



CA Gen Code Completion Editor

Mustafa Arikan, ARIKAN Productivity Group
mustafa.arikan@arikan.at

Session Track 3
12th October 15:30 16:30

Biography

Mustafa Arikan studied industrial engineering, mathematics and computer science in Istanbul and in Vienna and finished his education in 1986. He has meanwhile 29 years industrial experience in IT and operations research. He worked for vendors like IBM and as technology partner of Computer Associates for various large scale companies and won many IT awards throughout his career so far. His companies serve in Austria and Turkey and in cooperation with partners in over 10 countries mainly in software modernization.



Agenda

Building a Code Completion editor for CA Gen as an alternative to 'Click2Code'.

Code Completion Editor.

Name Space.

Formal Grammar.

Parsing.

Backus Naur Form.

CA Gen Grammar.

CA Gen Life Cycle

Demo.

Autocomplete

Autocomplete is a feature provided by many [source code editors](#). Autocomplete involves the program predicting a word or phrase that the user wants to type in without the user actually typing it in completely. This feature is effective when it is easy to predict the word being typed based on those already typed, such as when there are a limited number of possible or commonly used words (as is the case with [e-mail](#) programs, [web browsers](#), or [command line interpreters](#)), or when editing text written in a highly-structured, easy-to-predict language (as in [source code editors](#)).

Some parts of this presentation are taken from WIKIPEDIA.

www.wikipedia.org

Autocomplete

Autocomplete speeds up human-computer interactions in environments to which it is well suited. Autocomplete features are often enabled by default, and disabling or defeating them can sometimes be difficult for users to accomplish.

Code Completion

Text Autocomplete of source code is also known as code completion. In a [source code editor](#) autocomplete is greatly simplified by the regular structure of the [programming languages](#). There are usually only a limited number of words meaningful in the current context or [namespace](#), such as names of variables and functions.

Name Space

A namespace is an abstract container or environment created to hold a logical grouping of unique identifiers or symbols (i.e., names). An identifier defined in a namespace is associated with that namespace. The same identifier can be independently defined in multiple namespaces. That is, the meaning associated with an identifier defined in one namespace may or may not have the same meaning as the same identifier defined in another namespace. Languages that support namespaces specify the rules that determine to which namespace an identifier (i.e., not its definition) belongs.

Formal Grammar

A formal language is a set of words, i.e. finite strings of letters, symbols, or tokens. The set from which these letters are taken is called the alphabet over which the language is defined. A formal language is often defined by means of a formal grammar (also called its formation rules); accordingly, words that belong to a formal language are sometimes called *well-formed words* (or well-formed formulas).

A formal grammar (sometimes simply called a grammar) is a set of rules for forming strings in a formal language. These rules that make up the grammar describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings—only their location and the ways that they can be manipulated

Parsing

The process of recognizing a string by constructing a combination of applications of rules that generate it is known as [parsing](#). Most languages have the meanings of their utterances structured according to their syntax—a practice known as [compositional semantics](#). As a result, the first step to describing the meaning of an utterance in language is to break it down part by part and look at its analyzed form (known as its [parse tree](#) in computer science, and as its [deep structure](#) in [generative grammar](#)).

Backus Naur Form

In [computer science](#), Backus–Naur Form (BNF) is a [metasyntax](#) used to express [context-free grammars](#): that is, a formal way to describe [formal languages](#). [John Backus](#) and [Peter Naur](#) developed a context free grammar to define the syntax of a programming language by using two sets of rules: i.e., lexical rules and syntactic rules

CA Gen Grammar (M.Stankiewicz)

BNF for AllFusion Gen AD Language

NON-TERMINALS

Draft Version: 0.5 , January 29, 2006

```
CompilationUnit ::= ActionDeclaration <EOF>
ActionDeclaration ::= <ACTION> Name ( QualifierList )? ActionBlockBody
ActionBlockBody ::= "{" ( ImportsDeclaration )? ( LocalsDeclaration )? ( ExportsDeclaration )? ( EntityActionsDeclaration )? MainDeclaration ( EventDeclaration )? ( EventDeclaration )* }"
EventDeclaration ::= <EVENT> Name ( QualifierList )? "{" GroupOfStatements }"
ImportsDeclaration ::= <IMPORTS> "{" ( ViewDeclaration )* }"
ExportsDeclaration ::= <EXPORTS> "{" ( ViewDeclaration )* }"
LocalsDeclaration ::= <LOCALS> "{" ( ViewDeclaration )* }"
EntityActionsDeclaration ::= <ENTITY> <ACTIONS> "{" ( EntityViewDeclaration )* }"
ViewDeclaration ::= ( SimpleViewDeclaration | GroupViewDeclaration )
EntityViewDeclaration ::= <ENTITY> <VIEW> Name ( Name )? AttributeList
SimpleViewDeclaration ::= <VIEW> ( Name )? ( ( <WORK> | <ENTITY> ) ) Name QualifierList AttributeList
GroupViewDeclaration ::= <GROUP> Name QualifierList "{" ( ( SimpleViewDeclaration | GroupViewDeclaration ) )* }"
QualifierList ::= "[" Qualifier ( "," Qualifier )* "]"
Qualifier ::= QualifierKey "=" QualifierValue
QualifierKey ::= <IDENTIFIER>
QualifierValue ::= ( ( <INTEGER_LITERAL> | <STRING_LITERAL> | Name ) )
```

CA Gen ActionBlock Grammar

The screenshot displays the CA Gen IDE interface for a parser grammar file named `Gen_AB.g`. The main editor window shows the following grammar rules:

```
/* ===== Parser Grammar ===== */  
  
action_block :  
    {actionHandler.startRule("action_block");}  
    (common_comment  
    |stmt_statement  
    )+  
    EOF  
    {actionHandler.endRule("action_block");}  
    ;  
  
univName :  
    {actionHandler.startRule("univName");}  
    (IDENT {actionHandler.handleToken(TokenType.IDENT, $IDENT.text);}  
    |INTID {actionHandler.handleToken(TokenType.INTID, $INTID.text);})  
    {actionHandler.endRule("univName");}  
    ;  
  
common_name :  
    {actionHandler.startRule("common_name");}  
    univName (univName)*  
    {actionHandler.endRule("common_name");}  
    ;
```

The `univName` rule is highlighted in yellow. Below the editor, a parse tree diagram for the `univName` rule is shown, illustrating its derivation from the non-terminal `univName` into the terminals `IDENT` and `INTID`.

```
graph TD  
    univName --> IDENT  
    univName --> INTID
```

CA Gen Action Block Grammar

The screenshot displays the CA Gen IDE interface. The top window shows the grammar file `Gen_AB.g` with the following content:

```
stmt_statement : {actionHandler.startRule("stmt_statement");} {  
  stmt_create  
  | stmt_exitState  
  | stmt_move  
  | stmt_set  
  } {actionHandler.endRule("stmt_statement");}  
;  
  
stmt_create : {actionHandler.startRule("stmt_create");} {  
  'CREATE' univName  
  stmt_set*  
  stmt_when*  
  'END-CREATE'  
  } {actionHandler.endRule("stmt_create");}  
;
```

The bottom window shows the parse tree for the `action_block` rule. The tree structure is as follows:

- `action_block` (root)
 - `CREATE`
 - `univName` (child of `CREATE`)
 - `person`
 - `stmt_set` (child of `action_block`)
 - `SET` (child of `stmt_set`)
 - `common_name` (child of `stmt_set`)
 - `univName` (child of `common_name`)
 - `phone`
 - `TO` (child of `stmt_set`)
 - `common_name` (child of `stmt_set`)
 - `univName` (child of `common_name`)
 - `imp`
 - `univName` (child of `common_name`)
 - `person`
 - `univName` (child of `common_name`)
 - `phone`
 - `SET` (child of `stmt_set`)

The bottom-left window shows the corresponding SQL code for the `action_block` rule:

```
CREATE person  
SET phone TO imp person phone  
SET email TO imp person email  
SET jobdescription TO imp person jobdescription  
SET department TO imp person department  
SET surname TO imp person surname  
SET name TO imp person name  
SET number TO imp person number  
WHEN successful  
MOVE person TO exp person  
WHEN already exists  
EXIT STATE IS person_ae  
WHEN permitted value violation  
EXIT STATE IS person_pv  
END-CREATE
```

CA Gen Data Grammar

The screenshot displays an IDE window titled "Gen_View.g" containing ANTLR grammar code. The code defines a grammar for parsing a view definition. The grammar includes options for the number of look-ahead tokens (k=4) and the output format (AST). It defines a header with package and import statements, a lexer header, a rulecatch block, and a members block with a dummy action handler and a mismatch method.

```
grammar Gen_View;

options {
    k = 4;
    output=AST;
}

@header {
package at.arikan.cobol antlr;
import at.arikan.modelcvs.legacy antlr.parse.DummyActionHandler;
import at.arikan.modelcvs.legacy antlr.parse.IActionHandler;
import at.arikan.modelcvs.legacy antlr.parse.TokenType;
import java.util.ArrayList;
import java.util.LinkedHashMap;
}

@lexer:header {
package at.arikan.cobol antlr;
}

@rulecatch {
}

@members {
private IActionHandler actionHandler = new DummyActionHandler();

long lineNumber = 0;
protected void mismatch(IntStream input, int ttype, BitSet follow)
    throws RecognitionException {
}
```

The parse tree for the expression "Entity View imp person (mandatory,transient,import only) phone (mandatory) email (mandatory) jobdescription (mandatory) department (mandatory) surname (mandatory) name (mandatory) number (mandatory)" is shown below the code. The root node is "view", which branches into "Entity", "View", "univName", "univName", "(", "mandatory", ",", "transient", ",", "import", "only", and ")". The "univName" nodes further branch into "imp" and "person".

IMPORTS:
Entity View imp person
(mandatory,transient,import only)
phone (mandatory)
email (mandatory)
jobdescription (mandatory)
department (mandatory)
surname (mandatory)
name (mandatory)
number (mandatory)

EXPORTS:
Entity View exp person

AntLr->JAVA, AntLr->Metamodel

The rules from AntLr are exported and build the Parser Classes

The rules from AntLr are exported to build the Metamodel

Parser fills the metamodel and builds the instance

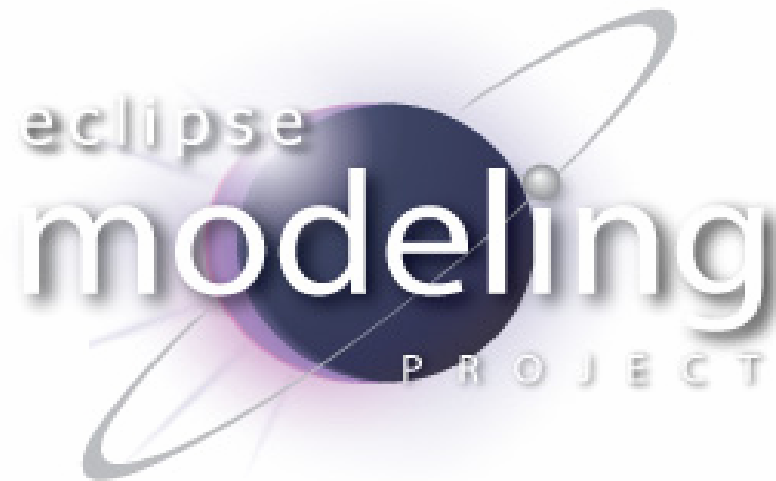
At the same time the Adaptor Classes are generated and build the persistence programs

Eclipse Modeling Framework

Eclipse Modeling Project

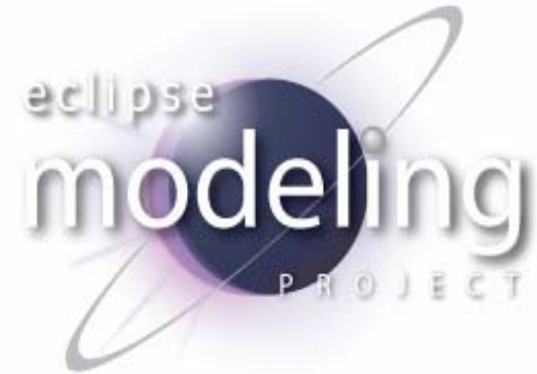
The Eclipse Modeling Project focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations.

The Modeling Project charter is posted [here](#) and inherits from the [Eclipse Standard Top-Level Charter v1.0](#).



Model Life Cycle

for **ca**
CA Gen



```
CREATE_PERSON
IMPORTS:
  Entity View imp person (mandatory,transient,import only)
    phone (mandatory)
    email (mandatory)
    jobdescription (mandatory)
    department (mandatory)
    surname (mandatory)
    name (mandatory)
    number (mandatory)
EXPORTS:
  Entity View exp person (transient,export only)
    phone
    email
    jobdescription
    department
    surname
    name
    number
LOCALS:
ENTITY ACTIONS:
  Entity View person
    phone
    email
    jobdescription
    department
    surname
    name
    number

CREATE person
SET phone TO imp person phone
SET email TO imp person email
SET jobdescription TO imp person jobdescription
SET department TO imp person department
SET surname TO imp person surname
```

```
CREATE person
  SET phone TO imp person phone
  SET email TO imp person email
  SET jobdescription TO imp person jobdescription
  SET department TO imp person department
  SET surname TO imp person surname
  SET name TO imp person name
  SET number TO imp person number
WHEN successful
  MOVE person TO exp person
WHEN already exists
  EXIT STATE IS person_ae
WHEN permitted value violation
  EXIT STATE IS person_pv
END-CREATE
```

EDGE / EMEA 2009

October 11-13, 2009
Amsterdam

Sequence of Activities

CA Gen Model is exported (converted) into an instance of its MM.

CA Gen Model will be transferred to EMF.

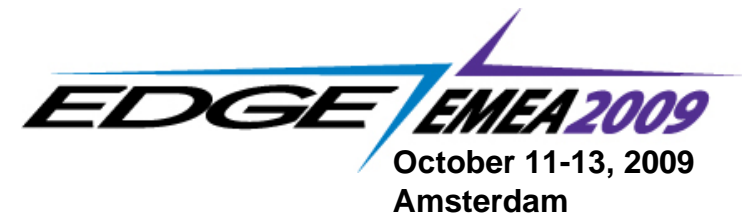
CA Gen Model Instance is shown as an CA Gen Code.

The Adress Space is loaded into editor.

The changes are traced and completed by the program.

The new instance will be converted to CA Gen Code and persisted in the CA Gen Ency.

Demo



Q&A