Extended BNF

• EBNF is an extension of BNF primarily to improve the readability and writability of grammars
• Optional parts are placed in brackets [ ]

<proc_call> → ident [(<expr_list>)]

<if_stmt> → <if> (<expr>) <stmt> [ else <stmt>]
Extended BNF

• Alternative parts of RHS’s are placed inside parentheses and separated via vertical bars

BNF: \( <\text{term}> \rightarrow <\text{term}> + \text{const} \mid <\text{term}> - <\text{const}> \)

EBNF: \( <\text{term}> \rightarrow <\text{term}> (+\mid-) \text{const} \)
Extended BNF

• Repetitions (0 or more) are placed inside braces { }

\[ \text{<ident>} \rightarrow \text{letter \ {\text{letter|digit}}} \]

\[ \text{<ident\_list>} \rightarrow \text{<identifier> \ {, \ <identifier>}} \]
BNF and EBNF

• BNF

<expr> → <expr> + <term>
| <expr> - <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>

• EBNF

<expr> → <term> { (+ | - ) <term> }
<term> → <factor> { (* | / ) <factor> }
BNF and EBNF

Modify following grammar from BNF to EBNF

**BNF:**
<assign> → <id> = <expr>
=id> → A | B | C
<expr> → <id> + <expr> | <id> * <expr> | ( <expr> ) | <id>

**EBNF:**
<assign> → <id> = <expr>
=id> → A | B | C
<expr> → <expr> (+ | * ) <expr> | (<expr>) | <id>
Recent Variations in EBNF

• Alternative RHS’s are put on separate lines
• Use of a colon instead of $=>$
• Use of $\text{opt}$ for optional parts
• Use of $\text{oneof}$ for choices
Sample Grammar

\[ <\text{pop}> ::= [<\text{bop}>,<\text{pop}>] \]

\[ | \quad <\text{bop}> \]

\[ <\text{bop}> ::= <\text{boop}> \]

\[ | \quad (<\text{pop}>)) \]

\[ <\text{boop}> ::= x | y | z \]
Static Semantics

• Nothing to do with meaning
• Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
• Categories of constructs that are trouble:
  – Context-free, but cumbersome (e.g., types of operands in expressions)
  – Non-context-free (e.g., variables must be declared before they are used)
Dynamic Semantics

- Describe the meaning of programs
  - English language descriptions neither precise nor complete
- "Holy Grail" to be able to prove a program correct
  - No universal notation for semantics
Describing Semantics

- **Programmers** need to know precisely what the statements of a language do before they can use them.
- **Compiler writers** must know exactly what language constructs *mean* to correctly design implementations for them.
- If there were a precise semantics specification, programs could be proven correct without testing.
Operational semantics

- Describe the meaning of a statement or program by specifying the effects of running it on a machine.

- The effects on the machine are viewed as the sequence of changes in its state.

- Often we write a small program in a new language to determine the meaning of some programming language construct.
Axiomatic Semantics

• Based on mathematical logic
• Rather than directly specifying the meaning of a program, axiomatic semantics specifies what can be proven about the program.
• Assertions are attached to program statements describing constraints on variable at that point in program.
Axiomatic Semantics (cont.)

• An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution
• An assertion following a statement is a postcondition
• A weakest precondition is the least restrictive precondition that will guarantee the postcondition
Axiomatic Semantics

• Postcondition assertions describe result of the statement
• Precondition assertions describe state before statement is executed
• Can derive the weakest precondition that guarantees a particular postcondition
Axiomatic Semantics Form

• Pre-, post form: \{P\} statement \{Q\}

• An example
  o \(a = b + 1\) \(\{a > 1\}\)
  o One possible precondition: \(\{b > 10\}\)
  o Weakest precondition: \(\{b > 0\}\)