UUAGC v.0.9.3
Release notes and Implementation notes

Jeroen Fokker

December 4th, 2006

This report describes the differences of UUAGC release candidate 0.9.3, as compared to version 0.9.1 of December 2005. The new version is currently available in the brach named *candidate* in the SVN archive.

In short, this version has:

- **Better input**
  - It was already possible to later add new alternatives to exisiting datatypes. But now you can also later add new children to existing alternatives.
  - You can also add children generically to multiple datatypes.
  - It was already possible to declare an attribute for multiple datatypes. But now you can also generically *define* these attributes in a single SEM-definition.
- **Better output**
  - Optionally, the computation of attributes can be scheduled over multiple visits.
  - The output is now ghc-6.6 compliant, because the lhs of a definition is no longer put in parentheses.
  - There are generally less superfluous parentheses in the output, which makes the generated code easier to read.
  - Comments can be generated not only listing the attributes, but also the children and local variables.
  - Error messages have a better layout.
- **Better source**
  - The five stages of processing that together form the main dataflow are made more uniform, treating their input (tree and options) and output (next tree, errors, and additional strings) in a more consistent way. This makes the source easier to understand and modify in the future.
  - In imported Haskell libraries, all needed functions are explicitly enumerated. This makes it more transparent why a module is actually needed.
  - Some code (especially the gathering of all information, and the generation of default rules) has been rewritten in order to make it easier to understand.

In section 1 we introduce the new features of the input language. In section 2 we describe the overall architecture of the program. In section 3 we list the modifications that were made for the generic attribute definitions. In section 4 we list the modifications that were made for the sequential visits. In section 5 we describe the build process as steered by *make*.

1 New features

1.1 Add children to existing datatypes

When defining a datatype in Haskell, you have to specify all alternatives in one declaration. In contrast, in UUAGC it is possible to add new alternatives to existing datatypes. This was already possible in earlier versions. For example:

```
-- initial definition
DATA Foo
   | One  a1 : {Int}
          a2 : {Int}
   | Two  b1 : {Int}
--other definitions
DATA Bar
   | First d : {Int}
-- add-on to first definition
DATA Foo
   | Three c1 : {Int}
```

In the new version, it is also possible to add new children to existing alternatives. For example, to add a second child to alternative `Two` of datatype `Foo`, you can extend the example above by:

```
-- add new child to existing alternative
data Foo
   | Two  b2 : {Int}
```

In earlier versions, this would result in a 'duplicate alternative' error, but now it is allowed. Of course, the name of the added child `b2` should differ from the name of the existing child `b1`.

1.2 Generically add children to multiple datatypes

If two datatypes share a common part, it is now possible to write it only once, and later extend it in different ways. For example:

```
-- common part
DATA Foo Bar
   | One a : {Int}
   | Two b : {Int}
-- two extensions
DATA Foo
   | Three c : {Int}
DATA Bar
   | Four  d : {Char}
```

In earlier versions, it was a syntax error to enumerate more than one type name in a DATA header. Even an attempt to capture both names in a set would fail in earlier versions:

```
SET Common = Foo Bar
DATA Common
   | One a : {Int}
   | Two b : {Int}
```

It would throw a 'duplicate synonym' error in earlier versions, but it is possible in the new version.

1.3 Generically define attribute values

In earlier versions, it was already possible to declare an attribute for multiple datatypes:

```
ATTR Foo Bar [ | | s : {Int} ]
```

But it was not possible to *define* the *value* of the attribute generically. In the new version, it is possible to have a single definition for two datatypes, provided that they have a common alternative. Of course, this situation will occur more often if the construction in the previous subsection is used frequently. We are now able to define in a single definition:

```
SEM Foo Bar
  | One  lhs.s = 3
```

In earlier versions, it was a syntax error to enumerate more than one type name in a SEM definition, and an apporach using SETs would also fail.

1.4 Using wildcards

In earlier versions, it was already possible to use a wildchard name `*`, and a name set difference operator `-`, to generically define an attribute for more than one alternative. For example:

```
SEM Foo
  | * - One  lhs.s = 4
```

would define the attribute for alternatives `Two` and `Three`.

It should be noted that in the new version, this notation has an even more generic meaning:

```
SEM Foo Bar
  | * - One  lhs.s = 4
```

will define the attribute for alternatives `Two` and `Three` of datatype `Foo`, *and* for alternatives `Two` and `Four` of datatype `Bar`.

It is even allowed to use wildcards in the header of a SEM-definition, as in:

```
SEM *
  | * - One  lhs.s = 4
```

Also, it is now allowed to use wildcards in the header of DATA and ATTR definitions. Thus, we can add an uniform alternative to all existing datatypes, for example to provide a placeholder for error situations:

```
DATA *
  | Error   why : {String}
```

And it is possoble to uniformly add a child to all alternatives of a (set of) datatype(s), for example to store the location in the source file in all alternatives of both the `Expr` and the `Stat` datatype:

```
DATA Expr Stat
  | *  pos : {Int}
```

2 Program architecture

2.1 Main data flow

Figure 1 shows the main dataflow of the UUAGC compiler. It is a graphical representation of the *main* function in source file *Ag.hs*.

The central column shows the various stages of processing. The synthesized attribute *output* of each stage is fed into the next stage as tree to be processed.

- The input string is parsed, resulting in an *AG* structure. This structure directly corresponds to the source constructs.
- The *AG* structure is transformed into a *Grammar* structure. Child definitions, attribute declarations, and attribute definitions are moved such that everything is grouped per nonterminal. Attribute declarations which are common to multiple nonterminals are copied. In this new version the same may happen to generic attribute definitions and child definitions.
- The Grammar structure is processed to automatically add default rules. The result is still a *Grammar*.
- (this would be the place where other transformations could be plugged in)
- The Grammar is ordered, that is the attributes are partinioned in various visits, that can be executed sequentially. The interfaces of the visits are stored, thereby augmenting the Grammar to a *CGrammar*.
- The CGrammar is used to generate Haskell code. Here, the attributes are encoded into tuples that are passed up and down by fold-like functions, which are also generated. The result is a structure that represents a Haskell *Program*.
- The abstract Haskell program is pretty-printed. The result is al list of *PP_Doc* documents, one for each toplevel definition.
- The output string is obtained by rendering each *PP_Doc* using the *disp* function, and written to the output file.

Most of the phases (including parsing) can generate error messages. The errors are collected in the synthesized attribute *errors*, which is a *Seq*uence of *Error* values. The error messages are separately prettyprinted and rendered on standard output.

Most of the phases (including error printing) take additional options through the inherited attribute *options*. The options originate frmm the command line of the program.

Some phases, apart from their regular output and error messages, also generate additional code *Blocks*. A *Block* is a named list of uninterpreted strings, which are supposed to contain Haskell code. This code is merged with the generated code and written to the output file. Blocks with a special name (`optpragmas` and `imports`) are moved to the front. The blocks are yielded in the syntesized attribute *blocks*.

2.2 UUAGC source structure

Figure 2 shows the various modules which make up the UUAGC program. The orange boxes in the rightmost column denote Haskell source files, of which file *AG.hs* contains the main program. The yellow boxes in the other columns denote AG source files.

The eight boxes in the central column correspond to separate tree transformations. The first five (*Transform*, *DefaultRules*, *Order*, *GenerateCode* and *PrintCode* are part of the main dataflow discussed in the previous subsection. The *PrintErrorMessages* transformation, of course, is for prettyprinting error messages. Finally, *SemHsTokens* and *InterfaceRules* are used to separately process attribute definitions and interfaces, respectively. Attribute definitions are special in UUAGC source, as they conform to Haskell syntax (with `@` as an escape character). These Haskell fragments are not parsed by UUAGC, but only processed at lexical level.

The nine boxes in the leftmost colums denote modules that describe datatypes (rather than their attributes). The datatypes that are described are shown in italics in the green pop-up boxes. Four

of these correspond to the intermediate types discussed in the main data flow:

- module *ConcreteSyntax* defines the *AG* datatype
- module *AbstractSyntax* defines the *Grammar* datatype
- module *CodeSyntax* defines the *CGrammar* datatype
- module *Code* defines the *Program* datatype

The other four modules describe auxiliary datatypes.

In short, the modules boxes in the lefmost column describe the *syntax* of the language. They are compiled separately using the `-d` flags, and thus only generate datatype definitions. The six modules in the central column describe the *semantics* of the language. They are compiled using the `-cfs` flags, and thus generate catamorphisms (c), semantics functions (f) and their signatures (s).

The 'syntax' modules are included by the 'semantics' modules, which need to know the datatypes they are attributing. But the 'semantics' modules don't generate code defining the datatypes. The separation of syntactical and semantical aspects is not only good coding practice, but also necessary. Otherwise (that is, if the semantics modules would be compiled with `-d` flag on) for example both the *DefaultRules* module and the *Order* module would generate the *Grammar* datatype, which is described in the *AbstractSyntax* module they both include. That would give a conflict when the generated Haskell files from these two modules are linked together.

There is one more column in figure 2: the two boxes left from the central column (*GenerateCodeSM* and *Dep*). They denote simple file inclusion, which in this case is only done to prevent the file that includes them from growing very big. These files are not compiled separately. There are only two files left in this category (the other two that used to exist are inlined now). Furthermore, these remaining two are obsolescent.

It should be noted that the arrows in figure 2 show the way the AG source files *include* each other. This is not the same as the way the generated Haskell modules *import* each other. The latter relationship is depicted in figure 3.

From this picture, it is more obvious how the various modules cooperate. Reading from right to left, we first note that *Ag.hs* is the module which contains the *main* function and sets other modules to work. Modules that perform a phase from the main dataflow import the description of their source *and* target languages. For example, the *Transform* module imports its source language *ConcreteSyntax* and its target language *AbstractSyntax*. This contrasts with the AG-compiletime inclusion relations from figure 2, where transformation modules need only to know the datatypes of their source language.

From the Haskell import-relations in figure 3 is also becomes clear that the three modules dealing with *HsToken* perform a subordinate task for the *GenerateCode* module. Similar observations can be made for three more clusters of files.

Definitions made in *Options*, *CommonTypes* and *ErrorMessages* are needed almost everywhere. Also the consumers of *Expression* and *Patterns* are too many to list.

3 Modifications for generic attribute definitions

For the generic attribute definition feature, and the general code streamlining, many files were edited, three were removed, and one was added. Not only were the new features described in section 1 implemented, but also some modifications were made that streamline the architecture. This will make future modifications easier. The architecture described in the previous section reflects the new situation.

3.1 New and obsolete files

A new Haskell module *Version.hs* was introduced. It only contains a definition of a banner string containing the version number. This used to be done in *Ag.hs*, the module containing the main function.

The insertion of the version number is performed by *configure*, which generates *Version.hs* from a template *Version.hs.in*. The fact that this is now isolated from the rest of *Ag.hs*, removes the need for preprocessing *Ag.hs* by *configure*. Therefore, *Ag.hs.in* has become obsolete.

The previous version contained two more source modules: a 'syntactical' unit *Rules* and a corresponding 'semantical' unit *SemRules*. The tasks performed by these modules are now integrated in *ConcreteSyntax* and *Transform*, respectively.

The file *Expr.hs*, which contained fossile code, is removed.

3.2 Modified files

The following 'syntactical' units were modified:

- *ConcreteSyntax*: new datatypes *SemDef(s)* were added, originally in the *Rules* module. Structures denoting DATA and SEM definitions were adapted to allow for more than one nonterminal name.
- *Patterns*: added a SELF attribute declaration
- *Code*: changed the type of two leafs from *PP_Doc* to a more structured type (*[String]* and *Pattern*, respectively). Prettyprinting to a *PP_Doc* belongs to a later phase.
- *ErrorMessages*: added a new alternative to denote parsing errors, in order to process parsing errors uniformly with errors in later phases.

The following 'semantical' units were modified:

- *Transform*: major rewrite. In the new version, eveything is first collected in lists, and only then checked for duplicates. (Originally, new declarations were checked for duplicates on encountering them, inserting them in an set that was passed as a threaded attribute).
- *DefaultRules*: drastically rewrote the implementation of use-rules and copy-rules, which makes them shorter and clearer.
- *GenerateCode*: adapted to changes in *Code*. In the included files furthermore:
  - *Comments*: generate better comments for the `-p` option, listing not only the attributes but also the children and local variables
  - *GenerateCodeSM*: outputs Haskell code as *Blocks*, for uniform treatment with the blocks generated by *Transform* (see figure 1)
- *PrintCode*: now also does prettyprinting of patterns, which used to be done too early. Also, suppresses the generation of superfluous parentheses, especially those that are not compliant to ghc-6.6.
- *PrintErrorMessages*: improved readability of error messages

The following Haskell files were modified:

- *Parser*: allowing more than one nonterminal in DATA and SEM definitions. Adapted error processing for uniform treatment.
- *Ag*: streamlined the main dataflow as much as possible, as described in the previous section.

4 Modifications for sequential visits

4.1 New and obsolete files

The new phase that orders the attributes in sequential visits in the main pipeline is modeled in the 'syntactical' unit *CodeSyntax.ag* and the 'semantical' unit *Order.lag*. Note that the latter is written in literate-programming style. The unit *GenerateCode* is also rewritten in literate style, changing the extension to *.lag*.

New units are introduced that describe the syntax and semantics of interfaces: *Interfaces.ag* and *InterfaceRules.lag*. The wrapping of an interface is steered from *SequentialComputation.hs*, which is an auxiliary file used in the ordering phase. Auxiliary types and functions related to ordering are in the new files *SequentialTypes.hs* and *GrammarInfo.hs*.

In the previous version there was optional support for generating so-called 'syntax macros'. This feature is not compatible with the new sequential codegeneration. Four files were related to this feature: two include-files to *GenerateCode* (*GenerateCodeSM.ag* and *Dep.ag*), and two auxiliary Haskell-files (*DepTypes.hs* and *Streaming.hs*). These files are still in the distribution, but their use is commented out.

The file *ExpressionAttr.ag*, formerly included by *GenerateCode.ag*, is now inlined in the new phase *Order.lag*, making the original file obsolete.

The file *Comments.ag*, formerly included by *GenerateCode.ag*, is now inlined in *GenerateCode.lag*, making the original file obsolete. The modifications described in the previous section are retained.

4.2 Modified files

A major rewrite was done of the *GenerateCode* unit. It is now split in two phases: *Order.lag* and *GenerateCode.lag*.

A new intermediate language is defined in *CodeSyntax.ag*. It defines a datatype *CGrammar*, which is similar to the datatype *Grammar* defined in *AbstractSyntax.ag*. The main differences are:

- While each *Production* contained *Alternatives*, now *CProduction* contains not only *CAlternatives* but also *CInterfaces*.
- While each *Alternative* contained *Rules*, now *CAlternative* contains *CVisits*, which in turn contain *CRules*.
- While *Rule* had only one alternative denoting an attribute definition, now em CRule also has an alternative denoting a child visit.

Syntax and semantics of a new auxiliary datastructure *Interface* is defined in *Interfaces.ag* and *InterfaceRules.ag*. Additional Haskell types and functions are defined in *SequentialTypes*, *SequentialComposition*, and *GrammarInfo*.

The source language is slightly enhanced to allow type signatures for local attributes. This brings small changes in *Parser.hs*, *ConcreteSyntax.ag*, *Transform.ag*, *AbstractSyntax.ag*, and *DefaultRules.ag*.

A notion of *unboxed tuples* is introduced, which brings small changes in *Code.ag* and *PrintCode.ag*.

The new features can be enabled by six new options introduced in *Options.hs*. New error situations are trapped in *(Print)ErrorMessages.ag*: three tastes of circularity replace the old *CircGrammar* error, and type signatures can be 'duplicate' or 'missing'.

The main file *Ag.hs* is updated to include the new phase, and to handle the new options.

5 Installation

As described in the readme document, compiling and installing UUAGC from the source is very easy. It is done by typing the following commands:

- autoconf
- configure
- make build
- make install

Due to the bootstrapping nature of the process (UUAGC is written using itself), the third step requires an existing UUAGC system. This is not included in the SVN archive, and should be downloaded separately.

The remainder of this section describes in some more detail what happens during the steps above. Understanding this is not necessary for simply installing UUAGC, but it is to be able to modify the installation procedure. The process is summarized in figure 4, and discussed below.

First, the GNU utility *autoconf* is run to generate a configuration script named *configure*. It is specified by the description in *configure.in*.

Next, the *configure* script is run. It basically inserts some configuration-dependent details (such as the compiler to use) in source files. Thus, these files can be generated from a template which is provided in the distribution. The template typically is named with suffix *.in*. For UUAGC, four files are generated in this way:

- *Makefile* generated from template *Makefile.in*, inserting the names of compilers and other utilities to use.
- *src/Version.hs* generated from template *src/Version.hs.in*, inserting the version number found in file *VERSION*.
- *uuagc.cabal* generated from template *uuagc.cabal.in*, again inserting the version number.
- *scripts/mkAgDepend.sh* generated from template *scripts/mkAgDepend.sh.in*, which is just copied because it doesn't contain configuration-specific details

Subsequently, *make build* is used to build the system. It performs the following actions, steered by the *Makefile* generated before:

- (only once): *Setup.hs* is compiled to generate a program *Setup* that can drive the Haskell compiler steered by a cabal-file
- The dependencies between AG-files are detected by chasing the include-statements, thus effectively determining the arrows in figure 2. The dependencies are temporarily stored in file *ag.depend*, and used to decide by *make* itself which files need to be recompiled
- The 'syntactic' AG-sources are compiled using an existing version of UUAGC, with `-d` flag
- The 'semantic' AG-sources are compiled using an existing version of UUAGC, with `-cfs` flag
- The Haskell program is build from the files generated by UUAGC, the original Haskell-files in the distribution, and the *Version.hs* that was generated by *configure*. Compilation is controlled by *Setup*. The dependencies between Haskell files are checked by Haskell itself, so *make* needs no information about them.

The setup process is steered by *uuagc.cabal*, which specifies that the executable should be placed in a subdirectory of directory *dist*.

Finally, *make install* is used to install the new UUAGC system. It copies the executable from the *dist* directory to the location where executables should be stored.

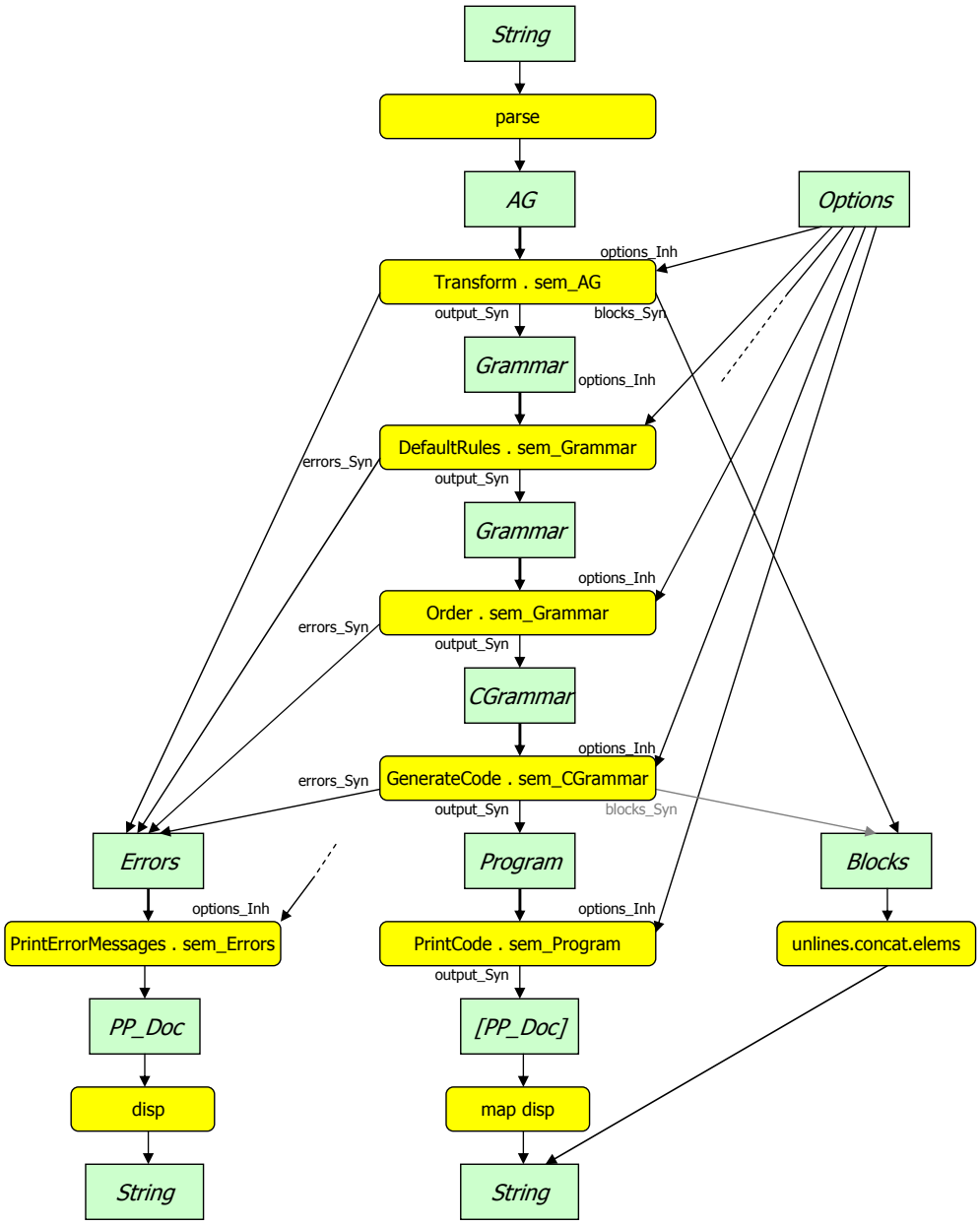

Figure 1: UUAGC main dataflow

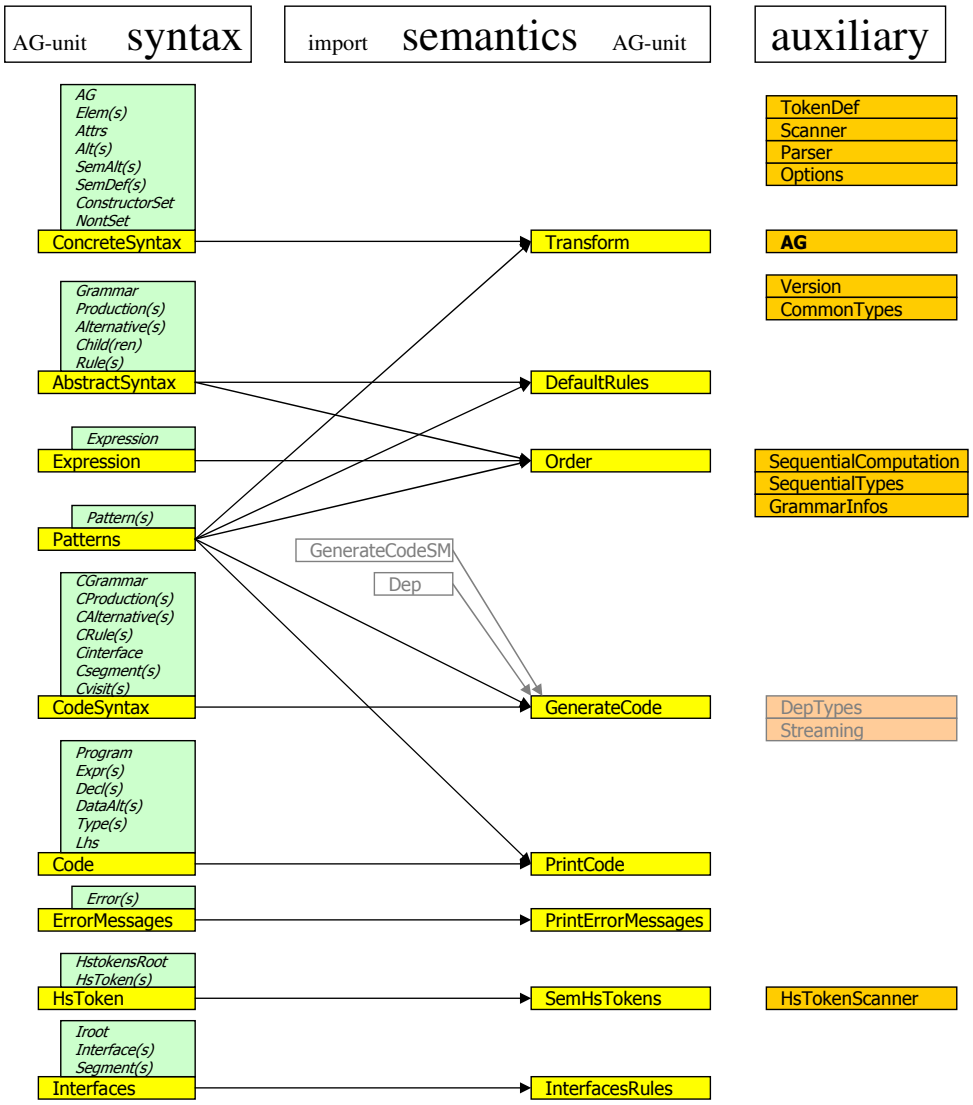

Figure 2: UUAGC source structure

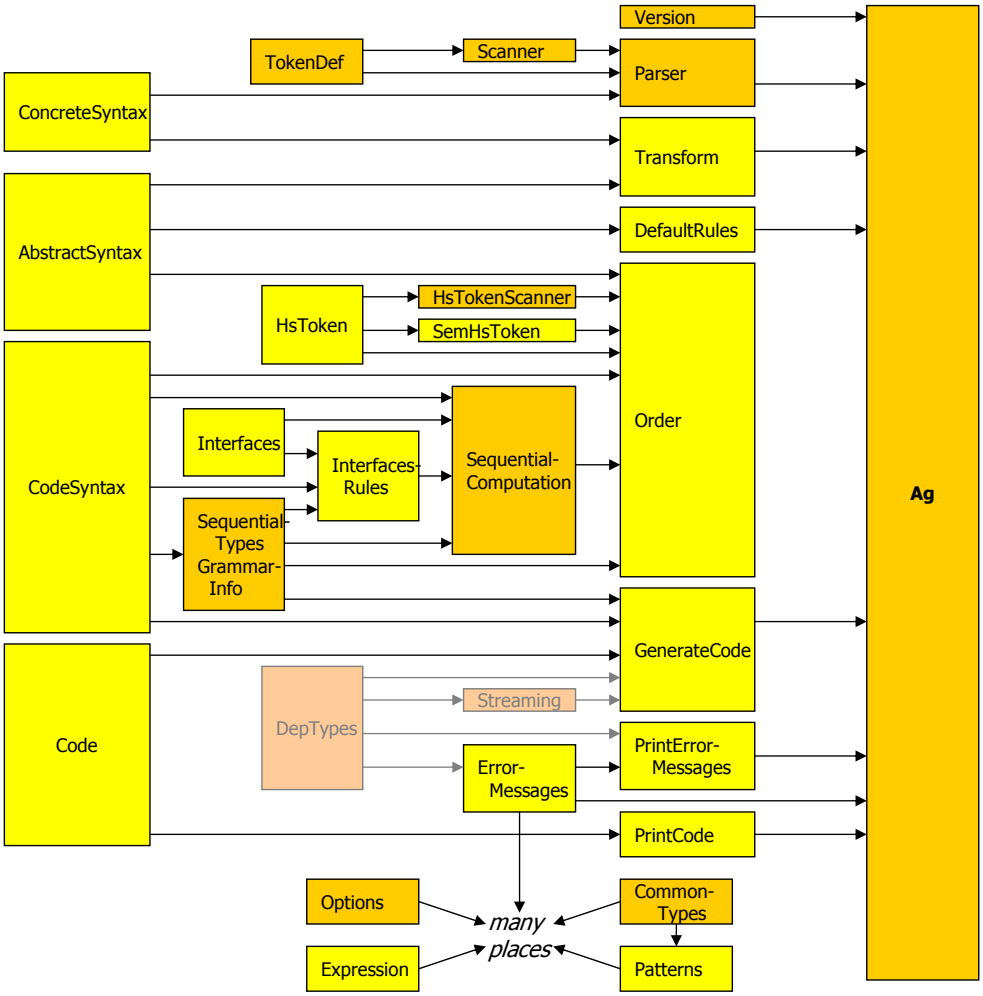

Figure 3: Dependencies in UUAGC generated code

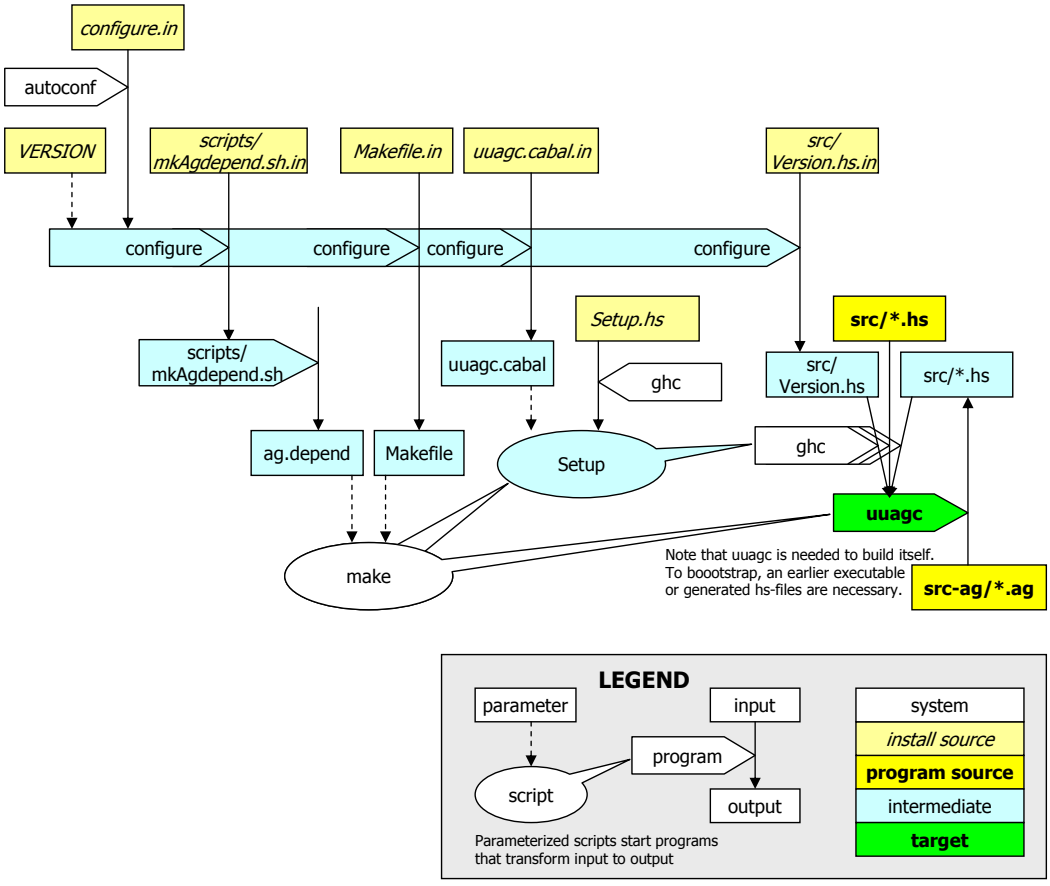

Figure 4: Installation of UUAGC