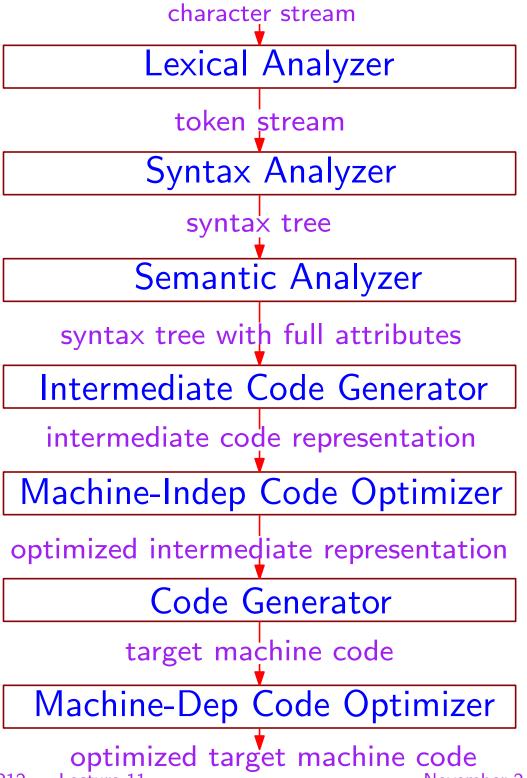
Syntactic Analysis

Phases of a Compiler

Symbol Table



Slide 2 of 22 CS4212 — Lecture 11 November 2, 2012

Example

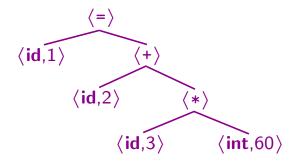
Symbol Table

1	position	• • •
2	initial	• • •
3	rate	• • •

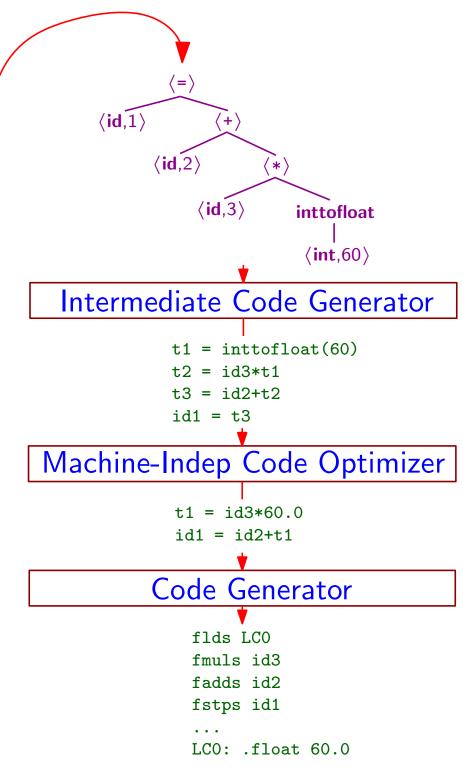
position = initial + rate*60



Syntax Analyzer



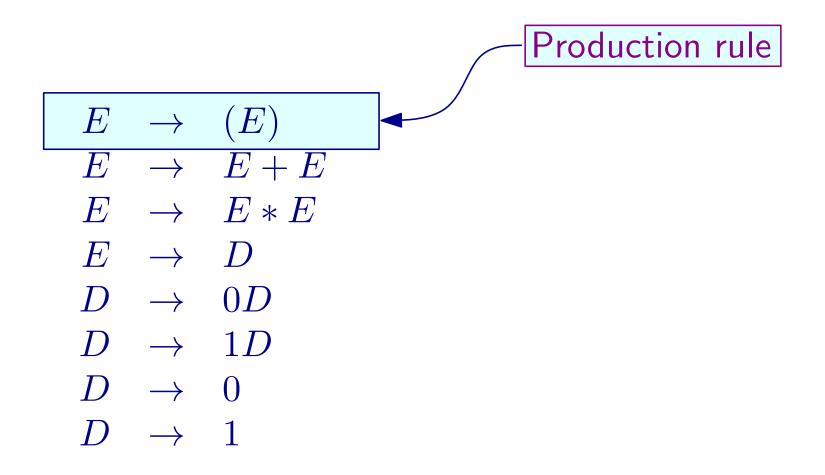
Semantic Analyzer

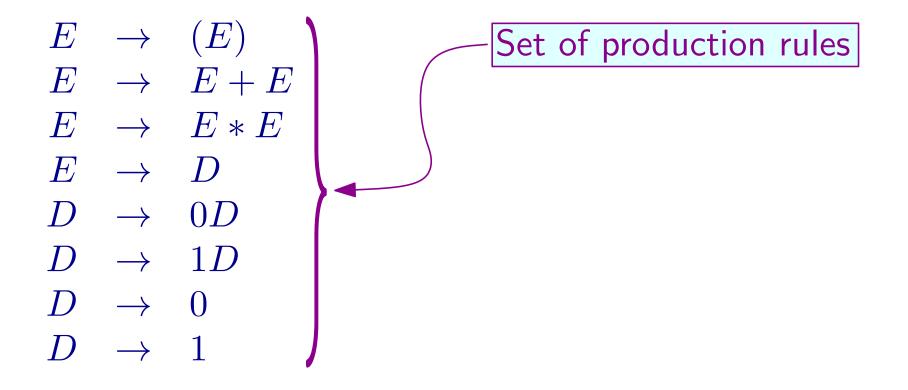


Grammars

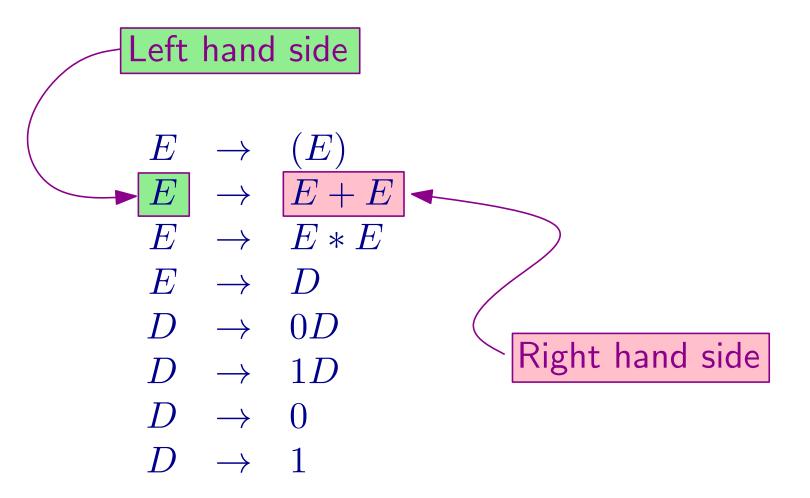
- Mathematical concept in the area of formal languages
- Language generator: specifies a mechanism for generating all elements of the language.
 - The language is the set of strings that can be generated.
 - ♦ Can be converted into an acceptor: a procedure that tests whether a string is in the language or not.
- Practitioner's approach: introduce concepts step-by-step

$$\begin{array}{cccc} E & \rightarrow & (E) \\ E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & D \\ D & \rightarrow & 0D \\ D & \rightarrow & 1D \\ D & \rightarrow & 0 \\ D & \rightarrow & 1 \end{array}$$



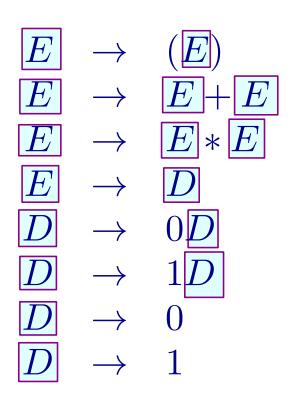


Slide 5 of 22 View: 3 CS4212 — Lecture 11 November 2, 2012



Left hand side can be *rewritten* as the right hand side.

Slide 5 of 22 View: 4 CS4212 — Lecture 11 November 2, 2012



Non-terminals

- Denoted by capitals
- Not part of the generated language.
- An aid to generating the language.

$$\begin{array}{cccc} E & \rightarrow & \textcircled{E} \\ E & \rightarrow & E \oplus E \\ E & \rightarrow & D \\ E & \rightarrow & D \\ D & \rightarrow & \textcircled{D} \\ \end{array}$$

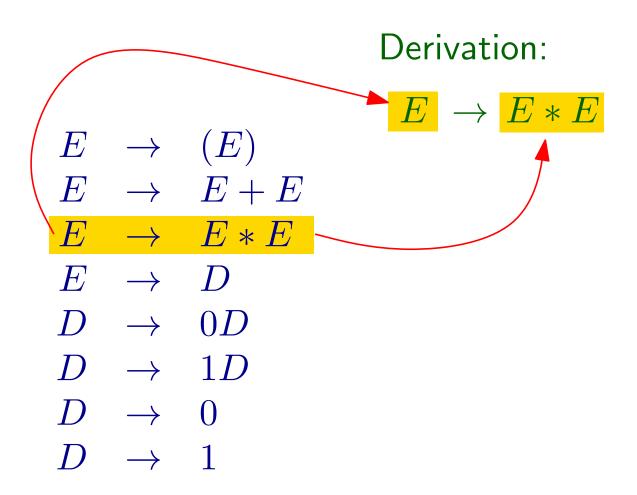
Terminals:

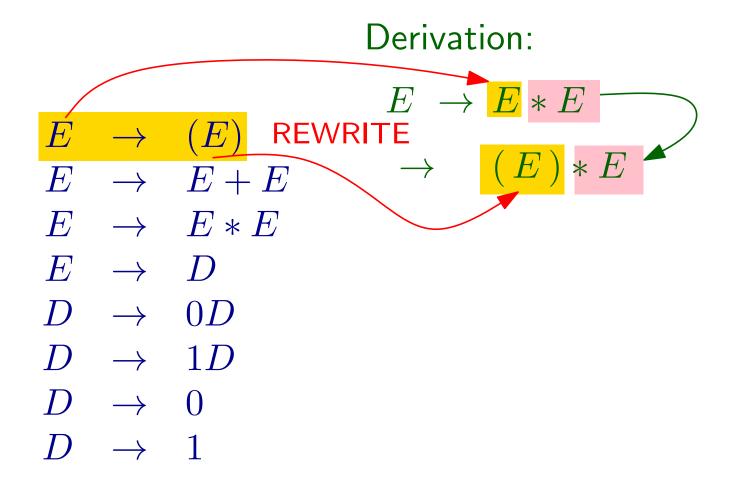
- ♦ The rest of the symbols.
- Part of the generated language.

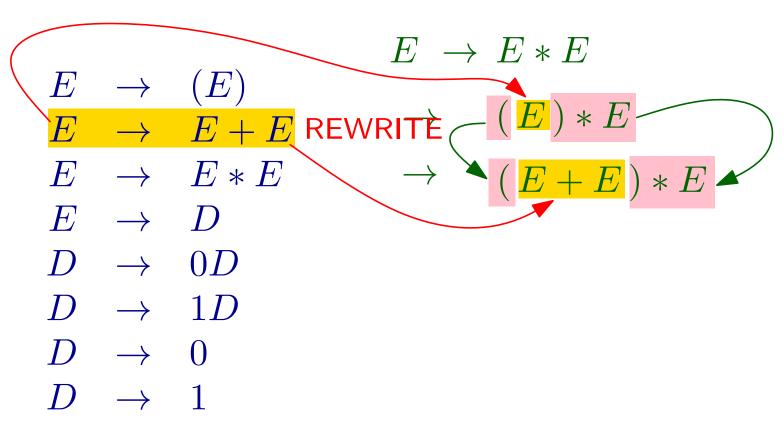
Slide 5 of 22 View: 6 CS4212 — Lecture 11 November 2, 2012

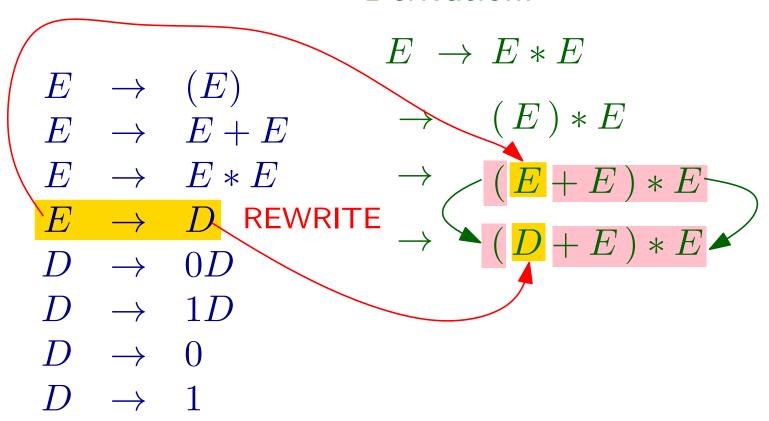
$$E \rightarrow$$

$$egin{array}{cccc} E &
ightarrow & (E) \ E &
ightarrow & E + E \ E &
ightarrow & D \ D &
ightarrow & DD \ D &
ightarrow & 1D \ D &
ightarrow & 1 \ D &
ightarrow & 1 \ \end{array}$$









$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow E * E$$

$$E \rightarrow D$$

$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 1$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow E + E \rightarrow (E) * E$$

$$E \rightarrow E * E \rightarrow (E + E) * E$$

$$E \rightarrow D \text{ REWRITE}$$

$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 0$$

$$D \rightarrow 1$$

$$D \rightarrow 1$$

$$D \rightarrow 1$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow E + E \rightarrow (E) * E$$

$$E \rightarrow E * E \rightarrow (E + E) * E$$

$$E \rightarrow D \rightarrow (D + E) * E$$

$$D \rightarrow 0D \rightarrow 1D \text{ REWRITE} \rightarrow (1 + E) * E$$

$$D \rightarrow 0 \rightarrow 1 \rightarrow 1$$

$$D \rightarrow 1 \rightarrow 1 \rightarrow (1 + D) * E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

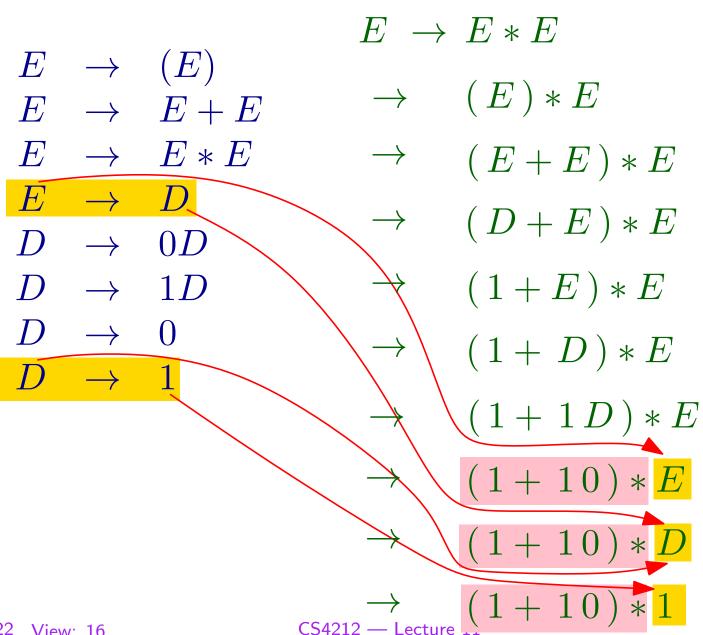
$$E \rightarrow D$$

$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 0$$

$$D \rightarrow 1$$



$$\begin{array}{cccc} E & \rightarrow & (E) \\ E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & D \\ D & \rightarrow & 0D \\ D & \rightarrow & 1D \\ D & \rightarrow & 0 \\ D & \rightarrow & 1 \end{array}$$

$$(1+10)*1 \in \mathcal{L}(E)$$

$$\rightarrow (1+E)*E$$

$$\rightarrow (1+D)*E$$

$$\rightarrow (1+1D)*E$$

$$\rightarrow (1+10)*E$$

$$\rightarrow (1+10)*D$$

$$\rightarrow (1+10)*1$$
Novel

Language generated by non-terminal E

$$\begin{array}{cccc} E & \rightarrow & (E) \\ E & \rightarrow & E + E \\ E & \rightarrow & E * E \\ E & \rightarrow & D \\ D & \rightarrow & 0D \\ D & \rightarrow & 1D \\ D & \rightarrow & 0 \\ D & \rightarrow & 1 \end{array}$$

Derivation:

$$\underbrace{(1+10)*1}_{\text{terminals only}} \in \mathcal{L}(E)$$

Similarly:

$$1001 \in \mathcal{L}(D)$$

$$\rightarrow (1+10)*E$$

$$\rightarrow$$
 $(1+10)*D$

$$\rightarrow$$
 $(1+10)*1$

Formal Definition of Grammar

Grammar: tuple $G \equiv \langle \Sigma, N, \Pi, S \rangle$

 Σ - alphabet of *terminal symbols*

N - set of non-terminal symbols

 Π - set of production rules

S - start non-terminal, $S \in N$

 $\mathcal{L}(G) \equiv \mathcal{L}(S)$ (contains only strings of terminal symbols)

Rules are enough in practice.

Grammar can be derived from the rules:

$$E \rightarrow (E)$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow D$$

$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 0$$

$$D \rightarrow 1$$

Rules are enough in practice.	\overline{E}	$\stackrel{\frown}{\rightarrow}$	(E)
	E	\rightarrow	E + E
Grammar can be derived from the rules:	E	\rightarrow	E * E
$\Sigma = \{ (,), +, *, 0, 1 \}$	\boldsymbol{E}	\rightarrow	D
	D	\rightarrow	$\Box 0D$
	$\bigcirc D$	\rightarrow	1D
	D	\rightarrow	0
	D	\rightarrow	_1

Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$

$$N = \{E, D\}$$

$$\begin{array}{cccc}
E & \rightarrow & (E) \\
E & \rightarrow & E + E \\
\hline
E & \rightarrow & E * E \\
\hline
-E & \rightarrow & D \\
D & \rightarrow & 0D \\
D & \rightarrow & 1D \\
D & \rightarrow & 0
\end{array}$$

Rules are enough in practice.
$$E \to (E)$$
 Grammar can be derived from the rules:
$$E \to E + E$$

$$E = \{(,), +, *, 0, 1\}$$

$$E \to D$$

$$D \to 0D$$

$$D \to 1D$$

$$D \to 0$$

$$D \to 0$$

$$D \to 0$$

Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$

$$N = \{ E, D \}$$

$$\Pi = \{ E \rightarrow (E), \dots \}$$

 $(E) \rightarrow (E)$ $E \rightarrow E + E$ $E \rightarrow E * E$ $E \rightarrow D$ $D \rightarrow 0D$ $D \rightarrow 1D$ $D \rightarrow 1$

Start non-terminal: E (the LHS of the first rule)

Rules are enough in practice.

Grammar can be derived from the rules:

$$\Sigma = \{ (,), +, *, 0, 1 \}$$
 $N = \{ E, D \}$
 $\Pi = \{ E \to (E), \dots \}$

$$\begin{array}{ccc} E & \rightarrow & (E) \\ E & \rightarrow & E + \end{array}$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow D$$

$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 0$$

$$D \rightarrow 1$$

Start non-terminal: E (the LHS of the first rule)

$$G = \langle \Sigma, N, \Pi, E \rangle$$

$$E \rightarrow (E)$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow D$$

$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 0$$

(1 + 1 0) * 1

$$E \rightarrow (E)$$

$$E \rightarrow E + E$$

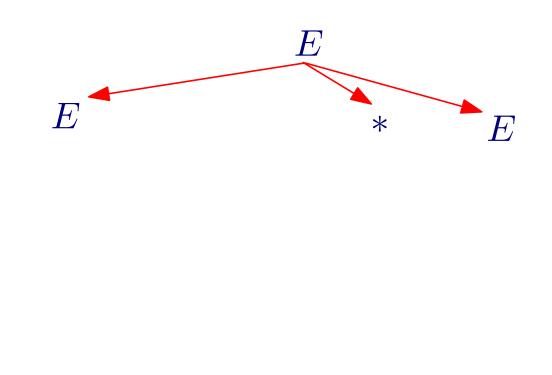
$$E \rightarrow E * E$$

$$E \rightarrow D$$

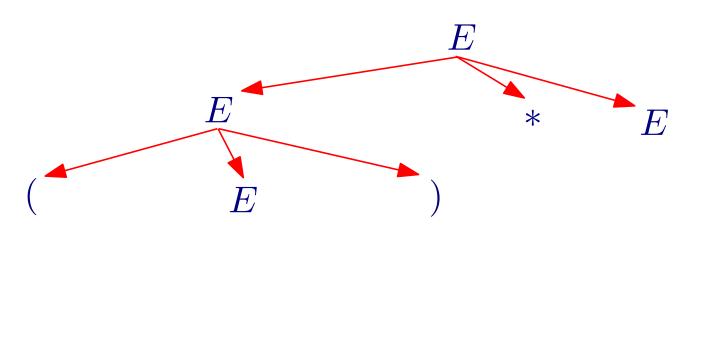
$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 1$$

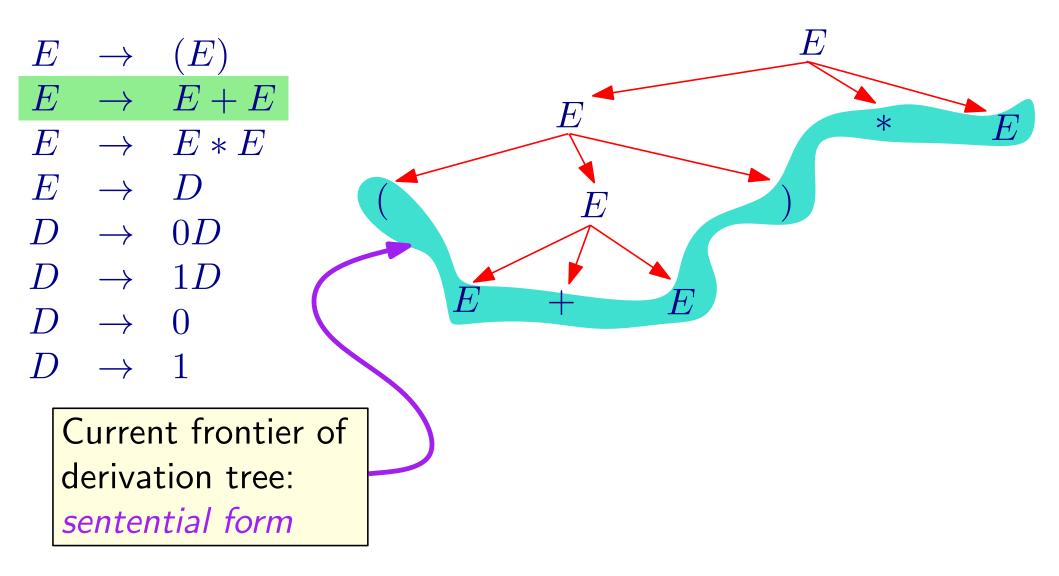


$$(1 + 1 0) * 1$$



 $\frac{\text{matched}:}{1} + 1 = 0$) * 1

Slide 8 of 22 View: 3 CS4212 — Lecture 11 November 2, 2012



matched: (1 + 1 0) * 1

$$E \rightarrow (E)$$

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

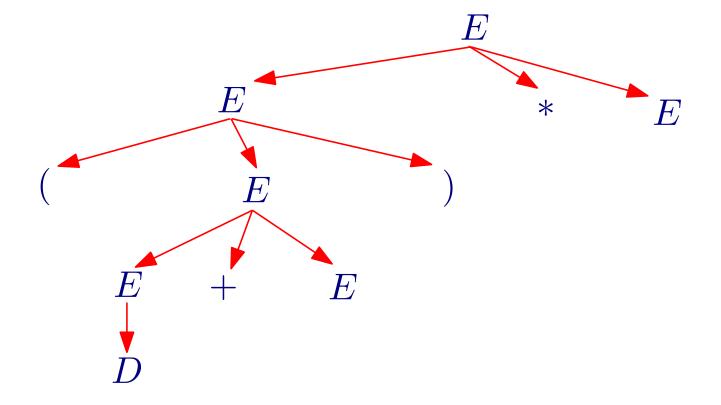
$$D \rightarrow D$$

$$D \rightarrow 0D$$

$$D \rightarrow 1D$$

$$D \rightarrow 0$$

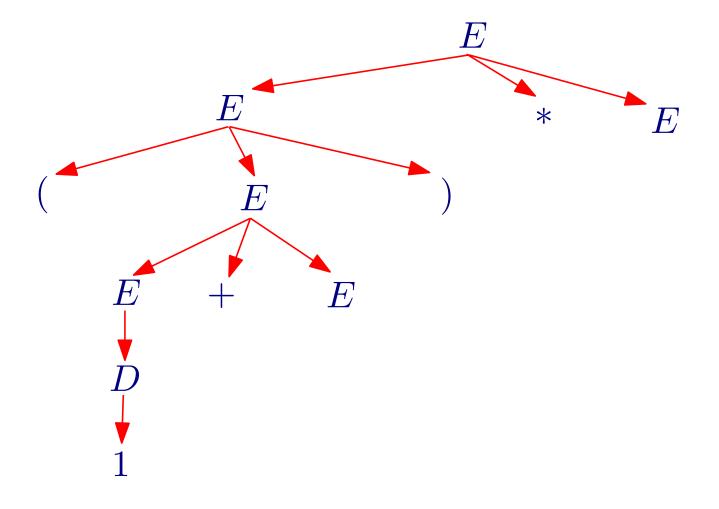
$$D \rightarrow 1$$



matched: (1 + 1 0) * 1

Slide 8 of 22 View: 5 CS4212 — Lecture 11 November 2, 2012

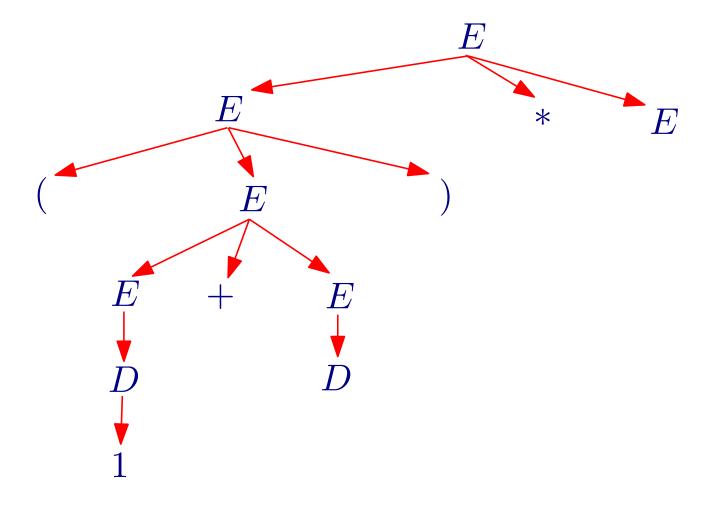
$$egin{array}{cccc} E &
ightarrow & (E) \ E &
ightarrow & E + E \ E &
ightarrow & D \ D &
ightarrow & DD \ D &
ightarrow & 1D \ D &
ightarrow & 0 \ D &
ightarrow & 1 \end{array}$$



 $\frac{\text{matched}:}{1} + 1 = 0$ * 1

Slide 8 of 22 View: 6 CS4212 — Lecture 11 November 2, 2012

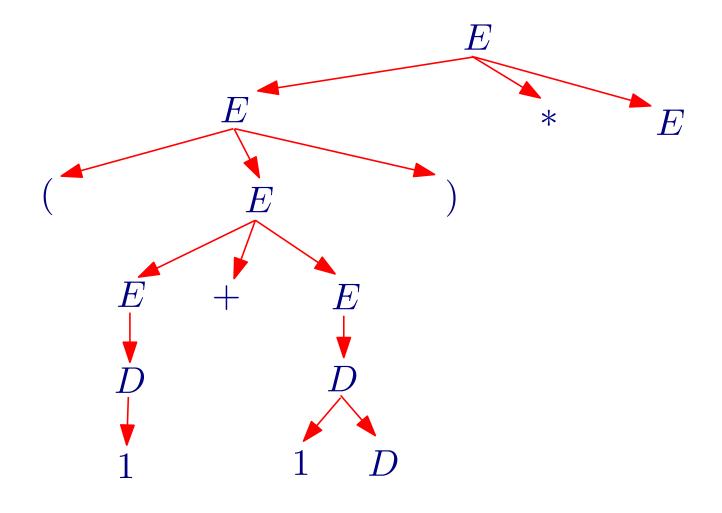
$$egin{array}{cccc} E &
ightarrow & (E) \ E &
ightarrow & E + E \ E &
ightarrow & D \ D &
ightarrow & D \ D &
ightarrow & 1D \ D &
ightarrow & 1 \ D &
ightarrow & 1 \ \end{array}$$



 $\frac{\text{matched}:}{1} + 1 = 0$) * 1

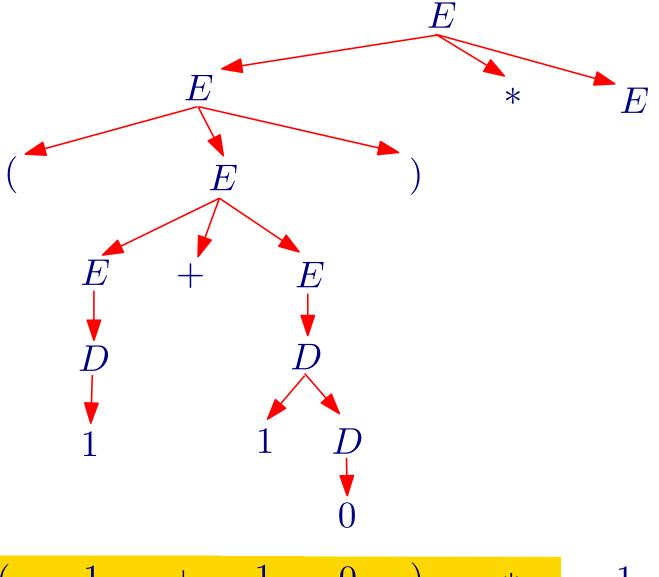
Slide 8 of 22 View: 7 CS4212 — Lecture 11 November 2, 2012

$$egin{array}{cccc} E &
ightarrow & (E) \ E &
ightarrow & E + E \ E &
ightarrow & D \ D &
ightarrow & 0D \ D &
ightarrow & 1D \ D &
ightarrow & 1 \ D &
ightarrow & 1 \ \end{array}$$



 $\frac{\text{matched}}{\text{matched}} : (1 + 1 0) * 1$

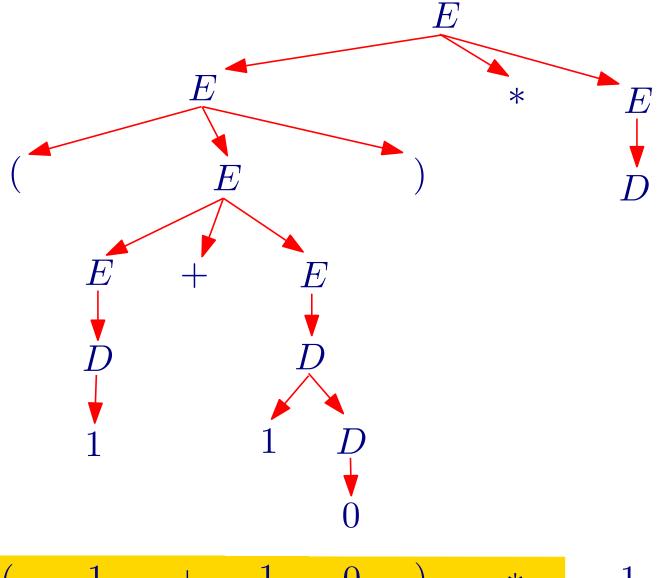
Slide 8 of 22 View: 8 CS4212 — Lecture 11 November 2, 2012



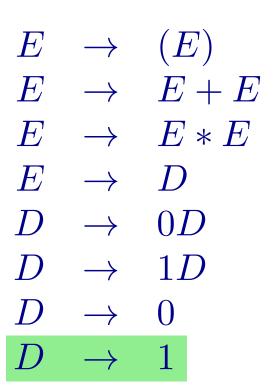
matched: (1 + 1 0) *

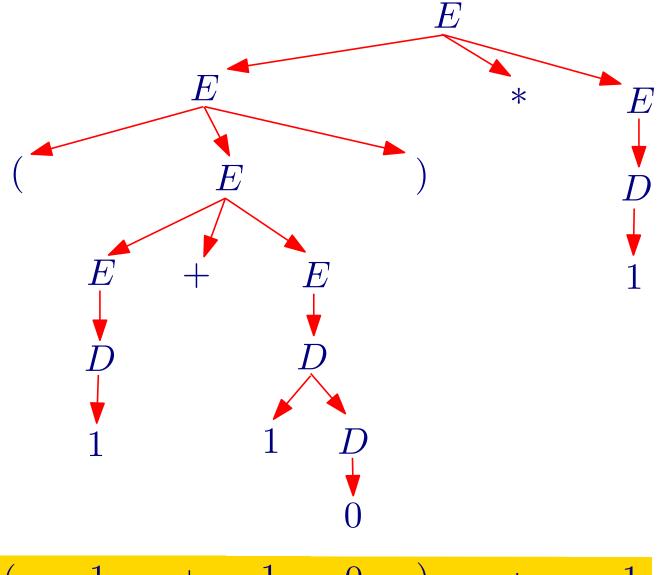
Slide 8 of 22 View: 9 CS4212 — Lecture 11 November 2, 2012

$$egin{array}{ccccc} E &
ightarrow & (E) \ E &
ightarrow & E+E \ E &
ightarrow & D \ D &
ightarrow & D \ D &
ightarrow & 0D \ D &
ightarrow & 1D \ D &
ightarrow & 0 \ D &
ightarrow & 1 \end{array}$$



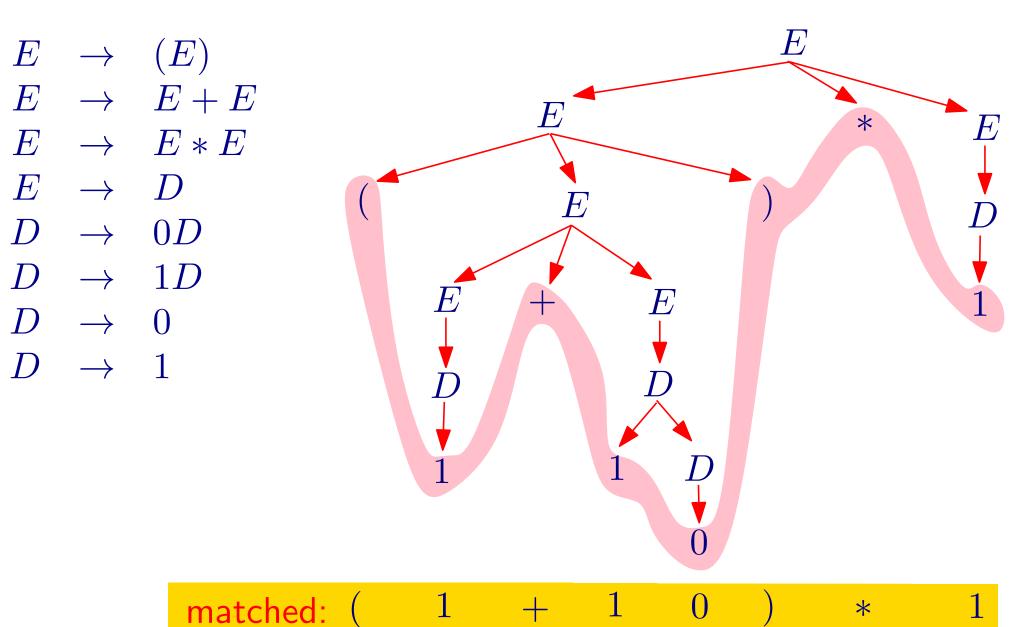
matched: (1 + 1 0) * 1





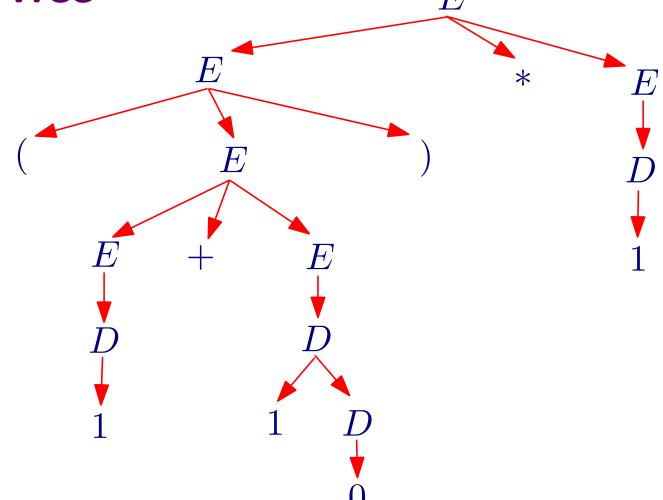
matched: (1 + 1 0) * 1

Slide 8 of 22 View: 11 CS4212 — Lecture 11 November 2, 2012

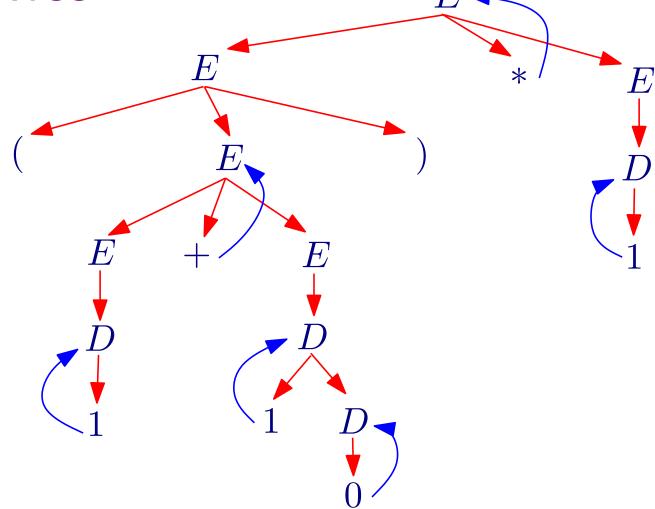


Slide 8 of 22 View: 12 CS4212 — Lecture 11 November 2, 2012

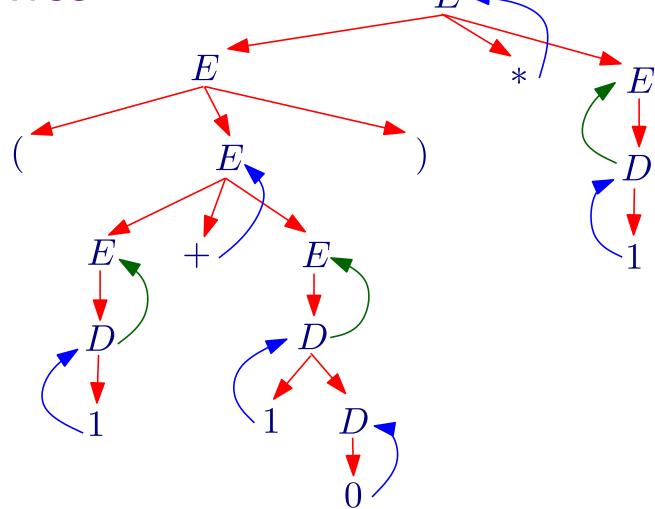
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



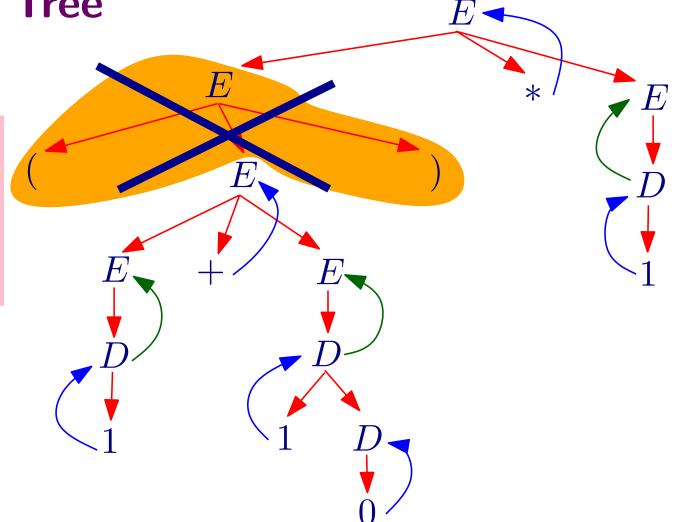
- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed



- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed

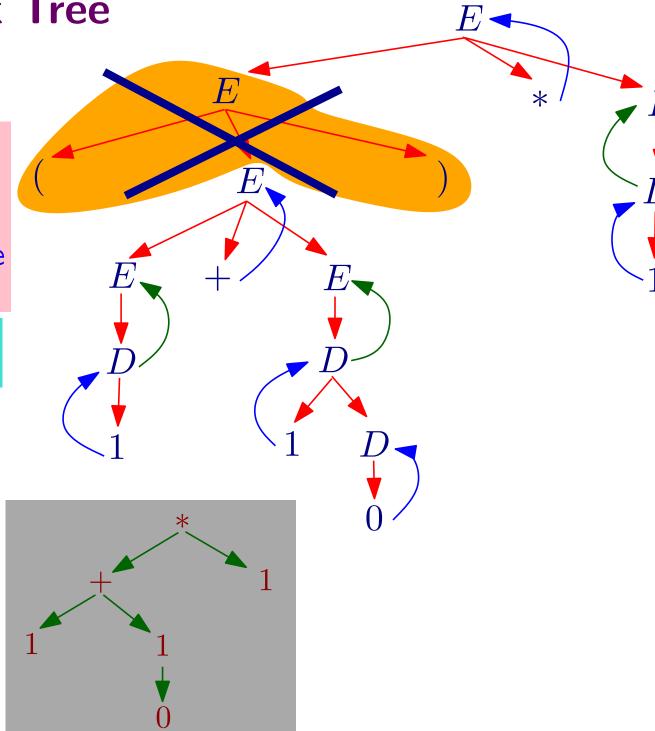


Simplified version of the parse tree:

- if node has only one terminal child, make the terminal the label of current node and remove the child
- if node has only one child, shortcut the node
- many other customized rules, discussed as they are needed

Resulting tree:

Simple, but still captures the structure of the expression.



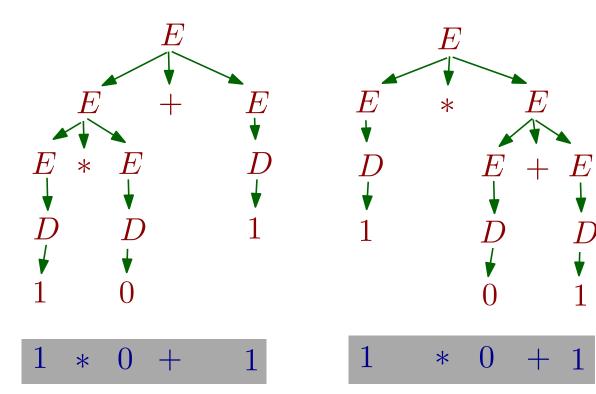
Slide 9 of 22 View: 5 CS4212 — Lecture 11 November 2, 2012

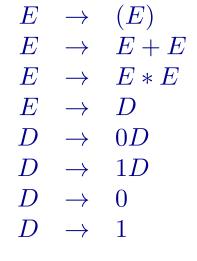
What Are Grammars Good For?

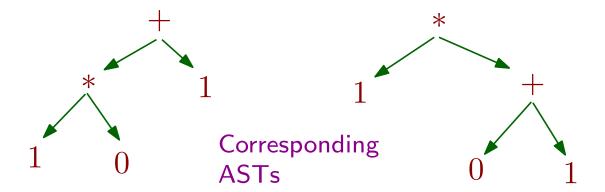
- Allow specification of (programming) languages generation.
- Allow deciding whether a string belongs to the language or not – acceptance.
- Allow syntactic analysis.
- Syntactic analysis: building the Abstract Syntax Tree from a string or program.
- The AST can be used by compilers, program analyzers, and other code manipulators.

Ambiguity

Non-unique parse trees: ambiguous grammar



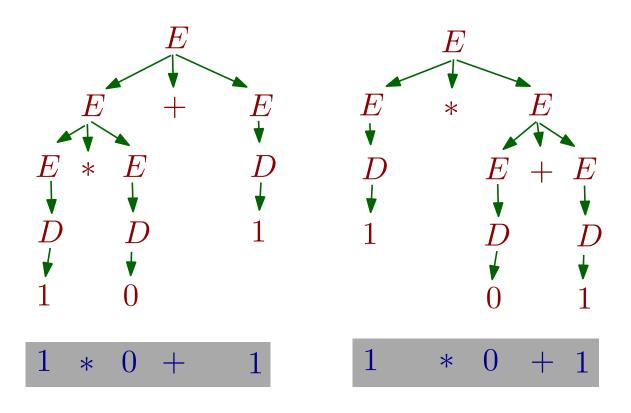


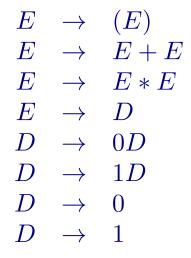


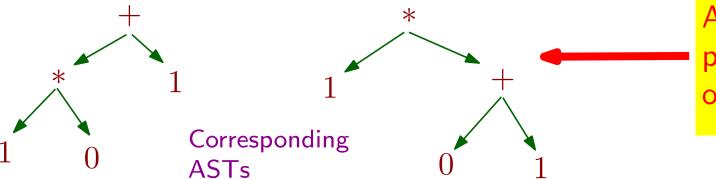
Slide 11 of 22 View: 1 CS4212 — Lecture 11 November 2, 2012

Ambiguity

Non-unique parse trees: ambiguous grammar







Ambiguous precedence of operators!

Slide 11 of 22 View: 2 CS4212 — Lecture 11 November 2, 2012

Ambiguity Should Be Avoided!

- An ambiguous grammar can always be replaced by a non-ambiguous one.
- Ambiguous grammars have fewer rules, but tend to capture less of the language's structure.
- Precedence and associativity of operators is crucial to structure of languages, and should be captured in the grammar.
- Languages with non-ambiguous grammars can be parsed more efficiently.

Original grammar:

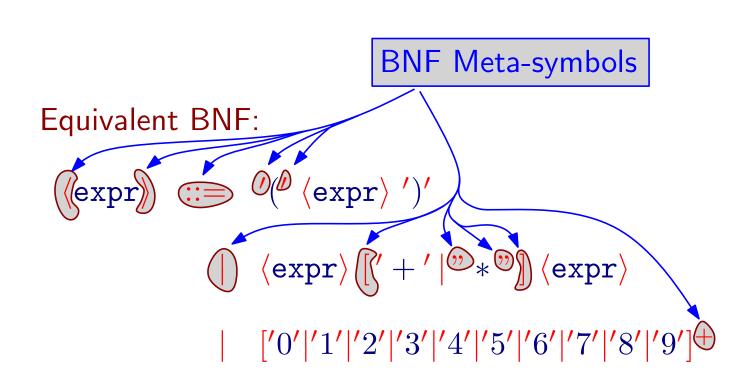
```
E \rightarrow (E)
E \rightarrow E + E
E \rightarrow E * E
E \rightarrow D
D \rightarrow 0D
D \rightarrow 1D
D \rightarrow 2D
D \rightarrow 3D
D \rightarrow
            4D
D \rightarrow 5D
D \rightarrow 6D
D \rightarrow 7D
D \rightarrow 8D
D \rightarrow 9D
D \rightarrow 0
D \rightarrow
D \rightarrow 2
D \rightarrow 3
D \rightarrow 4
D \rightarrow 5
D \rightarrow 6
D \rightarrow 7
D \rightarrow 8
D \rightarrow 9
```

Equivalent BNF:

```
\langle \expr \rangle ::= '(' \langle \expr \rangle ')'
 | \langle \expr \rangle [' + ' | " * "] \langle \expr \rangle 
 | ['0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9']^{+}
```

Original grammar:

```
(E)
    E + E
\rightarrow E * E
    D
    0D
    1D
    2D
    3D
    4D
    5D
\rightarrow 6D
    7D
    8D
    9D
    9
```



Original grammar:

```
(E)
        E + E
 \rightarrow E * E
       D
 \rightarrow
        0D
        1D
\rightarrow 2D
        3D
 \rightarrow
        4D
        5D
\rightarrow 6D
       7D
        8D
\rightarrow 9D
 \rightarrow 0
\rightarrow 2
\rightarrow 7
 \rightarrow 8
```

 \rightarrow 9

Non-terminals are enclosed in angle brackets

Equivalent BNF:

Terminals are enclosed in simple or double quotes.

Original grammar: \boldsymbol{E} $E \oplus E$ $E \otimes E$ D0D1D2D3D4D5D6D7D8D9D

9

Equivalent BNF:

$$\langle \expr \rangle ::= \langle (') \langle \expr \rangle (')' \rangle$$

$$\langle \exp r \rangle [(+)|(**)| \langle \exp r \rangle$$

(0) | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'] +

Single quote terminal: "'"

Double quote terminal: ""

Original grammar:

```
(E)
        E + E
 \rightarrow E * E
        D
 \rightarrow
        0D
        1D
        2D
 \rightarrow
        3D
        4D
        5D
\rightarrow 6D
       7D
        8D
\rightarrow 9D
 \rightarrow 0
 \rightarrow
\rightarrow 2
\rightarrow 7
 \rightarrow 8
\rightarrow 9
```

Equivalent BNF:

```
\(\left(\expr\right)'\)
\( \left(\expr\right)'\)
\( \left(\expr\right) \left['+'\color=*" *"] \left(\expr\right)\)
\( \left['0'\color='1'\color='2'\color='1'' *"] \left(\expr\right)\)
\( \left['0'\color='1'\color='2'\color='1'' *"] \left(\expr\right)\)
\( \left['0'\color='1'\color='1'' *"] \left(\expr\right)\)
```

 \rightarrow 7

9

Original grammar: Grouping E(E)E + EE * E**Equivalent BNF:** 0D1D2D(' \langle expr \rangle ')' $\langle expr \rangle$ 3D \rightarrow 4D5D $\langle \exp r \rangle [' + ' | " * "] \langle \exp r \rangle$ \rightarrow 6D 7D8D['0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9']+ $\rightarrow 9D$ \rightarrow 0

Original grammar:

```
\rightarrow (E)
      \rightarrow E + E
     \rightarrow E * E
    \rightarrow D
     \rightarrow 0D
           1D
     \rightarrow 2D
     \rightarrow 3D
             4D
      \rightarrow 5D
    \rightarrow 6D
           7D
D \rightarrow 8D
    \rightarrow 9D
    \rightarrow 0
    \rightarrow
D \rightarrow 2
D \rightarrow 4
D \rightarrow 7
D \rightarrow 8
```

 \rightarrow 9

Equivalent BNF:

$$\langle \exp r \rangle ::= '(' \langle \exp r \rangle ')'$$
 $\langle \exp r \rangle [' + '] " * "] \langle \exp r \rangle$
 $['0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9']^+$
Either-or meta-symbol

Original grammar:

```
(E)
\rightarrow
\rightarrow
        0D
        1D
        2D
\rightarrow
        3D
        4D
        5D
\rightarrow 6D
       7D
        8D
\rightarrow 9D
\rightarrow 0
\rightarrow 7
\rightarrow 9
```

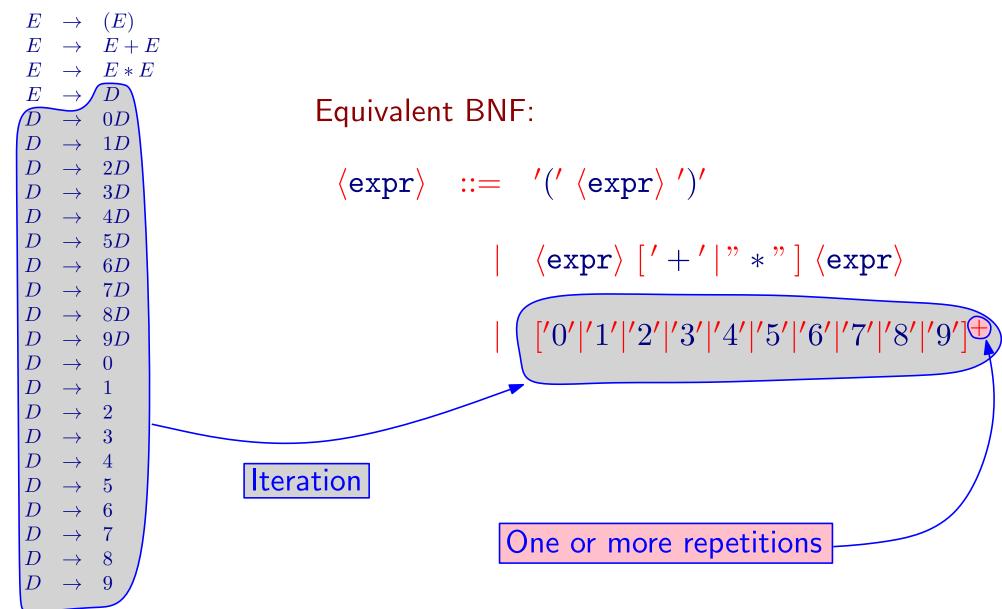
Equivalent BNF:

$$\langle \exp r \rangle ::= '(' \langle \exp r \rangle ')'$$

$$| \langle \exp r \rangle [' + ' | " * "] \langle \exp r \rangle$$

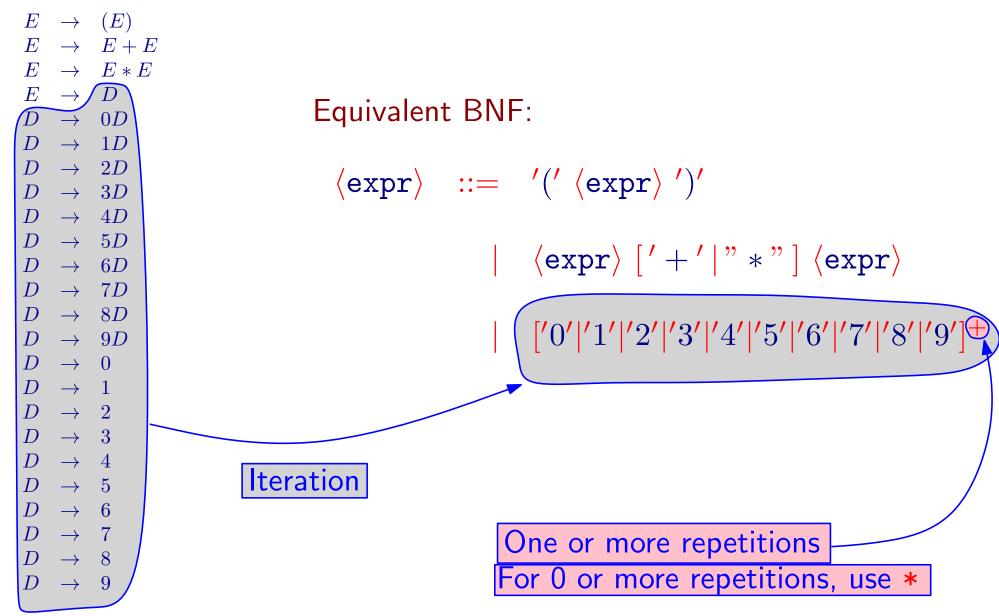
$$['0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9']^{+}$$
Factorization

Original grammar:



Slide 13 of 22 View: 9 CS4212 — Lecture 11

Original grammar:



Slide 13 of 22 View: 10 CS4212 — Lecture 11

A Non-Ambiguous Grammar for Expressions

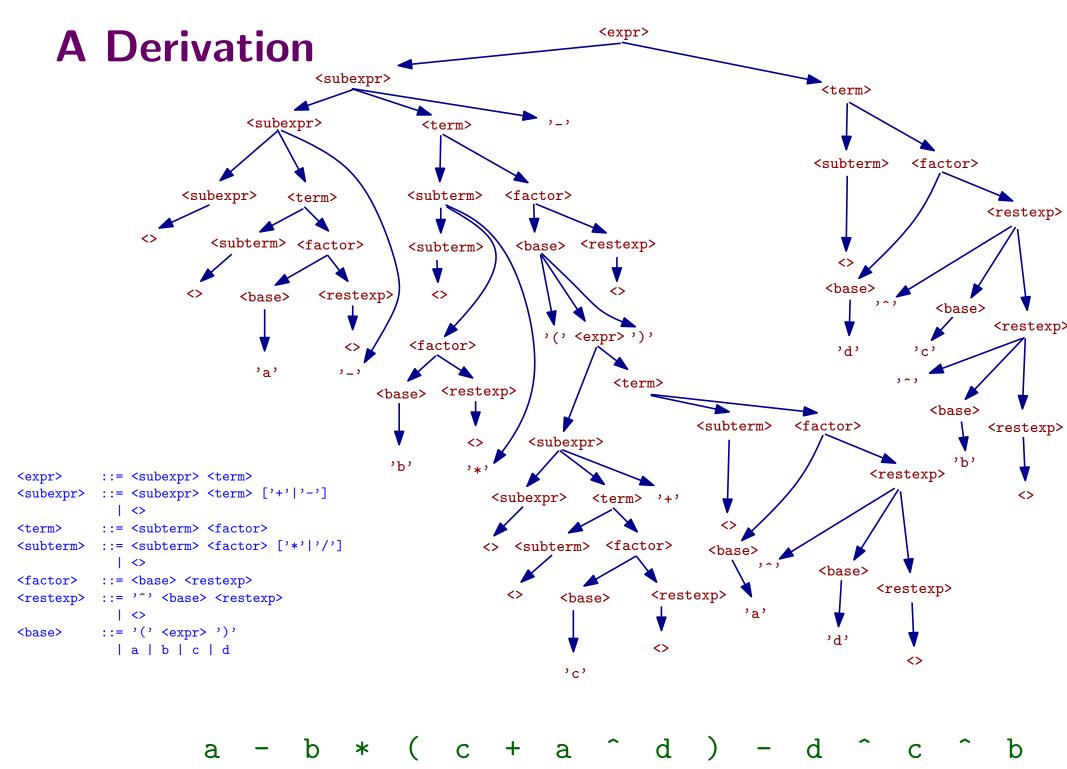
```
<expr> ::= <subexpr> <term>
<subexpr> ::= <subexpr> <term> ['+'|'-']
            | <>
<term> ::= <subterm> <factor>
<subterm> ::= <subterm> <factor> ['*'|'/']
             | <>
<factor> ::= <base> <restexp>
<restexp> ::= '^' <base> <restexp>
            | <>
<base> ::= '(' <expr> ')'
            | a | b | c | d
```

A Non-Ambiguous Grammar for Expressions

```
Empty string: can appear
                  inbetween any two terminals
<expr> ::= | <subexpr> <term>
<subexpr> ::= <subexpr> <term> ['+'|'-']
<term> ::= <subterm> <factor>
<subterm> ::= <subterm> <factor> ['*'|'/']
             | <>
<factor> ::= <base> <restexp>
<restexp> ::= '^' <base> <restexp>
             <>
<base> ::= '(' <expr> ')'
             | a | b | c | d
```

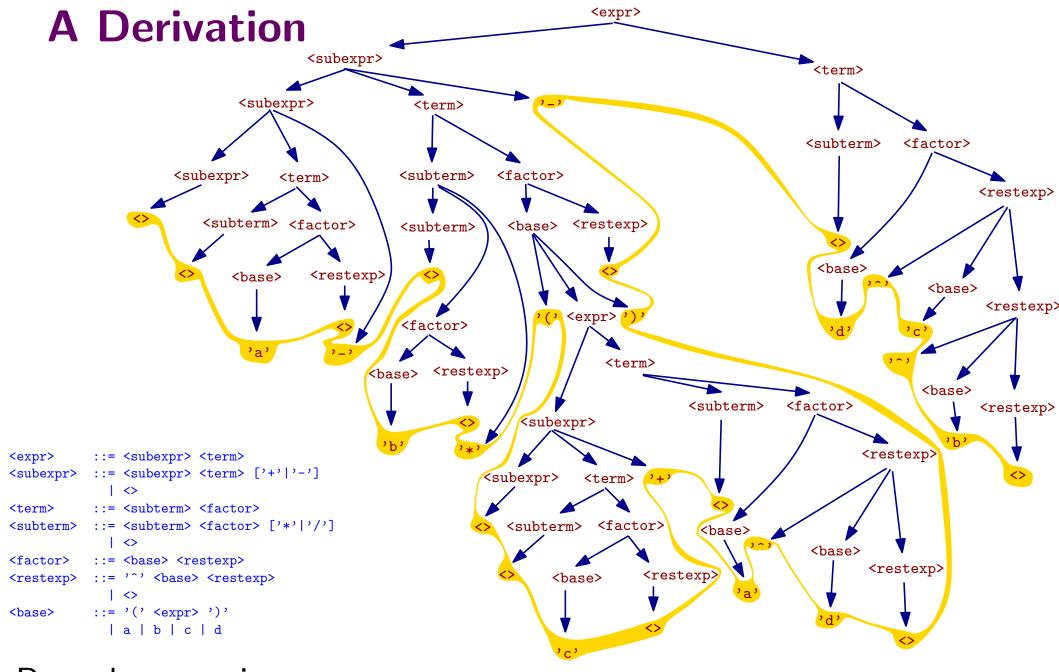
A Non-Ambiguous Grammar for Expressions

Compress the spec, but information about associativity is lost! <expr> ::= <subexpr> <term> <subexpr> ::= [<term> ['+'|'-']]* ::= <subterm> <factor> <term> <subterm> ::= <subterm> <factor> ['*'|'/'] **| <>** <factor> ::= <base> <restexp> ::= '^' <base> <restexp> <restexp> **<>** <base> ::= '(' <expr> ')' | a | b | c | d



Slide 15 of 22 View: 1

CS4212 — Lecture 11

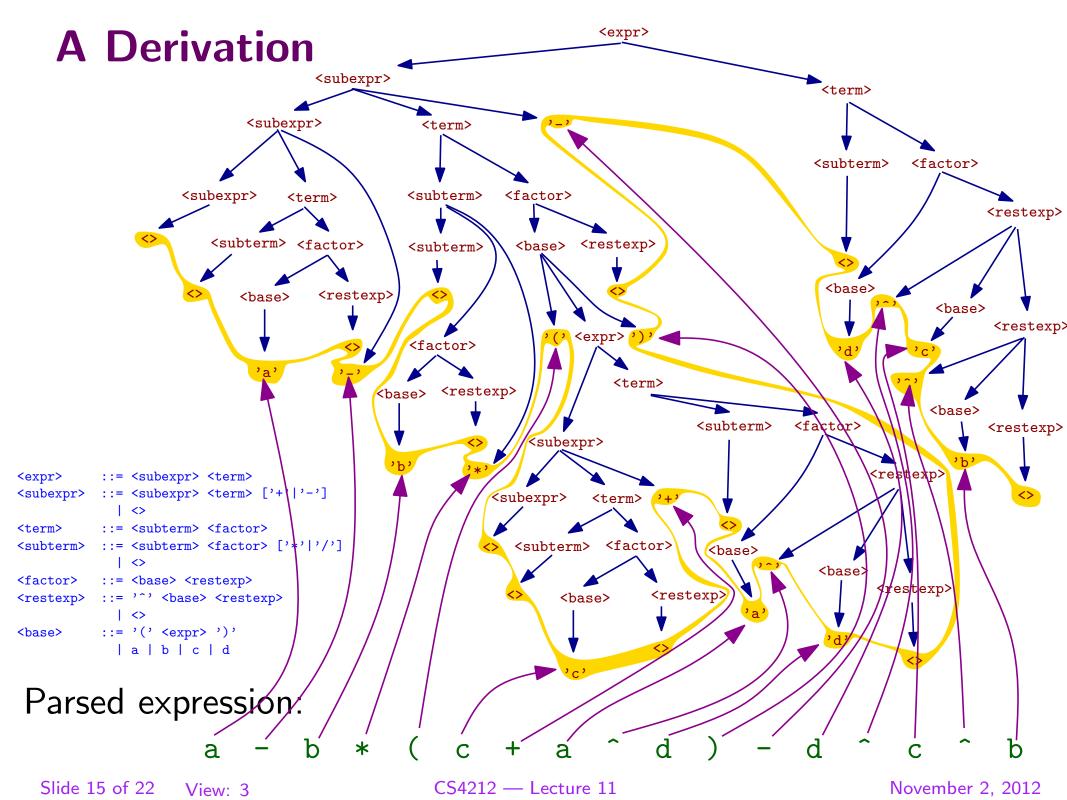


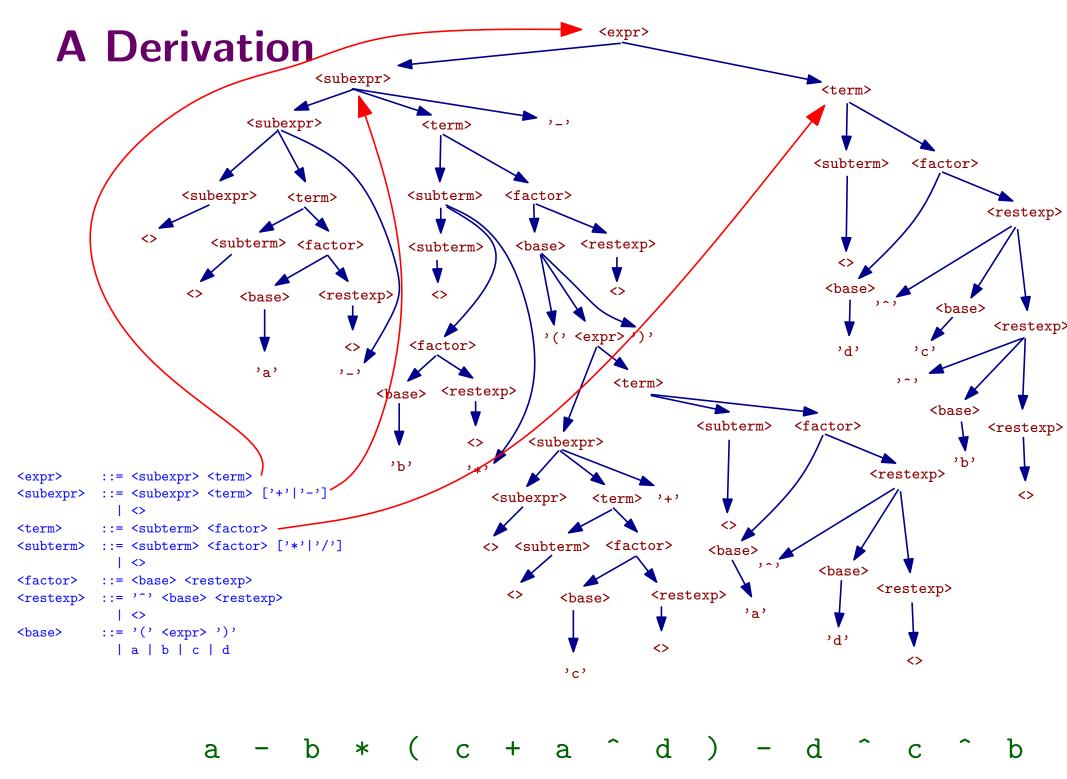
Parsed expression:

 $a - b * (c + a ^ d) - d ^ c ^ b$

Slide 15 of 22 View: 2

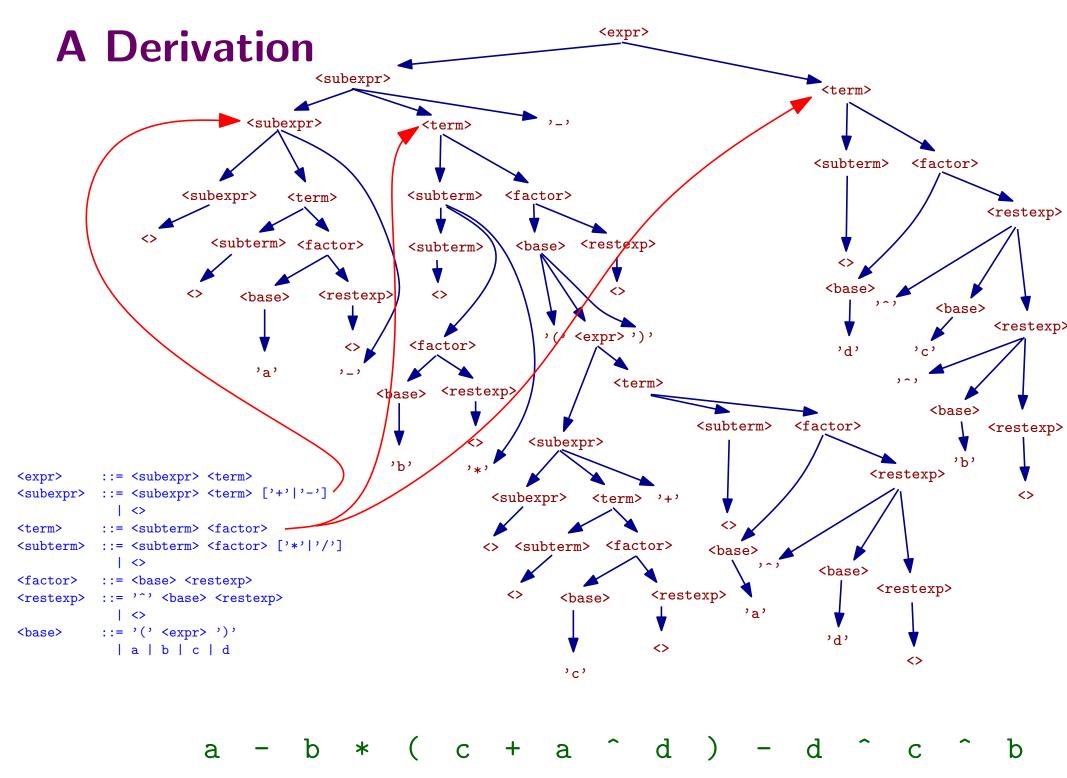
CS4212 — Lecture 11





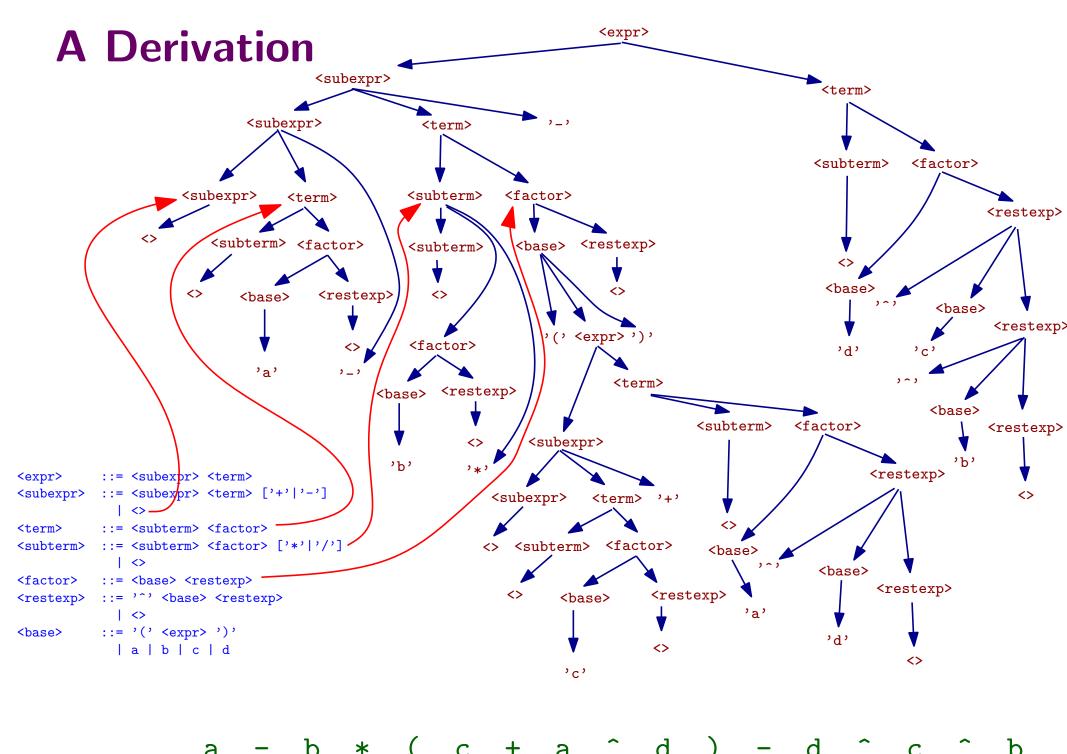
Slide 15 of 22 View: 4

CS4212 — Lecture 11

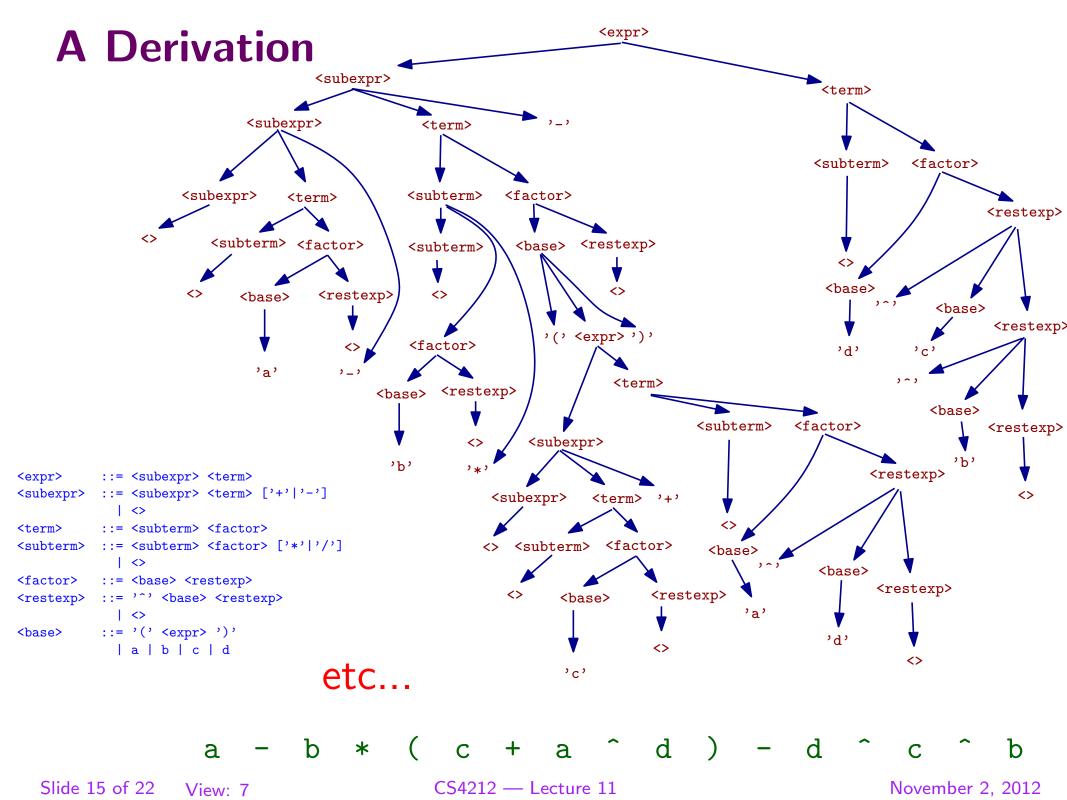


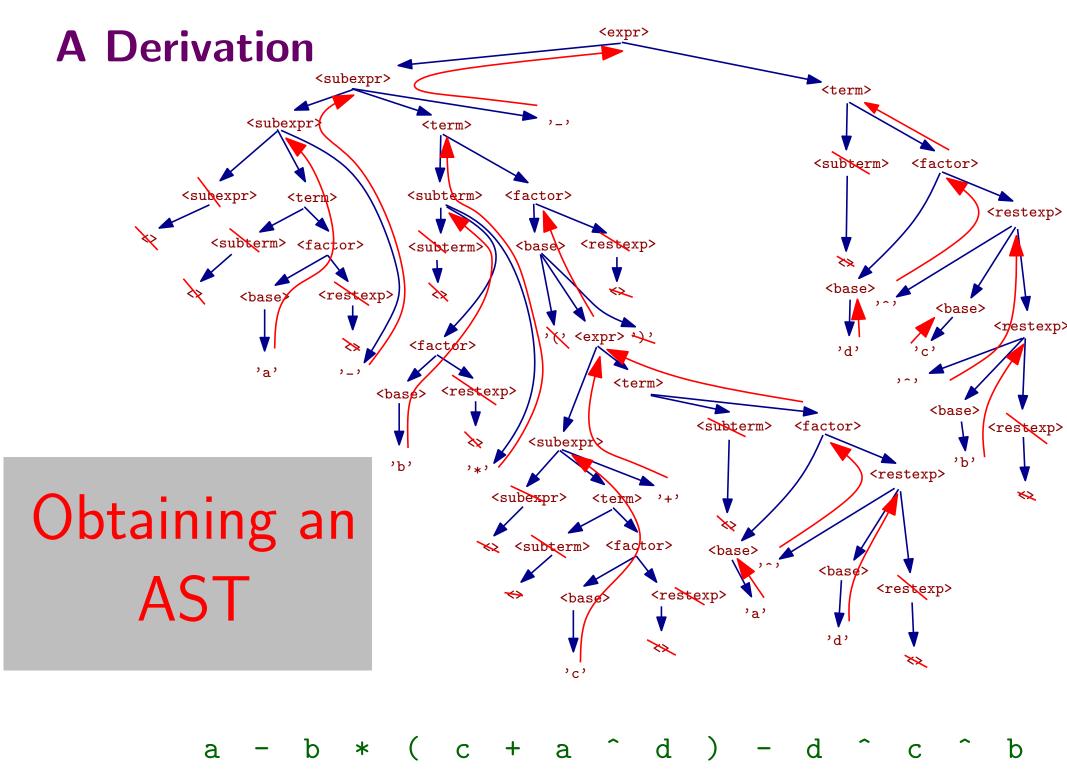
Slide 15 of 22 View: 5

CS4212 — Lecture 11

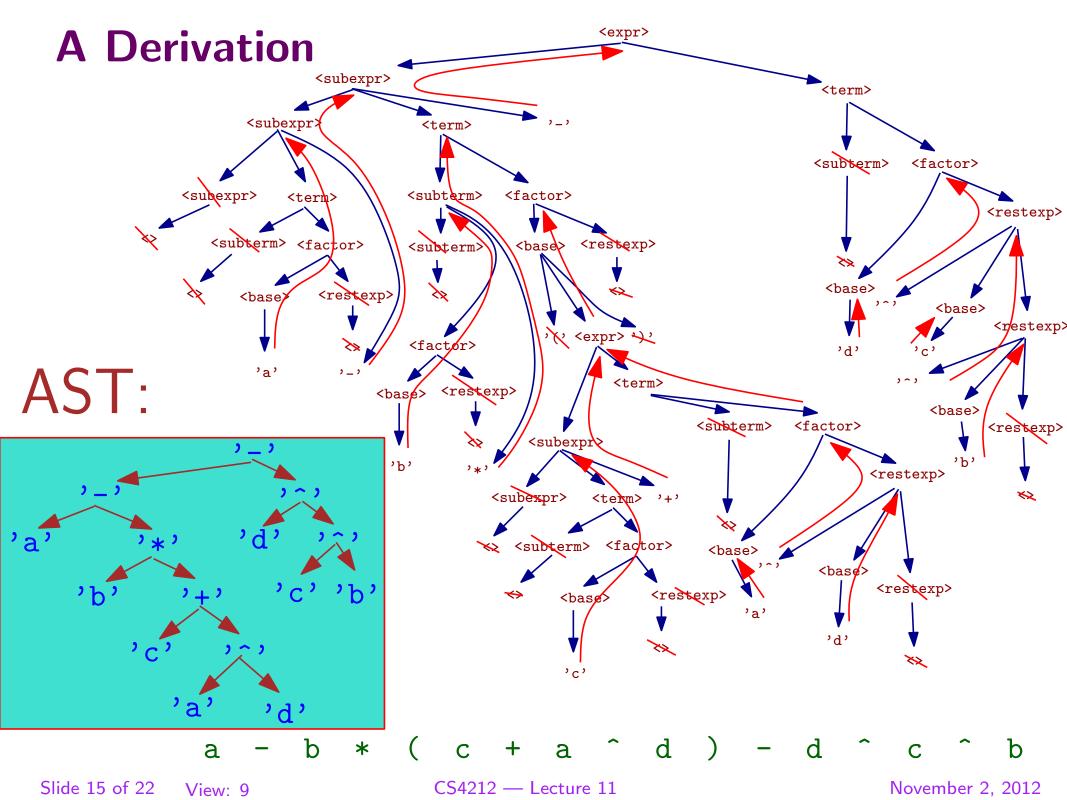


Slide 15 of 22 View: 6 CS4212 — Lecture 11 November 2, 2012





Slide 15 of 22 View: 8 CS4212 — Lecture 11 November 2, 2012



Lessons Learned

- ♦ There are many grammars for the same language.
- ♦ Parsing: Building an AST for a given string, provided it is in the grammar's language.
- Unambiguous grammars are preferred for that purpose.
- Grammar should try to capture structural properties of the language, such as associativity of operators, and nesting of blocks.
- The AST can be built from the parse tree through an algorithmic process.
- Each tree segment is processed in a systematic way.

$$\frac{\langle \operatorname{subexpr} \rangle + S_1 - \langle \operatorname{term} \rangle + S_2}{\langle \operatorname{subexpr} \rangle + S} \qquad s = s_1 s_2$$

$$\frac{\langle \operatorname{subexpr} \rangle + S_1 - \langle \operatorname{term} \rangle + S_2}{\langle \operatorname{subexpr} \rangle + S} \qquad s = s_1 s_2 s_3, \ s_3 \in \{+, -\}$$

$$\frac{\langle \operatorname{subexpr} \rangle + \langle \operatorname{subex$$

$$\frac{\langle \text{subexpr} \rangle \vdash s_1}{\langle \text{expr} \rangle \vdash s} \quad \text{$s = s_1 s_2$}$$

$$\frac{\langle \text{subexpr} \rangle \vdash s_1 \quad \langle \text{term} \rangle \vdash s_2}{\langle \text{subexpr} \rangle \vdash s} \quad s = s_1 s_2 s_3, \ s_3 \in \{+, -\}$$

$$\frac{\langle \text{subterm} \rangle \vdash s_1 \quad \langle \text{factor} \rangle \vdash s_2}{\langle \text{term} \rangle \vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{\langle subterm \rangle} \vdash s_1 \quad \texttt{\langle factor \rangle} \vdash s_2}{\texttt{\langle subterm \rangle} \vdash s} \quad s = s_1 s_2 s_3, \ s_3 \in \{*,/\}$$

<subterm>⊢<>

$$\frac{\text{} \vdash s_1 \quad \text{} \vdash s_2}{\text{} \vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{} \vdash s_1 \quad \texttt{} \vdash s_2}{\texttt{} \vdash \widehat{\ \ } s} \quad s = s_1 s_2$$

$$\frac{\langle \text{expr} \rangle \vdash S_1}{\langle \text{base} \rangle \vdash S}$$
 $s = (s_1)$

$$_{ \texttt{} \vdash s} \quad s \in \{a, \dots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

$$\frac{\langle \operatorname{subexpr} \rangle \vdash s_1 \quad \langle \operatorname{term} \rangle \vdash s_2}{\langle \operatorname{expr} \rangle \vdash s} \qquad s = s_1 s_2$$

$$\frac{\langle \operatorname{subexpr} \rangle \vdash s_1 \quad \langle \operatorname{term} \rangle \vdash s_2}{\langle \operatorname{subexpr} \rangle \vdash s} \qquad s = s_1 s_2 s_3, \ s_3 \in \{+, -\}$$

$$\frac{\langle \operatorname{subexpr} \rangle \vdash \langle \operatorname{subexpr}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

Then string s is generated by nonterminal $\langle expr \rangle$

```
<subterm>⊢<>
```

$$\frac{\langle \mathsf{base} \rangle \vdash s_1}{\langle \mathsf{factor} \rangle \vdash s_2} \qquad s = s_1 s_2$$

$$\frac{\langle \mathsf{base} \rangle \vdash s_1}{\langle \mathsf{subterm} \rangle \vdash \hat{s}} \qquad s = s_1 s_2$$

$$\frac{\langle \mathsf{base} \rangle \vdash s_1}{\langle \mathsf{subterm} \rangle \vdash \hat{s}} \qquad s = s_1 s_2$$

$$\frac{\langle \texttt{expr} \succ \mid s_1}{\langle \texttt{base} \succ \mid s} \quad s = (s_1)$$

$$\frac{\langle \texttt{base} \succ \mid s \mid}{\langle \texttt{base} \succ \mid s \mid} \quad s \in \{a, \dots, z\}$$

$$\frac{\langle \operatorname{subexpr} \rangle \vdash s_1 \quad \langle \operatorname{term} \rangle \vdash s_2}{\langle \operatorname{expr} \rangle \vdash s} \qquad s = s_1 s_2$$

$$\frac{\langle \operatorname{subexpr} \rangle \vdash s_1 \quad \langle \operatorname{term} \rangle \vdash s_2}{\langle \operatorname{subexpr} \rangle \vdash s} \qquad s = s_1 s_2 s_3, \ s_3 \in \{+, -\}$$

$$\frac{\langle \operatorname{subexpr} \rangle \vdash \langle \circ \rangle}{\langle \operatorname{subterm} \rangle \vdash s_1} \qquad \langle \operatorname{factor} \rangle \vdash s_2} \qquad s = s_1 s_2$$

$$\frac{\langle \operatorname{subterm} \rangle \vdash s_1 \quad \langle \operatorname{factor} \rangle \vdash s_2}{\langle \operatorname{subterm} \rangle \vdash s} \qquad s = s_1 s_2 s_3, \ s_3 \in \{*, /\}$$

$$\frac{\langle \operatorname{subterm} \rangle \vdash s_1}{\langle \operatorname{subterm} \rangle \vdash s_2} \qquad s = s_1 s_2 s_3, \ s_3 \in \{*, /\}$$

If string s_1 is generated by nonterminal subexpr>

And string s_2 is generated by nonterminal <term>

Then string s is generated by nonterminal $\langle expr \rangle$

$$\frac{\texttt{} \vdash s_1 \quad \texttt{} \vdash s_2}{\texttt{} \vdash s}$$

$$\frac{\text{} \vdash s_1 \quad \text{} \vdash s_2}{\text{} \vdash \widehat{\ \ \ } s}$$

$$\frac{\langle \texttt{expr} \rangle \vdash s_1}{\langle \texttt{base} \rangle \vdash s} \quad s = (s_1)$$

$$\frac{}{\text{}\vdash s} \quad s \in \{a, \dots, z\}$$

$$s = s_1 s_2$$

$$s = s_1 s_2$$

Where \dot{s} is s_1 concatenated with s_2 .

```
<subexpr>\vdash s_1 <term>\vdash s_2
                                       s = s_1 s_2
            \langle expr \rangle \vdash S
\langle subexpr \rangle \vdash S
 <subexpr>⊢<>
 <subterm> \vdash S_1 <factor> \vdash S_2
                                         s = s_1 s_2
              < term > \vdash S
\langle \text{subterm} \rangle \vdash S
 <subterm>⊢<>
 \langle base \rangle \vdash S_1 \quad \langle restexp \rangle \vdash S_2
                                     s = s_1 s_2
          \overline{\langle factor \rangle} \vdash S
\frac{s_1}{s_2} = \frac{s_1s_2}{s_2}
                                                                      <expr>
                                                                                    ::= <subexpr> <term>
                                                                                   ::= <subexpr> <term> ['+'|'-']
                                                                      <subexpr>
        \langle \text{subterm} \rangle \vdash \widehat{S}
                                                                                      | <>
                                                                                    ::= <subterm> <factor>
                                                                      <term>
<restexp>⊢<>
                                                                      <subterm> ::= <subterm> <factor> ['*'|'/']
                                                                                      | <>
                                                                      <factor> ::= <base> <restexp>
\frac{\langle \text{expr} \rangle \vdash S_1}{\langle \text{base} \rangle \vdash S} s = (s_1)
                                                                      <restexp> ::= '^' <base> <restexp>
                                                                                      | <>
\frac{}{{\sf \langle base \rangle} \vdash s} \quad s \in \{a, \dots, z\}
                                                                                   ::= '(' <expr> ')'
                                                                      <base>
                                                                                      lalblcld
```

```
\langle \text{subexpr} \rangle \vdash s_1 \quad \langle \text{term} \rangle \vdash s_2
                                      s = s_1 s_2
            \langle expr \rangle \vdash S
\langle subexpr \rangle \vdash S
 <subexpr>⊢<>
 < term > \vdash S
\langle \text{subterm} \rangle \vdash S
 <subterm>⊢<>
 \langle base \rangle \vdash s_1 \quad \langle restexp \rangle \vdash s_2
                                    s = s_1 s_2
          \langle factor \rangle \vdash S
\frac{s_1}{s_2} = \frac{s_1s_2}{s_2}
                                                                                  ::= <subexpr> <term>
                                                                     <expr>
                                                                     <subexpr>
                                                                                  ::= <subexpr> <term> ['+'|'-']
        \langle \text{subterm} \rangle \vdash \widehat{s}
                                                                                     | <>
                                                                                  ::= <subterm> <factor>
                                                                     <term>
<restexp>⊢<>
                                                                     <subterm> ::= <subterm> <factor> ['*'|'/']
                                                                                    | <>
                                                                     <factor> ::= <base> <restexp>
\frac{\langle \text{expr} \rangle \vdash S_1}{\langle \text{base} \rangle \vdash S} s = (s_1)
                                                                     <restexp> ::= '^' <base> <restexp>
                                                                                    | <>
\frac{}{{\sf \langle base \rangle} \vdash s} \quad s \in \{a, \dots, z\}
                                                                     <base> ::= '(' <expr> ')'
                                                                                   | a | b | c | d
```

```
\langle subexpr \rangle \vdash s_1 \quad \langle term \rangle \vdash s_2
                                    s = s_1 s_2
           \langle expr \rangle \vdash S
\langle subexpr \rangle \vdash S
 <subexpr>⊢<>
 <subterm> \vdash S_1 <factor> \vdash S_2
                                      s = s_1 s_2
             < term > \vdash S
\langle \text{subterm} \rangle \vdash S
<subterm>⊢<>
\langle factor \rangle \vdash S
\frac{s_1}{s_2} = \frac{s_1s_2}{s_2}
                                                                               ::= <subexpr> <term>
                                                                  <expr>
                                                                               ::= <subexpr> <term> ['+'|'-']
                                                                  <subexpr>
        \langle \text{subterm} \rangle \vdash \widehat{S}
                                                                                 | <>
                                                                               ::= <subterm> <factor>
                                                                  <term>
<restexp>⊢<>
                                                                  <subterm> ::= <subterm> <factor> ['*'|'/']
                                                                                | <>
                                                                  <factor>
                                                                              ::= <base> <restexp>
\frac{\langle \text{expr} \rangle \vdash S_1}{\langle \text{base} \rangle \vdash S} s = (s_1)
                                                                  <restexp> ::= '^' <base> <restexp>
                                                                                 | <>
\frac{}{{\sf \langle base \rangle} \vdash s} \quad s \in \{a, \dots, z\}
                                                                  <base> ::= '(' <expr> ')'
                                                                                | a | b | c | d
```

```
<subexpr>\vdash S_1 <term>\vdash S_2
                                       s = s_1 s_2
            \langle expr \rangle \vdash S
\langle subexpr \rangle \vdash S
 <subexpr>⊢<>
 <subterm> \vdash S_1 <factor> \vdash S_2
                                         s = s_1 s_2
              < term > \vdash S
\langle \text{subterm} \rangle \vdash S
 <subterm>⊢<>
 \langle base \rangle \vdash S_1 \quad \langle restexp \rangle \vdash S_2
                                     s = s_1 s_2
          \langle factor \rangle \vdash S
\frac{s_1}{s_2} = \frac{s_1s_2}{s_2}
                                                                                   ::= <subexpr> <term>
                                                                      <expr>
                                                                                   ::= <subexpr> <term> ['+'|'-']
                                                                      <subexpr>
        \langle \text{subterm} \rangle \vdash \widehat{S}
                                                                                      | <>
                                                                                   ::= <subterm> <factor>
                                                                      <term>
<restexp>⊢<>
                                                                      <subterm> ::= <subterm> <factor> ['*'|'/']
                                                  etc...
                                                                                     | <>
                                                                      <factor>
                                                                                   ::= <base> <restexp>
\frac{\langle \text{expr} \rangle \vdash S_1}{\langle \text{base} \rangle \vdash S} s = (s_1)
                                                                      <restexp> ::= '^' <base> <restexp>
                                                                                     | <>
\frac{}{{\sf \langle base \rangle} \vdash s} \quad s \in \{a, \dots, z\}
                                                                      <base> ::= '(' <expr> ')'
                                                                                     | a | b | c | d
```

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).
subexpr("").
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]).
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 = < S, S = < 122.
```

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).
subexpr("").
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1), term(S2), member([S1,S2,0],S).
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 = < S, S = < 122.
```

Prolog double quoted term: list of ASCII codes:

"abc" = [97,98,99]

```
expr(S) :=
    append(S1,S2,S), subexpr(S1), term(S2).
subexpr("").
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]).
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 = < S, S = < 122.
```

Prolog double quoted term: list of ASCII codes:

List of lists

"abc" = [97,98,99]

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).
subexpr("").
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]).
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
                                                     \langle subexpr \rangle \vdash S_1
                                                                       <term>\vdash S_2
    97 = < S, S = < 122.
                                                                                       s = s_1 s_2
                                                               <expr><math>\vdash S
```

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).
subexpr("").
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]).
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 = < S, S = < 122.
                                                          <term>\vdash S_2
                                       \langle subexpr \rangle \vdash S_1
                                                                          s = s_1 s_2 s_3, s_3 \in \{+, -\}
                                                \langle subexpr \rangle \vdash S
```

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).
                                                                                 <subexpr>⊢<>
subexpr("").
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]).
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 = < S, S = < 122.
```

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).
                       Empty string is prefix of empty string.
subexpr("")
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]).
                               Left recursion
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
                                                    Runs into infinite loop !!!
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
   97 = < S, S = < 122.
```

```
expr(S) :=
    append(S1,S2,S), subexpr(S1), term(S2).
subexpr("").
subexpr(S) :-
    append([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]).
term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).
subterm("").
subterm(S):-
    append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]).
factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).
restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).
base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 = < S, S = < 122.
```

Add heuristics!

Non-empty
Balanced brackets
No + or - outside
bracktes

Add heuristics! expr(S) :=append(S1,S2,S), subexpr(S1), term(S2). Non-empty Balanced brackets subexpr(""). subexpr(S) :-No + or - outsideappend([S1,S2,0],S), subexpr(S1),term(S2),member(0,["+","-"]). bracktes term(S) :append(S1,S2,S), subterm(S1), factor(S2). subterm(""). subterm(S):append([S1,S2,0],S), subterm(S1),factor(S2),member(0,["*","/"]). Non-empty factor(S) :append(S1,S2,S), base(S1), restexp(S2). Balanced brackets No * or / outside restexp(""). restexp(S) :bracktes append($["^",S1,S2],S$), base(S1), restexp(S2). base(S) :-Non-empty append(["(",S1,")"],S), expr(S1). base([S]) :-Balanced brackets 97 = < S, S = < 122.No outside bracktes

```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).
subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :=
    S1 = [\_|\_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
         ( 0 = [] \rightarrow member(0,0L) ; true ).
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"])
    !,subexpr(S1), term(S2).
subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :=
    S1 = [ | ], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
             0 = [] \rightarrow member(0,0L) ; true ).
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"])
    !, subexpr(S1), term(S2).
                       S2 is non-empty
subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
   _{S1} = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
             0 = [] \rightarrow member(0,0L) ; true ).
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"])
    !, subexpr(S1), term(S2).
subexpr("") :- !.
                   S is concatenation of S1 and S2
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"])
    !, subexpr(S1) term(S2).
subexpr("") :- !.
                    No + or - outside brackets in S2
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [ _ | _ ], append(L,S), balanced(S1,R1),
    findall(X, (member([X], OL), member(X, R1)), []),
             0 = [] \rightarrow member(0,0L) ; true ).
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Watch demo on findall and append/2

```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"])
    !.subexpr(S1), term(S2).
subexpr("") :- !. S2 should contain balanced brackets only
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :=
    S1 = [ | ], append(L,S), balanced(S1,R1),
    findall(X (mamban([X],CL),member(X,R1)),[]),
             0 = [] \rightarrow member(0,0L) ; true ).
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

```
expr(S) :=
                                                        constrain(S,S1,0,L,OL) :=
    constrain(S,S2,[],[S1,S2],["+","-"]),
                                                            S1 = [ ], append(L,S), palanced(S1,R1),
    !,subexpr(S1), term(S2).
                                                            findall(X,(member([X],OL),member(X,R1)),[]),
                                                                     0 = [] \rightarrow member(0,0L) ; true ).
subexpr("") :- !.
subexpr(S) :-
                                                        balanced("","") :- !.
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
                                                        balanced(S,"") :-
    !, subexpr(S1), term(S2).
                                                            append(["(",S1,")"],S),balanced(S1,_),!.
                                                        balanced(S,R) :-
term(S) :-
                                                            append([X],S1,S), \+ member([X],["(",")"]),!,
    constrain(S,S2,[],[S1,S2],["*","/"]
                                                            balanced(S1,R1), append([X],R1,R).
    !, subterm(S1), factor(S2).
                                                        balanced(S,R) :-
                                                            append(S1,[X],S), \+ member([X],["(",")"]),!,
subterm("") :- !.
                                                            balanced(S1,R1), append(R1,[X],R).
subterm(S):-
                                                        balanced(S,R) :-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
                                                            append(["(",S1,")",S2,"(",S3,")"],S),
    !, subterm(S1), factor(S2).
                                                            balanced(S1,_),balanced(S2,R),balanced(S3,_).
factor(S) :-
                                            S is S1 concatenated with S2 and O, where O is either + or -
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).
subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [ | ], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
         ( 0 = [] \rightarrow member(0,0L) ; true ).
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Apply heuristics in all places where it is useful!

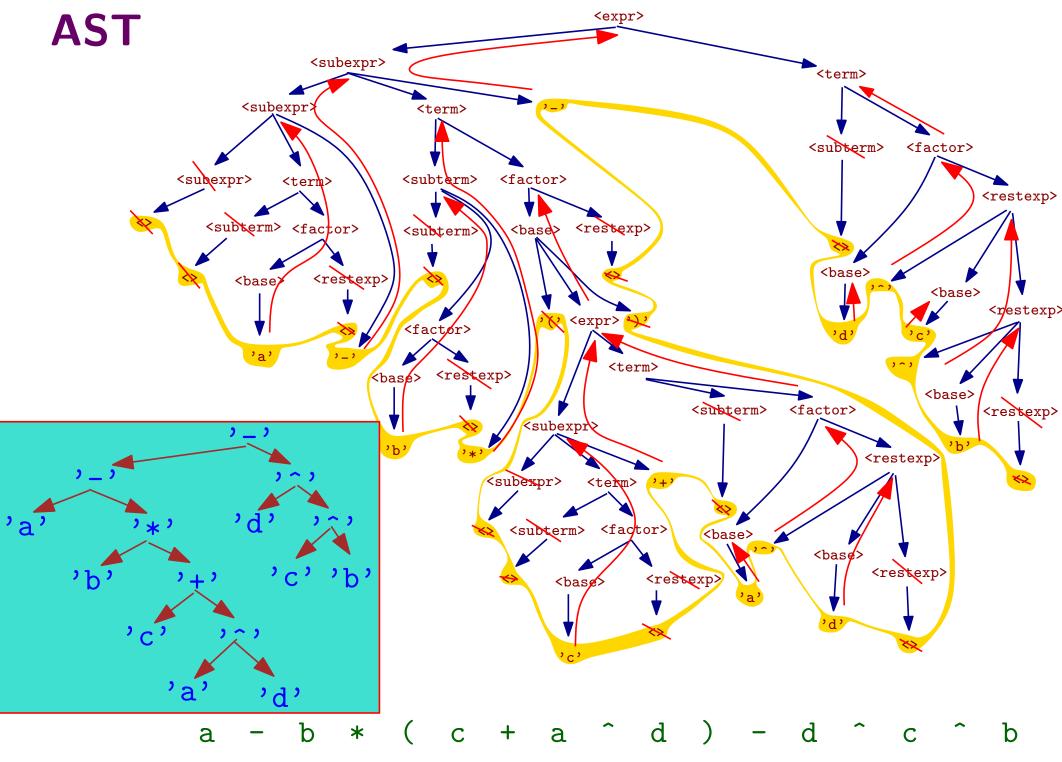
```
expr(S) :=
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).
subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !, subexpr(S1), term(S2).
term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1), factor(S2).
subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1), factor(S2).
factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).
restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).
base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [\_|\_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
         ( 0 = [] \rightarrow member(0,0L) ; true ).
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

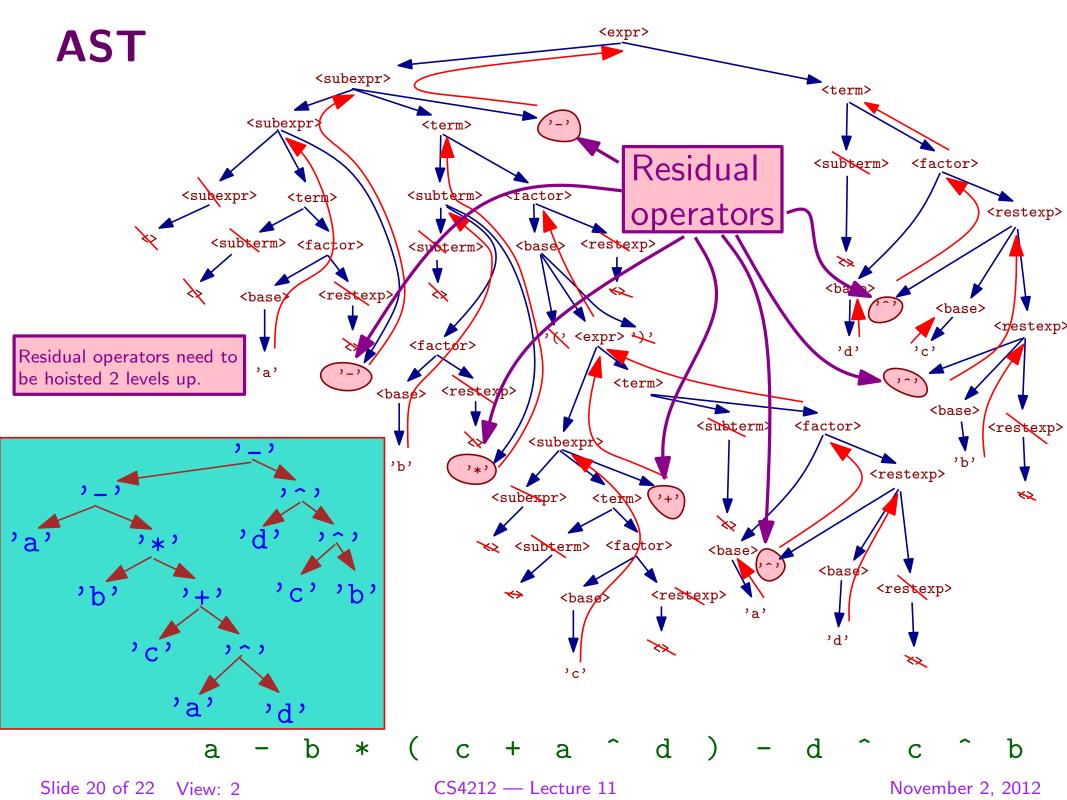
```
Query:

1 ?- S="(((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b)", expr(S).

S = [40, 40, 40, 97, 43, 98, 41, 42, 99|...].
```



Slide 20 of 22 View: 1 CS4212 — Lecture 11 November 2, 2012



```
expr(S,T) :-
                                                      restexp("",nil,nil) :- !.
    constrain(S,S2,[],[S1,S2],["+","-"]),
                                                      restexp(S,T,^) :-
    !, subexpr(S1,T1,O1), term(S2,T2),
                                                          constrain(S,S1,"^",["^",S1,S2],["^"]),
    build(T,T1,T2,[01,T1,T2]).
                                                           !, base(S1,T1), restexp(S2,T2,O2),
                                                          build(T,T2,T1,[02,T1,T2]).
subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
                                                      base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
                                                      base([S],A) :- 97 =< S, S =< 122, char_code(A,S).
    char_code(Op,O),
    !, subexpr(S1,T1,O1),term(S2,T2),
                                                      build(T,nil,T,_) := !.
    build(T,T1,T2,[01,T1,T2]).
                                                      build(T,_{-},_{-},L) :- T = ... L.
term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1, T1, O1), factor(S2, T2),
    build(T,T1,T2,[01,T1,T2]).
subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1,T1,O1), factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[01,T1,T2]).
factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[02,T1,T2]).
```

```
expr(S,T) :-
                                                      restexp("",nil,nil) :- !.
    constrain(S,S2,[],[S1,S2],["+","-"]),
                                                      restexp(S,T,^) :-
    !, subexpr(S1,T1,O1), term(S2,T2),
                                                          constrain(S,S1,"^",["^",S1,S2],["^"]),
    build(T,T1,T2,[01,T1,T2]).
                                                          !, base(S1,T1), restexp(S2,T2,O2),
                                                          build(T,T2,T1,[02,T1,T2]).
subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
                                                      base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
                                                      base([S],A) :- 97 =< S, S =< 122, char_code(A,S).
    char_code(Op,O),
    !, subexpr(S1,T1,O1),term(S2,T2),
                                                      build(T,nil,T,_) := !.
    build(T,T1,T2,[01,T1,T2]).
                                                      build(T,_,_,L) :- T = ... L.
term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1, T1, O1), factor(S2, T2),
    build(T,T1,T2,[01,T1,T2]).
subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1,T1,O1), factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[01,T1,T2]).
factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
```

Piggyback on the syntax

build(T,T2,T1,[02,T1,T2]).

```
Building an AST
expr(S,T) :=
                                                     restexp(",nil,nil) :- !.
    constrain(S, SZ, [], [S1, 52]
                                                     restexp(S,T,^) :-
    !, subexpr(S_2, T_1, O_1) term(S_2, T_2)
                                                         constrain(S,S1,"^",["^",S1,S2],["^"]),
    build(T,T1,T2,[01,T1,T2])
                                                          !, base(S1,T1), restexp(S2,T2,O2),
                                                         build(T,T2,T1,[02,T1,T2]).
subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
                                                     base(S_1) :- append(["(",S1,")"],S), !, expr(S1,T).
    constrain(S,S2,[0]/[S1/S2/[0]],["+","-"]),
                                                     base([S],A) :- 97 =< S, S =< 122, char_code(A,S).
    char_code(Op,O),
    !, subexpr(S1,T1,O1), term($2,T2),
                                                     build(T,nil,T,_) :- !.
    build(T,T1,T2,[01,T1,T2])
                                                     build(T,_,_,L) :- T = ... L.
                                          Second argument is where the AST is output
term(S,T) :-
    constrain(S,S2, [1, [$1,S1], ["*","/"]),
    !, subterm(S1, T1, O1), factor(S2, T2),
    build(T,T1,T2,[D1,T1,T2]).
subterm("",nil,nil) :-
subterm(S,T,Op):-
    constrain(S,S2, [0], [S1,S2,[0]], ["*", "/"]),
                                                              Piggyback on the syntax
    !, subterm(S1, 71, 01), factor(S2, T2), char_code(Op, 0),
    build(T,T1,T2,[01,T1,T2]).
                                                              analyzer
factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
```

build(T,T2,T1,[02,T1,T2]).

Slide 21 of 22 View: 3 CS4212 — Lecture 11 November 2, 2012

```
expr(S,T) :=
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !, subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[01,T1,T2]).
subexpr("",nil,nil) :- !.
subexpr(S,T,Qp)
    constrain [8, S2, [8], [S1, S2, [0]], ["+", "-"]),
    char_code(Op,O),
    !, subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[01,T1,T2]).
term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"])
    !, subterm(S1, T1, O1), factor(S2, T2),
    build(T,T1,T2,\{01,T1,T2\}).
subterm("",nil,11) :-!.
subterm(S,T,Op):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1,T1,O1), factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[01,T1,T2]).
factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[02,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .</pre>
```

Residual operator

Third argument of some nonterminals

Piggyback on the syntax analyzer

```
expr(S,T) :=
                                                      restexp("",nil,nil) :- !.
    constrain(S,S2,[],[S1,S2],["+","-"]),
                                                      restexp(S,T,^) :-
    !, subexpr(S1,T1,O1), term(S2,T2),
                                                          constrain(S,S1,"^",["^",S1,S2],["^"]),
    build(T,T1,T2,[01,T1,T2]).
                                                          !, base(S1,T1), restexp(S2,T2,O2),
                                                          build(T,T2,T1,[02,T1,T2]).
subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
                                                      base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
                                                      base([S],A) :- 97 =< S, S =< 122, char_code(A,S)
    char_code(Op,O),
    !, subexpr(S1,T1,O1),term(S2,T2),
                                                      build(T,nil,T,_) := !.
    build(T,T1,T2,[01,T1,T2]).
                                                      build(T,_,_,L) :- T = ... L.
term(S,T) :=
                                                                            Convert ASCII code S into atom A
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1, T1, O1), factor(S2, T2),
    build(T,T1,T2,[01,T1,T2]).
subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1,T1,O1), factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[01,T1,T2]).
```

Piggyback on the syntax

constrain(S,S1,[],[S1,S2],["^"]), !,base(S1,T1), restexp(S2,T2,O2),

build(T,T2,T1,[02,T1,T2]).

factor(S,T) :-

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !, subexpr(S1,T1,O1), term(S2,T2),
   build(T,T1,T2,[01,T1,T2]).
subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    char_code(Op,O),
    !, subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[01,T1,T2])
term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !, subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[01,T1,T2]).
subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !, subterm(S1,T1,O1), factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[01,T1,T2])
factor(S,T) :-
```

```
factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
   !,base(S1,T1), restexp(S2,T2,O2),
   build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,02),
    build(T,T2,T1,[02,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .</pre>
```

Piggyback on the syntax analyzer

Tree building:

- ♦ If the residual operator is nil, just pass the current tree up.
- ♦ If the residual operator is not nil, then L contains the tree components, which must be assembled into a term.

```
expr(S,T) :=
                                                      restexp("",nil,nil) :- !.
    constrain(S,S2,[],[S1,S2],["+","-"]),
                                                      restexp(S,T,^) :-
    !, subexpr(S1,T1,O1), term(S2,T2),
                                                          constrain(S,S1,"^",["^",S1,S2],["^"]),
    build(T,T1,T2,[01,T1,T2]).
                                                          !, base(S1,T1), restexp(S2,T2,O2),
                                                          build(T,T2,T1,[02,T1,T2]).
subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
                                                      base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
                                                      base([S],A) :- 97 =< S, S =< 122, char_code(A,S).
    char_code(Op,O),
    !, subexpr(S1,T1,O1),term(S2,T2),
                                                      build(T,nil,T,_) := !.
    build(T,T1,T2,[01,T1,T2]).
                                                      build(T,_{-},_{-},L) :- T = ... L.
term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
                                                         Query:
    !, subterm(S1,T1,O1), factor(S2,T2),
                                                         1 ?- S="(((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b)",
                                                              expr(S,T), T = ... L, S = ... X.
    build(T,T1,T2,[01,T1,T2]).
                                                         S = [40, 40, 40, 97, 43, 98, 41, 42, 99]...],
                                                         T = (((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b),
subterm("",nil,nil) :- !.
                                                         L = [*, ((a+b)*c/d^e^f-g)^ (a*b)+c, a+b],
subterm(S,T,Op):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
                                                         X = ['.', 40, [40, 40, 97, 43, 98, 41|...]].
    !, subterm(S1,T1,O1), factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[01,T1,T2]).
factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[02,T1,T2]).
```

YACC: separate set of slides