Programming Languages & Paradigms
PROP HT 2011

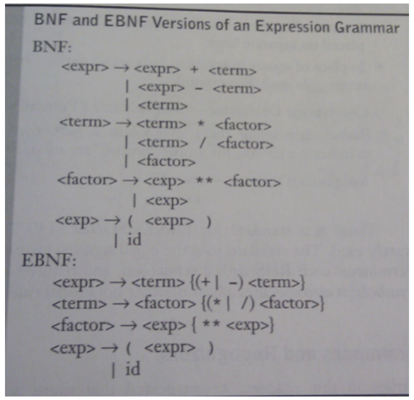
Lecture 3

Expressions, assignments and statements

Beatrice Åkerblom
beatrice@dsv.su.se

Questions from you

- BNF with $::=$ or $\longrightarrow$?
- Identifier names in grammars



BNF and EBNF Versions of an Expression Grammar

```
BNF:
    <expr> → <expr> + <term>
           | <expr> - <term>
           | <term>
    <term> → <term> * <factor>
           | <term> / <factor>
           | <factor>
    <factor> → <exp> ** <factor>
             | <exp>
    <exp> → ( <expr> )
          | id
EBNF:
    <expr> → <term> {(+ | -) <term>}
    <term> → <factor> {(* | /) <factor>}
    <factor> → <exp> { ** <exp>}
    <exp> → ( <expr> )
          | id
```

Arithmetic Expressions

Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of **operators**, **operands**, **parentheses**, and **function calls**

```
15 * (a + b) / log(x)
```

  - operator precedence rules
  - operator associativity rules
  - order of operand evaluation
  - operand evaluation side effects
  - operator overloading
  - mode mixing expressions

Operators

- A **unary** operator has one operand

```
    -7
```

- A **binary** operator has two operands

```
    7 + 2
```

- A **ternary** operator has three operands

```
    avg = (count == 0)? 0 : sum / count;
```

- The operator **precedence** rules for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated

Precedence levels

- Typical precedence levels
  - parentheses
  - unary operators
  - ** (exponent - if the language supports it)
  - *, /
  - +, -

Precedence levels, cont'd

- More **unusual** (?) precedence levels
  - Smalltalk - What's an operator?
    unary > binary > keyword > **assignment**
  - Forth, using postfix notation
    - 1 2 + 3 * 6 + 2 3 + /
    - (((1 + 2) * 3) + 6) / (2 + 3)
  - Of course precedence and associativity rules can be overridden with parentheses

Associativity

- The operator **associativity** rules for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated

- Typical associativity rules:
  - Left to right, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., FORTRAN)

What Happens on Evaluation?

- **Variables**: just fetch the value
- **Constants**: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
- **Parenthesized expressions**: evaluate all operands and operators first
- **Function calls**: order of evaluation may be crucial

```
a = 10;
b = a + foo(&a);
```

- Functional side effects (when a function changes a **two-way parameter** or a **nonlocal variable**)

Possible Solution 1(2)

- Write the language definition to disallow functional side effects
- No two-way parameters in functions
- No nonlocal references in functions

- Advantage: it “works”
- Disadvantage: Programmers want the flexibility of two-way parameters and nonlocal references. In other words this solution makes the language too restrictive and less “writable”.
- Copy all data structures always?
- Note also that the usage of global values is important method for execution speed improvement. The common tradeoff- safety, elegance, readability, etc. for performance

Possible Solution 2(2)

- Write the language definition to demand that operand evaluation order be fixed
- Disadvantages:
  - limits some compiler optimizations using operand reordering
  - does not really solve the problem

Overloaded Operators

- Some overloading is common (e.g., + for int and float, then we can go beyond arithmetic and see it use for strings)

- Potential trouble (e.g., * in C and C++, both pointers and arithmetic use)
- Loss of compiler error detection (omission of an operand should be a detectable error)
- Some loss of readability
- Can be avoided by introduction of new symbols (e.g., Pascal’s div)

- C++ and Ada allow user-defined overloaded operators
- Potential problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense

Type Conversions

- **Narrowing** conversion:
  - converts an object to a type that cannot include all of the values of the original type e.g., float to int usually somewhat troubling (downcast)
- **Widening** conversion:
  - converts an object to a type that can include at least approximations to all of the values of the original type e.g., int to float usually not so troubling (upcast)

Converts???

Eiffel

Type Conversions, cont’d

- A **coercion** is an implicit type conversion

- **Coercions** reduce the benefits of type checking; they may cause reliability problems
- On the other hand, coercions provide flexibilities
- In Ada, there are virtually no coercions in expressions

Explicit Type Conversions

- Explicit Type Conversions are often called casts

  - Ada: `FLOAT(INDEX) -- INDEX is INTEGER type`
  - Java: `(int) speed /* speed is float type */`

Substitution

- Subtype has same attributes and behaviour as supertype
- An object can be used wherever an object from a supertype is expected.
- The Liskov substitution principle:
  - Let *q(x)* be a property provable about objects *x* of type *T*. Then *q(y)* should be true for objects *y* of type *S* where *S* is a subtype of *T*.
- *A <: B* -- *A* is a subtype of *B*
- Any expression of type *A* may also be given type *B* if *A <: B*
  - in any situation
  - with no observable effect

Substitution in DPLs and STLs

- The importance of the principle of substitution differs between dynamically typed and statically typed languages
  - in statically typed languages objects are (typically) characterised by their class
  - in dynamically typed languages objects are (typically) characterised by their behaviour

Relational & Boolean Expressions

Relational Expressions

- Use **relational operators** and operands of various types
- Evaluate to some Boolean representation
- Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)

- Operands are Boolean and the result is Boolean

But not in C…

- C has no Boolean type--it uses **int** type
  - 0 for false
  - nonzero for true
- One odd characteristic of C's expressions:
  `a < b < c`
  is a legal expression, but the result is not what you might expect:
  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., c)

Short Circuit Evaluation

- The result is determined without evaluating all of the operands and/or operators
- Examples:

```
(13*a) * (b/13–1)
```

  - If a is zero, there is no need to evaluate (b/13-1)

```
(a > 3 && b < 5 * d)
```

  - If a <= 3,there is no need to evaluate b < 5 * d
  - How about

```
(c <= 7 || a != 3 * b)
```

Assignment

Assignment Statements

- General syntax
  <target_var> <assign_operator> <expression>
- Assignment is dominant statement in an imperative language, all about putting something in memory
- The assignment operator symbol itself can be the biggest issue in expressions

  = FORTRAN, BASIC, PL/I, C, C++, Java
  := ALGOLs, Pascal, Ada, Smalltalk
- = Can be bad if it is overloaded for the relational operator for equality (PL/I)
- Still difficult to see difference = and ==, e.g. known from C…

Ruby

More complicated assignments

- Multiple targets (PL/I)
  `A,B=10 /*does A get 10 or is it undefined*/`
- Conditional targets (C, C++, and Java)
  `(first == true) ? total : subtotal = 0`
- Compound assignment operators (C, C++, JavaScript and Java)
  Shorthand form to address commonly needed assignments;
  `sum += nextval;` vs. `sum = sum + nextval;`
- Unary assignment operators (C, C++, and Java)
  `a++; a--; --a; ++a;`
- C, C++, and Java treat = as an arithmetic binary operator e.g.
  `a = b * (c = d * 2 + 1) + 1`
  – This is inherited from ALGOL but seems to cause some readability concerns

Assignment Statements, cont'd

- In C, C++, and Java, the assignment statement produces a result so, they can be used as operands in expressions e.g.
  `while((ch=getchar()!=EOF){...}`
- Smalltalk: What is an assignment operator?
  `myVariable := 3.`
  assignment operator

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode, for example
  ```
  int a = 2, b = 3;
  float c;
  c = a / b;
  ```
- C allows mixed-mode assignments; The coercion takes place only after the right- side expression has been evaluated
- Java and C# allow only widening assignment coercions
- In Ada, there is no assignment coercion

Control Flow

Levels of Control Flow

- Within expressions
- Among program statements
- Among program units

Control Structure

- A **control structure** is a **control statement** and the **statements** whose execution it controls

- What control statements should a language have, beyond selection and pretest logical loops?
- The number of control statements in the language effects readability and writability. Too many makes the language is hard to learn, too few it can be confusing or difficult to write in.

Selection Statements

- A selection statement provides the means of choosing between two or more execution paths in a program
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- General (?) form:

```
if control_expression
  then clause
  else clause
```

- Smalltalk: What is a selection statement?

Multiple-Way Selection Statements

- Allow the selection of one of any number of statements or statement groups
- Design Issues:
  - What is the form and type of the control expression?
  - How are the selectable segments specified?
  - Is execution flow through the structure restricted to include just a single selectable segment?
  - How are the case values specified?
  - What is done about unrepresented expression values?

C's switch statement

```
switch (expression) {
    case constant_expression_1 : statement_1;
    ...
    case constant_expression_n : statement_n;
    [default: statement_n+1]
}
```

- Control expression only integer type
- Selectable segments can be statement sequences, blocks, or compound statements
- Construct is encapsulated
- Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments) (a trade-off between reliability and flexibility--convenience)
  - To avoid it, the programmer must supply a break statement for each segment.
- `default` clause is for unrepresented values (if there is no default, the whole statement does nothing)

Perl

Break Statements

- Suggested this approach in multi-way is a cause of programming problems.
- Break statements are actually a restricted form of a GOTO
  - Considered less harmful* because they tend to jump to areas near-by inspected code
  - Some language support labeled breaks
- We see break and continue also used of course in loop control

* See reference section

Multiple-Way Selection Using If

- Multiple Selectors can appear as direct extensions to two-way selectors, using `else-if` clauses, for example in Ada:

```
if ...
   then ...
elsif ...
   then ...
elsif ...
   then ...
   else ...
end if
```

- Far more readable than deeply nested if's
- Allows a Boolean gate on every selectable group

Iteration

Iterative Statements

- The repeated execution of a statement or compound statement is accomplished either by **iteration** or **recursion**

- General design issues for iteration control statements:
  - How is iteration controlled?
  - Where is the control mechanism in the loop?

Counter-Controlled Loops

- A counting iterative statement has a **loop variable**, and a means of specifying the initial and terminal, and stepsize values
- The initial and terminal, and stepsize specifications of a loop are called the loop parameters

- Design issues:
  - Type and scope of the loop variable?
  - Value of the loop variable at loop termination?
  - Legal or not for loop variable / loop parameters to be changed in the loop body (and if so, does the change affect loop control)?
  - Loop parameters evaluated only once, or once every iteration?

Counter-Controlled Loops - C

- The expressions can be whole statements, or even statement sequences, with the statements separated by commas

```
for ([expr_1] ; [expr_2] ; [expr_3]) statement
```

- The value of a multiple-statement expression is the value of the last statement in the expression e.g., `for(i=0,j=10;j==i; i++)`
- If the second expression is absent, it is an infinite loop

```
for (count1 = 0, count2 = 1.0;
count1 <= 10 && count2 <= 100.0;
sum = ++ count1 + count2, count2 *= 2.5);
```

Counter-Controlled Loops - C++

- C++ differs from C in two ways:

- The control expression can also be Boolean
- The initial expression can include variable definitions (scope is from the definition to the end of the loop body)

```
for (int count = 0; count < len; count++) {...}
```

- Java and C#
- Differs from C++ in that the control expression must be Boolean

Logically-Controlled Loops

- Design Issues:
  - Pre-test or post-test?
  - Should this be a special case of the counting loop statement (or a separate statement)?

```
while (ctrl_expr)        do
   loop body                loop body
                         while (ctrl_expr)
```

Examples

- **Pascal** has separate pretest and posttest logical loop statements (`while-do` and `repeat-until`)
- **C** and **C++** have both pre-test and post- test logical loop statements , but the *control expression for the post-test version is treated just like in the pre-test case* (`while-do` and `do-while`)
- **Java** is like C, except the control expression must be Boolean (and the body can only be entered at the beginning -- Java has no goto)
- **Ada** has a pretest version, but no posttest
- **FORTRAN** 77 and 90 have neither, just a counter loop
- **Perl** has two pretest logical loops, `while` and `until`, but no posttest logical loop

User-Located Loop Control Mechanisms

- Design issues:
  - Should the conditional be part of the exit?
  - Should the mechanism be allowed in an already controlled loop?
  - Should control be transferable out of more than one loop? (nesting)
- **C** , **C++**, **Ruby**, and **C#** have unconditional unlabeled exits (break) for any loop or switch; one level only
- **Java** and **Perl** have unconditional labeled break statement: control transfers to the label
- An alternative: continue statement; it skips the remainder of this iteration, but does not exit the loop

Examples in Java

```
for (row = 0; row < numRows; row++)
   for (col = 0; col < numCols; col++) {
      sum += mat[row][col];
      if (sum > 1000.0) {
         break;
}

for (x = 1; x <= 10; x++) {
   if (x == 5) { continue; }
   printf("%d ", x);
}

OuterLoop:
for (row = 0; row < numRows; row++)
   for (col = 0; col < numCols; col++) {
      sum += mat[row][col];
      if (sum > 1000.0) break OuterLoop;
   }
```

Iteration Based on Data Structures

- Use order and number of elements of some data structure to control iteration
- Control mechanism is a call to a function that returns the next element in some chosen order, if there is one; else exit loop
- C's `for loop` can be used to build a user-defined iterator e.g.
  `for (p=hdr; p; p=next(p)) { ... }`

Unconditional Branching

- Transfers execution control to a specified place in the program
- Represented one of the most heated debates in 1960's and 1970's
- Well-known mechanism: **`goto` statement**
- Major concern: Readability
- Some languages do not support goto statement (e.g., Module-2 and Java)
- C# offers goto statement (can be used in switch statements)
- Loop exit statements are restricted and somewhat camouflaged goto's

The End

References

- Sebesta, R - Concepts of Programming Languages
- For * see Dijkstra, E - "Letters to the editor: go to statement considered harmful", 1968 (http://dl.acm.org/citation.cfm?doid=362929.362947)
- Try Smalltalk http://www.pharo-project.org/home