Homework & Announcements

- Reading: Chapter
- Homework: Exercises at end
- Due: 11/5
COS 140: Foundations of Computer Science

Specifying Programming Languages: Backus–Naur Form (BNF)

Fall 2018
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
- **Syntactic category**: classes of lexemes that play the same role in the structure of a statement – analogous to parts of speech
- **Tokens**: Low-level syntactic categories corresponding to the lexemes
- **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.: \( aSc \rightarrow aSbSc \)
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.: \( S \rightarrow aS | b \) – generates \( a^+b \)
- **Context-free languages** – programming languages
  - LHS: single symbol; terminals, non-terminals in RHS
  - E.g.: \( S \rightarrow aSb | \text{nil} \) – 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 → 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

\[
\begin{align*}
\langle expr \rangle & \rightarrow \langle var \rangle \mid \langle var \rangle + \langle var \rangle \\
\langle var \rangle & \rightarrow A \mid B \mid C
\end{align*}
\]

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

- 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:
  \[ <expr> \rightarrow <var> | <var> + <expr> \]
- 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**

<table>
<thead>
<tr>
<th>Grammar: &lt;expr&gt;</th>
<th>&lt;var&gt;</th>
<th>&lt;var&gt; + &lt;expr&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;var&gt;</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

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

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

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

Parse tree:
A
Parsing Example

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

Grammar:

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

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

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

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

**Parse tree:**
```
<expr>  \
/   \
<var> + <expr>  \
        |   \
        A  \
```

- **Parsing**
- **Example**
- **Ambiguity**
- **Unambiguous grammar**
- **Precedence**
- **Example Grammar**
- **Parsing**
- **Example Parse**
- **Parsers**
Parsing Example

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

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 ⇒ 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 \(\Rightarrow\) first expression formed (lowest in tree) needs to be to the left
- Replace rule in previous with:

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

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

Example:

```
<var>
  +
  <var>
    |
    <var>
      |
      A
```

\[
<\text{expr}>
  +
  <\text{var}>
    |
    C
```

```
<expr>
```

```
<expr>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<var>
```

```
<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> → <expr> + <var>`

- Right associativity: recursion to the left.
Levels of Precendence

- 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

\[
\text{<expr>} \rightarrow \text{<expr>} \; + \; \text{<term>} \; | \\
\text{<expr>} \; - \; \text{<term>} \; | \\
\text{<term>}
\]

\[
\text{<term>} \rightarrow \text{<term>} \; * \; \text{<factor>} \; | \\
\text{<term>} \; / \; \text{<factor>} \; | \\
\text{<factor>}
\]

\[
\text{<factor>} \rightarrow \left( \text{<expr>} \right) \; | \; \text{id}
\]

\[
\text{id} \rightarrow \text{A} \; | \; \text{B} \; | \; \text{C} \; | \; \text{D}
\]
**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 $A + B * C - D$

State:
(empty)
Input left: $A + B * C - D$

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

State:
A

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

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:
$factor ::= id$
Parse of $A + B \times C - D$

State:

$$<\text{factor}>$$

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

Only rule that applies is:

$$<\text{term}> ::= <\text{factor}>$$

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

State:
$\text{<term>}$

Input left: $+ B * C - D$

Only rule that applies is:
$\text{<expr> ::= <term>}$

$<\text{term}>$

$<\text{factor}>$

$<\text{id}>$

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

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

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

\[ \langle \text{expr} \rangle \]
\[ \langle \text{term} \rangle \]
\[ \langle \text{factor} \rangle \]
\[ \langle \text{id} \rangle \]
\[ A, +, B, \ast, C, -, D \]
Parse of $A + B \cdot C - D$

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

No rule applies.
Read next token (B).

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

\[ A + B \cdot C - D \]
Parse of $A + B \ast C - D$

State:
$<expr> + B$

Input left: $\ast C - D$

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

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

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

Input left: $* C - D$

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

\[
\begin{array}{c}
\quad \langle \text{expr} \rangle \\
\quad \langle \text{term} \rangle \\
\quad \langle \text{factor} \rangle \\
\quad \langle \text{id} \rangle \\
\quad A + B * C - D
\end{array}
\]
Parse of $A + B \times C - D$

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

Input left: $\times C - D$

Can’t replace whole thing.
Use $\langle \text{term} \rangle ::= \langle \text{factor} \rangle$

```
 A + B * C - D
```

Copyright © 2002–2018 UMaine School of Computing and Information Science – 24 / 25
Parse of $A + B \times C - D$

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

Input left: $\times C - D$

Can apply $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$

```
<expr>
   /|
  <term> <term>
   |
  <factor> <factor>
   |
  <id> <id>
```

$A + B \times C - D$
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 ($\times$)

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

<table>
<thead>
<tr>
<th>State:</th>
<th>No grammar rule produces anything beginning with $expr *$ -- dead end. Backtrack to previous state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$expr$</td>
<td>$expr$</td>
</tr>
<tr>
<td>$term$</td>
<td>$term$</td>
</tr>
<tr>
<td>$factor$</td>
<td>$factor$</td>
</tr>
<tr>
<td>$id$</td>
<td>$id$</td>
</tr>
<tr>
<td>$A$</td>
<td>$B$ * $C$ - $D$</td>
</tr>
</tbody>
</table>
Parse of A + B * C - D

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

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

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:
\[
<\text{expr}> \ + \ <\text{term}> \ \times \ C
\]
Input left: $-D$

Nothing matches directly with $<\text{term}> \ \times \ C$
Use grammar rule: $<\text{id}> ::= C$

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

State:
<expr> + <term> * <id>

Input left: - D

Nothing matches directly with <term> * <id>
Use grammar rule: <factor> ::= <id>

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

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

Input left: $- D$

Use grammar rule:
$<term> ::= <term> \times <factor>$

```
<expr>
  /
 <term>            <term>
   \
 <factor>  <factor>  <factor>
     \
 <id>    <id>       <id>
```

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

State:
$<expr> + <term>$

Input left: $- D$

Use grammar rule:
$<expr> ::= <expr> + <term>$
Parse of $A + B \times C - D$

State:
<expr>
Input left: - D

Although <expr> is state, there is still input, so we’re not done.
No matching rules in grammar; so read next input token.
Parse of $A + B \cdot C - D$

State:
<expr> -
Input left: D

Doesn’t match anything in grammar. Read D from input.
Parse of $A + B * C - D$

State:
$\langle expr \rangle - D$

Input: none

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

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

Doesn’t match anything in grammar. So use <factor> ::= <id>
Parse of $A + B * C - D$

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

Doesn’t match anything in grammar.
So use <term> ::= <factor>
Parse of $A + B \times C - D$

State:
$\text{expr} - \text{term}$

Input: none

Matches grammar rule $\text{expr} ::= \text{expr} - \text{term}$
Parse of $A + B \times C - D$

State:
<expr>
Input: none

Done.

<expr>
<expr>
<term>
<term>
<factor>
<factor>
<factor>
<factor>

A + B * 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