CISS 445 Programming Languages

51. Grammars 1

Dr. Liow

Context Free Grammars

- Grammars are rules for generating strings.
- Example:
  - Consider the regex [1-9][0-9]* (i.e. positive integers). This is written as a regex. You can describe the same set of words in terms of a DFA or NFA.
  - Another way is to define a grammar.
    - $S \rightarrow 1T \mid 2T \mid 3T \mid 4T \mid 5T \mid 6T \mid 7T \mid 8T \mid 9T$
    - $T \rightarrow \varepsilon \mid 0T \mid 1T \mid 2T \mid 3T \mid 4T \mid 5T \mid 6T \mid 7T \mid 8T \mid 9T$

Context Free Grammars

- How to use the grammar
  - $S \rightarrow 1T \mid 2T \mid 3T \mid 4T \mid 5T \mid 6T \mid 7T \mid 8T \mid 9T$
  - $T \rightarrow \varepsilon \mid 0T \mid 1T \mid 2T \mid 3T \mid 4T \mid 5T \mid 6T \mid 7T \mid 8T \mid 9T$
- There are two kinds of symbols. Terminating symbols are symbols making up the words you want to generate.
- Nonterminating symbols are “variables”. There is a starting nonterminating symbol. Usually S.

Context Free Grammars

- You start with symbol S.
- Using the rules to replace variables to get new strings. You stop when there are no more non-terminating symbols (“variables”).
- Example:
- $\underline{S}$ => $2\underline{T}$ using the rule $S \to 2T$

  => $21\underline{T}$ using the rule $T \to 1T$

  => 21 using the rule $T \to \varepsilon$

Context Free Grammars

- S => 2T => 21T => 21 is a **derivation**
- Here's another derivation:

  <u>S</u> => 1<u>T</u> => 12<u>T</u> => 123<u>T</u> => 1234<u>T</u> => 1234
- Make sure you see the difference between a derivations and rules.

Context Free Grammars

- **Example.** Here's another example.

$$S \rightarrow 0S1 \mid \varepsilon$$

- There's only one rule.
- Here are some derivations:

$\underline{S} \Rightarrow \varepsilon$

$\underline{S} \Rightarrow 0\underline{S}1 \Rightarrow 00\underline{S}11 \Rightarrow 00\varepsilon11 \Rightarrow 0011$

$\underline{S} \Rightarrow 0\underline{S}1 \Rightarrow 00\underline{S}11 \Rightarrow 000\underline{S}111 \Rightarrow 000111$

Exercise

- Terminals: 0 1 + ( )
- Nonterminals: S (the start symbol)
- $S \rightarrow 0 \mid 1 \mid S + S \mid (S)$ (4 rules)
- Write down some derivations. Make sure you use all the rules.
- Is it possible to derive the string 1+1+1 from the grammar?
- Is is possible to derive the string ((1)+(1+1)) from the grammar?

Definitions

- A **context-free grammar** (CFG) is just like the above
  - A set of **nonterminating symbols** ("variables"). There is a starting symbol from this set of symbols.
  - A set of **terminating symbols**: These are symbols making up the words you want to generate.
  - A set of **rules** (**productions**) of the form

$$X \rightarrow x$$

where X is a nonterminating symbols and x is a string made up of nonterminating and terminating symbols

Definitions

- A word (of terminating symbols) is derived from a CFG if when starting with the start symbol, you can find productions to replace the variables until you reach that word.
- The sequence of strings to derive that word is called a ***derivation***.

Derivations

- Here's a CFG:
- S → Noun Verb Adjective Noun

  Noun → boy | girl | dog

  Verb → dates | emails | texts | walks

  Adjective → pretty | impatient | noisy
- Here are some words derived from the grammar:

  boy dates pretty girl

  dog walks noisy boy

Derivations

- Here's another CFG:
- $S \rightarrow S + S \mid S * S \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- Here are some derivation:
- $\underline{S} \Rightarrow \underline{S} + S \Rightarrow \underline{S} * S + S \Rightarrow 2 * \underline{S} + S$
  $\Rightarrow 2 * 5 + \underline{S} \Rightarrow 2 * 5 + 7$
- $\underline{S} \Rightarrow S * \underline{S} \Rightarrow S * S + \underline{S} \Rightarrow S * \underline{S} + 7$
  $\Rightarrow \underline{S} * 5 + 7 \Rightarrow 2 * 5 + 7$

Derivations

- $\underline{S} \Rightarrow S * \underline{S} \Rightarrow S * \underline{S} + S \Rightarrow \underline{S} * 5 + S$
  $\Rightarrow 2 * 5 + \underline{S} \Rightarrow 2 * 5 + 7$
- Note that
  - The first derivation picks the leftmost variable for replacement. This is a **leftmost derivation**.
  - The second derivation picks the rightmost variable for replacement. This is a **rightmost derivation**.

BNF Grammars

- There is another way to write the rules. Instead of writing $X \to x$, we can write

$$X ::= x$$

- **Backus-Naur forms (BNF)** are the same as CFG except that the rules are written

$$X ::= x$$

instead of $X \to x$

Example

- Terminals: 0 1 + ( )
- Nonterminals: S (also the start symbol)
- Rules: S ::= 0 | 1 | S + S | (S)
- Here's a rightmost derivation
- $\underline{S} \Rightarrow S + \underline{S}$

  $\Rightarrow S + (S + \underline{S})$

  $\Rightarrow S + (\underline{S} + 1)$

  $\Rightarrow \underline{S} + (0 + 1)$

  $\Rightarrow 1 + (0 + 1)$

Example

- Write down the rightmost derivation of

$$(1 + 0) + 1.$$

Example

- Write down the rightmost derivation of

$$(1 + (1 + (1 + 0))).$$

Example

- Write down the rightmost derivation of

  (((1 + 1) + 1) + 0).

Notation for Nonterminals

- In order to make grammars easier to read instead of a single character for nonterminals, you will find <…> where … is a descriptive word.
- Example:
- <expr> ::= <if-expr> | <let-expr> |…
- <if-expr> ::= if <bool-expr> then <expr> else
  <expr>
- …

Example

- Here's a grammar:

```
<expr>  ::= <factor>
          |  <factor> +  <factor>
<factor>  ::=  <bin>
            |  <bin>  *  <expr>
<bin> ::=  0  | 1
```

- Write down a derivation for 1 * 1 + 0

EBNF

- Some rules occur frequently. Extended BNF makes writing them less painful.
- Options [ ]: X::=*y*[*v*]*z*
  - Shorthand for X::=*yvz* | *yz*
  - I.e., [v] mean v U ε, i.e. v is optional
- Repetition { }*: X::=*y*{*v*}**z*
  - Shorthand for X::=*yz* | yVz, V::=*v* | *v*V where V is a new symbol.

Regular Grammars

- There is a subclass of BNF where the rules are of the form

  <nonterminal>::=<terminal><nonterminal>

  or

  <nonterminal>::=<terminal>
- Such grammars are called **regular grammars**
- The languages generated by grammars are the same as the languages generated by regex.

Example

- Regular grammar:

  <Balanced> ::= ε

  <Balanced> ::= 0<One>

  <Balanced> ::= 1<Zero>

  <One> ::= 1<Balanced>

  <Zero> ::= 0<Balanced>

Example

- Write down all the strings of lengths < 4 generated by this grammar.

Example

- Describe this grammar in words.

Parse Trees

- We can describe a derivation using a **tree**.
- Consider this grammar (see prev slide):

  <expr> ::= <factor>
  | <factor> + <factor>
  <factor> ::= <bin>
  | <bin> * <expr>
  <bin> ::= 0 | 1
- Draw the parse tree for 1 * 1 + 0 using rightmost derivation.

Parse Trees

<expr>

<expr>

Parse Trees

<expr>

=> <factor>

<expr>

|

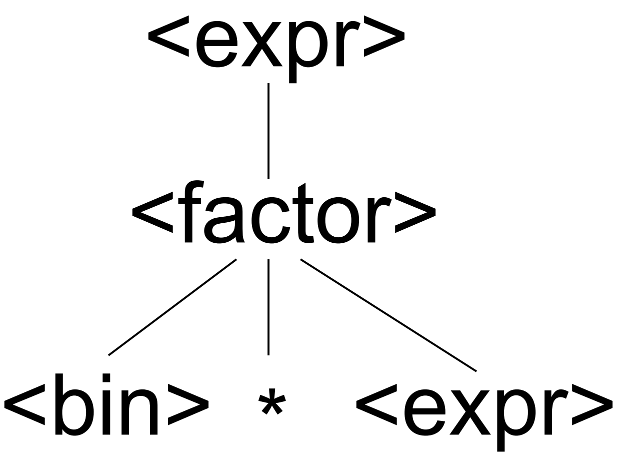
<factor>

Parse Trees

<expr>

=> <factor>

=><bin>*<expr>



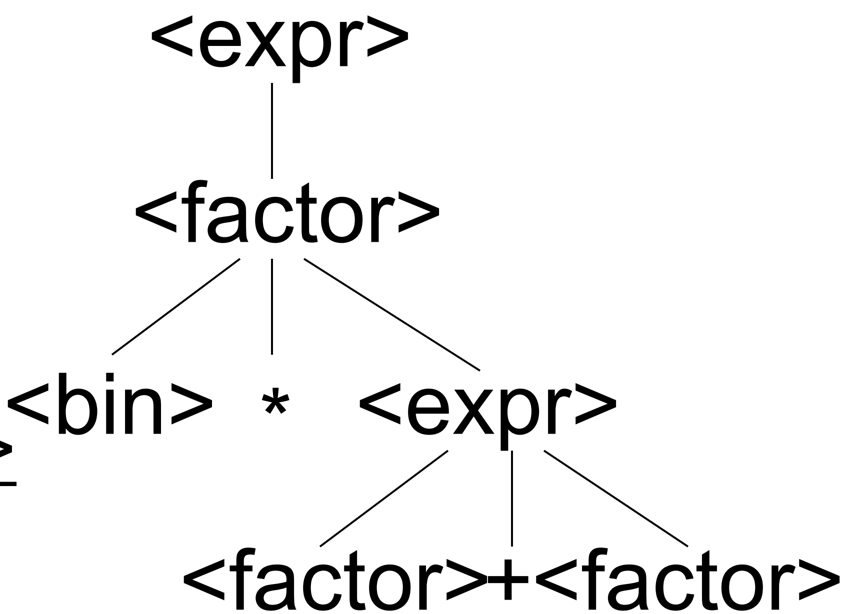
Parse Trees

<expr>

=> <factor>

=><bin>*<expr>

=><bin>*<factor>+<factor>



Parse Trees

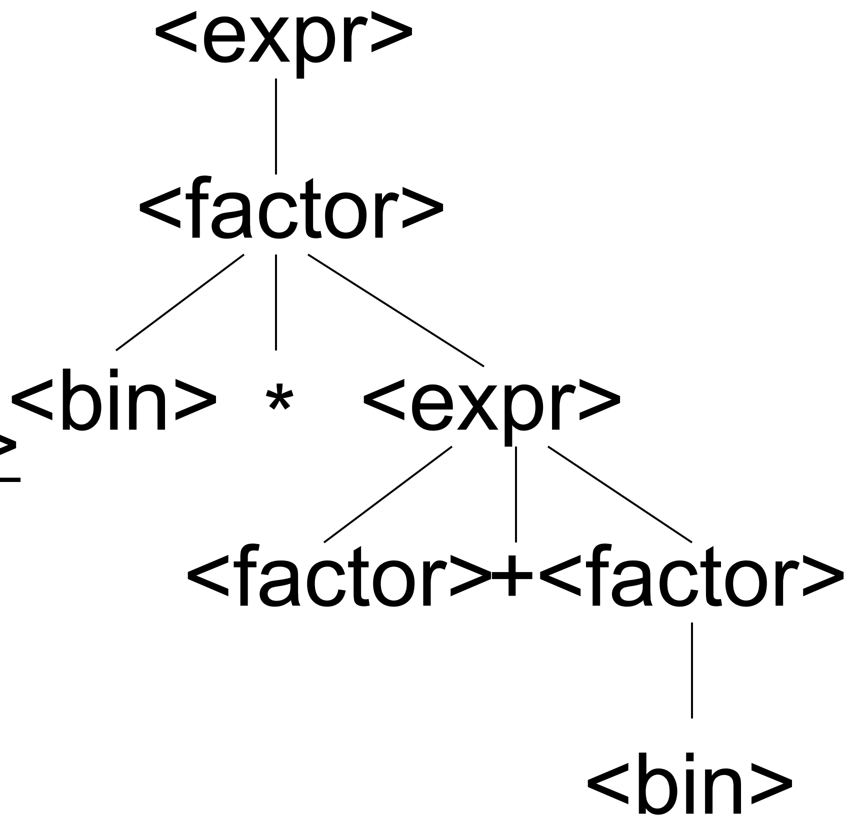
<expr>

=> <factor>

=><bin>*<expr>

=><bin>*<factor>+<factor>

=><bin>*<factor>+<bin>



Parse Trees

<expr>

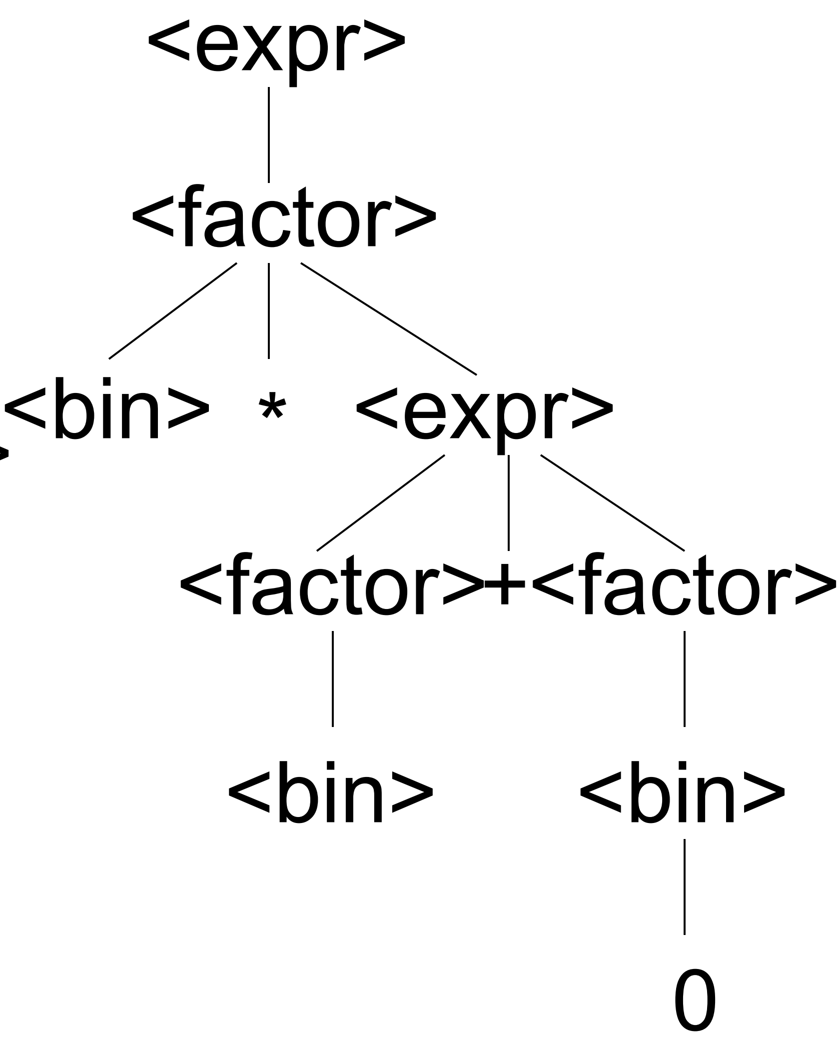
=> <factor>

=><bin>*<expr>

=><bin>*<factor>+<factor>

=><bin>*<factor>+<bin>

=><bin>*<bin>+0



Parse Trees

<expr>

=> <factor>

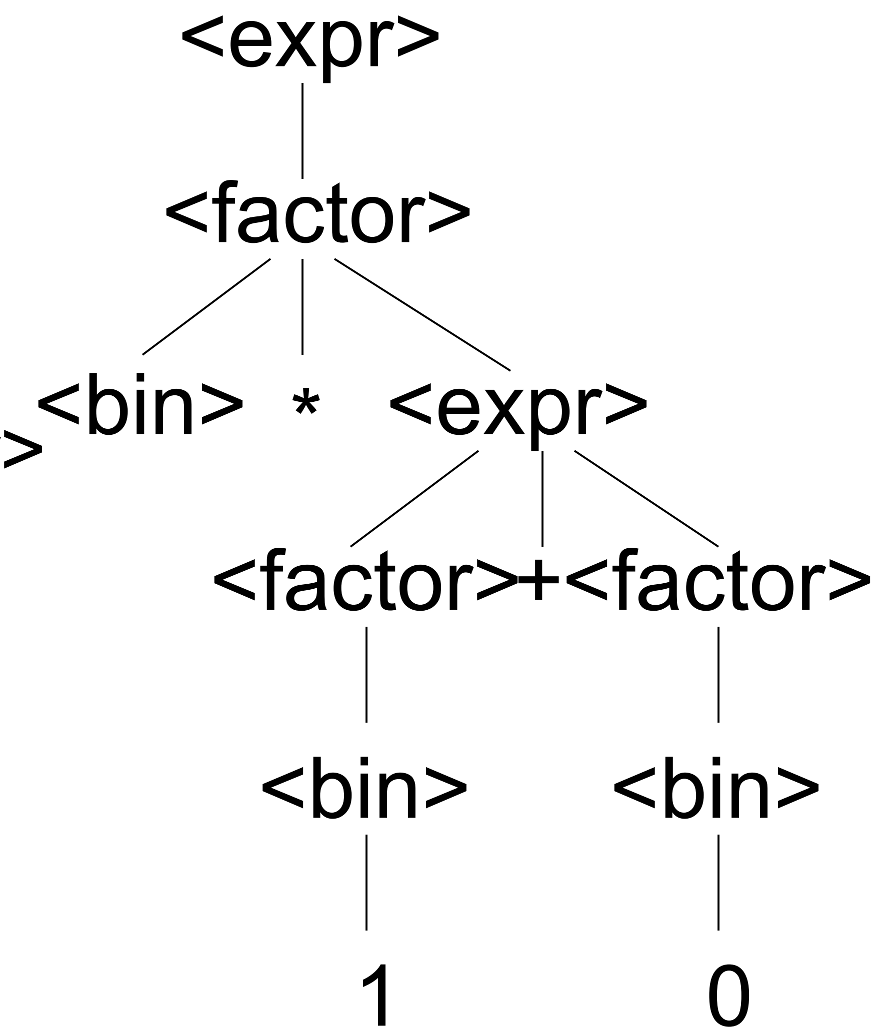
=><bin>*<expr>

=><bin>*<factor>+<factor>

=><bin>*<factor>+<bin>

=><bin>*<bin>+0

=><bin>*1+0



Parse Trees

<expr>

=> <factor>

=><bin>*<expr>

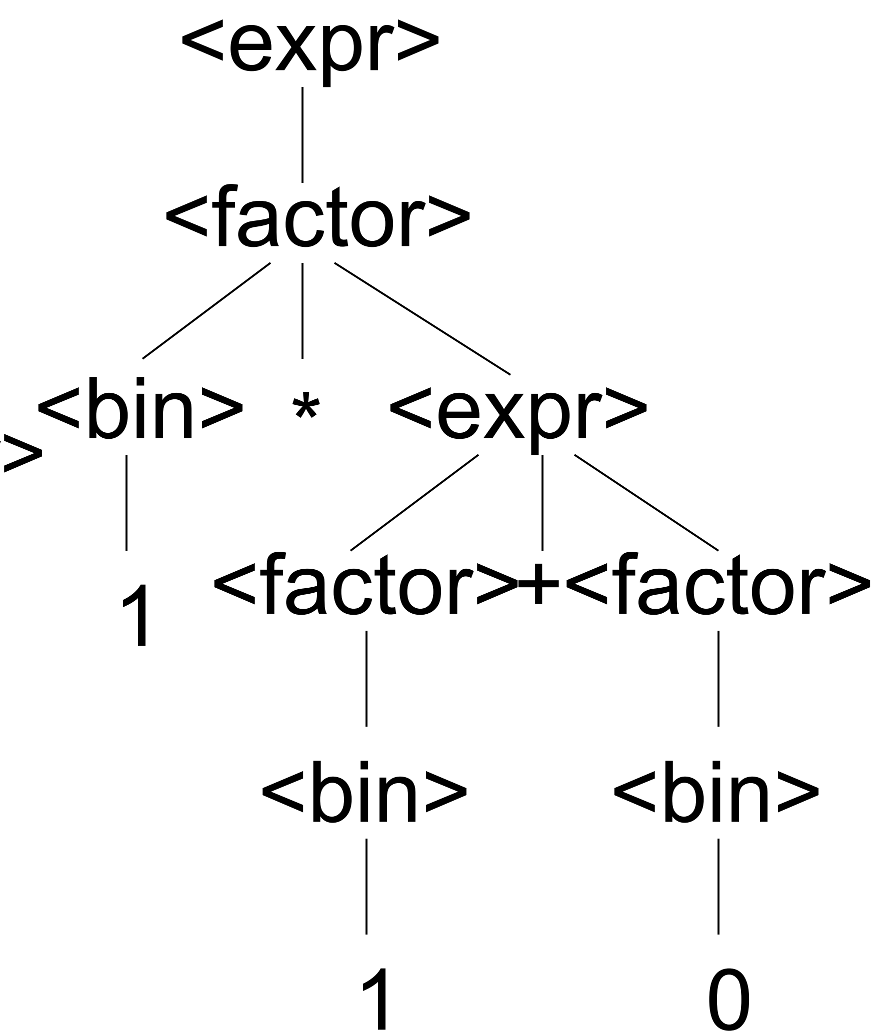
=><bin>*<factor>+<factor>

=><bin>*<factor>+<bin>

=><bin>*<bin>+0

=><bin>*1+0

=>1*1+0



Parse Trees

<expr>

=> <factor>

=><bin>*<expr>

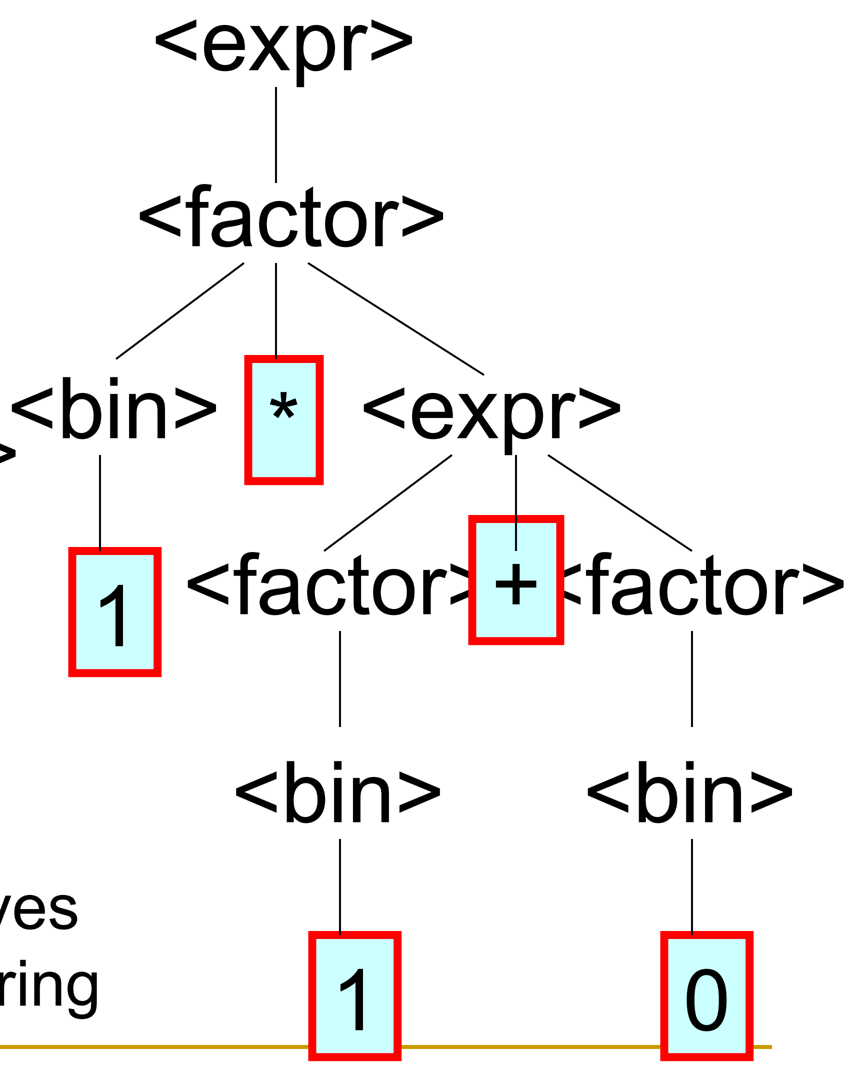
=><bin>*<factor>+<factor>

=><bin>*<factor>+<bin>

=><bin>*<bin>+0

=><u><bin></u>*1+0

=><u>1</u>*1+0



Note: Leaves give the string

Exercise

Using the grammar above, draw the parse tree for 1 * 0 + 0 * 1 using a rightmost derivation

OCAML

- Now to translate grammar to code.
- You can use OCAML to represent a parse tree.
- One type for each nonterminal
- One constructor for each rule
- Defined as <u>**mutually recursion**</u> of type declarations using `and` keyword

OCAML

- Example:
  <expr> ::= <factor> | <factor> + <factor>
  <factor> ::= <bin> | <bin> * <expr>
  <bin> ::= 0 | 1
- OCAML code:

```
type expr   = Factor2Expr of factor
            | Plus of factor * factor
and factor = Bin2Factor of bin
            | Mult of bin * expr
and bin     = Zero
            | One
```

OCAML

```
Factor2Expr
  (Mult(One,
   Plus(Bin2Factor One,
    Bin2Factor Zero)))
```

<expr>

=> <factor>

=><bin>*<expr>

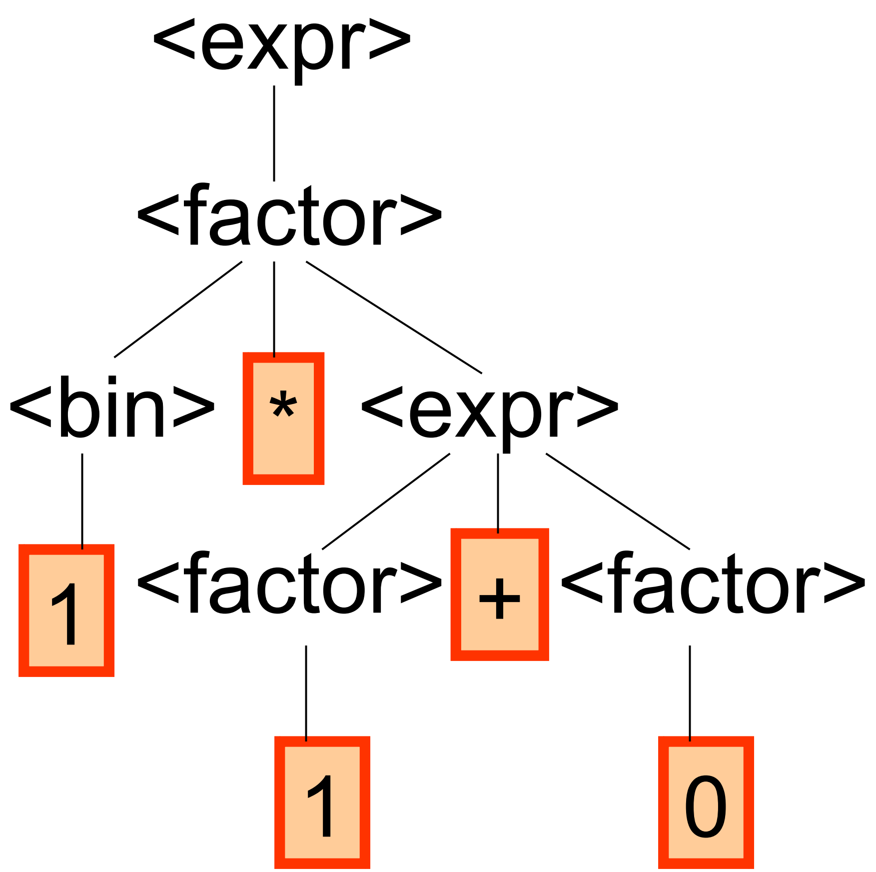
=><bin>*<factor>+<factor>

=><bin>*<factor>+ <bin>

=><bin>* <bin> + 0

=><bin>* 1 + 0

=> 1 * 1 + 0



Exercise

- Using OCAML, construct the parse tree for 1 * 0 + 0 * 1

Ambiguity

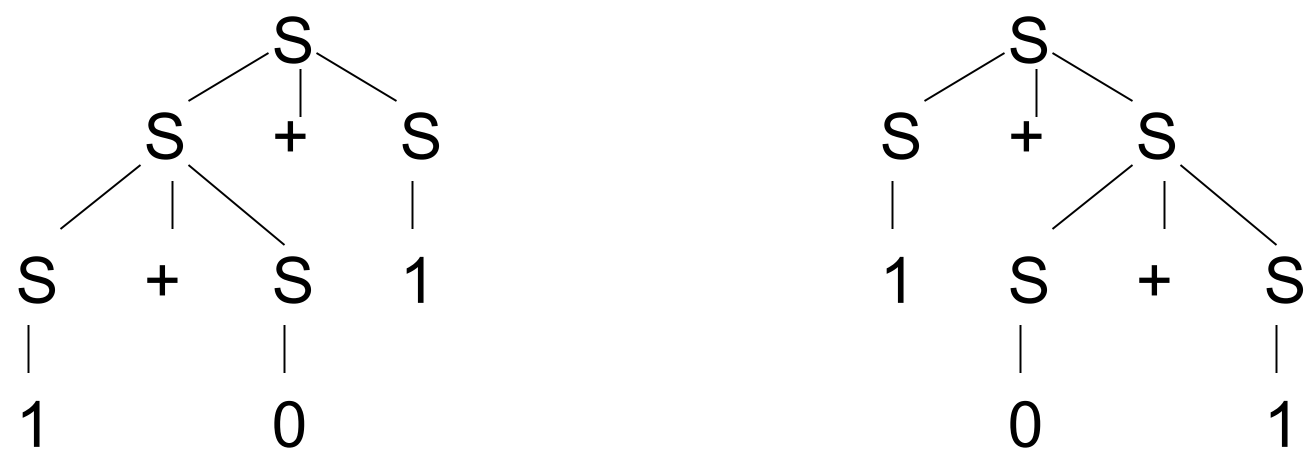
- A grammar is **ambiguous** if a string can have more than one parse tree.
- Given a language, if **all** grammars for that language are ambiguous then the language is **inherently ambiguous**.

Ambiguity

- Example:
  - S → S + S | 0 | 1 | ε
  - The string 1 + 0 + 1 can be derived by
  - $\underline{S} \Rightarrow S + \underline{S} \Rightarrow \underline{S} + 1 \Rightarrow S + \underline{S} + 1 \Rightarrow \underline{S} + 0 + 1 \Rightarrow 1 + 0 + 1$
  - $\underline{S} \Rightarrow S + \underline{S} \Rightarrow S + S + \underline{S} \Rightarrow S + \underline{S} + 1 \Rightarrow \underline{S} + 0 + 1 \Rightarrow 1 + 0 + 1$

Ambiguity

- Example (cont'd)
  - The parse trees are



Ambiguity

- For a string like "1 – 2 – 3", based on the parse tree it can interpreted as

  $$(1 - 2) - 3$$

  or

  $$1 - (2 - 3)$$

  giving different values.
- (Note: Pascal, C/C++, ML associates left to right. But APL associates right to left.)

Ambiguity

- Two major source of ambiguity:
  - Lack of determination of operator precedence
  - Lack of determination of operator associativity
- Next time we will see how to rewrite grammars to remove such problems (if possible at all).