More on Syntax

Judy Stafford
Comp 80 – Meeting 3
February 1, 2010

Agenda for the Day

- Administrative Stuff
  - Moodle…
  - Classlist at 55 without waiting list
- More on Syntax
- In-Class Exercise
- Using parse trees

2

Last time

- Syntax
  - **Problem**: how to precisely describe what a properly formed program looks like
  - Must cover all possible programs
- Solution: formal grammars
  - Inspired by work in linguistics
  - Notation: Backus-Naur Form (BNF)

Context-free grammar

#	Production rule
1	*non-terminal* → *terminals or non-terminals*
2	| …

- Formally: ***context-free grammar*** is
  - **G** = (**s**, **N**, **T**, **P**)
  - ***T*** : set of terminals — *(the "words")*
  - ***N*** : set of non-terminals — *(parts of speech)*
  - $P : N \rightarrow (N \cup T)^*$: production rules — *(sentence structure)*
  - $s \in N$: start or goal symbol
- **Note**:
  - In a complete grammar all non-terminals appear on the left-hand side of at least one production

Using a BNF

Two ways:

- Generate strings – called ***derivation***

  Idea: Use productions as ***rewrite rules***

  – Start with the start symbol (a non-terminal)

  – Apply productions:

  Choose a non-terminal and "expand" it to the right-hand side of one of its productions

  – When only terminal symbols remain, we have a legal string

- Recognize strings – called ***parsing***

  – Start with a string of terminals (e.g., a program)

  – Try to figure out if it can be derived from the grammar

  – Topic of comp181 – Compilers

Applied to programming

A real grammar

- Arithmetic expressions

  – Numbers and variables (terminals called "num" and "id")

  – Binary operators: +, -, *, /

  – For now, no parentheses

- Examples:

  – `3 + 5`

  – `3 + 5 * 6`

  – `5 / 9 * F – 32`

  – `x – 2 * y`

BNF for Expressions

Note:

- Special terminals
  - number and identifier
  - Categories of terminals
  - Examples:

    <num, “5”>

    <id, “foo”>

#	Production rule
1	*expr* → *expr* *op* *expr*
2	| `number`
3	| `identifier`
4	*op* → +
5	| -
6	| *
7	| /

- How are terminals specified?
  - Typically, using regular expressions
  - We probably won’t cover that topic

Language of expressions

- What’s in this language?

#	Production rule
1	*expr* → *expr* *op* *expr*
2	| `number`
3	| `identifier`
4	*op* → +
5	| -
6	| *
7	| /

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	<id,x> *op expr*
5	<id,x> - *expr*
1	<id,x> - *expr op expr*
2	<id,x> - <num,2> *op expr*
6	<id,x> - <num,2> * *expr*
3	<id,x> - <num,2> * <id,y>

We can build the string “`x - 2 * y`”

This string is in the language

More on derivations

- Different derivations are possible
  - At each step we can choose any non-terminal
  - ***Rightmost derivation***:
    - » Always choose right-most non-terminal
  - ***Leftmost derivation***:
    - » Always choose left-most non-terminal
  - What other derivations are possible?
    - » *"random" derivations – not of interest*
- **Question**:
  - Does it matter?

Left vs right derivations

- Two derivations of "`x – 2 * y`"

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

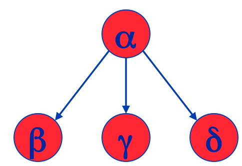
Left-most derivation

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

Right-most derivation

Derivations and parse trees

- Two different derivations
  - Both are correct
  - Let's look carefully at the differences
- Represent derivation as a ***parse tree***
  - Leaves are terminal symbols
  - Inner nodes are non-terminals
  - To depict production $\alpha \rightarrow \beta\ \gamma\ \delta$
    show nodes β,γ,δ as children of α



Tree is often used to represent semantics

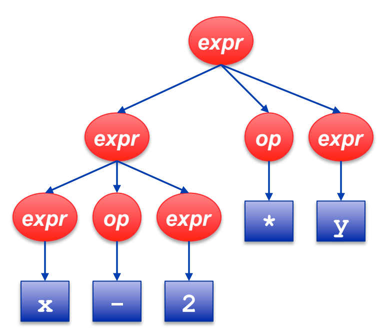
We want the ***structure*** of the parse tree to capture the ***meaning*** of the sentence

Example (I)

Right-most derivation

Rule	*Sentential form*
-	*expr*
1	*expr op expr*
3	*expr op <id,y>*
6	*expr * <id,y>*
1	*expr op expr * <id,y>*
2	*expr op <num,2> * <id,y>*
5	*expr - <num,2> * <id,y>*
3	*<id,x> - <num,2> * <id,y>*

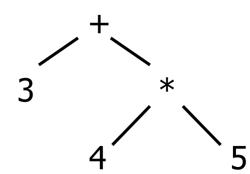
Parse tree



- "Concrete" syntax tree
  - Captures the exact grammatical structure
  - Often has lots of extraneous information

Concrete vs abstract

- ***Concrete*** syntax
  - The exact symbols used to write a program
  - The precise grammatical structure
- ***Abstract*** syntax
  - An abstraction of the program syntax
  - Eliminates uninteresting details of the derivation
  - Closer to the "meaning" of the program
  - Often something used inside the compiler or interpreter
- Examples
  - Infix: **3 + (4 * 5)**
  - Postfix: **3 4 5 * +**
  - Prefix: **(+ 3 (* 4 5))**



Concrete vs abstract

- Another example:

```
while i < N do begin
    i := i + 1
end
```

Pascal

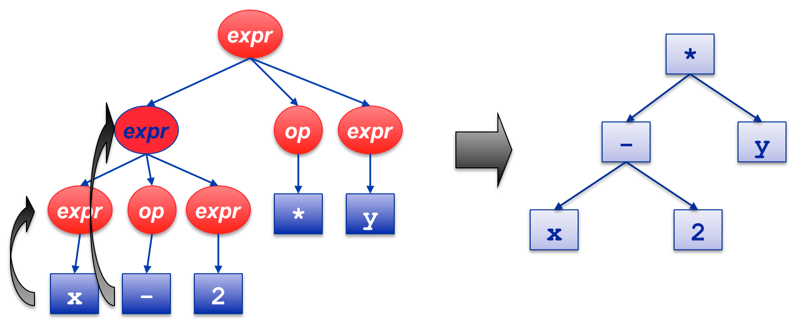
```
while (i < N) {
    i = i + 1;
}
```

C/C++

- What are some differences?
- **Note**: these programs do the same thing
  - Different concrete syntax
  - Same (or similar) abstract syntax
  - Identical semantics

Abstract syntax tree

- Turn concrete syntax into abstract syntax
  - Eliminate extra junk (intermediate nodes)
  - Move operators up to parent nodes
  - Result: ***abstract syntax tree***

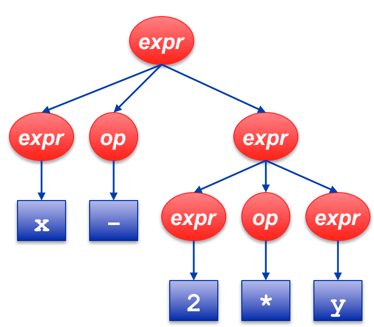


Back to derivations

Left-most derivation

Rule	Sentential form
-	*expr*
1	*expr op expr*
3	*<id, x> op expr*
5	*<id,x> - expr*
1	*<id,x> - expr op expr*
2	*<id,x> - <num,2> op expr*
6	*<id,x> - <num,2> * expr*
3	*<id,x> - <num,2> * <id,y>*

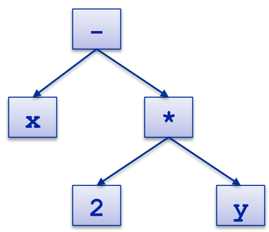
Parse tree



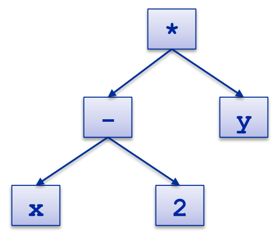
Derivations

- Two different abstract syntax trees for `x - 2 * y`

  Which one do I want?



Left-most derivation



Right-most derivation

Derivations and semantics

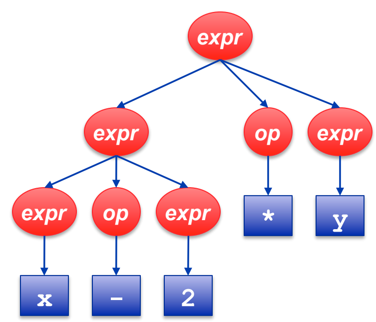
- **Problem**:
  - Two different valid derivations
  - One captures "meaning" we want

    (Arithmetic precedence rules)
  - **Key idea**: shape of tree implies its meaning
- Can we express precedence in grammar?

Derivations and precedence

- **Question**:
  - Which operations are evaluated first in the abstract syntax tree?
  - Operations **deeper** in tree
- **Solution**: add an intermediate production
  - New production isolates different levels of precedence
  - Force higher precedence "deeper" in the grammar

Parse tree



Adding precedence

- Two levels:

Level 1: lower precedence – higher in the tree (rules 1–3)

Level 2: higher precedence – deeper in the tree (rules 4–6)

#	Production rule
1	*expr* → *expr* + *term*
2	| *expr* - *term*
3	| *term*
4	*term* → *term* * *factor*
5	| *term* / *factor*
6	| *factor*
7	*factor* → number
8	| identifier

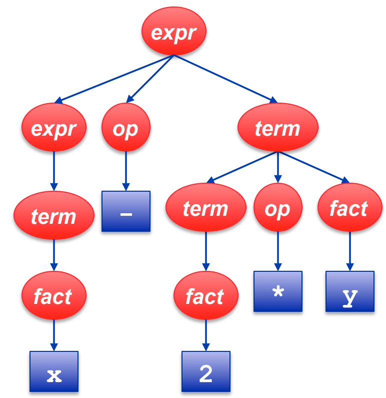
- Observations:
  - Larger: requires more rewriting to reach terminals
  - **But**, produces same abstract parse tree under both left and right derivations

Expression example

Right-most derivation

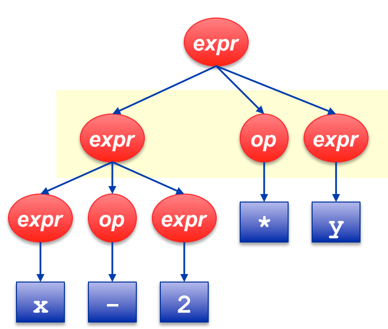
Rule	*Sentential form*
-	*expr*
2	*expr - term*
4	*expr - term * factor*
8	*expr - term * <id,y>*
6	*expr - factor * <id,y>*
7	*expr - <num,2> * <id,y>*
3	*term - <num,2> * <id,y>*
6	*factor - <num,2> * <id,y>*
8	*<id,x> - <num,2> * <id,y>*

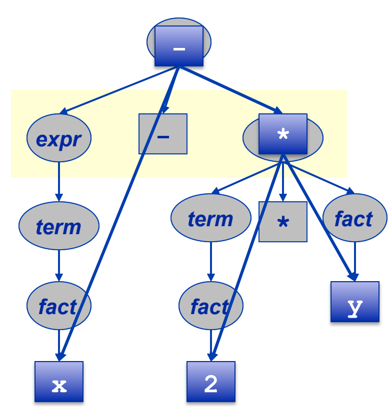
Parse tree



➡ Now right derivation yields `x – (2 * y)`

With precedence





In-Class Assignment

- Find a partner and get out a piece of paper
- Draw both an concrete and abstract syntax tree for: 2* A + 9 – B * 4

#	Production rule
1	*expr* → *expr* + *term*
2	| *expr* - *term*
3	| *term*
4	*term* → *term* * *factor*
5	| *term* / *factor*
6	| *factor*
7	*factor* → number
8	| identifier

Exchange with neighbors – grade

Send over to Chris

A quick look at BNF for C++

- http://www.nongnu.org/hcb/

24

Next time

- A bit more on Syntax
- Introduction to Scheme