Grammar Format

Motivation

- Look at structure of grammar, how to accomplish certain tasks
- These concepts are common to most/all programming languages

Example

- Goal: Grammar for arithmetic expression
- Initially: Just + and -
- Define terminal:
  NUM $\rightarrow$ -?\d+
- Then:
  S $\rightarrow$ NUM | S + S | S - S

Example

- Or, more concisely: Define another terminal:
  OP $\rightarrow$ [-+]
- Then define:
  - S $\rightarrow$ NUM | S OP S
- There's a problem. What is it?

Problem

- Our grammar derives any expression, but it’s ambiguous

Derivation

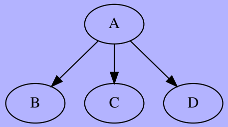
- Derive 1+2-3
  - $S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S - S \rightarrow 1 + 2 - S \rightarrow 1 + 2 - 3$
  - $S \rightarrow S - S \rightarrow S - 3 \rightarrow S + S \rightarrow 1 + S - 3 \rightarrow 1 + 2 - 3$
    - Note: Showing the OP’s as + or - explicitly so it’s clear what we’re doing
- Why is this a problem?

Problem

- We usually don't just care *if* a string is a valid program
  - We also care about program's structure

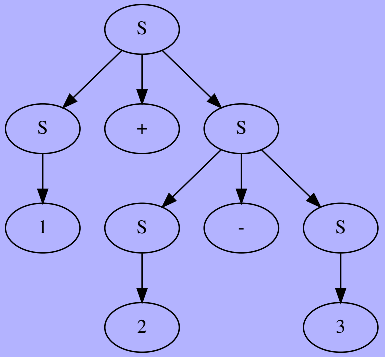
Parse Tree

- Parse tree construction:
  - If we see expansion A $\rightarrow$ B C D we create nodes for B, C, D and graft them as children of A



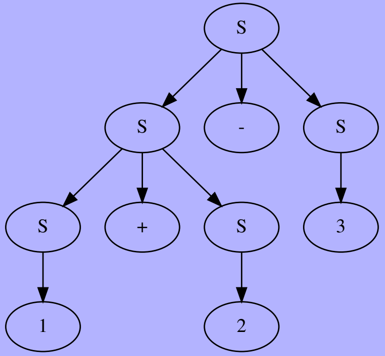
Derivation

- Suppose we derive
  S $\rightarrow$ S + S $\rightarrow$ 1 + S $\rightarrow$ 1 + S - S $\rightarrow$ 1 + 2 - S $\rightarrow$ 1 + 2 - 3
- Final tree: Has 2-3 in its own subtree



Derivation

- What if we derive:
  $S \rightarrow S - S \rightarrow S - 3 \rightarrow S + S \rightarrow 1 + S - 3 \rightarrow 1 + 2 - 3$
- We get a different parse tree:



Problem

- Why is different parse tree structure a problem?
- Suppose we have our tree node defined like so:

```
class Node:
    def __init__(self,sym,token):
        self.sym=sym
        self.children=[]
        self.token=token
```

Problem

- When we evaluate, we usually use recursive style algorithm:

```
def eval(node):
    if n has 3 children:
        v1=eval(n.children[0])
        v2=eval(n.children[2])
        if n.children[1].token.lexeme == '+':
            return v1+v2
        else:
            return v1-v2
    else:
        return int(n.children[0].lexeme)
```

- The two trees give different results: 1+(2-3) vs. (1+2)-3

Solution

- Rewrite grammar to eliminiate ambiguity:
  - $S \rightarrow S$ op num | num
- Now: No choice about tree: It must grow down left branch only
- Right side of tree will be just one item
- This produces *left associativity* of operators

Associativity

- Left associativity: 1 op 2 op 3 $\rightarrow$ (1 op 2) op 3
- Doesn't matter for addition
- But it does for subtraction:
  - 1 - 2 - 3 = (1 - 2) - 3 = -1 - 3 = -4
  - 1 - 2 - 3 = 1 - (2 - 3) = 1 - -1 = 1 + 1 = 2
- Usually, we think of arithmetic as left associative

Operators

- What if two operators at different priorities?
  - ADDOP → [-+]
  - MULOP → [*/]
- Suppose all operators are left associative
  - Our tree must grow down left branch
  - So all rules are of form:
    - X → X <op> NUM
- The + is “too weak” to pull anything away from a *
- So we make a product look atomic from a +’s perspective

Grammar

- sum $\rightarrow$ sum ADDOP product | product
  product $\rightarrow$ product MULOP NUM | NUM
- Ex: 1 + 2 * 3
  - Diagram the parse tree in class

Trees

- Notice precedence always respected
- Diagram in class:
  - 1*2+3
  - 1*2*3
  - 1+2*3
  - 1+2+3

Parentheses

- How about including parentheses?
- What can “take apart” a parenthesized expression?

Parentheses

- Nothing can “disassemble” parenthesized expression, so we must make them atomic from everything else’s perspective
- Add two terminals: LPAREN and RPAREN
- sum $\rightarrow$ sum ADDOP product | product
  product $\rightarrow$ product MULOP factor | factor
  factor $\rightarrow$ NUM | LPAREN sum RPAREN
- Notice how we must include ’sum’ in the parens so we can “go back to the top again”

Example

- (1+2)*3
- 1*(2+3)

Compression

- Note: If we have unit productions, we might opt to replace nodes in-place
  - When building tree: If we have unit production ($X \rightarrow Y$), we replace tree node for X with tree node for Y
- Ex: 1+2*3: Do in-class

Variables

- What about variables? How to modify the grammar to allow those?
  - Do in class

Multiple Levels

- What if more levels of precedence?
- Example: RELOP -> >=|<=|>|<|==|!=
- Where should RELOP be in precedence hierarchy?
  - Ex: while(5+x > 7*4){ ... }
- What should grammar look like?

Levels

- relexp → sum RELOP sum | sum
  sum → sum ADDOP product | product
  product → product MULOP factor | factor
  factor → NUM | LPAREN sum RPAREN
- Does this allow x > 5 > y ?
- *Should* we allow x > 5 > y ?

Right-Associative

- What if we want a right-associative operator?
  - Left associative: $A \oplus B \oplus C = (A \oplus B) \oplus C$
  - Right associative: $A \oplus B \oplus C = A \oplus (B \oplus C)$
- Most operators are left associative: + - * / %
  - Ex: 5-2-1 = (5-2)-1
- Many languages treat exponentiation as right associative
  - Note: C uses a function [pow()], so this doesn't apply there
  - Ex: a ** b ** c = a ** (b**c)
  - 10**2**5 =$10^{2^5} = 10^{32}$
  - If ** was left associative, we'd have $\left(10^2\right)^5 = 10^{10}$
- Suppose ** is the highest priority operator. How do we add it to the grammar?

Grammar

- relexp $\rightarrow$ sum RELOP sum | sum
  sum $\rightarrow$ sum ADDOP product | product
  product $\rightarrow$ product MULOP pow | pow
  pow $\rightarrow$ factor STARSTAR pow | factor
  factor $\rightarrow$ NUM | LPAREN sum RPAREN
- What would parse tree for 10**2**5 look like?

Unary Operators

- What about unary operators?
  - Ex: Negation, bitwise complement, boolean NOT
  - These are also right associative: ~~5 = ~(~5)
- Suppose they are higher priority than **
- What would our grammar look like?

Grammar

- relexp $\rightarrow$ sum RELOP sum | sum
  sum $\rightarrow$ sum ADDOP product | product
  product $\rightarrow$ product MULOP pow | pow
  pow $\rightarrow$ unary STARSTAR pow | unary
  unary $\rightarrow$ UNARYOP unary | factor
  factor $\rightarrow$ NUM | LPAREN sum RPAREN
- Common unary operators: +, -, ~, !

Pattern

- General pattern: If we have high priority left associative operator $\oplus$ and lower priority left associative operator $\otimes$ we create rules:
  - $X \to X \oplus Y \mid Y$
  - $Y \to Y \otimes Z \mid Z$
- If an operator is right associative: Make the production right-recursive instead

C

- Ex: C has 15 levels of precedence:
  1. () [] -> .
  2. Unary (+ - ~ ! & * ++), cast, sizeof
  3. Multiply/divide/modulo (* and / when binary operators)
  4. Add/subtract (+ - when binary)
  5. Shift
  6. Relational greater/less than
  7. Relational equal/not equal
  8. Bitwise and (&)
  9. Bitwise xor (^)
  10. Bitwise or (|)
  11. Logical and (&&)
  12. Logical or (||)
  13. Ternary (?:)
  14. Assignment (=), assign-and-op (+=, -=, etc.)
  15. Comma (,)

Hierarchies

- For some languages we want to restrict where expressions can be used (Ex: Java has some restrictions like these)
  - Illegal: if(x=y){ ... }
  - Illegal: if(x){ ... }
  - Illegal: if(x==y==z)
  - Illegal: if(x or y > z)
- Note: C allows all of these (with || instead of ‘or’), but they often don’t do what you might expect
- Note: Python correctly handles x < y < z ; C doesn’t do what you think

Assignment

- Design a grammar which has two arithmetic operations (+,*), a relational operator (>), and two boolean operator (&&, ||)
- The order of priority (low to high) is: ||, &&, >, +, *
- Use Java style rules:
  - Any number of operands chained by + and/or * are legal
  - > must not be chained. So 1>2>3 is illegal.
  - Boolean operators do allow chaining. So 1>2 && 3>4 is legal. However, the two sides of a boolean operator must themselves be boolean. So 1 && 2 is not legal, and neither is 1>2 && 3.

Sources

- Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques, & Tools (2nd ed). Addison Wesley Publishing.
- K. Louden. Compiler Construction: Principles and Practice. PWS Publishing.
- Linux man page for C operator precedence

Created using $\LaTeX$.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org