

Project Presentation

Recursive Descent Generator in C++ Language

Petr Máj

What is AGPG

- AGPG (Another Grammar Parser Generator) is a tool for automatic generation of context-free attributed grammar parsers in C++ language
- AGPG is highly extensible and theoretically is not limited by the C++ language and CFG grammars
- it also features the AGPG framework, the set of classes usable for all developers of language parsers & other tools

Contents

- Introduction to AGPG
- AGPG Framework
 - Basic principles
 - Extensions
 - Introduction to phases
 - Building the inner form tree
 - Checking & Modifying the inner form tree
 - Generating the output code
- AGPG Application
 - GDL2, C++ capabilities, C++ output
- Conclusion & Future development

Introduction to AGPG

- AGPG consists of two major parts:
 - AGPG Framework
 - Basic framework for general purpose language tools
 - Specific framework for AGPG base systems
 - Standardizes the AGPG applications
 - AGPG Application
 - Demonstration of AGPG framework base system
 - Generates the C++ source code of attributed LL(1) CFG grammar from GDL2 language

AGPG Framework I

- Defines the *Exceptions*, special way of handling errors in AGPG applications
 - Several error importance types (warnings, fatal errors, etc.)
 - Unified reporting
 - Error recovery (e. g. reports of multiple errors in only one pass)
- Defines basic wrappers for the resources (source files, etc)
- Defines the basic structure of the inner form tree

AGPG Framework II

- *AGPG Application*, a base class for all applications following the AGPG's ideas
 - Error management
 - Extension management
 - Phase management
 - User interaction basics
- Defines the basic *Extensions* and their classes
- Implements the basic steps of the translation process, e. g. the *phases*

Extensions I

- The functionality of AGPG application is almost none
- It is extended by the extensions, a plug-in classes performing the special tasks
- These tasks corresponds to the phases of the translation
- Extensions may be 3rd party products
- Statically or dynamically loaded
 - Dynamic loading is not implemented in the framework because it is dependent on other libraries (QT, etc.)

Extensions II

- Extensions are managed by the Application
- They are checked for their dependencies and the list of all loaded extensions is made
 - Loaded extensions are executed during the translation process and they thus determine the output
-

Translation Phases

- The translation process in AGPG is divided into six phases:
 - Phase 0 – Initialization
 - Phase 1 – Building of the inner form tree
 - Phase 2 – Checking the inner form tree
 - Phase 3 – Modifying the inner form tree
 - Phase 4 – Generating the output code
 - Phase 5 – Finalization & cleanup
- Phases 1-4 are called the translation phases
- Phases 0 and 5 mostly only interacts with the user and the environment

Initialization & Finalization

- Initialization
 - Creating the Application singleton
 - Registering all available extensions
 - Parsing the command line arguments
 - Loading the selected or needed extensions
 - Configuring the environment
- Finalization
 - Destroying all objects
 - Printing errors
 - Cleaning up any temporary resources

Building the inner tree

- Parsing all input files with the appropriate extensions
- Producing the basic inner form representation of all source code elements both in GDL and implementation language
- Reports usually only syntax and fatal (IO) errors

Checking & Modifying

- Checking the various aspects of the inner form tree
 - Typically, the semantic errors are reported in this phase
 - The checks should not modify the inner form
- Modifications of the inner form
 - Transformations of various inner form structures into different types
 - Modifies the inner form to standardized format

Generating output code

- Output code for the selected elements is generated using the implementation language
- Generation is highly dependent on the used implementation language
- Together with the SDK (including basic classes such as GDL String), the implementation code headers and the generated code the working parser can be compiled

GDL version 2 I.

- Grammar Definition Language is used for the definition of context-free attributed grammars
- Main features of GDL:
 - Strict separation of GDL code and the implementation language code
 - Easy and simple format
 - Similar to the formal definition of grammars
 - Platform and implementation language independent
- GDL syntax is influenced by many modern languages including Pascal and C++

GDL II

- GDL supports the following structures:
 - Namespaces
 - Comments & persistent comments
 - Grammars
 - Nonterminals
 - Terminals
 - Rules
- All other elements must be declared in the implementation language headers

C++ Parsing

- Type definitions
- Functions
- Classes
 - Including inheritance
- Not supported features of C++:
 - Templates
 - Specific types and multiple pointers
 - Other than private members
 - Variables and fields

C++ Code Generation

- Grammars are represented as classes
 - Grammar inheritance is the class inheritance
- Nonterminals are inner classes
- Terminal symbols all share the same class
 - Only the terminal type is different
- E-BNF rules are expanded as cycles and switches
- Rules are represented as methods
- Expansion is decided in virtual methods

Conclusion

- Comparison with other products (Bison, ANTLR):
 - AGPG lacks some of their advanced features
 - Is not as mature and experienced system
 - However when comparing only the features implemented in all these applications, AGPG offers easier coding and more transparent approach
- Future development
 - Addition of more error control, in rule expressions and better C++ support
 - Maybe switching the AGPG language from C++ to more scalable scripting language such as Python

Thank You