# LPG Automatic AST Generation

Gerry Fisher & Philippe Charles

## Requirement for Automatic AST Generation

Interpretation of parsed input is usually done in "semantic actions" associated with the grammar rules. Often these actions simply produce an AST (or Abstract Syntax Tree) which in effect is simply a condensed version of a syntax derivation tree. The AST is then traversed (or "walked") to produce the objects that interpret the syntax. Sometimes no AST is produced: instead the actions directly build the required objects. For example, the actions for a SQL parser might directly produce SQL-Model objects. Generating objects directly is generally more efficient since it eliminates the intermediate step of building an AST. If bottom-up parsing is used (and it is with LPG), the actions can only synthesize attributes (not inherit attributes), so the creation of objects is more difficult than it is with AST traversal. A tree walker can easily and transparently deal with inherited and synthesized attributes.

If the AST can be automatically generated, the parsing phase needs no user written actions and thereby the parser becomes a separate component available to various applications that process the parsed language. For example, several tools that process SQL in various ways can share a binary form of the SQL parser. With a generated AST it is not necessary to start with a copy of the grammar rules and write actions for them and then generate a parser. Instead, one takes the AST generation classes and parser as a separate component and writes an AST walker.

## The LPG Solution

Our solution is to construct AST classes and interfaces and base the generated rule actions on these constructs according to the following principles:

- There is an abstract class, Ast, containing location information, which contains all AST classes and interfaces
- Every non-terminal symbol defines an interface
- Every terminal symbol defines a subclass of Ast that identifies the token kind
- Every rule (except single productions with non-terminal right-hand side) extends Ast class and implements the interface defined by its left-hand side symbol; the class has an attribute for each right-hand side symbol whose type is determined as follows: for a non-terminal symbol the type of the attribute is its interface class, while for a terminal symbol the type is its Ast subclass
- The interface defined by a non-terminal symbol (e.g., B) that appears as the right-hand side of a single production with a given left-hand side (e.g., A) extends the interface defined by the left-hand side (that is, interface B extends interface A by virtue of the rule 'A ::= B')

## *Example*

Here is an example to show how this works. Consider the following grammar:
Here are the rules:

```
E ::= E + T
E ::= E - T
E ::= T
T ::= T * P
T ::= T / P
P ::= NUM
```

Here are the interfaces and classes:

```
public class Ast
{
    int leftToken, rightToken;
    Ast next;
    TokenStream tokStream;

    Ast(TokenStream tokStream, int leftToken, int rightToken)
    {
        this.tokStream = tokStream;
        this.leftToken = leftToken;
        this.rightToken = rightToken;
    }

    public int getLeftToken() { return leftToken; }

    public int getRightToken() { return rightToken; }

    interface IE { }
    interface IT extends IE { }
    interface IP extends IT { }

    public class TPLUS extends Ast
    {
        public TPLUS(TokenStream tokStream, int leftToken,
                                            int rightToken)
        {
            super(tokStream, leftToken, rightToken);
        }
        public int getPLUS() { return leftToken; }
    }

    public class TMINUS extends Ast
    {
        public TMINUS(TokenStream tokStream, int leftToken,
                                             int rightToken)
        {
            super(tokStream, leftToken, rightToken);
        }
        public int getMINUS() { return leftToken; }
    }

    public class TSTAR extends Ast
```

```
{
    public TSTAR(TokenStream tokStream, int leftToken,
                                        int rightToken)
    {
        super(tokStream, leftToken, rightToken);
    }
    public int getSTAR() { return leftToken; }
}

public class TSLASH extends Ast
{
    public TSLASH(TokenStream tokStream, int leftToken,
                                         int rightToken)
    {
        super(tokStream, leftToken, rightToken);
    }
    public int getSLASH() { return leftToken; }
}

public class TNUM extends Ast
{
    public TNUM(TokenStream tokStream, int leftToken,
                                       int rightToken)
    {
        super(tokStream, leftToken, rightToken);
    }
    public int getNUM() { return leftToken; }
}

class E1 extends Ast implements IE    // E ::= E + T
{
    private IE _E;
    private TPLUS _PLUS;
    private IT _T;

    public IE getE() { return _E; }
    public TPLUS getPLUS() { return _PLUS; }
    public IT getT() { return _T; }

    public E1 (TokenStream tokStream,
               int leftToken, int rightToken,
               IE _E, TPLUS _PLUS, IT _T)
    {
        super(tokStream, leftToken, rightToken);
        this._E = _E;
        this._PLUS = _PLUS;
        this._T = _T;
    }

}

class E2 extends Ast implements IE    // E ::= E - T
{
    private IE _E;
    private TMINUS _MINUS;
    private IT _T;
    public IE getE() { return _E; }
    public TMINUS getMINUS() { return _MINUS; }
    public IT getT() { return _T; }

    public E2 (TokenStream tokStream,
               int leftToken, int rightToken,
               IE _E, TMINUS _MINUS, IT _T)
    {
        super(tokStream, leftToken, rightToken);
        this._E = _E;
        this._MINUS = _MINUS;
        this._T = _T;
    }

}

class T1 extends Ast implements IT    // T ::= T * P
{
    private IT _T;
    private TSTAR _STAR;
    private IP _P;

    public IT getT() { return _T; }
    public TSTAR getSTAR() { return _STAR; }
    public IT getP() { return _P; }

    public T1 (TokenStream tokStream,
               int leftToken, int rightToken,
               IT _T, TSTAR _STAR, IP _P)
    {
        super(tokStream, leftToken, rightToken);
        this._T = _T;
        this._STAR = _STAR;
        this._T = _T;
    }

}

class T2 extends Ast implements IT    // T ::= T / P
{
    private IT _T;
    private TSLASH _SLASH;
    private IP _P;

    public IT getT() { return _T; }
    public TSLASH getSTAR() { return _SLASH; }
    public IT getP() { return _P; }

    public T2 (TokenStream tokStream,
               int leftToken, int rightToken,
               IT _T, TSLASH _SLASH, IP _P)
    {
        super(tokStream, leftToken, rightToken);
        this._T = _T;
        this._SLASH = _SLASH;
        this._T = _T;
    }
    }

    class P1 extends Ast implements IP    // P ::= NUM
    {
        private TNUM _NUM;

        public TNUM getNUM() { return _NUM; }

        public P1 (TokenStream tokStream,
                   int leftToken, int rightToken,
                   TNUM _NUM)
        {
            super(tokStream, leftToken, rightToken);
            this._NUM = _NUM;
        }

    }

}
```

Note that there are no classes for the single productions `E ::= T` and `T ::= P`. There will be no actions generated for these rules. Note also that the class name is simply the left-hand side name suffixed with an occurrence number. Note, as well, that the attribute getters for each rule class are not shown.

Let us see what will be generated automatically for each rule.

For the rule `E1 (E ::= E + T)` we generate:

```
{ $setResult(new E1 (getPrsStream(),$getLeftSpan(),$getRightSpan(),
                     (IE)$getSym(1),
                     new TPLUS($getToken(2)),
                     (IT)$getSym(3)); }
```

Generation for rule `E2 (E ::= E - T)` is similar, the only difference is the class of the generated object (so in fact we could assign both of these rules to the same class and distinguish the operator by the token kind).

For the rule `T1 (T ::= T * P)` we generate:

```
{ $setResult(new T1 (getPrsStream(),$getLeftSpan(),$getRightSpan(),
                     (IT)$getSym(1),
                     new TSTAR($getToken(2)),
                     (IP)$getSym(3)); }
```

Generation for rule `T2 (T ::= T - P)` is similar.

For the rule `P1 (P ::= NUM)` we generate:

```
{ $setResult(new P1 (getPrsStream(),$getLeftSpan(),$getRightSpan(),
                     new TNUM($getToken(1)) ); }
```

## *Semantic Analysis of the AST*

The AST class 'Ast' will actually be the "action" class for the parser and will (most probably) extend PrsStream. It will then contain a "parse()" method which builds the classes and interfaces, performs the rule actions and returns the root Ast. In addition, rule actions may specify additional methods, attributes and initializations for the rule classes.

There are many ways to analyze the generated AST. One common way of processing is to synthesize or inherit attributes on which various operations are performed. For example, we may add a `value()` method to each AST rule and token class of our example. What the `value()` method returns depends on the class of the object to which it is applied – if, for example, the object is an `E1`, the method will return `_E.value() + _T.value()`, where `E` and `T` are the attributes of `E1` defined above; if the object is a `P1`, the method returns `_NUM.value()`, which converts the token text to an integer value. If we start with the root Ast node (returned from parsing the input), invoking its value method will recursively walk the tree and evaluate the expression.