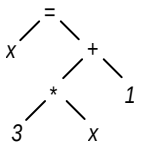
How to Build Abstract Syntax Trees with Coco/R

Hanspeter Mössenböck
September 2011

Many compilers translate a source program into an *abstract syntax tree* (AST), which serves as an internal program representation on which optimizations can be performed and from which target code can be generated. An abstract syntax tree differs from a *concrete syntax tree* (or *parse tree*) in that it does not reflect the parsed productions but rather the logical structure of the compiled program. Its inner nodes are operators and its leaves are operands. An AST for the statement `x = 3 * x + 1;` could look as follows:



Some compiler generators use special notations for describing the transformation of the source program into an AST. In Coco/R[1] we deliberately decided to avoid such special notations and rather build the AST with ordinary semantic actions. This simplifies the compiler description and gives the compiler writer more flexibility. The following tutorial shows how to build an AST for programs of a simple Java-like language.

Basic Idea

The general idea is that every nonterminal symbol has an output attribute which returns the AST of this nonterminal. The AST of a production's left-hand-side nonterminal is built from the ASTs obtained from the right-hand-side nonterminals. In a simplified pseudo-notation the production

```
Expr ↑e = Term ↑e1 "+" Term ↑e2          (. e = new BinExpr(e1, Operator.PLUS, e2); .).
```

obtains the subtrees *e1* and *e2* from the Terms and creates from it a new subtree *e* with the operator "+" as its root and *e1* and *e2* as its children.

The nodes of the AST are objects of classes derived from a common base class *Node*. For the above example we could have the following types (here denoted in Java):

```
class Node {}                       // base class of all nodes

class Expr extends Node {}          // base class of nodes that form (sub-) expressions

class BinExpr extends Expr {        // class describing binary expressions
    Operator op;
    Expr left;                      // left sub-expression
    Expr right;                     // right sub-expression
    BinExpr (Expr e1, Operator o, Expr e2) { op = o; left = e1; right = e2; }
}

enum Operator { PLUS, MINUS, TIMES, DIV, ... }
```

[1] http://ssw.jku.at/coco/

The full Coco/R production for *Expr* would read as follows:

```
Expr <out BinExpr e>          (. Operator op; Expr e2; .)
=  Term <out e>
   {  AddOp <out op>
      Term <out e2>           (. e = new BinExpr(e, op, e2); .)
   }.
```

Note, that no special notation is necessary for describing the construction of the AST. Everything can be done with the familiar Coco/R notation for productions, attributes and semantic actions.

In the following sections we first introduce our sample language *Taste* and then show how to build abstract syntax trees for expressions, statements, declarations and procedures of this language.

The Input Language Taste

In order to explain the construction of an AST we introduce a simple Java-like language that we call *Taste*. It features programs with global variables and parameterless procedures. Procedures can have local variables. The only two types in *Taste* are *int* and *bool*. The language has been kept simple in order to make this tutorial short. Readers should find it straightforward to extend the language with more sophisticated features.

```
Taste      =  "program" ident "{" { VarDecl | ProcDecl } "}".
VarDecl    =  Type ident { "," ident } ";".
Type       =  "int" | "bool".
ProcDecl   =  "void" ident "(" ")" Block.
Block      =  "{" { Stat | VarDecl } "}".
Stat       =  ident "=" Expr ";"
           |  ident "(" ")" ";"
           |  "if" "(" Expr ")" Stat [ "else" Stat ]
           |  "while" "(" Expr ")" Stat
           |  "read" ident ";"
           |  "write" Expr ";"
           |  Block.
Expr       =  SimExpr [ RelOp SimExpr ].
SimExpr    =  Term { AddOp Term }.
Term       =  Factor { MulOp Factor }.
Factor     =  ident | number | "-" Factor | "true" | "false".
RelOp      =  "==" | "<" | ">".
AddOp      =  "+" | "-".
MulOp      =  "*" | "/".
```

Abstract Syntax Trees for Expressions

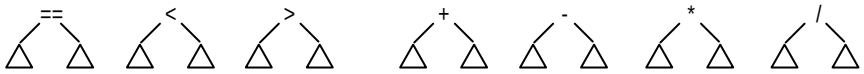
When designing the AST for a language we first have to think about the kinds of nodes that we need and how to create and link those nodes. Each node type is implemented as a class that has fields for the children of this node as well as a constructor for creating the node and linking it with its children. All nodes are derived from a common base class *Node* that can contain general information about the node (e.g., position information). In order to keep our example simple, however, our node class is empty.

```
class Node {}
```

Since we are designing abstract syntax trees for expressions we introduce a class *Expr*, which is the base class of more specific expression nodes. For reasons of simplicity, again, this class is empty in our example.

```
class Expr extends Node {}
```

Now we have to think about which kinds of expressions we have in *Taste* and how we want to represent them in the AST. In *Taste* there are *binary expressions*



unary expressions

and *leaf expressions* (identifiers, numbers and boolean constants).

ident intCon boolCon

Therefore we design classes that represent binary expressions, unary expressions and leaf expressions:

```
class BinExpr extends Expr {
    Operator op;
    Expr left, right;
    BinExpr (Expr e1, Operator o, Expr e2) { op = o; left = e1; right = e2; }
}

class UnaryExpr extends Expr {
    Operator op;
    Expr e;
    UnaryExpr (Operator x, Expr y) { op = x; e = y; }
}

class Ident extends Expr {
    Obj obj;
    Ident (Obj o) { obj = o; }
}

class IntCon extends Expr {
    int val;
    IntCon (int x) { val = x; }
}

class BoolCon extends Expr {
    boolen val;
    BoolCon (boolean x) { val = x; }
}
```

The type *Obj* that is used in class *Ident* denotes the declaration of a named object in the symbol table (see later). The type *Operator* is defined as follows:

```
enum Operator { EQU, LSS, GTR, ADD, SUB, MUL, DIV }
```

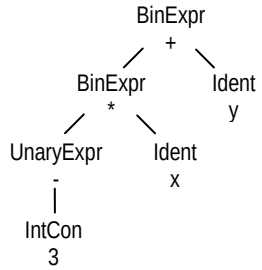
Finally, we write a Coco/R attributed grammar in which we build the abstract syntax trees for expressions:

```
Expr <out Expr e>                (. Operator op; Expr e2; .)
=  SimExpr <out e>
   [   RelOp <out op>
       SimExpr <out e2>          (. e = new BinExpr(e, op, e2); .)
   ].

SimExpr <out Expr e>             (. Operator op; Expr e2; .)
=  Term <out e>
   {   AddOp <out op>
       Term <out e2>             (. e = new BinExpr(e, op, e2); .)
   }.
```

```
Term <out Expr e>                (. Operator op; Expr e2; .)
=   Factor <out e>
    {   MulOp <out op>
        Factor <out e2>          (. e = new BinExpr(e, op, e2); .)
    }.

Factor <out Expr e>              (. String name; .)
=                                (. e = null; .)
(   Ident <out name>             (. e = new Ident(curProc.find(name)); .)
|   number                       (. e = new IntCon(Integer.parseInt(t.val)); .)
|   "-" Factor <out e>           (. e = new UnaryExpr(Operator.SUB, e); .)
|   "true"                       (. e = new BoolCon(true); .)
|   "false"                      (. e = new BoolCon(false); .)
 ).

Ident <out String name>
=   ident                        (. name = t.val; .).

AddOp <out Operator op>
=                                (. op = Operator.ADD; .)
(   "+"
|   "-"                          (. op = Operator.SUB; .)
).

MulOp <out Operator op>
=                                (. op = Operator.MUL; .)
(   "*"
|   "/"                          (. op = Operator.DIV; .)
).

RelOp <out Operator op>
=                                (. op = Operator.EQU; .)
(   "=="
|   "<"                          (. op = Operator.LSS; .)
|   ">"                          (. op = Operator.GTR; .)
).
```

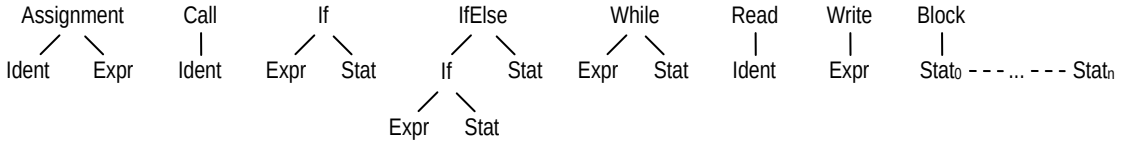
The method *curProc.find*(*name*) looks up a name in the local declarations of the current procedure and its enclosing program and returns the *Obj* that represents the declaration of this name (see later).

For example, the expression -3 * x + y is translated to the following AST:



Abstract Syntax Trees for Statements

Like for expressions, we have to think about which kinds of statements we have in our language and how we want to represent them as abstract syntax trees. In Taste, we have assignments, procedure calls, if statements (with and without else part), while statements, read statements, write statements and blocks. These should be represented as follows:



Note, that an if statement with an else part is represented as an *IfElse* node whose left son is an if statement without else part and whose right son is the statement of the else part. A block is represented as a list of statements.

Again we design classes that represent these statement nodes:

```
class Stat extends Node {}

class Assignment extends Stat {
    Obj  left;
    Expr right;
    Assignment (Obj o, Expr e) { left = o; right = e; }
}

class Call extends Stat {
    Obj proc;
    Call (Obj o) { proc = o; }
}

class If extends Stat {
    Expr cond;
    Stat stat;
    If (Expr e, Stat s) { cond = e; stat = s; }
}

class IfElse extends Stat {
    Stat ifPart;
    Stat elsePart;
    IfElse (Stat i, Stat e) { ifPart = i; elsePart = e; }
}

class While extends Stat {
    Expr cond;
    Stat stat;
    While (Expr e, Stat s) { cond = e; stat = s; }
}

class Read extends Stat {
    Obj obj;
    Read (Obj o) { obj = o; }
}

class Write extends Stat {
    Expr e;
    Write (Expr x) { e = x; }
}

class Block extends Stat {
    List<Stat> stats = new ArrayList<Stat>();
    void add (Stat s) { stats.add(s); }
}
```

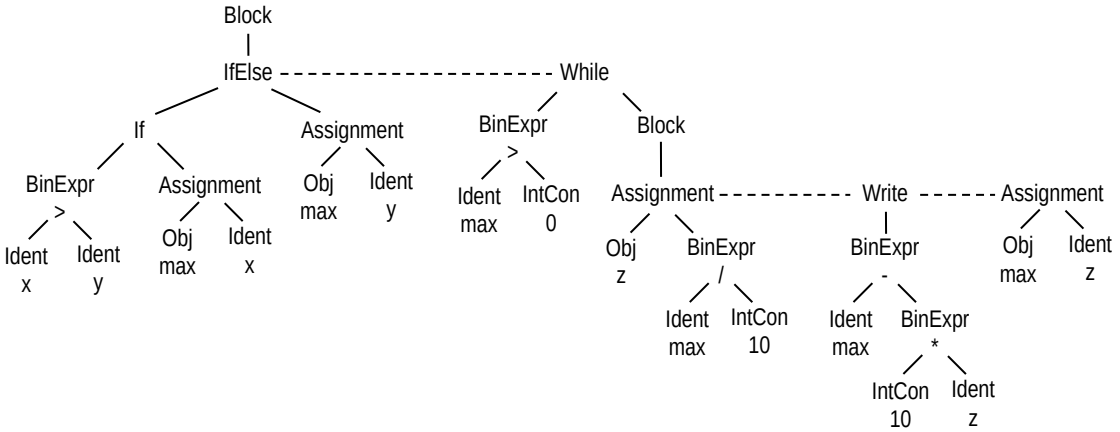
Finally we write an attributed grammar that builds abstract syntax trees for statements:

```
Block <out Block b>              (. Stat s; .)
=   "{"                          (.b = new Block(); .)
    {   Stat <out s>             (. b.add(s); .)
    |   VarDecl
    }
    "}" .

Stat <out Stat s>                (. String name; Expr e; Stat s2; Block b; .)
=                                (. s = null; .)
(   Ident <out name>             (. Obj obj = curProc.find(name); .)
    (   "=" Expr <out e> ";"     (. s = new Assignment(obj, e); .)
    |   "("  ")"  ";"            (. s = new Call(obj); .)
    )
|   "if" "(" Expr<out e> ")"
    Stat <out s>                 (. s = new If(e, s); .)
    [   "else" Stat <out s2>     (. s = new IfElse(s, s2); .)
    ]
|   "while" "(" Expr <out e> ")"
    Stat<out s>                  (. s = new While(e, s); .)
|   "read" Ident <out name> ";"  (. s = new Read(curProc.find(name)); .)
|   "write" Expr <out e> ";"     (. s = new Write(e); .)
|   Block <out b>                (. s = b; .)
) .
```

For example, the *Taste* statement block

```
{   if (x > y) max = x; else max = y;
    while (max > 0) {
        z = max / 10;
        write max - 10 * z;
        max = z;
    }
}
```

is translated to the following AST:



Abstract Syntax Trees for Declarations and Procedures

Declarations introduce names and associate them with properties such as a type or an address. Every declaration belongs to the program unit in which it appears, i.e., a procedure contains the declarations of its local variables and a program contains the declarations of the global variables and procedures. All declarations together form the *symbol table* of the compiled program.

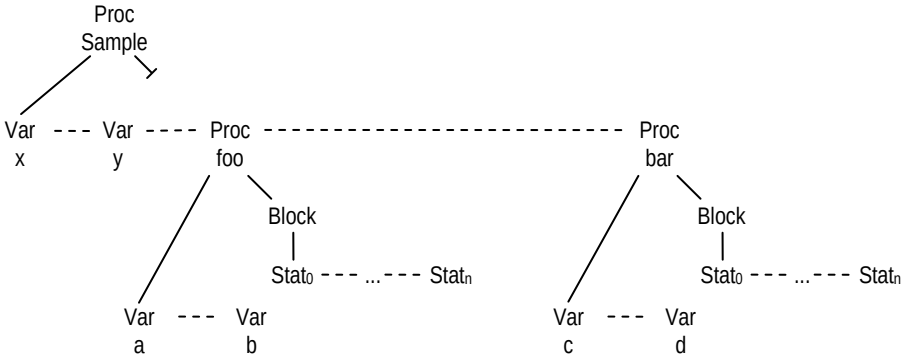
In *Taste*, we store the symbol table as part of the AST although it could be kept as a separate data structure as well. Every declaration creates a *Var* node or a *Proc* node which are subclasses of *Obj* nodes. The program itself is also represented as a *Proc* node. A *Proc* node maintains a list of its local declarations using the methods *add*(*obj*) and *find*(*name*).

```
enum Type { UNDEF, INT, BOOL }

class Obj extends Node {          // any declared object that has a name
    String name;                  // name of this object
    Type type;                    // type of this object (UNDEF for procedures)
    Obj (String s, Type t) { name = s; type = t; }
}

class Var extends Obj {           // variable
    int adr;                      // address in memory
    Var (String name, Type type) { super(name, type); }
}

class Proc extends Obj {          // procedure (also used for the program)
    List<Obj> locals;             // objects declared in this procedure
    Block block;                  // block of this procedure (null for the main program)
    int nextAdr;                  // next free address in this procedure
    Proc program;                 // link to the Proc node of the main program or null

    Proc (String name, Proc program) {
        super(name, Type.UNDEF);
        locals = new ArrayList<Obj>();
        this.program = program;
    }

    void add (Obj obj) {
        for (Obj o: locals) {
            if (o.name.equals(obj.name)) SemErr(obj.name + " declared twice");
        }
        locals.add(obj);
        if (obj instanceofVar) ((Var)obj).adr = nextAdr++;
    }

    Obj find (String name) {
        for (Obj o: locals) { if (o.name.equals(name)) return o; }
        if (program != null) {
            for (Obj o: program.locals) { if (o.name.equals(name)) return o; }
        }
        SemErr(name + " undeclared");
        return new Obj("_undef", Type.INT); // dummy
    }
}
```

The AST of a procedure has two subtrees, one for its local declarations and one for its block. For example, the *Taste* program

```
program Sample {
    int x;
    bool y;
    void foo() { int a, b; ... }
    void bar() { int c, d; ... }
}
```

is translated to the following AST:



Here is the attributed grammar that processes the *Taste* declarations:

```
Taste                              (.  String name; .)
= "program" Ident <out name>       (.  curProc = new Proc(name, null); .)
    "{"
    { VarDecl | ProcDecl }
    "}" .

VarDecl                            (.  String name; Type type; .)
=   Typ <out type>
    Ident <out name>               (.  curProc.add(new Var(name, type)); .)
    { "," Ident <out name>         (.  curProc.add(new Var(name, type)); .)
    } ";" .

Typ <out Type type>
=                                  (.  type = Type.INT; .)
    ( "int"
    | "bool"                       (.  type = Type.BOOL; .)
    ) .

ProcDecl                           (.  String name; .)
=   "void" Ident <out name>        (.  Proc oldProc = curProc;
                                       curProc = new Proc(name, oldProc);
                                       oldProc.add(curProc); .)
    "("  ")"
    Block <out curProc.block>      (.  curProc = oldProc; .) .
```

This concludes our tutorial. We have shown that it is straightforward to build an abstract syntax tree with Coco/R by just using attributes and semantic actions. No special notation was necessary to specify the AST and its construction. The nodes of the AST are declared as classes that are linked with their children in their constructors. If the AST should store additional information such as types or optimization hints this information can easily be added in the form of additional fields of the node classes. This gives the compiler writer full control over the contents of the AST nodes and how the nodes are linked into a tree.

The full source code of this example including the attributed grammar can be obtained from http://ssw.jku.at/coco/Doc/AST.zip.