Extended BNF

* Optional parts are placed in brackets []
  + <proc\_call> -> ident [(<expr\_list>)]
* Alternative parts of RHSs are placed inside parenthesis and separated via vertical bars
  + <term> -> <term> (+|-) const
* Repetitions (0 or more) are placed inside braces {}
  + <ident> -> letter {letter|digit}

BNF and EBNF

* BNF
  + <expr> -> <expr> + <term>

| <expr> - <term>

| <term>

* + <term> -> <term> \* <factor>

| <term> / <factor>

| <factor>

* EBNF
  + <expr> -> <term> { ( + | - ) <term> }
  + <term> -> <factor> { ( \* | / ) <factor> }
    - Int x = “banana”;
      * Syntax OK, Static Semantics Wrong (type checking)

Attribute Grammars

* An attribute grammar is a context-free grammar G = (S, N, T, P) with the following additions:
  + For each grammar symbol x there is a set A(x) of attribute values
  + Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  + Each rule has a (possibly empty) set of predicates to check for attribute consistency
* Rule
  + X(0) -> X(1) … X(n)
  + Functions of the form S(X(0)) = f(A(X(1))), …, A(X(n)) define synthesized attributes
  + Functions of the form I(X(j)) = f(A(X(0))), …, A(X(j-1)) define inherited attributes
* An Example
  + Syntax
    - <assign> -> <var> = <expr>
    - <expr> -> <var> + <var> | <var>
    - <var> -> A | B | C
  + actual\_type: synthesized for <var> and <expr>
  + expected\_type: inherited for <expr>
  + Syntax Rule: <expr> -> <var>[1] + <var>[2]
  + Semantic Rules:
    - <expr>.actual\_type <- if
      * (<var>[1].actual\_type = int) and
      * (<var>[2].actual\_type = int)
        + then int
    - else
      * real
    - end if
  + Predicate:
    - <expr>.expected\_type == <expr>.actual\_type
* How are attribute values computed?
  + If all attributes were inherited, the tree could be decorated in top down order
  + If all attributes were synthesized the tree could be decorated in bottom up order

Semantics

* There is no single widely acceptable notation of formalism for describing semantics
* Several needs for a methodology and notation for semantics:
  + Programmers need to know what statements mean
  + Compiler writers must know exactly what language constructs do
  + Correctness proofs would be possible
  + Compiler generators would be possible
  + Designers could detect ambiguities and inconsistencies
* Operational Semantics
  + Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.)
* Denotational Semantics
  + Based on recursive function theory
  + The most abstract semantics description method
* Axiomatic Semantics
  + Based on formal logic (predicate calculus)
  + Original purpose was for formal program verification
  + Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
  + The logic expressions are called assertions
  + 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
  + Form
    - Pre-, post form: {P} statement {Q}
    - Example
      * a = b + 1 { a > 1 }
      * one possible precondition: { b > 10 }
      * weakest precondition: { b > 0 } because it’s obvious given that { a > 1 }
  + Program Proof Process
    - The postcondition for the entire program is the desired result
      * Work back through the program to the first statement. If the precondition on the first statement Is the same as the program specification, it is correct
  + Loops - An inference rule for logical pretest loops