Homework & Announcements

- New schedule on line.
- Reading: Chapter 18
- Homework: Exercises at end
- Due: 11/1
COS 140: Foundations of Computer Science

Specifying Programming Languages: Backus–Naur Form (BNF)

Fall 2017
Problem

- Problem: how to *specify* a programming language?
- Have to have a way to describe its syntax (and, possibly, its semantics)
Syntax and Semantics

- **Syntax**:
  - Form of something (e.g., language)
  - Describes relationships between components

- **Semantics**:
  - Meaning of something
  - Describes what the statements will do

- Need to capture formally so it’s clear how language is to be used.
Syntax Formalisms

- What is needed in a syntax formalism?
  - Able to specify language’s grammar for users and compiler designers
  - Easy to write and understand
  - Expressive enough to capture all programming languages
  - Easy to translate into algorithms for machine translation of language
Syntax Formalisms

- Formalisms often expressed as *production rules*:
  - $LHS \rightarrow RHS$
  - Match LHS with what you have, produce RHS
  - E.g.: $S \rightarrow NP \; VP$

- As we’ll see: can also think of going backwards
  - If you have NP and VP $\Rightarrow$ have a valid sentence $S$. 
Backus–Naur Form

- BNF is the most common way of specifying *grammar* of a programming language
- Also important for:
  - Computability theory
  - Natural language processing
  - Many other places in CS where you need to specify a grammar
Language Terminology

- **String**: any combination of characters in the language – may be the whole program
- **Lexemes**: lowest-level unit of language – analogous to words in English – sometimes called “tokens”
- **Syntactic category**: classes of lexemes that play the same role in the structure of a statement – analogous to parts of speech – also called “tokens” – just to be confusing, maybe?
- **Constituents**: tokens or groups of tokens that are put together in ways specified by the grammar – analogous to noun phrases, etc.
- **Grammar**: specification of the syntax; a set of rules describing the legal strings of the language
- **Terminal** and non-terminal symbols
Types of Languages: The Chomsky Hierarchy

- As go down the hierarchy, languages increase in complexity, more machinery needed to recognize them
Types of Languages: The Chomsky Hierarchy

- As go down the hierarchy, languages increase in complexity, more machinery needed to recognize them
- *Regular languages* (regular expressions) – tokens
  - Single symbol on the LHS; RHS has at most one non-terminal:
  - E.g.: \( S \rightarrow aS \mid b \) – generates \( a^+b \)
Types of Languages: The Chomsky Hierarchy

- As go down the hierarchy, languages increase in complexity, more machinery needed to recognize them

- **Regular languages** (regular expressions) – tokens
  - Single symbol on the LHS; RHS has at most one non-terminal:
    - E.g.: \( S \rightarrow aS \mid b \) – generates \( a^+b \)

- **Context-free languages** – programming languages
  - LHS: single symbol; terminals, non-terminals in RHS
  - E.g.: \( S \rightarrow aSb \mid \text{nil} \) – generates \( a^n b^n \)
Types of Languages: The Chomsky Hierarchy

- As go down the hierarchy, languages increase in complexity, more machinery needed to recognize them

- **Regular languages** (regular expressions) – tokens
  - Single symbol on the LHS; RHS has at most one non-terminal:
  - E.g.: \( S \rightarrow aS \mid b \) – generates \( a^+ b \)

- **Context-free languages** – programming languages
  - LHS: single symbol; terminals, non-terminals in RHS
  - E.g.: \( S \rightarrow aSb \mid \text{nil} \) – generates \( a^n b^n \)

- **Context-sensitive languages**
  - RHS has at least as many symbols as LHS
  - E.g.: \( aS \rightarrow aSbSc \)
Types of Languages: The Chomsky Hierarchy

- As go down the hierarchy, languages increase in complexity, more machinery needed to recognize them
- **Regular languages** (regular expressions) – tokens
  - Single symbol on the LHS; RHS has at most one non-terminal:
    - E.g.: $S \rightarrow aS \mid b$ – generates $a^+b$
- **Context-free languages** – programming languages
  - LHS: single symbol; terminals, non-terminals in RHS
  - E.g.: $S \rightarrow aSb \mid \text{nil}$ – generates $a^n b^n$
- **Context-sensitive languages**
  - RHS has at least as many symbols as LHS
  - E.g.: $aSc \rightarrow aSbSc$
- **Recursively-enumerable languages** – Turing-equivalent representation
  - No restrictions on LHS, RHS
  - E.g.: $aaaS \rightarrow aaaTQ$
Backus-Naur Form (BNF)

- Grammar formalism for context-free languages (only one symbol on LHS)
- Terminology:
  - *Rewrite (production) rules* – rules that rewrite current pattern into a new one
  - *LHS, RHS* – parts of rule: LHS \( \rightarrow \) RHS
  - *Terminal symbols* – symbols that appear only on RHS – i.e., do not get replaced by anything
  - *Non-terminal symbols* – symbols that appear in some LHS
  - *Alternatives* (\( | \)) – different RHS’s that can be used with the LHS
Backus-Naur Form (BNF)

- E.g.: \( <\text{expr}> \rightarrow <\text{var}> \mid <\text{var}> + <\text{var}> \)
  \( <\text{var}> \rightarrow A \mid B \mid C \)
- Often written with \( ::= \) instead of \( \rightarrow \):
  \( <\text{var}> ::= A \mid B \mid C \)
Example: Producing a String from BNF

\[
<\text{expr}> \rightarrow <\text{var}> | <\text{var}> + <\text{var}>
\]
\[
<\text{var}> \rightarrow A | B | C
\]

- Start with the kind of constituent you want to produce
  \[
  <\text{expr}>
  \]
- Replace each non-terminal in LHS with a RHS that appears in the rule for which that non-terminal is in the LHS
  \[
  <\text{var}> + <\text{var}>
  \]
- Keep doing this until there are only non-terminals in the string
  \[
  A + <\text{var}> \Rightarrow \ldots \Rightarrow A + B
  \]
Derivation

- **Start symbol**: where derivations begin to derive all possible strings (programs)
- **Sentential form**: any string derived from the start symbols using the rewrite rules
- **Derivation**: rewrite sentential forms until have all terminal symbols
- **Options**:
  - Order of rewriting – left-most, right-most derivations
  - Alternative used for rewriting
Recursion

- What is recursion?
- In BNF: LHS appears in the RHS
- Allows infinite language from finite grammar
- Cannot specify the number of times rule will be applied
- E.g.: to allow any number of additions in an expression:
  \[ <\text{expr}> \rightarrow <\text{var}> | <\text{var}> + <\text{expr}> \]
- Need an alternative that can stop the recursion!
Parsing

- What is parsing?
  - Checking the legality of input against a grammar
  - Producing a parse tree that shows the relationships between the tokens
  - Parse trees record derivations

- Start with a string of only tokens (terminal symbols)
- Find a RHS pattern in the string, replace with the LHS.
- Continue until you have the start symbol
- If you can do this: the input was a valid sentence in the language
- Create a parse tree by showing how we replace terminal/non-terminal symbols using appropriate rules
Parsing Example
Parsing Example

Input: A + B + C
Processed so far:
Parse tree:

Grammar:
<expr> → <var> | <var> + <expr>
<var> → A | B | C
Parsing Example

Input: A + B + C
Processed so far: A
Parse tree:

Grammar: \( <\text{expr}> \rightarrow <\text{var}> \mid <\text{var}> + <\text{expr}> \)

\( <\text{var}> \rightarrow A \mid B \mid C \)
Input: A + B + C
Processed so far: A
Parse tree:

<var>
  |
A
**Parsing Example**

**Input:** $A + B + C$

**Grammar:**
- $\text{<expr>} \rightarrow \text{<var>} | \text{<var> + <expr>}$
- $\text{<var>} \rightarrow A \mid B \mid C$

**Processed so far:** $A +$

**Parse tree:**

```
     <expr>
     |   +  <expr>
    /   |
   <var> A <var>
```

Copyright © 2002–2017 UMaine School of Computing and Information Science – 16 / 25
**Parsing Example**

**Input:** A + B + C

**Processed so far:** A + B + C

**Parse tree:**

```
<expr>  
   /   
<var> + <expr>  
   |      /     
   A     <var> + <expr>  
       |      /     
       B     <var>  
           |  
           C
```

**Grammar:**

- `<expr> -> <var> | <var> + <expr>`
- `<var> -> A | B | C`
Ambiguity

- More than one different, legal, correct parse trees for the same string
- Problem, since parse tree indicates relationship between the constituents
- Property of the grammar
Ambiguity Example

- Grammar:
  
  \[<\text{expr}> \rightarrow <\text{var}> \mid <\text{expr}> + <\text{expr}>\]
  
  \[<\text{var}> \rightarrow A \mid B \mid C\]

- Parse trees:

- Sub-expressions that are more deeply nested in the tree are evaluated first
An Unambiguous Grammar for Addition Expressions

- Need to ensure that the expression can be evaluated in only one way
- Want the expression to be evaluated left to right $\Rightarrow$ first expression formed (lowest in tree) needs to be to the left
An Unambiguous Grammar for Addition Expressions

- Need to ensure that the expression can be evaluated in only one way
- Want the expression to be evaluated left to right ⇒ first expression formed (lowest in tree) needs to be to the left
- Replace rule in previous with:

  \[ \langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{var} \rangle \]
An Unambiguous Grammar for Addition Expressions

- Need to ensure that the expression can be evaluated in only one way
- Want the expression to be evaluated left to right ⇒ first expression formed (lowest in tree) needs to be to the left
- Replace rule in previous with:

\[
<\text{expr}> \rightarrow <\text{expr}> + <\var>
\]
Operator Precedence

- Need a constituent for each set of operators with the same precedence
- Operators with the highest precedence need to be built at the lowest level in the tree
- Get left associativity if we parse left → right, keep recursion to the left
  \[ <expr> \rightarrow <expr> + <var> \]
- Right associativity: recursion to the left.
# Levels of Precedence

- **Items in parentheses and identifiers, numbers:** factor
  A, B, (C + D)

- **Multiplication and division:** term
  A*B, A/B

- **Addition and subtraction:** expression
  A+B, A-B
Example Grammar

\[ <expr> \rightarrow <expr> + <term> | <expr> - <term> | <term> \]
\[ <term> \rightarrow <term> * <factor> | <term> / <factor> | <factor> \]
\[ <factor> \rightarrow ( <expr> ) | <id> \]
\[ <id> \rightarrow A | B | C | D \]
Parsing

- Can think of there being a “state” of the parse
  - Records where we are in the process
  - E.g.: `<expr> + <term>` would mean that we’ve turned the tokens we’ve read so far into the non-terminal `<expr>`, the terminal symbol (token) `+`, and the non-terminal `<term>`

- Idea of state can be used to parse using a *state machine* or automata – we’ll talk about this later in course

- Can also think of there being some tokens that have not yet been read

- Start with an empty state, and with all terminals unread
## Parse of A + B * C - D

<table>
<thead>
<tr>
<th>State:</th>
<th>(empty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input left:</td>
<td>A + B * C - D</td>
</tr>
</tbody>
</table>

No grammar rules apply to empty state. Read next token (A).
Parse of $A + B * C - D$

State:
$A$

Input left: $+ B * C - D$

Only rule that applies is:
$id$ ::= $A$
Parse of $A + B \times C - D$

State:
$id$

Input left: $+ B \times C - D$

Only rule that applies is:
$\langle factor \rangle ::= \langle id \rangle$
Parse of $A + B * C - D$

State:

$<factor>$

Input left: $+ B * C - D$

Only rule that applies is:

$<term> ::= <factor>$

```
<factor>
   <id> A + B * C - D
```
Parse of $A + B \times C - D$

State:
<term>
Input left: $+ B \times C - D$

Only rule that applies is:
<expr> ::= <term>
Parse of $A + B * C - D$

- State:
  - $<\text{expr}>$
  - Input left: $+ B * C - D$

Although $<\text{expr}>$ is the current state, there are still input tokens left, so we’re not done. Read next token (+).

```
<\text{expr}>
  <\text{term}>
    <\text{factor}>
      <\text{id}>
```

A + B * C - D

Copyright © 2002–2017 UMaine School of Computing and Information Science – 24 / 25
Parse of $A + B \times C - D$

State:

\[
\begin{align*}
\text{<expr> +} \\
\text{Input left: B \times C - D}
\end{align*}
\]

No rule applies.
Read next token (B).

\[
\begin{align*}
\text{<expr>}
\end{align*}
\]

\[
\begin{align*}
\text{<term>}
\end{align*}
\]

\[
\begin{align*}
\text{<factor>}
\end{align*}
\]

\[
\begin{align*}
\text{<id>}
\end{align*}
\]

\[
\begin{align*}
\text{A} + \text{B} \times \text{C} - \text{D}
\end{align*}
\]
Parse of $A + B \ast C - D$

State:
\[
\text{<expr>} + B
\]
Input left: $\ast C - D$

Can’t replace whole thing.
Use $\text{id} ::= B$

\[
\text{<expr>}
\]
\[
\text{<term>}
\]
\[
\text{<factor>}
\]
\[
\text{id}
\]

$A + B \ast C - D$
Parse of $A + B \times C - D$

State:
$<expr> + <id>$

Input left: $\times C - D$

Can’t replace whole thing.
Use $<factor> ::= <id>$

```
  <expr>
    <term>
      <factor>
        <id>  <id>
          A    +    B   *    C    -    D
```
Parse of $A + B * C - D$

State:
$<expr> + <factor>$

Input left: $* C - D$

Can’t replace whole thing.
Use $<term> ::= <factor>$

```
<expr>
  <term>
    <factor> <factor>
      <id>    <id>
      A       +       B       *       C       -       D
```
Parse of $A + B * C - D$

State:
$<\text{expr}> + <\text{term}>$

Input left: $* C - D$

Can apply $<\text{expr}> ::= <\text{expr}> + <\text{term}>$

\[
\begin{align*}
\text{expr} & \\
\text{term} & \\
\text{factor} & \\
\text{id} & \\
A & + B & * C & - D
\end{align*}
\]
Parse of $A + B \times C - D$

State:
$<expr>$
Input left: $\times C - D$

Can’t stop -- still have input tokens left.
Read next token ($\times$)
Parse of \( A + B * C - D \)

State:
\[
<\text{expr}> *
\]
Input left: \( C - D \)

No grammar rule produces anything beginning with \( <\text{expr}> * \) -- dead end.
Backtrack to previous state.
Parse of $A + B \times C - D$

State:

<expr> + <term>

Input left: $\times C - D$

Although we could apply $<expr> ::= <expr> + <term>$, we’ve already tried that with no luck -- so read next input token ($\times$).

```
<expr>
   /
  /<term>
 /  /
/    /
<factor> <factor>
   /
  /
/  <id> <id>
/   /A + B
/    /   *
C - D
```
Parse of $A + B * C - D$

State:
$<expr> + <term> *$
Input left: $C - D$

Nothing matches directly with $<term> *$
Read next token (C).

```
<expr>
  <term>  <term>
    <factor> <factor>
      <id>    <id>
```

$A + B * C - D$
Parse of $A + B \times C - D$

State:
$<expr> + <term> \times C$

Input left: $- D$

Nothing matches directly with $<term> \times C$

Use grammar rule: $<id> ::= C$

```
<expr>
   <term>       <term>
   <factor>     <factor>
   <id>         <id>
   A + B \times C - D
```
Parse of $A + B * C - D$

State:
$\langle \text{expr} \rangle + \langle \text{term} \rangle * \langle \text{id} \rangle$

Input left: $- D$

Nothing matches directly with $\langle \text{term} \rangle * \langle \text{id} \rangle$

Use grammar rule: $\langle \text{factor} \rangle ::= \langle \text{id} \rangle$

```
<expr>
  <term> + <term> * <id>

<term>
  <factor>
    <id>

A + B * C - D
```
Parse of $A + B \times C - D$

State:
$\langle \text{expr} \rangle + \langle \text{term} \rangle \times \langle \text{factor} \rangle$

Input left: $- D$

Use grammar rule:
$\langle \text{term} \rangle ::= \langle \text{term} \rangle \times \langle \text{factor} \rangle$

```
<expr>  
   <term>  
      <factor>  
         <id>   
        A  
   +  
   <term>  
      <factor>  
         <id>   
        B  
      \times  
      <factor>  
         <id>   
        C  
    -  
    D
```
Parse of $A + B \times C - D$

State:
\[ \text{expr} + \text{term} \]
Input left: $- D$

Use grammar rule:
\[ \text{expr} ::= \text{expr} + \text{term} \]

```
<expr>
  <term>
  <factor> + <factor>
  <id> + <id> + <id>
  A + B * C - D
```

```
<term>
  <factor>
  <factor> * <factor>
  <id> * <id> * <id>
```

```
<factor>
  <id>
  <id>
  A
```
Parse of \( A + B \cdot C - D \)

State:
\(<\text{expr}>\)

Input left: - D

Although \(<\text{expr}>\) is state, there is still input, so we’re not done.

No matching rules in grammar; so read next input token.
Parse of $A + B * C - D$

- State:
  - $<expr>$ - 
  - Input left: D

- Doesn’t match anything in grammar. Read D from input.

- Example Parse
  - $<expr>$$<term>$$<factor>$$<id>$
    - A + B * C - D
  - Precedence
  - Example Grammar
  - Parsing
  - Example Parse
  - Parsers
Parse of A + B * C - D

State:  
<expr> - D  
Input: none

Doesn’t match anything in grammar directly. So use <id> ::= D

<expr>
  ↓
<term>
  ↓
<factor>
  ↓
{id>
A
  +
B
  *
C
  -
D
Parse of $A + B \times C - D$

State:
$\langle expr \rangle - \langle id \rangle$

Input: none

Doesn't match anything in grammar. So use $\langle factor \rangle ::= \langle id \rangle$
Parse of $A + B \cdot C - D$

State:
$<\text{expr}> - <\text{factor}>$

Input: none

Doesn’t match anything in grammar. So use $<\text{term}> ::= <\text{factor}>$

Copyright © 2002–2017 UMaine School of Computing and Information Science – 24 / 25
Parse of A + B * C - D

State:  
<expr> - <term>
Input: none

Matches grammar rule <expr> ::= <expr> - <term>

Diagram:

```
<expr>  
 |     |  
<term>  
 |  |  |  
<factor> <factor> <factor> <factor>  
 |  |  |  
<id> <id> <id> <id>  
A + B * C - D
```
Parse of \( A + B \times C - D \)
Parsers

- Kinds of grammars/kinds of parsing:
  - LL: left-to-right, leftmost derivation
  - LR: left-to-right, rightmost derivation
  - Different amounts of lookahead: LR(1), e.g.

- LL parsing: e.g., recursive-descent parsers
- LR parsing: e.g., shift-reduce parsers – typically table-driven