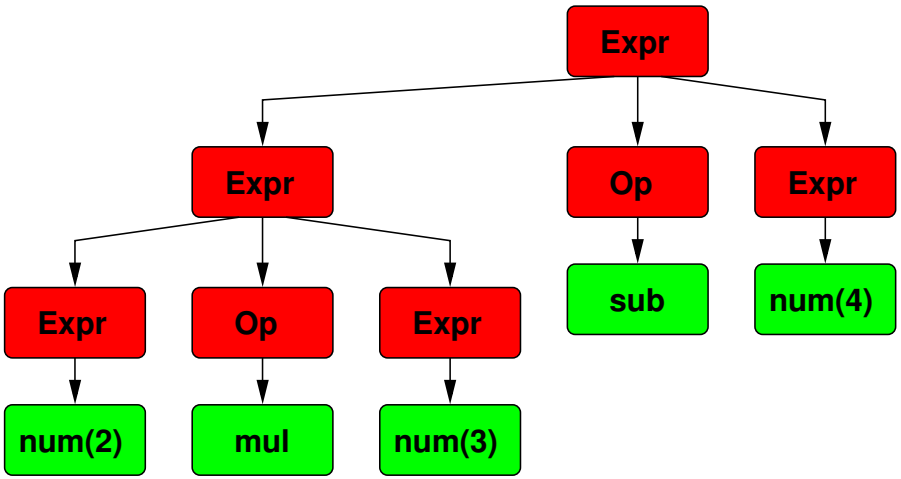
Which derivation order is correct ?

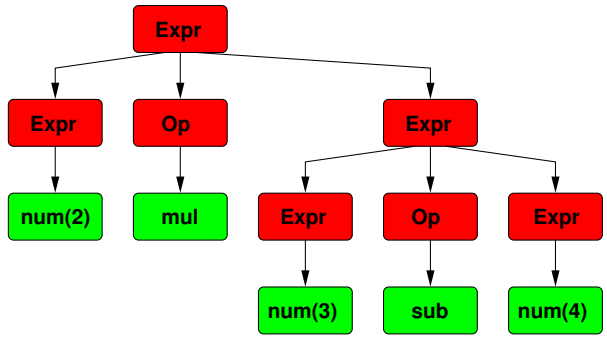
Derivation Order and Ambiguity

Example:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Derivation of 2*3-4:





Left-most derivation !

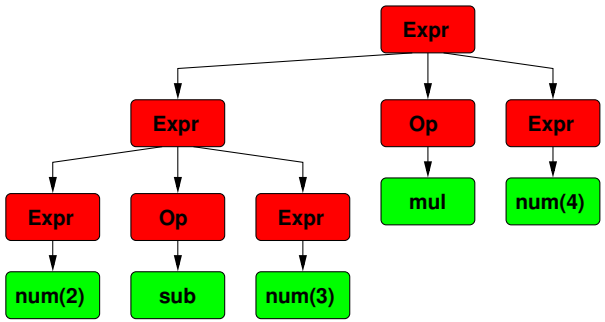
Higher precedence of multiplication !!

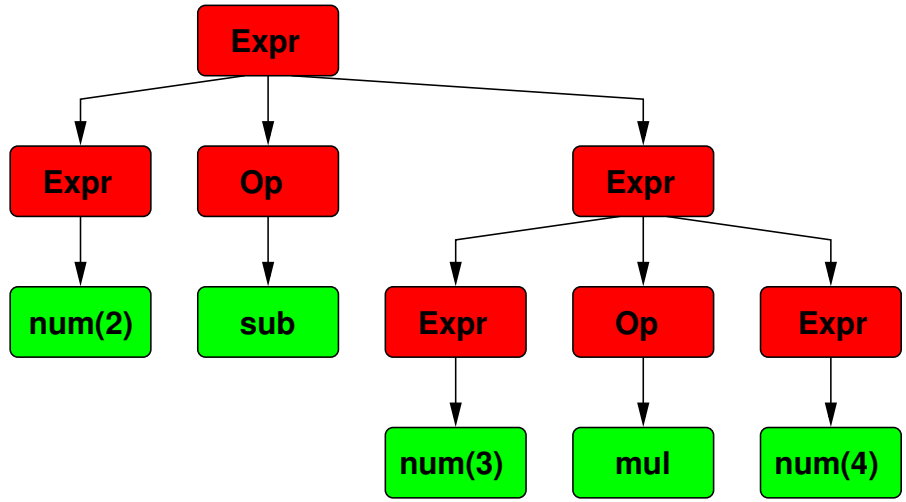
Derivation Order and Ambiguity

Example:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Derivation of 2-3*4:





Right-most derivation !

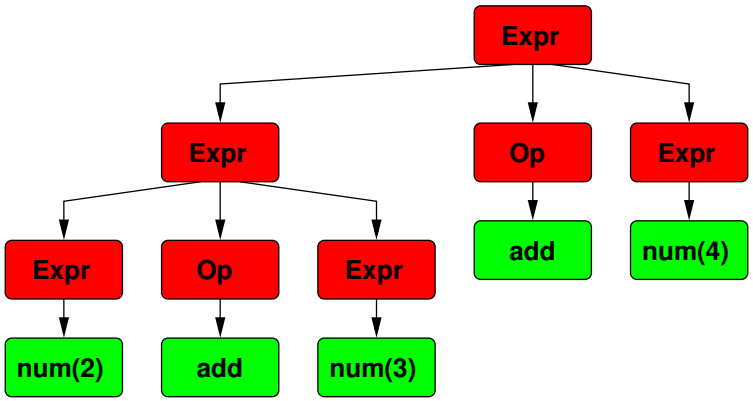
Higher precedence of multiplication !!

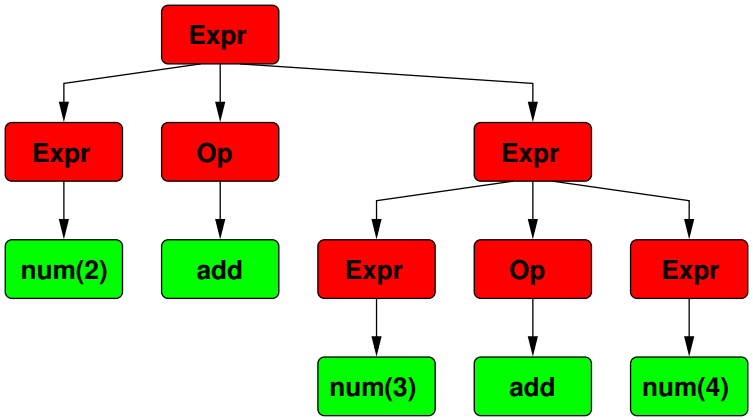
Derivation Order and Ambiguity

Example:

$$\begin{array}{lcl} \mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \textbf{num} \\ \mathit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Derivation of 2+3+4:





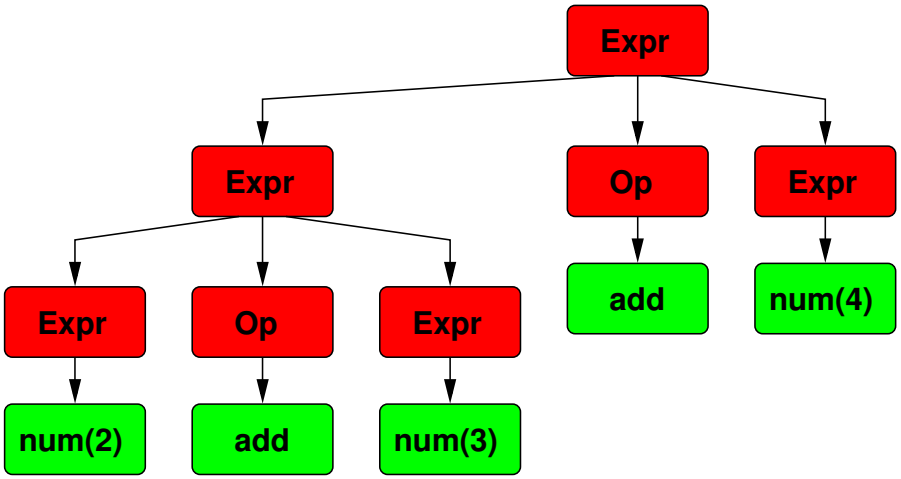
Which derivation order is correct ?

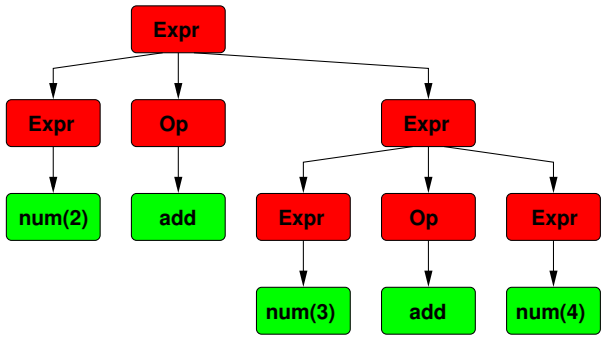
Derivation Order and Ambiguity

Example:

$$\begin{array}{lcl} \mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \mathbf{num} \\ \mathit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Derivation of 2+3+4:





Left-most derivation !

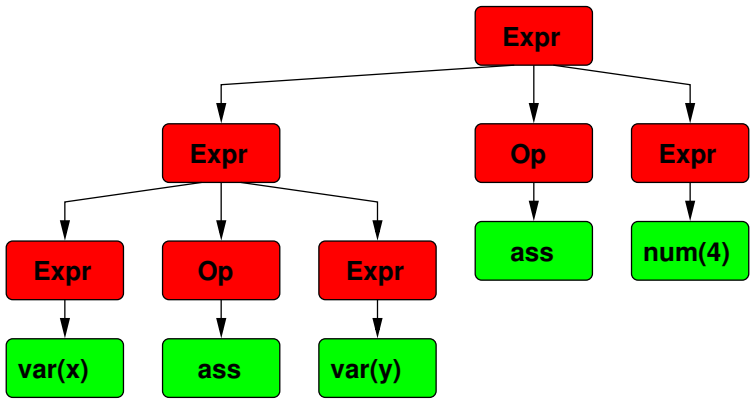
Left associativity of addition !!

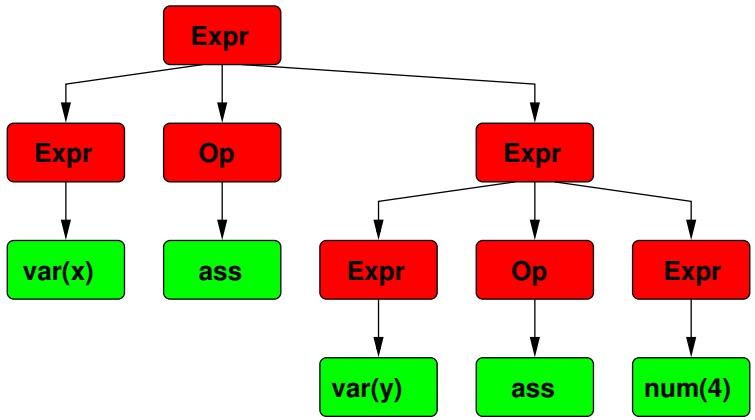
Derivation Order and Ambiguity

Example:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \textbf{num} \mid \textbf{var} \\
\mathit{Op} & \Rightarrow & + \mid - \mid * \mid / \mid =
\end{array}
$$

Derivation of x=y=4:





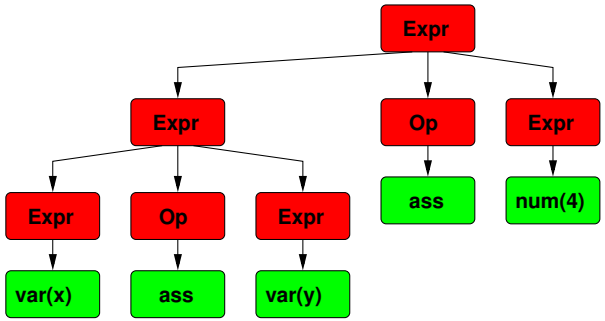
Which derivation order is correct ?

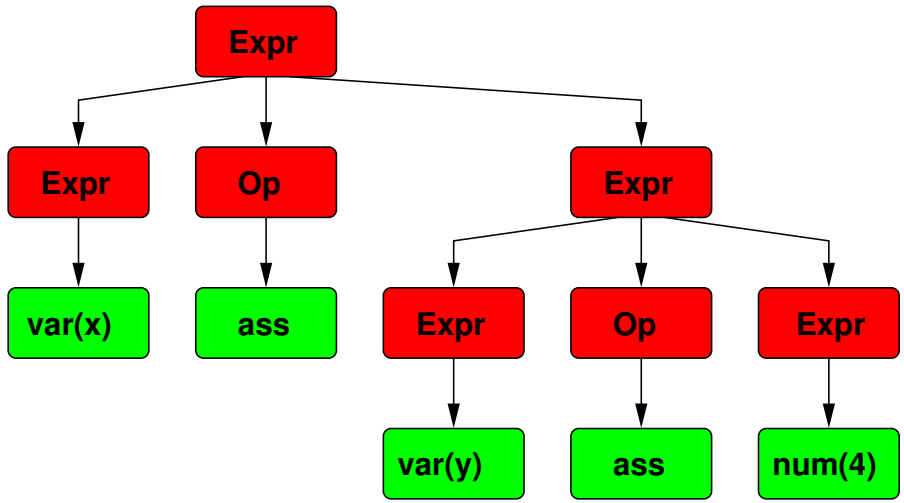
Derivation Order and Ambiguity

Example:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr Op Expr} \mid \textbf{num} \mid \textbf{var} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \mid = \end{array}$$

Derivation of x=y=4:





Right-most derivation !

Right associativity of assignment !!

Derivation Order and Ambiguity

Definiton:

A context-free grammar is ambiguous if and only if there exists a sentence in L(G) that has multiple leftmost (or rightmost) derivations.

Consequences:

- Ambiguous grammars are not suitable for parsing !
- Ambiguous grammars must be rewritten !

Derivation Order and Ambiguity

Definiton:

A context-free grammar is ambiguous if and only if there exists a sentence in L(G) that has multiple leftmost (or rightmost) derivations.

Consequences:

- Ambiguous grammars are not suitable for parsing !
- Ambiguous grammars must be rewritten !

Problem:

$$\begin{array}{lcl} \mathit{Expr} & \Rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \textbf{num} \\ \mathit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Our grammar neither encodes associativity nor precedence !!

Operator Associativity

Encoding left associativity into a grammar:

Expr	$\Rightarrow$	*Expr Op* **num** $\mid$ **num**
Op	$\Rightarrow$	+ $\mid$ - $\mid$ * $\mid$ /

Idea:

- Make first production asymmetric
- Force left-most derivation

Operator Associativity

Encoding left associativity into a grammar:

$$
\begin{array}{lcl}
\textit{Expr} & \Rightarrow & \textit{Expr}\ \textit{Op}\ \textbf{num} \mid \textbf{num} \\
\textit{Op} & \Rightarrow & + \mid - \mid * \mid /
\end{array}
$$

Idea:

- Make first production asymmetric
- Force left-most derivation

Easy:

- All arithmetic operators are left-associative
- No need to categorise operators

Operator Associativity

Encoding left associativity into a grammar:

$$\begin{array}{lcl} \textit{Expr} & \Rightarrow & \textit{Expr}\ \textit{Op}\ \textbf{num} \mid \textbf{num} \\ \textit{Op} & \Rightarrow & + \mid - \mid * \mid / \end{array}$$

Idea:

- Make first production asymmetric
- Force left-most derivation

Easy:

- All arithmetic operators are left-associative
- No need to categorise operators

Encoding right associativity:

- Works exactly the same way:

$$\textit{Expr} \Rightarrow \textbf{num}\ \textit{Op}\ \textit{Expr} \mid \textbf{num}$$

Precedence

Encoding precedence into a grammar:

$$
\begin{array}{lcl}
\textit{Expr} & \Rightarrow & \textit{Expr OpLowPrec Term} \\
 & | & \textit{Term} \\
 & & \\
\textit{Term} & \Rightarrow & \textit{Term OpHighPrec } \textbf{num} \\
 & | & \textbf{num} \\
 & & \\
\textit{OpLowPrec} & \Rightarrow & + \mid - \\
 & & \\
\textit{OpHighPrec} & \Rightarrow & * \mid /
\end{array}
$$

Idea:

- Have one non-terminal symbol per precedence level
- Start with derivation of low precedence expressions
- Switch to derivation of high precedence expressions as needed
- Distinguish operator classes
- Keep the same trick for associativity encoding

Precedence

Adding parentheses:

Expr	$\Rightarrow$	*Expr OpLowPrec Term*
	|	*Term*
Term	$\Rightarrow$	*Term OpHighPrec Factor*
	|	*Factor*
Factor	$\Rightarrow$	(*Expr*)
	|	**num**
OpLowPrec	$\Rightarrow$	+ | -
OpHighPrec	$\Rightarrow$	* | /

Precedence

Even more precedence levels:

- Relational operators (lower)

Precedence

Even more precedence levels:

- Relational operators (lower)
- Logic operators (even lower)

Precedence

Even more precedence levels:

- Relational operators (lower)
- Logic operators (even lower)
- Array subscripts (higher)

Precedence

Even more precedence levels:

- Relational operators (lower)
- Logic operators (even lower)
- Array subscripts (higher)
- Type casts (higher)

Precedence

Even more precedence levels:

- Relational operators (lower)
- Logic operators (even lower)
- Array subscripts (higher)
- Type casts (higher)

Life is complicated:

- C has 13 precedence levels !

Precedence

Even more precedence levels:

- Relational operators (lower)
- Logic operators (even lower)
- Array subscripts (higher)
- Type casts (higher)

Life is complicated:

- C has 13 precedence levels !

Implementation strategy:

- Introduce one non-terminal per precedence level

Back to Parsing

Given:

- A list of tokens generated by the scanner
- Example: num(2) mul num(3) sub num(4)
- An unambiguous context-free grammar

Wanted:

- Unique, unambiguous derivation
- Unique, unambiguous parse tree
- Or a (good) error message

Alternative Approaches: Top-down vs Bottom-up Parsing

Top-down Parsing:

- Begin with root of parse tree
- Grow tree towards leaves
- At each step, select one non-terminal on the lower fringe
- Extend parse tree downward from that node

Bottom-up Parsing:

- Begin with leaves of parse tree
- Grow the tree towards the root
- At each step, add new node that
  - abstracts from lower nodes or leaves
  - extends partial parse tree upwards

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly
- Continue until the left most symbol on the fringe is terminal

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly
- Continue until the left most symbol on the fringe is terminal
- Compare terminal symbol with token on input stream

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly
- Continue until the left most symbol on the fringe is terminal
- Compare terminal symbol with token on input stream
- Match: consume token and continue with next symbol on fringe

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly
- Continue until the left most symbol on the fringe is terminal
- Compare terminal symbol with token on input stream
- Match: consume token and continue with next symbol on fringe
- No match: backtrack to previous choice point

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly
- Continue until the left most symbol on the fringe is terminal
- Compare terminal symbol with token on input stream
- Match: consume token and continue with next symbol on fringe
- No match: backtrack to previous choice point
- Continue until no non-terminals on the fringe and token sequence exhausted: **Success !!**

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly
- Continue until the left most symbol on the fringe is terminal
- Compare terminal symbol with token on input stream
- Match: consume token and continue with next symbol on fringe
- No match: backtrack to previous choice point
- Continue until no non-terminals on the fringe and token sequence exhausted: **Success !!**
- Only one condition met: **Failure !!**

Top-Down Parsing: Algorithmic Idea

- Begin with root of parse tree: start non-terminal
- Choose a derivation of the left-most non-terminal
- Extend the parse tree accordingly
- Continue until the left most symbol on the fringe is terminal
- Compare terminal symbol with token on input stream
- Match: consume token and continue with next symbol on fringe
- No match: backtrack to previous choice point
- Continue until no non-terminals on the fringe and token sequence exhausted: **Success !!**
- Only one condition met: **Failure !!**
- Still tokens on input stream: **Failure !!**

Top-Down Parsing: Algorithm

$word \leftarrow nextWord()$
$tree \leftarrow start\_symbol$
$node \leftarrow start\_symbol$
loop forever
  if $node \in T \wedge node \equiv word$ *then*
    $node \leftarrow nextNode()$
    $word \leftarrow nextWord()$
  else if $node \in T \wedge node \not\equiv word$ *then*
    backtrack
  else if $node \in N$ *then*
    $deriv \leftarrow selectDerivation(node)$
    $tree \leftarrow extendTree(node, deriv)$
    $node \leftarrow leftMostSymbol(deriv)$
  if $node = \epsilon \wedge word = eof$ *then*
    return success
  if $node = \epsilon \wedge word \neq eof$ *then*
    return failure

Top-Down Parsing: Example

Expression grammar:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Expr} \;+\; \mathit{Term} \\
 & | & \mathit{Expr} \;-\; \mathit{Term} \\
 & | & \mathit{Term} \\
 & & \\
\mathit{Term} & \Rightarrow & \mathit{Term} \;*\; \textbf{num} \\
 & | & \mathit{Term} \;/\; \textbf{num} \\
 & | & \textbf{num}
\end{array}
$$

Expression:

2*3+4

Token stream:

num(2) $\lhd$ mul $\lhd$ num(3) $\lhd$ add $\lhd$ num(4)

Top-Down Parsing: Trouble Shooting

Problem:

- Left-recursive grammars

Top-Down Parsing: Trouble Shooting

Problem:

- Left-recursive grammars

Description:

- Productions of the form: $\mathit{Foo} \rightarrow \mathit{Foo}\ \alpha \mid \beta$
- Possibly with indirections
- Infinite expansion of parse tree
- Problem: only "wrong" token causes backtracking

Top-Down Parsing: Trouble Shooting

Problem:

- Left-recursive grammars

Description:

- Productions of the form: $\mathit{Foo} \rightarrow \mathit{Foo}\ \alpha \mid \beta$
- Possibly with indirections
- Infinite expansion of parse tree
- Problem: only "wrong" token causes backtracking

Solution:

- Every left-recursive grammar can be automatically transformed into equivalent right-recursive grammar.

Top-Down Parsing: Trouble Shooting

Problem:

- Left-recursive grammars

Description:

- Productions of the form: $\mathit{Foo} \rightarrow \mathit{Foo}\ \alpha \mid \beta$
- Possibly with indirections
- Infinite expansion of parse tree
- Problem: only "wrong" token causes backtracking

Solution:

- Every left-recursive grammar can be automatically transformed into equivalent right-recursive grammar.
- Immediately left-recursive productions:
  $\mathit{Foo} \rightarrow \beta\ \mathit{Foo\prime} \quad , \quad \mathit{Foo\prime} \rightarrow \alpha\ \mathit{Foo\prime} \mid \epsilon$

Top-Down Parsing: Trouble Shooting

Problem:

- Left-recursive grammars

Description:

- Productions of the form: $\mathit{Foo} \rightarrow \mathit{Foo}\ \alpha \mid \beta$
- Possibly with indirections
- Infinite expansion of parse tree
- Problem: only "wrong" token causes backtracking

Solution:

- Every left-recursive grammar can be automatically transformed into equivalent right-recursive grammar.
- Immediately left-recursive productions:
  $\mathit{Foo} \rightarrow \beta\ \mathit{Foo\prime} \quad , \quad \mathit{Foo\prime} \rightarrow \alpha\ \mathit{Foo\prime} \mid \epsilon$
- Indirectly left-recursive productions:
  See literature

Top-Down Parsing: Trouble Shooting

Problem:

- Selection of derivations

Top-Down Parsing: Trouble Shooting

Problem:

- Selection of derivations

Description:

- No hint for choosing "right" derivation
- Backtracking leads to exhaustive search in large design space
- Inefficient

Top-Down Parsing: Trouble Shooting

Problem:

- Selection of derivations

Description:

- No hint for choosing "right" derivation
- Backtracking leads to exhaustive search in large design space
- Inefficient

Solution:

- Right-recursive grammars very much reduce the need for backtracking.

Top-Down Parsing: Trouble Shooting

Example: right-recursive expression grammar:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' & \Rightarrow & + \ \mathit{Term}\ \mathit{Expr}' \\
 & | & - \ \mathit{Term}\ \mathit{Expr}' \\
 & | & \epsilon \\
 & & \\
\mathit{Term} & \Rightarrow & \textbf{num}\ \ \mathit{Term}' \\
\mathit{Term}' & \Rightarrow & * \ \ \textbf{num}\ \ \mathit{Term}' \\
 & | & / \ \ \textbf{num}\ \ \mathit{Term}' \\
 & | & \epsilon
\end{array}
$$

Top-Down Parsing: Trouble Shooting

Example: right-recursive expression grammar:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' & \Rightarrow & + \ \mathit{Term}\ \mathit{Expr}' \\
 & | & - \ \mathit{Term}\ \mathit{Expr}' \\
 & | & \epsilon \\
 & & \\
\mathit{Term} & \Rightarrow & \textbf{num}\ \ \mathit{Term}' \\
\mathit{Term}' & \Rightarrow & * \ \ \textbf{num}\ \ \mathit{Term}' \\
 & | & / \ \ \textbf{num}\ \ \mathit{Term}' \\
 & | & \epsilon
\end{array}
$$

Observation:

- Derivation can be uniquely selected by inspection of next token on input stream ($\rightarrow$ no backtracking).

Top-Down Parsing: Trouble Shooting

Example: right-recursive expression grammar:

$$
\begin{array}{lcl}
\mathit{Expr} & \Rightarrow & \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' & \Rightarrow & + \ \mathit{Term}\ \mathit{Expr}' \\
 & | & - \ \mathit{Term}\ \mathit{Expr}' \\
 & | & \epsilon \\
 & & \\
\mathit{Term} & \Rightarrow & \textbf{num}\ \ \mathit{Term}' \\
\mathit{Term}' & \Rightarrow & * \ \ \textbf{num}\ \ \mathit{Term}' \\
 & | & / \ \ \textbf{num}\ \ \mathit{Term}' \\
 & | & \epsilon
\end{array}
$$

Observation:

- Derivation can be uniquely selected by inspection of next token on input stream ($\rightarrow$ no backtracking).
- Many right-recursive grammars can be transformed into predictive grammars ($\rightarrow$ left-factoring).

Recursive Descent Parsing

Automating parser generation:

- From a predictive, right-recursive grammar we can easily deduce a top-down recursive-descent parser.

Recursive Descent Parsing

Automating parser generation:

- From a predictive, right-recursive grammar we can easily deduce a top-down recursive-descent parser.

Approach:

- Turn the grammar into a start-separated grammar.
  $\rightarrow$ Add a fresh start symbol that does not appear in any derivation:

Recursive Descent Parsing

Automating parser generation:

- From a predictive, right-recursive grammar we can easily deduce a top-down recursive-descent parser.

Approach:

- Turn the grammar into a start-separated grammar.
  $\rightarrow$ Add a fresh start symbol that does not appear in any derivation:

Example:

$$
\begin{array}{lcl}
\mathit{Start} & \Rightarrow & \mathit{Expr} \\
\mathit{Expr} & \Rightarrow & \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' & \Rightarrow & +\ \mathit{Term}\ \mathit{Expr}' \mid -\ \mathit{Term}\ \mathit{Expr}' \mid \epsilon \\
\mathit{Term} & \Rightarrow & \textbf{num}\ \ \mathit{Term}' \\
\mathit{Term}' & \Rightarrow & *\ \ \textbf{num}\ \ \mathit{Term}' \mid /\ \ \textbf{num}\ \ \mathit{Term}' \mid \epsilon
\end{array}
$$

Recursive Descent Parsing

Example:

$$
\begin{array}{lcl}
\mathit{Start} & \Rightarrow & \mathit{Expr} \\
\mathit{Expr} & \Rightarrow & \mathit{Term}\ \mathit{Expr}' \\
\mathit{Expr}' & \Rightarrow & +\ \mathit{Term}\ \mathit{Expr}' \mid -\ \mathit{Term}\ \mathit{Expr}' \mid \epsilon \\
\mathit{Term} & \Rightarrow & \textbf{num}\ \ \mathit{Term}' \\
\mathit{Term}' & \Rightarrow & *\ \ \textbf{num}\ \ \mathit{Term}' \mid /\ \ \textbf{num}\ \ \mathit{Term}' \mid \epsilon
\end{array}
$$

Automating parser generation:

For each non-terminal symbol generate a function that

- selects the proper derivation based on the prefix of the input stream,
- calls the corresponding function for each non-terminal in the derivation,
- checks the token stream for each terminal in the derivation,
- combines all these checks with logical conjunction.

Recursive Descent Parsing: Example

```
Start() {
  return Expr() && (nextToken() == eof);
}

Expr() {
  return Term() && ExprP();
}

ExprP() {
  token = nextToken();
  switch (token)
    case add: return Term() && ExprP();
    case sub: return Term() && ExprP();
    default:  ungetToken( token);
              return true;  // due to epsilon production
                            // otherwise, we would
                            // return false
}
```

Recursive Descent Parsing: Example (Continued)

```
Term() {
  return (nextToken() == num) && TermP();
}

TermP() {
  switch (nextToken())
    case mul: return (nextToken() == num) && TermP();
    case div: return (nextToken() == num) && TermP();
    default:  ungetToken( token);
              return true;  // due to epsilon production
                            // otherwise, we would
                            // return false
}
```

Scanner-Parser Integration

Theory:

- First, scanner generates whole sequence of tokens.
- Second, parser processes sequence of tokens.

Scanner-Parser Integration

Theory:

- First, scanner generates whole sequence of tokens.
- Second, parser processes sequence of tokens.

Practice:

- Scanner integrated into parser.
- Parser asks for next token on demand.

Scanner-Parser Integration

Theory:

- First, scanner generates whole sequence of tokens.
- Second, parser processes sequence of tokens.

Practice:

- Scanner integrated into parser.
- Parser asks for next token on demand.

Motivation:

- No explicit representation for token sequence.
- No intermediate data structure.
- Quick detection of syntax errors.

Top-Down Parsing Wrap-up

Recursive descent parsers recognise LL(k) languages:

- L: input read left to right
- L: construct leftmost derivation
- k: inspecting k input tokens for disambiguation

Top-Down Parsing Wrap-up

Recursive descent parsers recognise LL(k) languages:

- L: input read left to right
- L: construct leftmost derivation
- k: inspecting k input tokens for disambiguation

Recursive descent parsers:

- Can be table-driven
- Can be direct-encoded
- Support good error reporting
- But not popular for automatic parser generation

Automatic Parser Generation with YACC

Wanted:

- Auto-generate parsers from context-free grammars !

Automatic Parser Generation with YACC

Wanted:

- Auto-generate parsers from context-free grammars !

Tools:

- yacc — standard Unix tool
  ("Yet Another Compiler Compiler")
- bison — GNU version of yacc
- Many equivalent tools for other source languages and operating systems

Automatic Parser Generation with YACC

Wanted:

- Auto-generate parsers from context-free grammars !

Tools:

- yacc — standard Unix tool
  ("Yet Another Compiler Compiler")
- bison — GNU version of yacc
- Many equivalent tools for other source languages and operating systems

Parsing strategy:

- Bottom-up

Using YACC

Usage of parser generator:

- Source file: `foo.y`

Using YACC

Usage of parser generator:

- Source file: `foo.y`
- Invocation: `bison foo.y`

Using YACC

Usage of parser generator:

- Source file: `foo.y`
- Invocation: `bison foo.y`
- Yields two files:
  - source file `y.tab.c`
  - header file `y.tab.h`

Using YACC

Usage of parser generator:

- Source file: `foo.y`
- Invocation: `bison foo.y`
- Yields two files:
  - source file `y.tab.c`
  - header file `y.tab.h`
- C compiler creates executable parser

Using YACC

Usage of parser generator:

- Source file: `foo.y`
- Invocation: `bison foo.y`
- Yields two files:
  - source file `y.tab.c`
  - header file `y.tab.h`
- C compiler creates executable parser
- (f)lex gets token specification

Using YACC

Usage of generated parser:

- Parse function:
  - `void yyparse(void) // starts parser`

Using YACC

Usage of generated parser:

- Parse function:
  - `void yyparse(void) // starts parser`
- Global variables:
  - `FILE *yyin // stream to read from`

Using YACC

Usage of generated parser:

- Parse function:
  - `void yyparse(void) // starts parser`
- Global variables:
  - `FILE *yyin // stream to read from`
- Parse tree result:
  - user defined

YACC Specification

YACC File:

```
%{
   /* Any C code here, mostly declarations */
%}
yacc definitions
%%
parse rules
%%
/* Any C code here, e.g. functions for error handling */
```

YACC Specification

YACC File:

```
%{
   /* Any C code here, mostly declarations */
%}
yacc definitions
%%
non-terminal: derivation { action }
            | derivation { action }
            ;
%%
/* Any C code here, e.g. functions for error handling */
```

YACC Definitions

Data type definition:

- Defines values associated with scanned tokens / terminals
- Refers to C types
  - standard C types
  - user-defined C types from C section of yacc file

Example:

```
%union {
 nodetype   mynodetype;
 char*      myid;
 int        myint;
 float      myfloat;
 binop      mybinop;
 node*      mynode;
}
```

YACC Definitions

Definition of terminal symbols:

- Used both by yacc and lex
- Terminals/tokens without type/value
- Terminals/tokens with type/value from data type definition

Example:

```
%token BRACKET_L BRACKET_R COMMA SEMICOLON
%token MINUS PLUS STAR SLASH PERCENT
%token LE LT GE GT EQ NE OR AND
%token TRUEVAL FALSEVAL LET

%token <myint> NUM
%token <myfloat> FLOAT
%token <myid> ID
```

YACC Definitions

Definition of non-terminal symbols:

- Non-terminals without value/type
- Non-terminals with type/value from data type definition

Example:

```
%type <mynode>  intval floatval boolval constant expr
%type <mynode>  instrs instr assign vardef program
%type <mybinop> binop
```

YACC Definitions

Associativity:

- Operator associativity can be declared to yacc
- Simplifies production rules

Example:

```
%right LET
%left ADD SUB
```

Illustration:

```
a = (b = (c = 42));
d = (a + b) + c;
```

YACC Definitions

Precedence:

- Operator precedence can be declared to yacc
- Each associativity declaration defines a precedence level
- Low precedence to high precedence
- Special rule for non-associative operators

Example:

```
%right LET
%left ADD SUB
%left MUL DIV
%nonassoc FOO
```

YACC Definitions

Definition of start symbol:

- Start symbol must be declared as non-terminal.
- Optional definition
- Default: first production

Example:

```
%start program
```

YACC Production Rules

Simple syntax checker:

- Only production rules
- Literally identical with Backus-Naur-form
- Terminals and non-terminals from declaration section

Example:

```
expr: constant
    | ID
    | BRACKET_L expr binop expr BRACKET_R
    ;
```

YACC Production Rules

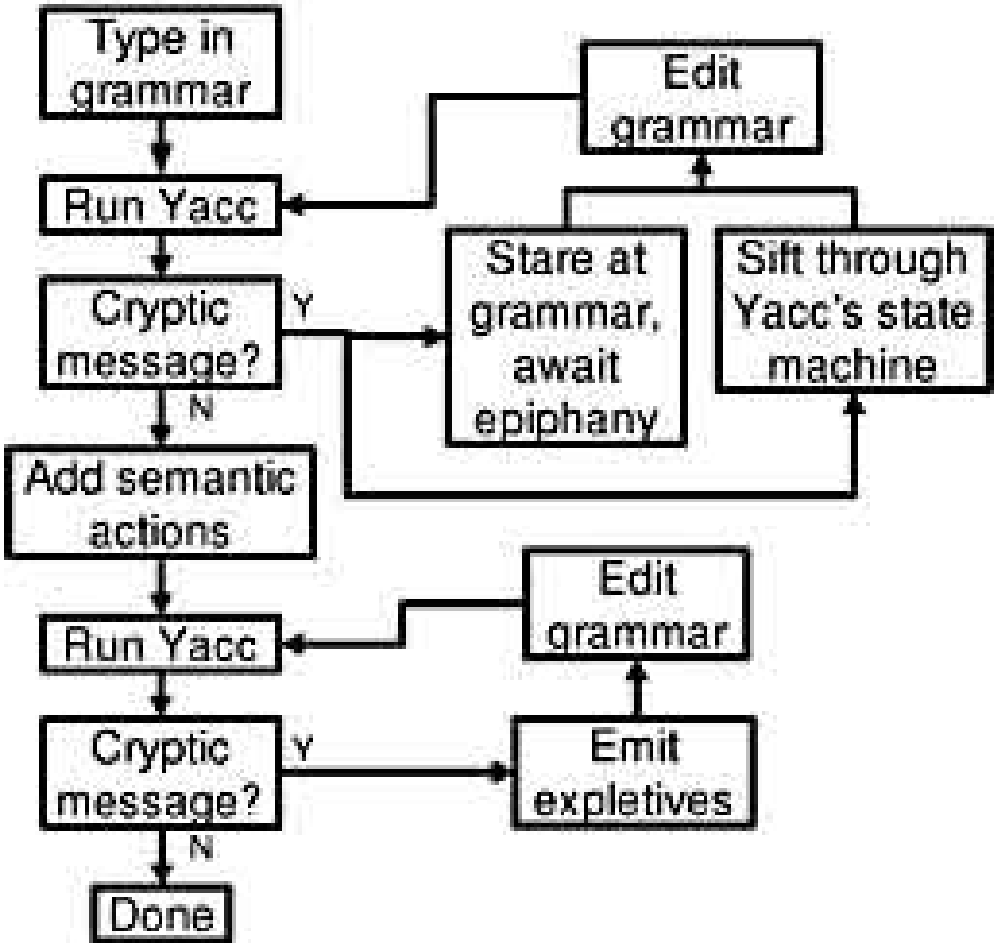
Proper IR generator:

- Associates each derivation with an action (plain C code)
- Allows for construction of IR
- $-variables provide access to values of terminals and non-terminals
- $$ denotes value of non-terminal defined by production

Example:

```
expr: constant {
        $$ = $1;
      }
    | ID {
        $$ = TBmakeVar( $1);
      }
    | BRACKET_L expr binop expr BRACKET_R {
        $$ = TBmakeBinop( $3, $2, $4);
      }
    ;
```

Having Fun with YACC



Some Hints on Using YACC in Practice

Shift/reduce and reduce/reduce conflicts:

- A good yacc parser has **ZERO** conflicts !
- Conflicts are often hard to find.
- Conflicts are sometimes hard to understand.
- More about conflicts after bottom-up parsing

How to get there:

- Change the yacc file in small (tiny) steps.
- Resolve conflicts as they arise.
- Never continue extending the parser if you still have conflicts
- Read the conflict resolution tutorial on Blackboard

Back to Theory: Top-down vs Bottom-up Parsing

Top-down Parsing:

- Begin with root of parse tree
- Grow tree towards leaves
- At each step, select one non-terminal on the lower fringe
- Extend parse tree downward from that node

Bottom-up Parsing:

- Begin with leaves of parse tree
- Grow the tree towards the root
- At each step, add new node that
  - abstracts from lower nodes or leaves
  - extends partial parse tree upwards

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream
- Push token onto stack

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream
- Push token onto stack
- Check if the top-most stack entries occur on the right hand side of any production

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream
- Push token onto stack
- Check if the top-most stack entries occur on the right hand side of any production
- If yes:
  - pop those elements from the stack
  - push left hand side of production onto stack
  - Extend partial parse tree

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream
- Push token onto stack
- Check if the top-most stack entries occur on the right hand side of any production
- If yes:
  - pop those elements from the stack
  - push left hand side of production onto stack
  - Extend partial parse tree
- If no:
  - read next token from input stream
  - push token onto stack

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream
- Push token onto stack
- Check if the top-most stack entries occur on the right hand side of any production
- If yes:
  - pop those elements from the stack
  - push left hand side of production onto stack
  - Extend partial parse tree
- If no:
  - read next token from input stream
  - push token onto stack
- Continue until end-of-file reached

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream
- Push token onto stack
- Check if the top-most stack entries occur on the right hand side of any production
- If yes:
  - pop those elements from the stack
  - push left hand side of production onto stack
  - Extend partial parse tree
- If no:
  - read next token from input stream
  - push token onto stack
- Continue until end-of-file reached
- If stack contains only start symbol: Success !

Back to Theory: Bottom-Up Parsing

Algorithmic idea:

- Read token from input stream
- Push token onto stack
- Check if the top-most stack entries occur on the right hand side of any production
- If yes:
  - pop those elements from the stack
  - push left hand side of production onto stack
  - Extend partial parse tree
- If no:
  - read next token from input stream
  - push token onto stack
- Continue until end-of-file reached
- If stack contains only start symbol: Success !
- Otherwise: Failure !

Bottom-Up Parsing: Algorithm

Algorithm:

$word \leftarrow nextWord()$
$stack \leftarrow \emptyset$
loop forever
  if top of stack matches β *for some production* $A \rightarrow \beta$
    $pop(\ stack, |\ \beta\ |)$
    $push(\ stack, A)$
  else if $word \neq eof$
    $push(\ stack, word)$
    $word \leftarrow nextWord()$
  else if $stack = start\_symbol$
    return success
  else
    return failure

Bottom-Up Parsing: Example

Expression grammar:

$$
\begin{array}{lcl}
\textit{Expr} & \Rightarrow & \textit{Expr} \ + \ \textit{Term} \\
 & | & \textit{Expr} \ - \ \textit{Term} \\
 & | & \textit{Term} \\
 & & \\
\textit{Term} & \Rightarrow & \textit{Term} \ * \ \textbf{num} \\
 & | & \textit{Term} \ / \ \textbf{num} \\
 & | & \textbf{num}
\end{array}
$$

Expression:

2−3∗4

Token stream:

num(2) ◁ sub ◁ num(3) ◁ mul ◁ num(4)

Bottom-Up Parsing

Terminology:

- Technique is also named shift-reduce parsing.
- Shift word from input to stack.
- Reduce handle on stack.

Bottom-Up Parsing

Terminology:

- Technique is also named shift-reduce parsing.
- Shift word from input to stack.
- Reduce handle on stack.

Notorious for shift-reduce parsing:

- Shift/Reduce Conflicts
- Reduce/Reduce Conflicts

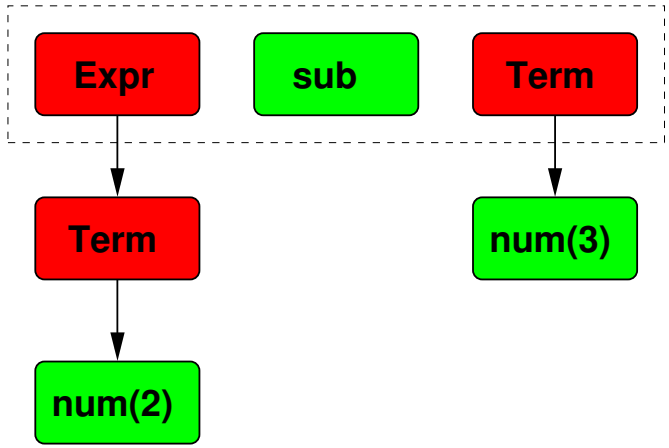
Choosing Handles

Observation:

- Efficient selection of handles is critical.
- In the previous example we used some magic to choose the correct handle.

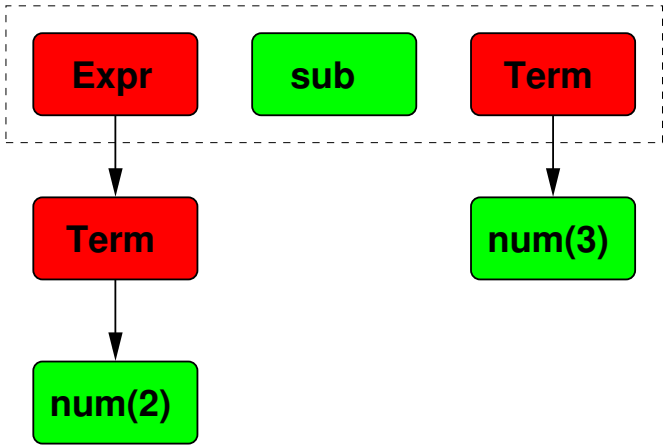
Finding Handles: Example

Magic step:



Finding Handles: Example

Magic step:



Potential Handles:

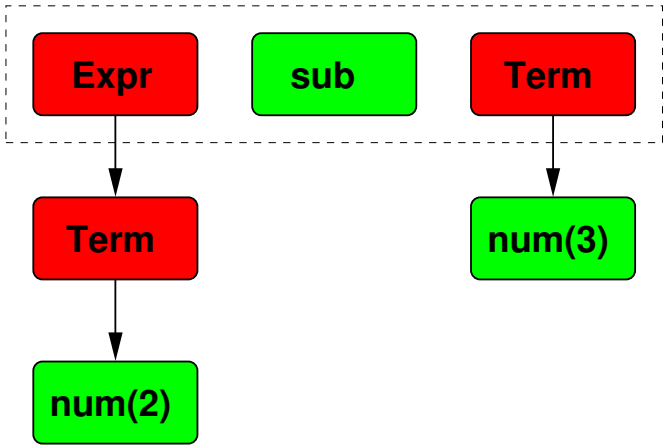
$Expr \rightarrow Expr\ sub\ Term$

$Term \rightarrow Term\ mul\ num$

$Term \rightarrow Term\ div\ num$

Finding Handles: Example

Magic step:



Potential Handles:

$Expr \rightarrow Expr\ sub\ Term$

$Term \rightarrow Term\ mul\ num$

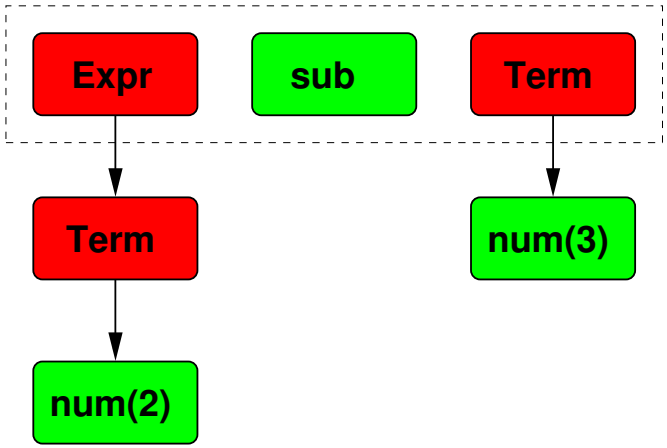
$Term \rightarrow Term\ div\ num$

Token stream:

$\text{num}(2) \lhd \text{sub} \lhd \text{num}(3) \lhd \text{mul} \lhd \text{num}(4)$

Finding Handles: Example

Magic step:



Potential Handles:

$Expr \rightarrow Expr\ sub\ Term$

$Term \rightarrow Term\ mul\ num$

$Term \rightarrow Term\ div\ num$

Token stream:

$\text{num}(2) \lhd \text{sub} \lhd \text{num}(3) \lhd \text{mul} \lhd \text{num}(4)$

Decision:

Only the second handle allows us to parse a 'mul' token

Table-driven Shift-Reduce Parsers

Action table:

- rows: states of handle selection DFA
- columns: terminal symbols
- elements: action to be taken
  - shift *state*
  - reduce *production*
  - accept
  - fail (empty field)

Example:

State	eof	add	sub	mul	div	num
0						s5
1	acc	s7	s8			
2	r4	r4	r4	s9	s10	
3	r7	r7	r7	r7	r7	
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Actions Revisited

Shift action:

- Format: shift *state*
- Shift lookahead word onto stack
- Shift *state* onto stack
- Scan new lookahead
- Continue with *state*

Actions Revisited

Shift action:

- Format: shift *state*
- Shift lookahead word onto stack
- Shift *state* onto stack
- Scan new lookahead
- Continue with *state*

Reduce action:

- Format: reduce *handle*
- Replace current handle by its non-terminal
- Discard all intermediate state
- Determine new state looking up state underneath handle and non-terminal in goto-table
- Push new state onto stack

Actions Revisited

Accept action:

- Format: accept
- Can only be reached with goal symbol on stack and eof as lookahead symbol

Actions Revisited

Accept action:

- Format: accept
- Can only be reached with goal symbol on stack and eof as lookahead symbol

Fail action:

- Format: fail
- Reached anytime the action table does not contain one of the above actions

Tables for Generated Parser

Goto table:

- rows: states of handle selection DFA
- columns: non-terminal symbols
- elements: follow-up state after reduce action

Example:

State	Expr	Term	Factor
0	1	2	3
1			
2			
3			
⋮	⋮	⋮	⋮

Driver for Generated Parser

$stack \leftarrow \emptyset$
$push(\ stack,\ NIL);\quad push(\ stack,\ 0)$
$word \leftarrow nextWord()$
loop forever
  $s \leftarrow top(\ stack)$
  if $Action[s, word]\ =\ shift\ i$
    $push(\ stack,\ word)$
    $push(\ stack,\ i)$
    $word \leftarrow nextWord()$
  else if $Action[s, word]\ =\ reduce\ A \rightarrow \beta$
    $pop(\ stack, 2\times \mid \beta \mid)$
    $s \leftarrow top(\ stack)$
    $push(\ stack,\ A)$
    $push(\ stack,\ Goto[s, A])$
  else if $Action[s, word]\ =\ accept$
    return success
  else return failure

Shift-Reduce Parsing: Example

Start-separated expression grammar:

1	*Goal*	$\Rightarrow$	*Expr*
2	*Expr*	$\Rightarrow$	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	$\Rightarrow$	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	$\Rightarrow$	(*Expr*)
9		|	**num**
10		|	**ident**

Example: Action and Goto Tables

State	eof	"+"	"-"	"*"	"/"	"("	")"	num	ident	Expr	Term	Factor
0						s4		s5	s6	1	2	3
1	acc	s7	s8									
2	r4	r4	r4	s9	s10							
3	r7	r7	r7	r7	r7							
4						s14		s15	s16	11	12	13
5	r9	r9	r9	r9	r9							
6	r10	r10	r10	r10	r10							
7						s4		s5	s6		17	3
8						s4		s5	s6		18	3
9						s4		s5	s6			19
10						s4		s5	s6			20
11		s21	s22				s23					
12		r4	r4	s24	s25		r4					
13		r7	r7	r7	r7		r7					
14						s14		s15	s16	26	12	13
15		r9	r9	r9	r9		r9					
16		r10	r10	r10	r10		r10					
17	r2	r2	r2	s9	s10							
18	r3	r3	r3	s9	s10							
19	r5	r5	r5	r5	r5							
20	r6	r6	r6	r6	r6							
21						s14		s15	s16		27	13
22						s14		s15	s16		28	13
23	r8	r8	r8	r8	r8							
24						s14		s15	s16			29
25						s14		s15	s16			30
26		s21	s22				s31					
27		r2	r2	s24	s25		r2					
28		r3	r3	s24	s25		r3					
29		r5	r5	r5	r5		r5					
30		r6	r6	r6	r6		r6					
31		r8	r8	r8	r8		r8					

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	reduce 7
9	mul	NIL 0 Expr 1 sub 8 Term 18	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	reduce 7
9	mul	NIL 0 Expr 1 sub 8 Term 18	shift 9
10	ident(y)	NIL 0 Expr 1 sub 8 Term 18 mul 9	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	reduce 7
9	mul	NIL 0 Expr 1 sub 8 Term 18	shift 9
10	ident(y)	NIL 0 Expr 1 sub 8 Term 18 mul 9	shift 6
11	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 ident 6	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	reduce 7
9	mul	NIL 0 Expr 1 sub 8 Term 18	shift 9
10	ident(y)	NIL 0 Expr 1 sub 8 Term 18 mul 9	shift 6
11	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 ident 6	reduce 10
12	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 Factor 19	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	reduce 7
9	mul	NIL 0 Expr 1 sub 8 Term 18	shift 9
10	ident(y)	NIL 0 Expr 1 sub 8 Term 18 mul 9	shift 6
11	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 ident 6	reduce 10
12	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 Factor 19	reduce 5
13	eof	NIL 0 Expr 1 sub 8 Term 18	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	reduce 7
9	mul	NIL 0 Expr 1 sub 8 Term 18	shift 9
10	ident(y)	NIL 0 Expr 1 sub 8 Term 18 mul 9	shift 6
11	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 ident 6	reduce 10
12	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 Factor 19	reduce 5
13	eof	NIL 0 Expr 1 sub 8 Term 18	reduce 3
14	eof	NIL 0 Expr 1	

Example Parser Execution: `x-2*y`

Step	Symbol	Stack	Action
1	ident(x)	NIL 0	shift 6
2	sub	NIL 0 ident 6	reduce 10
3	sub	NIL 0 Factor 3	reduce 7
4	sub	NIL 0 Term 2	reduce 4
5	sub	NIL 0 Expr 1	shift 8
6	num(2)	NIL 0 Expr 1 sub 8	shift 5
7	mul	NIL 0 Expr 1 sub 8 num 5	reduce 9
8	mul	NIL 0 Expr 1 sub 8 Factor 3	reduce 7
9	mul	NIL 0 Expr 1 sub 8 Term 18	shift 9
10	ident(y)	NIL 0 Expr 1 sub 8 Term 18 mul 9	shift 6
11	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 ident 6	reduce 10
12	eof	NIL 0 Expr 1 sub 8 Term 18 mul 9 Factor 19	reduce 5
13	eof	NIL 0 Expr 1 sub 8 Term 18	reduce 3
14	eof	NIL 0 Expr 1	**accept**

Parser Generation

Table Construction:

- Tedious, but systematic
- Can be automated
- YACC and co

How it really works:

- See literature: further reading

Shift-Reduce Conflicts

Characteristic:

- Grammar is defined such that with given lookahead parser cannot decide whether to shift or to reduce
- Typically, some decision needs to be made prematurely

Shift-Reduce Conflicts

Example:

$$
\begin{array}{lcl}
\textit{Program} & \Rightarrow & \textit{Globals Fundefs} \\
\textit{Globals} & \Rightarrow & \textit{Globals Global} \mid \epsilon \\
\textit{Global} & \Rightarrow & \textit{Type}\ \textbf{id}\ \ ; \\
\textit{Fundefs} & \Rightarrow & \textit{Fundefs Fundef} \mid \epsilon \\
\textit{FunDef} & \Rightarrow & \textit{FunHeader}\ \{\ \textit{FunBody}\ \} \\
\textit{FunHeader} & \Rightarrow & \textit{RetType}\ \textbf{id}\ \ (\ \textit{Params}\ ) \\
\textit{RetType} & \Rightarrow & \textbf{void} \mid \textit{Type} \\
\textit{Type} & \Rightarrow & \textbf{int} \mid \textbf{bool} \mid \textbf{float}
\end{array}
$$

Shift-Reduce Conflicts

Example:

Program	$\Rightarrow$	*Globals Fundefs*
Globals	$\Rightarrow$	*Globals Global* $\mid \epsilon$
Global	$\Rightarrow$	*Type* **id** **;**
Fundefs	$\Rightarrow$	*Fundefs Fundef* $\mid \epsilon$
FunDef	$\Rightarrow$	*FunHeader* $\{$ *FunBody* $\}$
FunHeader	$\Rightarrow$	*RetType* **id** (*Params*)
RetType	$\Rightarrow$	**void** $\mid$ *Type*
Type	$\Rightarrow$	**int** $\mid$ **bool** $\mid$ **float**

Problem:

- Token sequence: int $\lhd$ id $\lhd$ "(" $\lhd$ ...
- We shift 'int' and then reduce it to 'Type'
- Then, we could reduce 'Type' to 'RetType'
- But that prematurely opts for a function definition
- Prematurely because the lookahead is limited to 'Id'
- And that's still common for both globals and fundefs

Shift-Reduce Conflicts

What to do:

- YACC solution: shift
- Better: Rewrite grammar
- Use precedence declarations (if you know what you do)

Shift-Reduce Conflicts

Solution (almost):

$$
\begin{array}{lcl}
\mathit{Program} & \Rightarrow & \mathit{Globals}\ \mathit{FunDefs} \\
\mathit{Globals} & \Rightarrow & \mathit{Globals}\ \mathit{Global} \mid \epsilon \\
\mathit{Global} & \Rightarrow & \mathit{Type}\ \mathit{Id}\ \texttt{;} \\
\mathit{Fundefs} & \Rightarrow & \mathit{Fundefs}\ \mathit{Fundef} \mid \epsilon \\
\mathit{FunDef} & \Rightarrow & \mathit{FunHeader}\ \{\ \mathit{FunBody}\ \} \\
\mathit{FunHeader} & \Rightarrow & \mathit{Type}\ \mathit{Id}\ (\ [\mathit{Param}\ [\ ,\ \mathit{Param}\ ]^*\ ]\ ) \\
 & \mid & \textbf{void}\ \mathit{Id}\ (\ [\mathit{Param}\ [\ ,\ \mathit{Param}\ ]^*\ ]\ ) \\
\mathit{Type} & \Rightarrow & \textbf{int} \mid \textbf{bool} \mid \textbf{float}
\end{array}
$$

Idea:

- Avoid non-terminal RetType that enforces early decision between Fundef and Global

Shift-Reduce Conflicts

Solution (almost):

$$
\begin{array}{lcl}
\textit{Program} & \Rightarrow & \textit{Globals FunDefs} \\
\textit{Globals} & \Rightarrow & \textit{Globals Global} \mid \epsilon \\
\textit{Global} & \Rightarrow & \textit{Type Id} \ ; \\
\textit{Fundefs} & \Rightarrow & \textit{Fundefs Fundef} \mid \epsilon \\
\textit{FunDef} & \Rightarrow & \textit{FunHeader} \ \{ \ \textit{FunBody} \ \} \\
\textit{FunHeader} & \Rightarrow & \textit{Type Id} \ (\ [\textit{Param} \ [\ , \ \textit{Param} \ ]^* \ ] \ ) \\
 & \mid & \textbf{void} \ \textit{Id} \ (\ [\textit{Param} \ [\ , \ \textit{Param} \ ]^* \ ] \ ) \\
\textit{Type} & \Rightarrow & \textbf{int} \mid \textbf{bool} \mid \textbf{float}
\end{array}
$$

Idea:

- Avoid non-terminal RetType that enforces early decision between Fundef and Global

Problem:

- We can still not decide a-priori where globals end and fundefs starts
- Left-recursion of Globals rule is the show-stopper

Shift-Reduce Conflicts

Solution:

$$
\begin{array}{lcl}
\textit{Program} & \Rightarrow & \textit{Globals FunDefs} \\
\textit{Globals} & \Rightarrow & \textit{Global Globals} \mid \epsilon \\
\textit{Global} & \Rightarrow & \textit{Type Id} \ \texttt{;} \\
\textit{Fundefs} & \Rightarrow & \textit{Fundefs Fundef} \mid \epsilon \\
\textit{FunDef} & \Rightarrow & \textit{FunHeader} \ \{ \ \textit{FunBody} \ \} \\
\textit{FunHeader} & \Rightarrow & \textit{Type Id} \ (\ [\textit{Param} \ [\ , \ \textit{Param} \ ]^* \ ] \ ) \\
 & \mid & \textbf{void} \ \textit{Id} \ (\ [\textit{Param} \ [\ , \ \textit{Param} \ ]^* \ ] \ ) \\
\textit{Type} & \Rightarrow & \textbf{int} \mid \textbf{bool} \mid \textbf{float}
\end{array}
$$

Idea:

- Switch to right-recursion for Globals
- BUT: see notes on right-recursion later on

Shift-Reduce Conflicts

Solution:

Program	$\Rightarrow$	*Globals FunDefs*
Globals	$\Rightarrow$	*Global Globals* $\mid \epsilon$
Global	$\Rightarrow$	*Type Id* **;**
Fundefs	$\Rightarrow$	*Fundefs Fundef* $\mid \epsilon$
FunDef	$\Rightarrow$	*FunHeader* **{** *FunBody* **}**
FunHeader	$\Rightarrow$	*Type Id* **(** $[$*Param* $[$ **,** *Param* $]^*$ $]$ **)**
	$\mid$	**void** *Id* **(** $[$*Param* $[$ **,** *Param* $]^*$ $]$ **)**
Type	$\Rightarrow$	**int** $\mid$ **bool** $\mid$ **float**

Idea:

- Switch to right-recursion for Globals
- BUT: see notes on right-recursion later on

Alternative:

- Reconsider if you really need the strong separation of globals and fundefs

Reduce-Reduce Conflicts

Characteristic:

- Grammar is defined such that with given lookahead parser cannot decide among different reduce opportunities
- Typically grammar has identical right hand side productions

Reduce-Reduce Conflicts

Characteristic:

- Grammar is defined such that with given lookahead parser cannot decide among different reduce opportunities
- Typically grammar has identical right hand side productions

Example:

$$
\begin{array}{lcll}
\textit{Expr} & \Rightarrow & \textit{Term} * \textbf{num} \\
 & | & \textit{Term} \\
 & & \\
\textit{Term} & \Rightarrow & \textit{Term} * \textbf{num} \\
 & | & \textbf{num}
\end{array}
$$

Reduce-Reduce Conflicts

Characteristic:

- Grammar is defined such that with given lookahead parser cannot decide among different reduce opportunities
- Typically grammar has identical right hand side productions

What to do:

- YACC solution: use handle defined first
- Must be: Rewrite grammar !!

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & \mid & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & \mid & \textbf{elem} \end{array}$$

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Parsing 3-element list: elem elem elem:

List List

Left Recursion vs Right Recursion

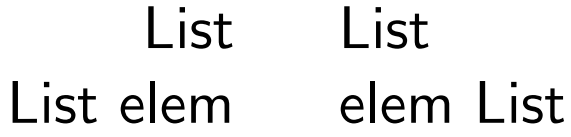
Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Parsing 3-element list: elem elem elem:



Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Parsing 3-element list: elem elem elem:

List	List
List elem	elem List
List elem elem	elem elem List

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Parsing 3-element list: elem elem elem:

List	List
List elem	elem List
List elem elem	elem elem List
elem elem elem	elem elem elem

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	reduce
List	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	reduce
List	elem	shift
List elem	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	reduce
List	elem	shift
List elem	elem	reduce
List	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	reduce
List	elem	shift
List elem	elem	reduce
List	elem	shift
List elem	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	reduce
List	elem	shift
List elem	elem	reduce
List	elem	shift
List elem	elem	reduce
List	eof	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Left-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	reduce
List	elem	shift
List elem	elem	reduce
List	elem	shift
List elem	elem	reduce
List	eof	**accept**

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	shift
elem elem	elem	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	shift
elem elem	elem	shift
elem elem elem	eof	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	shift
elem elem	elem	shift
elem elem elem	eof	reduce
elem elem List	eof	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	shift
elem elem	elem	shift
elem elem elem	eof	reduce
elem elem List	eof	reduce
elem List	eof	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	shift
elem elem	elem	shift
elem elem elem	eof	reduce
elem elem List	eof	reduce
elem List	eof	reduce
List	eof	

Left Recursion vs Right Recursion

Left-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textit{List}\ \ \textbf{elem} \\ & | & \textbf{elem} \end{array}$$

Right-recursive grammar:

$$\begin{array}{lcl} \textit{List} & \Rightarrow & \textbf{elem}\ \ \textit{List} \\ & | & \textbf{elem} \end{array}$$

Right-recursive shift-reduce parsing:

stack	lookahead	action
ϵ	elem	shift
elem	elem	shift
elem elem	elem	shift
elem elem elem	eof	reduce
elem elem List	eof	reduce
elem List	eof	reduce
List	eof	**accept**

Left Recursion vs Right Recursion: Summary

Left recursion:

- We alternate between shift and reduce
- Stack size bounded

Right recursion:

- We must shift the whole input before we can reduce
- Stack size linear in the length of the list

Left Recursion vs Right Recursion: Summary

	left recursion	right recursion
recursive-decent parsing	non-termination	required
shift-reduce parsing	bounded stack size	unbounded stack size

Conclusion: Four Perspectives on Syntax

1. Context-free grammar:

- Theoretician's approach to syntax definition
- Sets of symbols, production, ...
- *There* **is** *a solution ...*

Conclusion: Four Perspectives on Syntax

1. Context-free grammar:

- Theoretician's approach to syntax definition
- Sets of symbols, production, ...
- *There* **is** *a solution ...*

2. Extended Backus-Naur form:

- Practician's approach to syntax definition
- Lists of symbols, productions
- Syntactic variables used for documentation and clarification
- Readability more important than size

Conclusion: Four Perspectives on Syntax

3. Parser grammar:

- Technically extended Backus-Naur form, but ...
- Restrictions regarding token look-ahead
- Restrictions to avoid ambiguity (conflicts)
- Size of grammar, speed of derived parser matter (maybe)

Conclusion: Four Perspectives on Syntax

3. Parser grammar:

- Technically extended Backus-Naur form, but ...
- Restrictions regarding token look-ahead
- Restrictions to avoid ambiguity (conflicts)
- Size of grammar, speed of derived parser matter (maybe)

4. Abstract syntax tree:

- Technically nothing but yet another grammar, but ...
- Must be helpful for further program transformations
- Size and expressivity matter
- How to store additional inferred information ?
- Can deviate from exact syntax

Conclusion: Four Different Purposes and Audiences

Context-free grammar:

- Purpose: Theoretical considerations
- Audience: Programming language theoreticians

Extended Backus-Naur form:

- Purpose: Language specification
- Audience: Programmers

Parser grammar:

- Purpose: Parser generation
- Audience: Parser generator tool

Abstract syntax tree:

- Purpose: intermediate program representation
- Audience: Compiler implementer

Conclusion: Four Different Purposes and Audiences

One solution fits them all ?

Conclusion: Four Different Purposes and Audiences

One solution fits them all ?

Advantages of one uniform representation:

- Any language extension requires similar additions to spec, parser and IR
- More agile development
- No consistentency issues

Advantages of different individual representations:

- Audience-focussed:
  - programmer
  - parser generator
  - compiler implementer
- Different representations tailor-made for different purposes

The End: Questions ?