Syntax Meets Semantics

Three "Equivalent" Grammars

- These grammars all define the same language: the language of strings that contain one or more `a`'s, `b`'s or `c`'s separated by minus signs.
- But the parse trees are not equivalent.

Example ($\mathbb{G}1$)

```
<subexp> ::= a | b | c | <subexp> - <subexp>
```

Example ($\mathbb{G}2$)

```
<subexp> ::= <var> - <subexp> | <var>
<var> ::= a | b | c
```

Example ($\mathbb{G}3$)

```
<subexp> ::= <subexp> - <var> | <var>
<var> ::= a | b | c
```

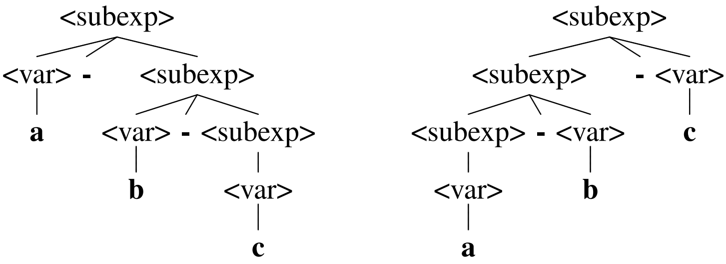
Parse of a-b-c

Example (𝔾2)

```
<subexp> ::= <var> - <subexp> | <var>
<var> ::= a | b | c
```

Example (𝔾3)

```
<subexp> ::= <subexp> - <var> | <var>
<var> ::= a | b | c
```



Why Parse Trees Matter

- We want the structure of the parse tree to correspond to the semantics of the string it generates
- This makes grammar design much harder: we're interested in the structure of each parse tree, not just in the generated string
- Parse trees are where syntax meets semantics

Operators

Operators

- Special syntax for frequently-used simple operations like addition, subtraction, multiplication and division
- The word operator refers both to the token used to specify the operation (like **+** and *) and to the operation itself
- Usually predefined, but not always
- Usually a single token, but not always

Operator Terminology

- Operands are the inputs to an operator, like 1 and 2 in the expression 1+2
- Unary operators take one operand: -1
- Binary operators take two: 1+2
- Ternary operators take three: a?b:c

More Operator Terminology

- In most programming languages, binary operators use an infix notation: **a + b**
- Sometimes you see prefix notation: **+ a b**
  - What is **(* (+ 3 4) 2)** ?
  - What is *** + 3 4 2** ?
- Sometimes postfix notation: **a b +**
  - What is **(2 (3 4 +) *)** ?
  - What is **2 3 4 + *** ?
- Unary operators, similarly:
  - (Can't be infix, of course)
  - Can be prefix, as in **-1**
  - Can be postfix, as in **a++**

Precedence

Expression Grammar

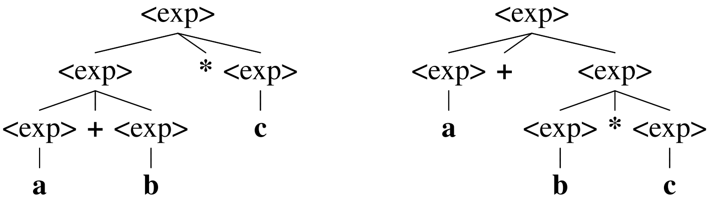
- This generates a language of arithmetic expressions using parentheses, the operators + and *, and variables `a`, `b` and `c`

Example ($\mathbb{G}4$)

```
<exp> ::= <exp> + <exp> |
          <exp> * <exp> |
          (<exp>) |
          a | b | c
```

Precedence Issue

- Our $\mathbb{G}4$ grammar can generate two trees for **a+b*c**.
- In the left tree, the addition is performed before the multiplication, which is not the usual convention for operator precedence.



Operator Precedence

- Applies when the order of evaluation is not completely decided by parentheses
- Each operator has a precedence level, and those with higher precedence are performed before those with lower precedence, as if parenthesized
- Most languages put * at a higher precedence level than +, so that: $a + b * c \equiv a + (b * c)$

Precedence Examples

Example (C: 15 precedence levels)

```
a = b < c ? * p + b * c : 1 << d ()
```

Example (Pascal: 5 precedence levels (error here!))

```
a <= 0 or 100 <= a
```

Example (Smalltalk: 1 precedence level)

```
a + b * c
```

Precedence in Grammar

- Fix precedence: Modify grammar to put * below + in the parse tree in $\mathbb{G}5$, * is higher precedence than +.
- <exp> defined in terms of <mulexp> therefore higher in parse tree and lower precedence.

Example ($\mathbb{G}5$)

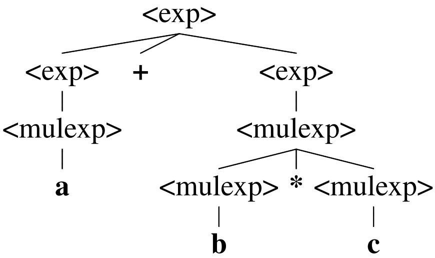
```
<exp> ::= <exp> + <exp> | <mulexp>
<mulexp> ::= <mulexp> * <mulexp>
           | (<exp>)
           | a | b | c
```

Correct Precedence

Example ($\mathbb{G}5$)

```
<exp> ::= <exp> + <exp> | <mulexp>
<mulexp> ::= <mulexp> * <mulexp>
           | (<exp>)
           | a | b | c
```

a+b*c



Exercise

Example (Grammar)

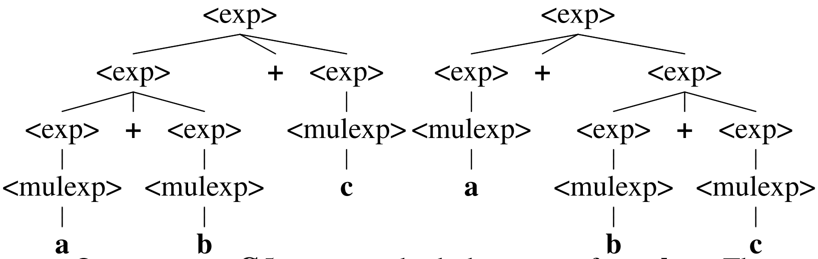
```
<exp> ::= <exp> - <exp> | <mulexp>
<mulexp> ::= <mulexp> * <mulexp> | 1 | 2 | 3
```

Draw parse tree and find value for each

1. $1 - 2 * 3$
2. $1 - 2 - 3$
3. $1 * 2 * 3$
4. $2 * 3 - 1 * 2$

Associativity

Associativity Issue



- Our grammar $\mathbb{G}5$ generates both these trees for **a+b+c**. The second one is not the usual convention for operator *associativity*.

Example ($\mathbb{G}5$)

```
<exp> ::= <exp> + <exp> | <mulexp>
<mulexp> ::= <mulexp> * <mulexp>
           | (<exp>)
           | a | b | c
```

Operator Associativity

- Applies when the order of evaluation is not decided by parentheses or by precedence
- *Left-associative* operators group left to right: $a+b+c+d \equiv ((a+b)+c)+d$
- *Right-associative* operators group right to left: $a+b+c+d \equiv a+(b+(c+d))$
- Most operators in most languages are left-associative, but there are exceptions

Syntax Meets Semantics

Operators

Precedence

Associativity

Associativity

Operator Associativity

Associativity In Grammar

Correct Associativity

Exercise

Ambiguities

Cluttered grammars

Abstract Syntax Trees

Postfix

Examples

Example (C)

```
a<<b<<c // most operators are left-associative
a=b=0   // right-associative (assignment)
```

Example (F#)

```
3-2-1    // most operators are left-associative
1::2::[] // right-associative (list builder)
```

Example (Fortran)

```
c     most operators are left-associative
      a/b*c
c     right-associative (exponentiation)
      a**b**c
```

Associativity In Grammar

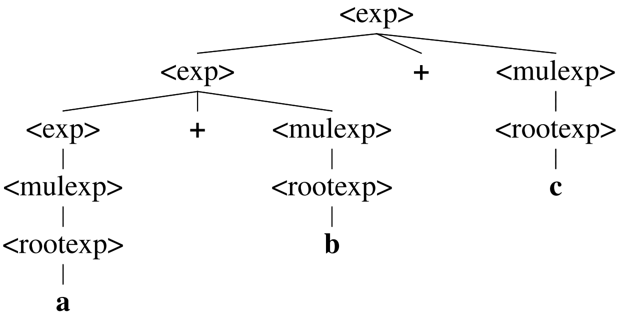
- Fix associativity: Modify the grammar to make trees of +'s grow down to the left (and likewise for *'s) in $\mathbb{G}6$.
- $\mathbb{G}5$ ambiguous: **<exp> + <exp>** is right and left recursive.
- $\mathbb{G}6$: Left-recursive rule alone defines left associativity, **<exp> + <mulexp>** and **<mulexp> * <rootexp>**

Example ($\mathbb{G}6$)

```
<exp> ::= <exp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= (<exp>) | a | b | c
```

Correct Associativity

- Our new grammar generates this tree for **a+b+c**. It generates the same language as before, but no longer generates trees with incorrect associativity.



Exercise

Example (Grammar)

```
<exp> ::= <exp> - <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= 1 | 2 | 3
```

Give parse tree and find value for each

1. $1 - 2 * 3$
2. $1 - 2 - 3$
3. $1 - 3 - 2 * 3$

Ambiguities

Ambiguity

- $\mathbb{G}4$ was ambiguous: it generated more than one parse tree for the same string
- Fixing the precedence (in $\mathbb{G}5$) and associativity (in $\mathbb{G}6$) problems eliminated all the ambiguity
- This is usually a good thing: the parse tree corresponds to the meaning of the program, and we don't want ambiguous meanings
- Not all ambiguity stems from confusion about precedence and associativity…

Dangling Else In Grammars

Example

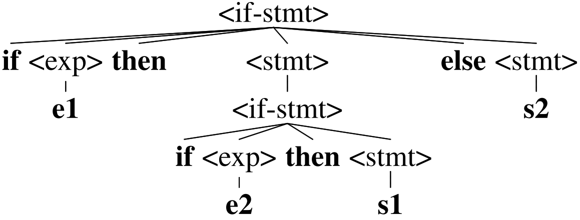
```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
               | if <expr> then <stmt>
<expr> ::= e1 | e2
```

- This grammar has a classic “dangling-else ambiguity.”
- How to interpret **if e1 then if e2 then s1 else s2** ?

Syntax Meets Semantics
Operators
Precedence
Associativity
Ambiguities
Ambiguity
Dangling Else
Eliminating The Ambiguity
Dangling Else
Clearer Styles
Languages That Don't Dangle
Cluttered grammars
Abstract Syntax Trees
Postfix

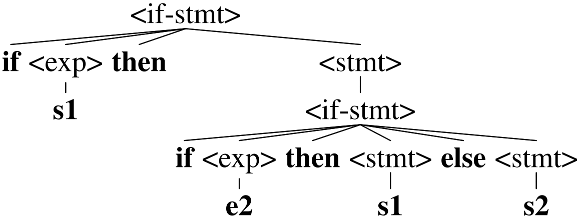
Ambiguous Parse Trees

if e1 then (if e2 then s1) else s2



---

if e1 then (if e2 then s1 else s2)



Eliminating The Ambiguity

- We want to insist that if **<stmt>** expands into an **if**, that **if** must already have its own **else**.
- First, we make a new non-terminal **<full-stmt>** that generates everything **<stmt>** generates.
- Except that it can not generate **if** statements with no **else**.

Example ($\mathbb{G}7$)

```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
            | if <expr> then <stmt>
<expr> ::= e1 | e2
```

Eliminating The Ambiguity

Example ($\mathbb{G}8$)

```
<if-stmt> ::= if <expr> then <full-stmt>
                        else <stmt> |
              if <expr> then <stmt>
<full-stmt> ::= <full-if> | s1 | s2
<full-if> ::= if <expr> then <full-stmt>
                        else <full-stmt>
<stmt> ::= <if-stmt> | s1 | s2
<expr> ::= e1 | e2
```

- Use the new **<full-stmt>** non-terminal.
- The effect is the new grammar can match an **else** part with an **if** part only if all the nearer **if** parts are already matched.

𝔾8 Derivation

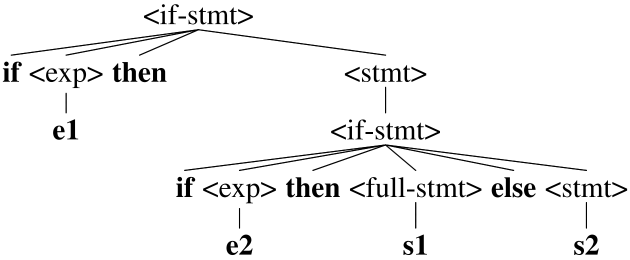
Example (𝔾8)

```
<if-stmt> ::= if <expr> then <full-stmt> else <stmt> |
              if <expr> then <stmt>
<full-stmt> ::= <full-if> | s1 | s2
<full-if> ::= if <expr> then <full-stmt> else <full-stmt>
<stmt> ::= <if-stmt> | s1 | s2
<expr> ::= e1 | e2
```

Example (if e1 then if e2 then s1 else s2)

```
<stmt>
= <if-stmt>
= if <expr> then <stmt>
= if <expr> then <if-stmt>
= if <expr> then if <expr> then <full-stmt> else <stmt>
= if e1 then if <expr> then <full-stmt> else <stmt>
= if e1 then if e2 then <full-stmt> else <stmt>
= if e1 then if e2 then s1 else <stmt>
= if e1 then if e2 then s1 else s2
```

Parse Tree



Example

```
if e1 then if e2 then s1 else s2
```

Dangling Else

- We fixed the grammar, but…
- The grammar trouble reflects a problem with the language, which we did not change
- A chain of if-then-else constructs can be very hard for people to read
- Especially true if some but not all of the `else` parts are present

Clearer Styles

Example (Good: correct indentation)

```
if (0==0)
  if (0==1) a=1;
  else a=2;
```

Example (Better: block structure)

```
if (0==0) {
  if (0==1) a=1;
  else a=2;
}
```

Example (Best: if/endif pairing)

```
if (0==0)
  if (0==1) a=1;
     else a=2;
  endif
endif
```

Languages That Don't Dangle

- Some languages define **if-then-else** in a way that forces the programmer to be more clear
- Algol does not allow the **then** part to be another **if** statement – though it can be a block containing an **if** statement
- Ada (and Visual Basic) requires each **if** statement to be terminated with an **endif**

Cluttered grammars

Clutter

- The new if-then-else grammar is harder for people to read than the old one
- It has a lot of clutter: more productions and more non-terminals
- Same with $\mathbb{G}4$, $\mathbb{G}5$ and $\mathbb{G}6$: we eliminated the ambiguity but made the grammar harder for people to read
- This is not always the right trade-off

Reminder: Multiple Audiences

- In lecture 8 we saw that grammars have multiple audiences:
  - Novices want to find out what legal programs look like
  - Experts – advanced users and language system implementers – want an exact, detailed definition
  - Tools – parser and scanner generators – want an exact, detailed definition in a particular, machine-readable form
- Tools often need ambiguity eliminated, while people often prefer a more readable grammar

Options

- Rewrite grammar to eliminate ambiguity
- Leave ambiguity but explain in accompanying text how things like associativity, precedence, and the dangling else should be parsed
- Do both in separate grammars

Abstract Syntax Trees

Full-Size Grammars

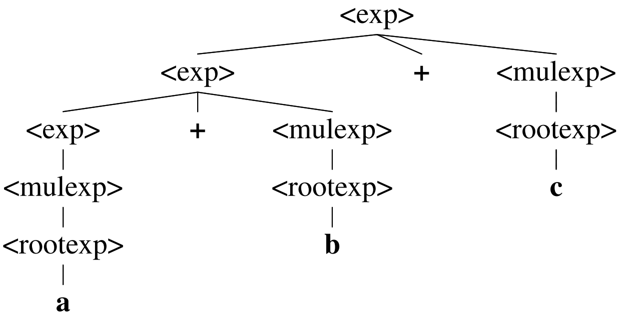
- In any realistically large language, there are many non-terminals
- Especially true when in the cluttered but unambiguous form needed by parsing tools
- Extra non-terminals guide construction of unique parse tree
- Once parse tree is found, such non-terminals are no longer of interest

Abstract Syntax Trees

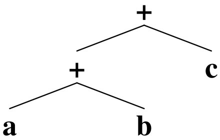
- Language systems usually store an abbreviated version of the parse tree called the Abstract Syntax Tree
- Details are implementation-dependent
- Usually, there is a node for every operation, with a subtree for every operand

AST Example

Parse tree:



Corresponding AST:



AST Conversion Guidelines

- ASTs should have no non-terminals
- If there is a node that followed a production with an operator, the operator is promoted to be the parent of the operand(s)
- Parentheses are discarded

Postfix

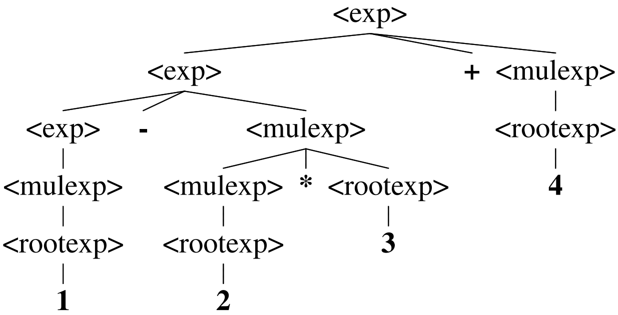
AST to Postfix Notation and Evaluation

- Abstract Syntax Tree holds the infix expression operations and operands.
- Traversing the AST in *post-order* produces postfix notation.
- Also called *Reverse Polish Notation*, or **RPN**.
- RPN operator precedence is unambiguous.
- Operands are arranged in proper order for post-order evaluation.
- Easily evaluated on hardware using a stack.

Parse Tree: 1-2*3+4

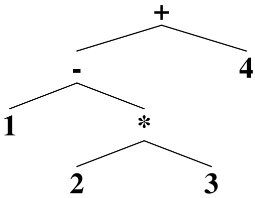
Example

```
<exp> ::= <exp> - <mulexp> |
          <exp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= 1 | 2 | 3 | 4
```

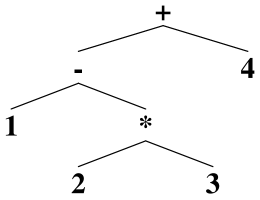


AST: 1-2*3+4

- Much simpler tree to analyze



Post-order traversal



$\Rightarrow$

RPN format:
$123 * -4+$

POSTORDER(tree T)

```
if T ≠ null then
    POSTORDER(T.LEFT)
    POSTORDER(T.RIGHT)
    print T.data
```

RPN Stack Evaluation

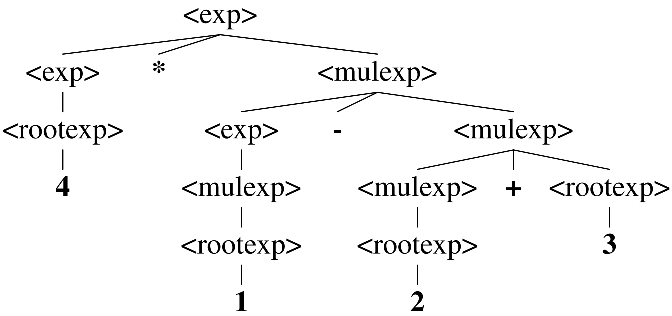
- Scan RPN input from left to right.
- input: **1 | 2 | 3 | 4**
  - push input onto stack.
- input: **+ | - | ***
  - pop operand2
  - pop operand1
  - push operand1 <operator> operand2
- After RPN is scanned, the expression value is stack top.

Input	**Stack**	**Pushed**
1 2 3 * - 4 +	1	1
2 3 * - 4 +	1 2	2
3 * - 4 +	1 2 3	3
* - 4 +	1 6	2 * 3
- 4 +	-5	1 - 6
4 +	-5 4	4
+	-1	-5+ 4

Syntax Meets Semantics
Operators
Precedence
Associativity
Ambiguities
Cluttered grammars
Abstract Syntax Trees
Postfix
AST to Postfix
Parse Tree
AST
Post-order traversal
RPN Stack Evaluation
Exercise
Parsing
Conclusion

Exercise

- Give the AST
- Give the resulting RPN
- Give the value of the expression



Exercise

Example

```
<exp> ::= <exp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= (<exp>) | 1 | 2 | 3 | 4 | 5
```

- Given the above BNF rules, parse and generate AST for expressions:
  - $2 + 3 * 4$
  - $2 + 3 * (4 + 1)$
- Traverse the AST nodes in post order to produce the Reverse Polish Notation.
- Evaluate the results

Parsing, Revisited

- When a language system parses a program, it goes through all the steps necessary to find the parse tree
- But it usually does not construct an explicit representation of the parse tree in memory
- Most systems construct an AST instead

Parsing, Revisited

- Grammars define syntax, *plus more*
- They define not just a set of legal programs, but a parse tree for each program
- The structure of a parse tree corresponds to the order in which different parts of the program are to be executed
- Thus, grammars contribute to the definition of semantics