Abstract Syntax Trees

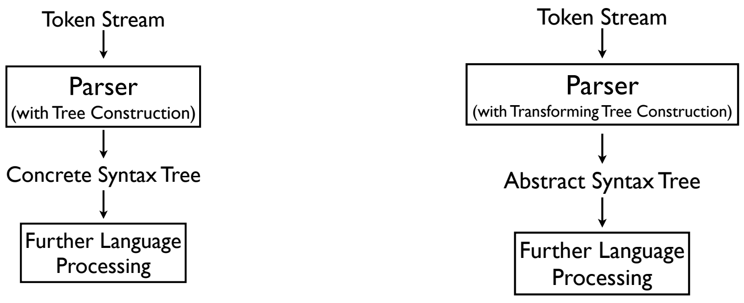
April 6, 2017

Contents

1 Abstract syntax vs. Concrete syntax

Let $\mathcal{P}$ be some (programming) language with CFG Γ and $p \in \mathcal{P}$ be a program. The **concrete syntax** of a programming language $\mathcal{P}$ determines the actual textual representation of programs (incl. keywords, separators, etc.). The syntax tree of p according to Γ is the **concrete syntax tree of** p.
The **abstract syntax** of $\mathcal{P}$ describes the tree structure of programs in a form that is sufficient and suitable for further processing in subsequent phases.The abstract syntax

usually more compact as it focuses on the semantically relevant parts and does not need to deal with white space, comments, separators, keywords and similar things. However, to improve error messages for later phases (e.g. during semantical analysis) or debugging, the abstract syntax tree usually also comprises information about source code positions. A tree for representing a program p according to the abstract syntax is called the **abstract syntax tree of** p.
Programming language specifications have to define all aspects of the concrete syntax of the corresponding language, while the abstract syntax is often not specified.
It is possible to use either the concrete or the abstract syntax as interface between the parser and the following phases.



When using the concrete syntax, the tree construction can be done automatically by parser. No additional specification of the abstract syntax is required, and no translation to an abstract syntax needs to be specified.
However, the simplified structure of the abstract syntax makes it much more feasible for later phases. Some tools, like SableCC, offer the possibility to specify abstract syntax and the translation as part of the parser specification.

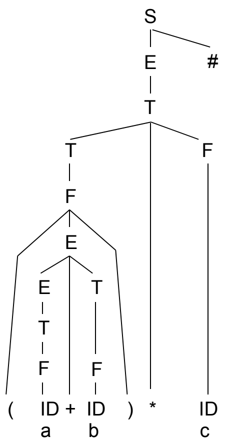
Example: Concrete vs. abstract syntax For the expression language, the concrete syntax can be specified as:

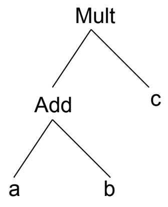
$$\begin{array}{lcl} S & \rightarrow & E\# \\ E & \rightarrow & T + E \mid T \\ T & \rightarrow & F * T \mid F \\ F & \rightarrow & (E) \mid ID \end{array}$$

A corresponding abstract syntax is usually expressed using algebraic data types (see below):

```
Exp = Add | Mult | Ident
Add  (Exp left, Exp right)
Mult (Exp left, Exp right)
Ident(String name)
```

For the expression `(a + b) * c`, the respective syntax trees are :





When transforming from concrete to abstract syntax, the tree structure can usually be simplified. For example, the abstract syntax does not require different nonterminals to reflecting operator precedences. Instead, the precedence is represented in the tree structure itself.

Abstract syntax: Specification and tree construction For representing abstract syntax trees, we use algebraic data types.

Definition 1. ***Algebraic data types*** *are defined using (possibly recursive) sum types and product types.*

- *A sum type $T = T_1 \,|\, T_2 \,|\, \ldots \,|\, T_n$ denotes different alternatives for a type T. Each value of type T falls in exactly one of the cases T_1 to T_n.*
- *A product type $C(T_1, T_2, \ldots, T_m)$ defines a tuple (or record) of values. Each value of type C contains values of type T_1 to T_m as part of the value.*

In most programming languages the notation of sum and product types is combined. For example the abstract syntax of the expression language (see above) can also be written as follows:

```
Exp = Add(Exp left, Exp right)
    | Mult(Exp left, Exp right)
    | Ident(String name)
```

Algebraic data types provide a very compact form for type declaration. They are syntactically supported in many specification languages (e.g. OBJ3, SableCC, Katja, ...). Further, they are directly supported by a number of functional languages and have a canonical implementation in object-oriented languages:

- A sum type $T = T_1 \,|\, T_2 \,|\, \ldots \,|\, T_n$ is represented as an abstract class or interface T with concrete subclasses T_1 to T_n.
- A product type $C(T_1, T_2, \ldots, T_m)$ is represented by a class with fields of type T_1 to T_m.

The abstract syntax for the expression language (see above) can be represented with the following Java classes:

```
public abstract class Exp {
    // ...
}
class Add extends Exp {
    private Exp left;
    private Expr right;
    // ...
}
public class Mult extends Exp {
    private Exp left;
    private Expr right;
    // ...
}
public class Ident extends Exp {
    private String name;
    // ...
}
```

2 Abstract Syntax Tree Generation

In the exercises we generate Java classes and interfaces from a short description of the AST structure. The description of abstract syntax trees we use consists of three different constructs.

- **Product types** (or “Constructors”) have a name and a list of parameters. For example an if-statement consists of an expression for the condition, and two statements for the then- and else-branches and would be written as:

  ```
  StmtIf(Expr condition, Statement ifTrue, Statement ifFalse)
  ```

- **Lists** have a name and an element type. For example a block-statement is a list of `Statement`s and would be written as:

  ```
  Block * Statement
  ```

- **Sum types** (or “case definitions”) also have a name and a list of different cases. The different cases are separated by a vertical bar symbol “|”. For example a Statement can be a `Block`, a `StmtIf`, or a `StmtAssign`, which can be written as:

  ```
  Statement = Block | StmtIf | StmtAssign
  ```

As a shorthand notation, it is possible to define constructors inside the definition of a sum type, for example:

```
Statement =
    Block
  | StmtIf(Expr condition, Statement ifTrue, Statement ifFalse)
  | StmtAssign(Expr left, Expr right)
```

2.1 Syntax of AST-description

The syntax of abstract syntax definitions is given by the EBNF grammar below:

```
spec ::=
    'package' qID
    ('typeprefix:' ID )?
    'abstract␣syntax:' element*
    'attributes:'     attributeDef*

element ::=
    contructorDef
  | listDef
  | caseDef

contructorDef ::=
    ID '(' (paramDef (',' paramDef )*)? ')'

paramDef ::=
   'ref'? javaType ID

listDef ::=
    ID '*' 'ref'? ID

caseDef ::=
    name=ID '=' choice ('|' choice)*

choice ::=
    ID
  | contructorDef

attributeDef ::=
    ID '.' ID ( '(' (javaType ID (',' javaType ID)*)? ')' )?
    STRVAL?
    'returns' javaType
    'implemented' 'by' qID
    ('circular' qID)?
  | STRVAL? javaType ID '.' ID

javaType ::=
    ('@Nullable')? qID  ('<' javaType  (',' javaType )*  '>')?;

qID ::=
    ID  ('.' ID)*;
```

Whitespace and Java-style comments are ignored.
As an example consider the following AST definition for MiniJava:

```
package minijava.ast

typeprefix: MJ

abstract syntax:

Program(MainClass mainClass, ClassDeclList classDecls)

MainClass(String name, String argsName, Block mainBody)

ClassDecl(String name, Extended extended, VarDeclList fields, MethodDeclList methods)

Extended =
    ExtendsNothing()
  | ExtendsClass(String name)
```

```
VarDecl(Type type, String name)

MethodDecl(Type returnType, String name, VarDeclList formalParameters, Block methodBody)

MemberDecl =
    VarDecl
  | MethodDecl

Type =
    TypeIntArray()
  | TypeInt()
  | TypeBool()
  | TypeClass(String name)

Statement =
    Block
  | StmtIf(Expr condition, Statement ifTrue, Statement ifFalse)
  | StmtWhile(Expr condition, Statement loopBody)
  | StmtReturn(Expr result)
  | StmtPrint(Expr printed)
  | VarDecl
  | StmtExpr(Expr expr)
  | StmtAssign(Expr left, Expr right)

Expr =
    ExprBinary(Expr left, Operator operator, Expr right)
  | ExprUnary(UnaryOperator unaryOperator, Expr expr)
  | ArrayLookup(Expr arrayExpr, Expr arrayIndex)
  | ArrayLength(Expr arrayExpr)
  | FieldAccess(Expr receiver, String fieldName)
  | MethodCall(Expr receiver, String methodName, ExprList arguments)
  | BoolConst(boolean boolValue)
  | VarUse(String varName)
  | Number(int intValue)
  | ExprThis()
  | ExprNull()
  | NewIntArray(Expr arraySize)
  | NewObject(String className)
  | Read()

Operator =
    And()
  | Plus()
  | Minus()
  | Times()
  | Div()
  | Less()
  | Equals()

UnaryOperator =
    UnaryMinus()
  | Negate()


// List types
ClassDeclList * ClassDecl
VarDeclList * VarDecl
MethodDeclList * MethodDecl
Block * Statement
ExprList * Expr
```

```
attributes:

"information␣about␣the␣source␣code"
frontend.SourcePosition Element.sourcePosition
```

2.2 Running the Generator

The main class of the generator is `asg.Main`. The program accepts two arguments: First the filepath to the AST-specification file and second the filepath to the output folder. Usually the tool will be run from a build-tool. For example the following task can be used with Gradle:

```
task genAst {
    description = 'Compile␣java␣cup␣specifications'
    fileTree(dir: 'src/main/java', include:'**/*.ast').each { file ->

        Pattern PACKAGE_PATTERN = Pattern.compile("package\\s+(\\S+)\\s*;");
        String fileContents = file.text

        Matcher matcher = PACKAGE_PATTERN.matcher(fileContents);
        String packageName = "";
        if (matcher.find()) {
            packageName = matcher.group(1);
        }

        String targetDir = "$genDir/" + packageName.replace(".", "/")

        inputs.file(file)
        outputs.dir(targetDir)

        doLast {
            javaexec {
                classpath configurations.compileOnly
                main = "asg.Main"
                args = [file, targetDir]
            }
        }
    }
}
```

2.3 Generated Code

In the specification it is possible to define a “typeprefix”. If a typeprefix is given, it will be added to the name of every generated type.

2.3.1 The Factory class

A factory class is generated, which contains static methods to create instances of every concrete type.
The name of the factory class is equal to the given typeprefix, or equal to the last part of the package name if no typeprefix is given.
For every constructor definition, a static method is generated, which constructs a new instance of the type. The signature of the constructor is equivalent to the constructor definition in the specification.

For every list definition, two static methods are created: One method takes any number of list elements and constructs a list containing these elements. The other method takes an `Iterable` of the elements and uses the elements contained in the `Iterable`.
For example consider the following Java method:

```
int foo(int x, int y) {
    int z;
    z = (x + y);
    return z;
}
```

An abstract syntax tree representing this program can be constructed with the factory methods mentioned above as follows:

```
MJMethodDecl method = MJ.MethodDecl(MJ.TypeInt(), "foo",
        MJ.VarDeclList(MJ.VarDecl(MJ.TypeInt(), "x"), MJ.VarDecl(MJ.TypeInt(), "y")),
        MJ.Block(
                MJ.VarDecl(MJ.TypeInt(), "z"),
                MJ.StmtAssign(MJ.VarUse("z"), MJ.ExprBinary(MJ.VarUse("x"),
                                                  MJ.Plus(), MJ.VarUse("y"))),
                MJ.StmtReturn(MJ.VarUse("z"))
        ));
```

It is also possible to add a static import and thus make the construction shorter and more readable:

```
import static minijava.ast.MJ.*;

...
MJMethodDecl method = MethodDecl(TypeInt(), "foo",
        VarDeclList(VarDecl(TypeInt(), "x"), VarDecl(TypeInt(), "y")),
        Block(
                VarDecl(TypeInt(), "z"),
                StmtAssign(VarUse("z"), ExprBinary(VarUse("x"), Plus(), VarUse("y"))),
                StmtReturn(VarUse("z"))
        ));
```

2.3.2 Sum types

For every sum type an interface type with name `typeprefix + name` is generated. All cases of the sum type are subtypes of the generated interface.
The sum type contains getters and setters for the fields, which are common to all cases.
A sum type provides a `match` method to distinguish the different cases (see Section 2.6).

2.3.3 Constructors

For each constructor definition an interface and a class are generated. The name of the interface is `typeprefix + name`.
The generated class is package private and never accessed from the outside. For creating instances of the class, the factory class (see Section 2.3.1) is used. Programs should always refer to the interface type.
The generated interface contains getters and setters for all fields given in the constructor definition.

2.3.4 Lists

For each list definition an interface and a class are created. The generated list implements the `java.util.List` interface, so all common list methods can be used.
Additionally, the methods `void addFront(T t)` and `List<T> removeAll()` are provided.
The method `removeAll` removes all elements from the list and returns them as a normal Java list. This is useful for moving elements from one part of the tree to another (see Section 2.4).

2.3.5 The Element interface

An interface with name `typeprefix + "Element"` is generated, which is a supertype of all other generated types. In general, the Element interface behaves like a sum type, which is the sum of all other types.
The Element interface provides some general methods to work with ASTs:

```
MJElement getParent();
MJElement copy();
int size();
void clearAttributes();
void clearAttributesLocal();
MJElement get(int i);
MJElement set(int i, MJElement newElement);
void setParent(MJElement parent);
void replaceBy(MJElement other);
boolean structuralEquals(MJElement elem);
<T> T match(Matcher<T> s);
void match(MatcherVoid s);
void accept(Visitor v);
```

Each element maintains a link to the parent element, where it is used. The parent element can be accessed using the `getParent` method. The `setParent` method should usually not be called manually, except for circumventing the invariant checks (see Section 2.4).
The methods `size`, `get`, and `set` provide generic access to the children of an AST node. The method `size` returns the number of children and the `get` and `set` methods can be used to access the specific children.
The `copy` method creates a deep copy of the AST-node. However, it will not copy external types and fields marked as `ref` fields.
The `replaceBy` method replaces the current AST node with another node. This method only changes the reference in the parent element. Other references will still point to the same object.
The `structuralEquals` method checks, if the structure of an element is equal to another element. Two elements are structurally equal, iff they have the same type, if all children are structurally equal, and if all other fields are equal in the sense of Javas `Objects.equals`.
Furthermore, the element interface provides `match` methods (see Section 2.6), an `accept` method for visitors (see Section 2.5), and methods to clear calculated Attributes (see section 2.7).

2.4 Invariants

The generated code tries to maintain some invariants, which are supposed to make programming with the ASTs easier:

1. No null references: The constructors and setters make sure, that no fields are set to `null`. Therefore the getters for the fields will never return `null`.

2. Tree structure and parents: The constructors and setters try to maintain the invariant, that the parent always points to the right element. In particular this means, that each element can have at most one parent and thus be used only once. So it is not possible to construct ASTs which are not trees.

   More formally the following invariant should always hold for any element e:

$$(\mathit{e.getParent()} \mathrel{!=} \mathit{null} \longrightarrow (\exists! i \,:\, 0 \leq i < \mathit{e.getParent().size()} \wedge \mathit{e.getParent().get(i)} == e))$$
$$\wedge\ (\forall i \,:\, 0 \leq i < \mathit{e.size()} \longrightarrow \mathit{e.get(i).getParent()} == e)$$

   If an element which already has a parent is added to a different part of a tree an `Error` with the message “Cannot change parent of element ...” is thrown.

   To circumvent this problem, the easiest method is to use the `copy` method to create a copy of a tree. For lists it is possible to use the `removeAll` method to remove the elements from their current parent, before adding them to another parent. It is also possible to manually call `setParent(null)` on an element before moving, however this might lead to violations of the invariant.

2.5 Visitors

A task that often comes up when working with ASTs involves recursively walking the tree to compute some value for the tree.
For example consider the following definition:

```
Expr =
    ExprBinary(Expr left, Operator operator, Expr right)
  | VarUse(String varName)
  | Number(int intValue)

Operator =
    Plus()
  | Times()
```

If we want to write code to evaluate an `Expr`, we have to distinguish the different cases. One way to do this in Java is using `instanceof` checks and casts, as in the following code:

```
public int evaluate(MJExpr e, Map<String, Integer> varValues) {
    if (e instanceof MJNumber) {
        MJNumber n = (MJNumber) e;
        return n.getIntValue();
    } else if (e instanceof MJVarUse) {
        MJVarUse v = (MJVarUse) e;
        return varValues.get(v);
    } else if (e instanceof  MJExprBinary){
        MJExprBinary b = (MJExprBinary) e;
        int l = evaluate(b.getLeft(), varValues);
        int r = evaluate(b.getRight(), varValues);
        return evaluateOperator(b.getOperator(), l, r);
    }
    throw new Error("unhandled␣case:␣" + e);
}
```

A problem with the code above is, that it is easy to miss a case or to get one of the casts wrong. In particular there is no compiler error, when a case is added to the abstract syntax specification later.
The generated code implements the visitor pattern to support these kind of tasks. A general introduction to the visitor pattern can be found in:

- Oliveira, Bruno CdS, Meng Wang, and Jeremy Gibbons. The visitor pattern as a reusable, generic, type-safe component.
- E. Gamma and R. Helm. Design Patterns, Elements of Reusable Object-Oriented Software.
- http://en.wikipedia.org/wiki/Visitor_pattern

To implement a Visitor, create a class which extends `typeprefix + "Element.DefaultVisitor"`. This class comes with `visit` methods for all concrete cases of the AST that recursively invoke the accept method on all children. You can just override the `visit` methods, which are relevant for the task at hand.
When overriding a `visit` method, you can visit the children by calling the `visit` method of the super-class. For more fine-grained control you can instead invoke the `accept` methods of child nodes individually.
For example the evaluation from above could be implemented using a Visitor:

```
class Evaluator extends MJElement.DefaultVisitor {

  private Map<String, Integer> varValues;
  private int value;

  public Evaluator(Map<String, Integer> varValues) {
      this.varValues = varValues;
  }

  @Override
  public void visit(MJVarUse varUse) {
      value = varValues.get(varUse.getVarName());
  }

  @Override
  public void visit(MJExprBinary exprBinary) {
      exprBinary.getLeft().accept(this);
      int left = value;
      exprBinary.getRight().accept(this);
      int right = value;
      value = evaluateOperator(exprBinary.getOperator(), left, right);
  }
  @Override
  public void visit(MJNumber number) {
      value = number.getIntValue();
  }
}


public int evaluate(MJExpr e, Map<String, Integer> varValues) {
  e.accept(new Evaluator(varValues));
  return value;
}
```

With this principle it is very easy to implement a method, which calculates the set of used variables in an expression:

```
class VariableFinder extends MJElement.DefaultVisitor {
    Set<String> variables = new TreeSet<>();

    @Override
    public void visit(MJVarUse varUse) {
        variables.add(varUse.getVarName());
    }
}

public Set<String> usedVariables(MJExpr e) {
    VariableFinder variableFinder = new VariableFinder();
    e.accept(variableFinder);
    return variableFinder.variables;
}
```

Since no fine-grained control over the order of traversing the children is required, we need only override the `visit` method for variable usages.
It is also possible to implement the `Visitor` using an anonymous class:

```
public Set<String> usedVariables(MJExpr e) {
    Set<String> variables = new TreeSet<>();
    e.accept(new MJElement.DefaultVisitor() {
        @Override
        public void visit(MJVarUse varUse) {
            variables.add(varUse.getVarName());
        }
    });
    return variables;
}
```

2.6 Matchers

Matchers are special cases of visitors. Instead of the complete AST, matchers only distinguish between the different cases of a sum type.
Comming back to our evaluation example:

```
public int evaluateOperator(MJOperator operator, int left, int right) {
  if (operator instanceof MJPlus) {
      return left + right;
  } else if (operator instanceof  MJTimes) {
      return left * right;
  }
}
```

As an alternative, it is possible to use the `match` methods provided by the generated interfaces. There are two overloads for the `match` method: One returns `void` and takes a `MatcherVoid` and the other expects a `Matcher<T>` and returns `T`.
The interfaces `MatcherVoid` and `Matcher<T>` are generated as inner classes for every sum type and for the general `Element` type.
To handle the different cases of a sum type `X`, we can simply implement the interface `X.Matcher<T>` or `X.MatcherVoid` and then pass an instance of the matcher to the `match` method. The interface contains a method for every possible case for the sum type and the `match` method will call the correct method based on the actual type of the element.
For example the `evaluateOperator` method from above could be implemented using a Matcher:

```
private int evaluateOperator(MJOperator operator, int left, int right) {
    return operator.match(new MJOperator.Matcher<Integer>() {

        @Override
        public Integer case_Plus(MJPlus plus) {
            return left + right;
        }

        @Override
        public Integer case_Times(MJTimes times) {
            return left * right;
        }
    });
}
```

2.7 Additional attributes, methods, and fields

It is possible to extend the generated classes with additional attributes, methods, and fields, which are declared in the specification after “attributes:”.

2.7.1 Fields

An additional field is declared by the syntax:

```
STRVAL? javaType ID '.' ID
```

The definition can start with a String which is used for documentation in the generated code. Afterwards the field type is declared followed by the name of the type where the field should be added. The name of the field is then given after a dot. For example the following definition would add a field of type `int` named `evaluationResult` to the type `Expr`:

```
int Expr.evaluationResult
```

For each declared field a getter and a setter method are generated.

2.7.2 Methods

An additional method is declared using the following syntax:

```
ID '.' ID '(' (javaType ID (',' javaType ID)*)? ')'
    STRVAL?
    'returns' javaType
    'implemented' 'by' qID
```

The first identifier is the type, for which the method should be added. The second identifier is the name of the method, which is followed by a parameter list as in Java. After the parameter list there can be a string value for documentation. Then the return type is given after "`returns`".
The actual implementation of the method must be in a static method. The fully qualified name of that method is given at the end after "`implemented by`".
In the generated code, the method is created and the call is forwarded to the given static method. Overloading can be used to distinguish different cases of sum types.

2.7.3 Attributes

An attribute is defined like a method, but without a parameter list:

```
ID '.' ID
    STRVAL?
    'returns' javaType
    'implemented' 'by' qID
```

In contrast to a method, an attribute is only evaluated once on the first call. All subsequent calls use a cached version of the result. The methods `clearAttributes` and `clearAttributesLocal` can be used to clear the cache.
If the value of an attribute is requested while it is being calculated, a `CyclicDependencyError` will be thrown.

2.8 Ref fields

For intermediate languages it sometimes make sense to have cross references to other elements in the tree. For example a variable use in an intermediate language could directly defer to the declaration of the variable instead of just storing the name of the variable, so that it is not necessary to do a second name analysis.
As those references don't obey to the tree structure (there can be many uses of the same variable), they have to be distinguished from other fields. This can be done by writing `ref` in front of the type of a field in a constructor or list definition. For example:

```
ExprVarAccess(ref Local var)
TypeRefList * ref Type
```

For fields marked with `ref` the parent invariant is not enforced and the parent is not changed for referenced elements.

3 Use with Java Cup

The recommended way to use the generated AST classes with a parser generator like Java CUP is to add a static import like:

```
import minijava.ast.*;
import static minijava.ast.MJ.*;
```

With the static import in place, the AST can be constructed in the parser actions, by simply calling the factory methods (See 2.3.1).

For example an AST for expressions can be build as follows:

```
expr ::=
        expr:l AND expr:r
            {: RESULT = ExprBinary(l, And(), r); :}
      | expr:l PLUS expr:r
            {: RESULT = ExprBinary(l, Plus(), r); :}
      ...
```

When building lists, the usual methods from `java.util.List` can be used, as shown in the following example to build a list of class declarations:

```
classDeclList ::=
        classDecl:c classDeclList:l
            {: RESULT = l; l.add(0, c); :}
      | /* empty */
            {: RESULT = ClassDeclList(); :}
      ;
```

Remember, that any valid Java code can be written in the parser actions. In particular, you can also call your own helper-methods, if more complicated transformations are required.