A Gentle Introduction to *Compiler Generation* Using HACS*

Kristoffer H. Rose

March 30, 2014

Abstract

Higher-order Attribute Contraction Schemes—or HACS—is a language for programming compilers. With HACS it is possible to create a fully functional compiler from a single source file. This document explains how to get HACS up and running, the basic conventions to follow, and how to program (and use) an example (simple) compiler; in addition the document includes reference information on HACS in appendix.

1 Getting Started

In this section we make sure you have a functional HACS installation on your computer.

1.1 Requirements. To run the HACS examples here you need a *nix system (including a shell and the usual utilities) with these common programs: a Java development environment (at least Java 1.6 SE SDK, with `java` and `javac` commands); a C99 compiler with the *icu* libraries; the `flex` lexical generator; the `wget` internet retrieval and `unzip` unpacking utility; and the GNU `make` command. In addition, the setup process needs internet access to retrieve the JavaCC parser generator [3].

1.2 Commands. Retrieve the *hacs.zip* archive, extract it to a new directory, and intialize the `Makefile`, for example with the following commands:[1]

```
$ mkdir myfirst
$ cd myfirst
$ wget http://krisrose.net/hacs.zip
$ unzip hacs.zip
$ cp Makefile-hacs-template Makefile
```

Now check that all the commands defined in the `SETUP` section near the top of you freshly copied `Makefile` are correct; if any need changing then uncomment the line and change it to reflect the required commands on your system. Then check that it works with these command, which will need Internet access the first time:[2]

*UNFINISHED DRAFT—Feedback Appreciated

[1] User input is blue.

[2] Note that the spacing of the output may vary—this is not a stable capability yet.

```
$ make first.run
...
$ ./first.run --action=Compile \
              --term="{initial := 1; rate := 1.0; position := initial + rate * 60;}"
     LDF T ,  #1
     STF initial , T
     LDF T_40 ,  #1.0
     STF rate , T_40
     LDF T_1 ,  initial
     LDF T_1_90 ,  rate
     LDF T_2 ,  #60
     MULF  T_2_84   ,  T_1_90   ,  T_2
     ADDF  T_25   ,  T_1   ,  T_2_84
     STF position , T_25
```

Congratulations—you just built your first compiler! (It may have taken a few minutes, as HACS also performed the initial setup, which is only needed on the first run.)

1.3 Plan. In the remainder of this document we introduce the most important features of the HACS language by explaining the corresponding parts of the *first.hx* specification used above, which is adapted from [1, Fig. 1.7], as well as several other minor examples. We explain lexical analysis in Sec. 2, syntax analysis in Sec. 3, basic semantic sorts, schemes, and rules in Sec. 4, semantic analysis in Sec. 5, Code Generation in Sec. 6, and the invokation conventions in Sec. 7. Appendices include reference information (App. A), some bonus examples (App. B) and explanations for a few cryptic error messages (App. C).

2 Lexical Analysis

Lexical analysis is the process of splitting the input text into tokens. HACS uses a fairly standard variation of *regular expressions* for this. As is common with parser generators, unique tokens (such as keywords) can be declared along with the syntax productions where they are used.

We shall give examples from *first.hx* of the HACS distribution. The first part uses regular expression notation to define the spacing conventions and tokens used as "terminal symbols" by the compiler.

2.1 Example (white space)**.** The regular expression for white space is setup by `space` followed by the regular expression of what to skip – here spaces, tabs, and newlines, where HACS uses backslash for escaping in character classes with usual C-style language escapes.

```
space [ \t\n] ;
```

2.2 Example (tokens)**.** Here is the relevant fragment for setting up the concrete syntax of integers, basic floating point numbers, and identifiers:

```
token Int    | ⟨Digit⟩+ ;                        // tokens
token Float  | ⟨Int⟩ "." ⟨Int⟩ ;
token Id     | ⟨Lower⟩+ ('_'? ⟨Int⟩)* ;

token fragment Digit  | [0-9] ;
token fragment Lower  | [a-z] ;
```

The example illustrates the following particulars of HACS in general and lexical expressions in particular:

- Declarations generally start with a keyword or two and are terminated by a `;` (semicolon).
- `token` declarations in particular have the token name followed by a regular expression between a `|` (vertical bar) and a `;` (semicolon).
- A regular expressions is a sequence of units, corresponding to the concatenation of sequences of characters that match the units. Each unit can be a *character class* such as `[a-z]`, which matches a single character in the indicated range, a string such as `'.'`, or a reference to a token or fragment such as ⟨`Lower`⟩.[3]
- A `token fragment` declaration means that the defined token can only be used in other token declarations, and not in syntax productions defined below.
- Every regular expression component can be followed by a repetition marker `?`, `+`, or `*`.
- HACS supports C/Java-style comments.
- Finally, the special trailing `('_'? [0-9]+)*` of the `Id` token regular expression is a special convention required for symbols that are automatically renamed by the system, linked to the `symbol` qualifier for productions, as we shall see below.

2.3 Commands (lexical analysis)**.** The generated command, *first.run*, from above, can be used as a lexical analyzer with two arguments: a *token sort* and a *token term*.[4] Execution then proceeds by parsing the string following the syntax of the token. We can, for example, check the lexical analysis of a number:

```
$ ./first.run --sort=Float --term=34.56
34.56
```

If there is an error, the lexical analyzer will inform us of this:

```
$ ./first.run --sort=Int --term=34.56
Exception in thread "main" java.lang.RuntimeException: net.sf.crsx.CRSException:
net.sf.crsx.parser.ParseException:
 Parse error in embedded firstInt term at line 1, column 27: ⟦ 34.56 ⟧ ...
 Encountered " <T_T_M_FLOAT> "34.56 "" at line 1, column 29.
Was expecting one of:
    <T_meta1_T_M_INT> ...
    "\u27e9" ...
    <T_T_M_INT> ...
```

(where the trail of Java exceptions has been truncated: the important information is in the first few lines).

3 Syntax Analysis

Once we have tokens, we can use HACS to program a complete syntax analysis with a grammar that specifies how the input text is decomposed according to a *concrete syntax* and how the desired *abstract syntax tree* (AST) is constructed from that. Notice that HACS does not provide a "parse tree" in the traditional sense, *i.e.*, a tree that represents the full concrete syntax parse: only the AST

[3] The used Unicode characters are summarized in Appendix A.11.

[4] The command has more options that we shall introduce as we need them.

is built. Grammars are structured following the *sorts* of AST nodes,[5]with concrete syntax details managed through annotations and "syntactic sugar" declarations.

3.1 Example. The *first.hx* example contains a syntax analyzer. Our small example source language merely has blocks, statements, and a few forms of expression. We first show the declaration of and productions for expressions:

```
sort Exp   | ⟦ ⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1
           | ⟦ ⟨Exp@2⟩ * ⟨Exp@3⟩ ⟧@2
           | ⟦ ⟨Int⟩ ⟧@3
           | ⟦ ⟨Float⟩ ⟧@3
           | ⟦ ⟨Name⟩ ⟧@3
           | sugar ⟦ (⟨Exp#⟩) ⟧@3 → # ;

sort Name  | symbol ⟦ ⟨Id⟩ ⟧ ;
```

The grammar structure here has two sorts: `Exp` for expressions and `Name` for names. Simple HACS grammars such as this follow these conventions:[6]

- Each sort is defined by a `sort` declaration followed by a number of *productions*, each introduced by a `|` (bar). (The first `|` corresponds to what is usually written "::=" or "→" in grammars.)
- Concrete syntax is enclosed in ⟦...⟧ (known as "double" or "white" brackets). Everything inside double brackets should be seen as literal syntax, even \ (backslash), *except* for HACS white space (corresponding to `[ \t\n\r]`), which is ignored, and fragments in ⟨...⟩ (angle brackets), which are special (explained next).
- References to *nonterminals* (other productions) are wrapped in ⟨...⟩ (angle brackets).
- *Precedence* is indicated with `@`n, where higher numbers n designate higher (tighter) precedence. Any reference in as well as the alternative itself may have a precedence in this way; in the example we establish that `*` binds tighter than `+`, and that both operators are left recursive. Note that we specify the precedence of both the entire expression, after the ⟦⟧s, and of each component, inside the ⟨⟩. (In fact where the precedence is omitted we could have written `@0`.)
- The special `sugar` declaration expresses that the concrete syntax can use parentheses to raise the precedence of the enclosed expression to 3: it is the first example of a *rewrite rule* with a → that we see, where we remark that the expression is marked `#` so we can use the `#` to indicate that it is extracted as the abstract result of concrete syntax with parenthesis. (In fact the general rule is that when an → is used then all sort specifiers must be "disambiguated" with distinct markers like `#` or `#5` in this way.)
- The `Name` sort is defined as a `symbol`, which is only allowed because the underlying `Id` token permits a trailing `_`n (underscore and count); this permits the use as binders and automatic symbol generation.

3.2 Commands. We can parse an expression from the command line:

[5]We use the concept of "sort" from ("many-sorted") algebra in HACS: a sort is essentially the same as a "type" but this avoids confusion between the type structures that described compilers operate on and the pieces of the HACS specification.

[6]As before the details of the Unicode characters are in the appendix.

```
$ ./first.run --sort=Exp --term="(2+(3*(4+5)))"
2 + 3 * ( 4 + 5 )
```

Notice that the printout differs slightly from the input term as it has been "resugared" from the AST with minimal insertion of parentheses.

3.3 Example. Our example language also includes statements with declarations of local variables. These we define with HACS in a *tail-recursive* manner, to have the scope structure follow the AST structure:

```
sort Stat  | ⟦ ⟨[x:Id]⟩ := ⟨Exp⟩ ; ⟨Stat[x:Id]⟩ ⟧
           | ⟦ { ⟨Stat⟩ } ⟨Stat⟩ ⟧
           | ⟦ ⟧ ;
```

These declarations illustrate the *higher-order binding* feature of HACS:

- When a non-terminal reference to a sort is given as $\langle$`[`x`:`T`]`$\rangle$ instead of $\langle T \rangle$ for T a *token underlying a symbol*, then the parser will behave the same as for $\langle T \rangle$ except the actual symbol will be recorded as a *binder*, referenced as x, and not put in the generated AST.
- When a binder x has been defined, then a reference can be given as $\langle N$`[`x`:`T`]`$\rangle$ to specify that an N subtree is the *scope* of the binder.

4 Semantic Sorts and Schemes

When analyzing the AST, which we cover in the next section, we shall need some utility sorts and functions. This section details how these are defined.

4.1 Example. The next part of the *first.hx* example has the semantic sorts and operations used. For our toy language that just means the notion of *type* used and the way that types are unified.

```
sort Type | Int | Float ;

| scheme Unif(Type,Type) ;
Unif(Int, Int) → Int;       //(Int-Int)
Unif(#t1, Float) → Float;   //(Any-Float)
Unif(Float, #t2) → Float;   //(Float-Any)
```

The code declares a new sort, `Type`, which is a *semantic* sort because it does not include any syntactic cases: all the possible values (as usual listed after leading `|`s) are simple term structures written without any custom syntax in ⟦⟧s. Structures are written with a leading "constructor," which should be a capitalized word (the same as sort names), optionally followed by some "arguments" in `()`s, where the declaration gives the sort for each argument.

The semantic sort also includes a `scheme` declaration for the `Unif` constructor followed by an argument list with two `Type` arguments. The scheme declaration is followed by *rules* of the form "pattern→replacement," which must specify for each possible shape of `Unif`-construction how it should be simplified by the scheme. Rules may include "meta-variables" starting with `#` (hash), like `#t1`, to designate "arguments" that should be copied from the pattern to the replacement.

The above rules can, for example, be used to simplifify a term as follows:

```
Unif(Float, Unif(Int, Float)) → Unif(Float, Float)
                              → Float
```

using first the (Any-Float) rule on the inner `Unif` and then the (Any-Float) rule (again) on the remaining `Unif`. However, we could also have rewritten by applying the (Float-Any) rule immediately on the top `Unif`, like this:

$$\texttt{Unif(Float, Unif(Int, Float))} \rightarrow \texttt{Float}$$

This is characteristic for rewriting systems such as HACS: there is no predetermined "reduction order" and thus we have to pay attention to create rules that have a well defined result.

Trouble is possible when there are *overlaps*: when some term form can be simplified by two different rules then a good rule of thumb is to ensure that the two possible rewrites give the same result. In our example, the term `Unif(Float, Float)` matches both (Any-Float) and (Float-Any), and thus we have to be careful to check that the two rules give the same result, which is indeed the case (the result is `Float` for both rewrites).[7]

5 Semantic Analysis

Once the AST is loaded we can analyze it. In this section, we demonstrate this by fully developing a simple type checking analysis of expressions.

5.1 Example. In Ex. 3.3 we presented the abstract syntax of the small expression language of *first.hx*. A type analysis of the expressions of the language might look as follows as a standard *syntax directed definition* (SDD), where we use E for the `Exp` non-terminal, and these attributes:

- $E.e$: inherited *environment* (*aka.* symbol table) mapping names to types.
- $E.t$: synthesized *type* of expression.

In the notations of [1], the SDD can be specified something like this:

PRODUCTION	SEMANTIC RULES	
$E \rightarrow E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \mathrm{Unif}(E_1.t, E_2.t)$	$(E1)$
$\mid E_1 * E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \mathrm{Unif}(E_1.t, E_2.t)$	$(E2)$
$\mid$ **int**	$E.t = \mathrm{Int}$	$(E3)$
$\mid$ **float**	$E.t = \mathrm{Float}$	$(E4)$
$\mid$ **name**	$E.t = \text{if } \mathrm{defined}(E.e, \textbf{name}.sym) \text{ then } \mathrm{lookup}(E.e, \textbf{name}.sym)$ $\text{else } \mathrm{TypeError}$	$(E5)$

The SDD has one location where the attribute dependencies are interesting: $(E5)$ defines the synthesized attribute $E.t$ in terms of the inherited attribute $E.e$ in a "down then up" configuration.

To translate an SDD to HACS, we follow some simple rules:

- Environment (*aka.* symbol table) attributes correspond to maps.
- Synthesized attribute assignments translate into synthesis rules for the corresponding pattern.
- Inherited attributes are associated to one or more recursive *schemes*, with inherited assignments corresponding to propagation of the inherited attribute recursively to all appropriate subterms.

[7] The details of this property—called *confluence*—are complex and beyond the scope of this introduction: for precise results consult the references.

- When an inherited attribute depends on synthesized attribute of a preceding non-terminal in the production,[8] then a *helper scheme* needs to be introduced to capture the action to take when the synthesized attribute becomes available.

The following example applies these rules to the SDD above.

5.2 Example. Let us first do the part of the SDD of Ex. 5.1, which implements the synhtesized attribute.

```
attribute ↑t(Type);       // synthesized expression type
sort Exp | ↑t;
⟦ ⟨Exp#1 ↑t(#t1)⟩ + ⟨Exp#2 ↑t(#t2)⟩ ⟧ ↑t(Unif(#t1,#t2));
⟦ ⟨Exp#1 ↑t(#t1)⟩ * ⟨Exp#2 ↑t(#t2)⟩ ⟧ ↑t(Unif(#t1,#t2));
⟦ ⟨Int#⟩ ⟧ ↑t(Int);
⟦ ⟨Float#⟩ ⟧ ↑t(Float);
```

The first block defines the synthesized $E.t$ attributes, using HACS ↑ for synthetic, with the sort in parenthesis, `(Type)`.

We then associate the synthetic attribute `t` to the `Exp` sort—all synthetic attributes are associated with one or more sorts—and we give a special *synthesis rule* that shows for each form of `Exp`, where `t` is not populated by `TA`, how synthetic attributes matches on fragments propagate to the entire expression, *e.g.*, the rule for the SDD's (1) is

```
⟦ ⟨Exp#1 ↑t(#t1)⟩ + ⟨Exp#2 ↑t(#t2)⟩ ⟧ ↑t(Unif(#t1,#t2));
```

and should be read as something like "When considering an `Exp` (the current sort) which has the shape $[\![\langle Exp\rangle + \langle Exp\rangle]\!]$ where furthermore the first expression has a value matching `#t1` for the synthesized attribute `t`, and the second expression has a value matching `#t2` for the synthesized attribute `t`, then the entire expression has the value `Unif(#t1,#t2)` for the synthesized attribute `t`." Notice that the attribute patterns have the same shape as the attribute declaration—a recurring feature of HACS. The following three lines correspond in the same way to $(E2, E3, E4)$.

5.3 Example. The inherited attributes have to be attached to so-called *schemes* that are recursive functions encoding recursive traversals of the AST.

```
attribute ↓e{Name:Type};  // inherited type environment

sort Exp | scheme ⟦ TA ⟨Exp⟩ ⟧ ↓e ;

⟦ TA id ⟧ ↓e{⟦id⟧ : #t} → ⟦ id ⟧ ↑t(#t);
⟦ TA id ⟧ ↓e{¬⟦id⟧} → error⟦Undefined identifier ⟨id⟩⟧;

⟦ TA ⟨Int#⟩ ⟧ → ⟦ ⟨Int#⟩ ⟧;
⟦ TA ⟨Float#⟩ ⟧ → ⟦ ⟨Float#⟩ ⟧;
⟦ TA (⟨Exp#1⟩ + ⟨Exp#2⟩) ⟧ → ⟦ (TA ⟨Exp#1⟩) + (TA ⟨Exp#2⟩) ⟧;
⟦ TA (⟨Exp#1⟩ * ⟨Exp#2⟩) ⟧ → ⟦ (TA ⟨Exp#1⟩) * (TA ⟨Exp#2⟩) ⟧;
```

The first block sets up the and inherited $E.e$ attributes, using HACS ↓ for inherited, and the special notation for a symbol mapping (or symbol table), `{Name:Type}`, with the sorts involved.

This is followed by a definition of the recursive `TA` scheme for *distributing* all types in the symbol table to actual symbol occurrences. Specifically, `scheme⟦TA⟨Exp⟩⟧↓e` means that (the sort `Exp`) has a scheme with the specified syntax that distributes the inherited attribute `e`. Note that the scheme

[8] This is always true for so-called *L-attributed* grammars [1, §5.1], which are the kind we shall discuss here.

uses *concrete syntax* for applications of the scheme: this is a convenience that makes it possible to invoke specific schemes directly from expressions entered in the command line. It does mean that we have to be careful with using the syntax correctly: we have not, for example, given a precedence to `TA`, so in the recursive rules we have expressions like

```
⟦ (TA⟨Exp#1⟩) + (TA⟨Exp#2⟩) ⟧
```

which uses the `(...)` syntactic sugar to properly nest the expression. We could also have written

```
⟦ ⟨Exp ⟦TA ⟨Exp#1⟩⟧⟩ + ⟨Exp ⟦TA ⟨Exp#2⟩⟧⟩ ⟧
```

using nested `⟨...⟩` constructs instead, which does not depend on the syntactic sugar.

Otherwise, the scheme just exploits the syntax to express *where* the recursion goes, which then automatically means that the inherited attribute will follow, as HACS inherited attributes always follow a scheme. The first two "`TA`-rules" have the cases for looking up a symbol in the environment: if it is there, *i.e.*, if the `#id` specified can make the environment pattern `↓{#id:#t}` match, instantiating `#t` to the type of `#id`, then the first rule is used and the result is the `#id` with the added synthesized attribute value `↑t(#t)`, *i.e.*, the t attribute gets the value we value we extracted from the environment. The rule with the negated pattern `↓e{¬#id}` will match if the `#id` symbol is *not* defined in the environment, then printing a simple error message. In a sense the two `Name` cases correspond to $(E5)$ in the SDD.

We continue the analysis by explaining how an environment of global variables is initialized and maintained across statements, where new names are introduced.

5.4 Example. Our type analysis from Ex. 5.1 is extended to S for the `Stat` non-terminal, with this attribute:

- $S.e$ inherited "environment" symbol table mapping names to types.

with these rules:

PRODUCTION	SEMANTIC RULES	
$S \rightarrow \textbf{name} := E_1; S_2$	$E_1.e = S.e; S_2.e = \text{extend}(S.e, \textbf{name}.sym, E_1.t)$	$(S1)$
$\mid \{\ S_1\ \}\ S_2$	$S_1.e = S.e; S_2.e = S.e$	$(S2)$
$\mid \epsilon$		$(S3)$

The SDD has one locations where the attribute dependencies are interesting: $(S1)$ copies the synthesized $E_1.t$ into the extended environment $S_2.e$, which is a "left to right" dependency for the e attribute. To translate this into HACS, we create a specific scheme for each such "left to right" dependency.

Here is the code from *first.hx*, which implements the SDD above in HACS.

```
sort Stat | scheme ⟦ TA { ⟨Stat⟩ } ⟧ ↓e ;

⟦ TA { id := ⟨Exp#2⟩; ⟨Stat#3[id]⟩ } ⟧ → ⟦ TA2 { id := TA ⟨Exp#2⟩; ⟨Stat#3[id]⟩ } ⟧;
{
  | scheme ⟦ TA2 { ⟨Stat⟩ } ⟧ ↓e;
  ⟦ TA2 { id := ⟨Exp#2 ↑t(#t2)⟩; ⟨Stat#3[id]⟩ } ⟧ →
    ⟦ id := ⟨Exp#2⟩; ⟨Stat ⟦TA {⟨Stat#3[id]⟩}⟧ ↓e{id:#t2}⟩ ⟧;
}

⟦ TA { { ⟨Stat#1⟩ } ⟨Stat#2⟩ } ⟧ → ⟦ { TA { ⟨Stat#1⟩ } } TA { ⟨Stat#2⟩ } ⟧;

⟦ TA { } ⟧ → ⟦ { } ⟧;
```

Again, we use a parsed form of the scheme, with added {}s to disambiguate in nested cases.

The first rule handles the hard case $(S1)$, with two schemes: the main scheme TA appeals to a helper scheme, TA2, which handles what needs to be done after the type of the expression becomes available. Essentially, the first rule recurses the TA scheme into the expression to associate types with any names. A new, second scheme, TA2, then takes over once the synthesized attribute has been instantiated for the expression, passing the environment extension to the subsequent statement. (The {} wrappers serve as a grouping for the "current sort" status, which here is not in fact changed.)

6 Intermediate and Code Generation

After the analysis we are ready for generating code.

6.1 Example. The part of *first.hx* that handles the translation from abstract syntax to the intermediate representation is shown in Fig. 1. The fragment contains the usual components: a syntax specification, rewrite schemes, and rewrite rules for the ICG scheme.

The code only uses two new features: ¶ markers in the syntax to indicate newlines, and rules that introduce *fresh* variables (of Tmp sort): when the replacement of a rule uses a symbol, which was not in the pattern, then this corresponds to *generating* a new globally unique symbol. So each time the rule

```
⟦ ICG id := ⟨Exp#2 ↑HasType(#t2)⟩; ⟧
  → ⟦ { ⟨I_Progr ⟦ICGExp T ⟨Exp#2⟩⟧ ↓TmpType{T:#t2}⟩ } id = T; ⟧ ;
```

is used, T denotes a new so-called "fresh" symbol. When printed, the various incarnations of T will be named T_1, T_86, *etc.*

6.2 Example. The code generation part of *first.hx* is the final translation CG from the intermediate representation to assembly code. This uses no new features, and is shown in Fig. 2, however, it is still worth a sanity check, walking through the CG scheme and checking that all syntactic cases are covered.

6.3 Remark (concrete *vs.* raw syntax)**.** In the presentation we have chosen to use *concrete syntax* even for semantic operations. This has the advantage of allowing direct invocation of even complex structured calculations from the command line but it does "pollute" the syntax of the defined language. (Production versions of HACS (not yet released) will have the option of generating parsers that ignore concrete syntax of schemes when running the compiler.) It is sometimes practical to define "bridge schemes" that make schemes available both in syntax and raw; we give an example of this in the following section.

7 Main

Finally, we put everything together.

7.1 Example. The last fragment of *first.hx* is the main compilation function, which pulls together the three stages defined in previous sections.

```
1  sort A_Progr | scheme ⟦ Compile ⟨Stat⟩ ⟧ ;
2  ⟦ Compile ⟨Stat#1⟩ ⟧ → ⟦ CG ICG TA ⟨Stat#1⟩ ⟧ ;
```

```
token T | T ('_' ⟨Int⟩)* ; // temporary

// Concrete syntax & abstract syntax sorts.

sort I_Progr  | ⟦⟨I_Instr⟩ ⟨I_Progr⟩⟧ | ⟦⟧ ;

sort I_Instr  | ⟦⟨Tmp⟩ = ⟨I_Arg⟩ + ⟨I_Arg⟩;¶⟧
              | ⟦⟨Tmp⟩ = ⟨I_Arg⟩ * ⟨I_Arg⟩;¶⟧
              | ⟦⟨Tmp⟩ = ⟨I_Arg⟩;¶⟧
              | ⟦⟨Name⟩ = ⟨Tmp⟩;¶⟧
              ;

sort I_Arg    | ⟦⟨Name⟩⟧
              | ⟦⟨Float⟩⟧
              | ⟦⟨Int⟩⟧
              | ⟦⟨Tmp⟩⟧
              ;

sort Tmp | symbol ⟦ ⟨T⟩ ⟧ ;

// Translation scheme.

attribute ↓TmpType{Tmp:Type} ;

sort I_Progr;

| scheme ⟦ ICG { ⟨Stat⟩ } ⟧ ↓TmpType ;

⟦ ICG { id := ⟨Exp#2 ↑t(#t2)⟩; ⟨Stat#3[id]⟩ } ⟧
   → ⟦ { ⟨I_Progr ⟦ICGExp T ⟨Exp#2⟩⟧ ↓TmpType{⟦T⟧:#t2}⟩ } id = T; ICG { ⟨Stat#3[id]⟩ }  ⟧ ;
⟦ ICG { { ⟨Stat#1⟩ } ⟨Stat#2⟩ } ⟧ → ⟦ { ICG { ⟨Stat#1⟩ } } ICG { ⟨Stat#2⟩ } ⟧ ;
⟦ ICG { } ⟧ → ⟦ ⟧;

| scheme ⟦ ICGExp ⟨Tmp⟩ ⟨Exp⟩ ⟧ ;

⟦ ICGExp T ⟨Int#1⟩ ⟧ → ⟦ T = ⟨Int#1⟩; ⟧ ;
⟦ ICGExp T ⟨Float#1⟩ ⟧ → ⟦ T = ⟨Float#1⟩; ⟧ ;
⟦ ICGExp T id ⟧ → ⟦ T = id; ⟧ ;

⟦ ICGExp T ⟨Exp#1⟩ + ⟨Exp#2⟩ ⟧
  → ⟦ {ICGExp T_1 ⟨Exp#1⟩} {ICGExp T_2 ⟨Exp#2⟩} T = T_1 + T_2; ⟧ ;

⟦ ICGExp T ⟨Exp#1⟩ * ⟨Exp#2⟩ ⟧
  → ⟦ {ICGExp T_1 ⟨Exp#1⟩} {ICGExp T_2 ⟨Exp#2⟩} T = T_1 * T_2; ⟧ ;

// Helper to flatten code sequence.
| scheme ⟦ {⟨I_Progr⟩} ⟨I_Progr⟩ ⟧;
⟦ {} ⟨I_Progr#3⟩ ⟧ → #3 ;
⟦ {⟨I_Instr#1⟩ ⟨I_Progr#2⟩} ⟨I_Progr#3⟩ ⟧ → ⟦ ⟨I_Instr#1⟩ {⟨I_Progr#2⟩} ⟨I_Progr#3⟩ ⟧;
```

Figure 1: Intermediate Code Generation.

```
1  sort A_Progr | ⟦ ⟨A_Instr⟩ ⟨A_Progr⟩ ⟧ | [[]] ;

2  sort A_Instr | ⟦ LDF ⟨Tmp⟩, ⟨A_Arg⟩ ¶⟧
3               | ⟦ STF ⟨Name⟩, ⟨Tmp⟩ ¶⟧
4               | ⟦ ADDF ⟨A_Arg⟩, ⟨A_Arg⟩, ⟨A_Arg⟩ ¶⟧
5               | ⟦ MULF ⟨A_Arg⟩, ⟨A_Arg⟩, ⟨A_Arg⟩ ¶⟧ ;

6  sort A_Arg | ⟦ #⟨Float⟩ ⟧ | ⟦ #⟨Int⟩ ⟧ | ⟦ ⟨Name⟩ ⟧ | ⟦ ⟨Tmp⟩ ⟧ ;

7  sort A_Progr | scheme ⟦ CG ⟨I_Progr⟩ ⟧ ;

8  ⟦ CG ⟧ → [[]] ;

9  ⟦ CG T = ⟨I_Arg#1⟩ + ⟨I_Arg#2⟩ ; ⟨I_Progr#⟩ ⟧
10   → ⟦ ADDF T, [⟨I_Arg#1⟩], [⟨I_Arg#2⟩] CG ⟨I_Progr#⟩ ⟧ ;

11 ⟦ CG T = ⟨I_Arg#1⟩ * ⟨I_Arg#2⟩ ; ⟨I_Progr#⟩ ⟧
12   → ⟦ MULF T, [⟨I_Arg#1⟩], [⟨I_Arg#2⟩] CG ⟨I_Progr#⟩ ⟧ ;

13 ⟦ CG T = ⟨I_Arg#1⟩ ; ⟨I_Progr#⟩ ⟧
14   → ⟦ LDF T, [⟨I_Arg#1⟩] CG ⟨I_Progr#⟩ ⟧ ;

15 ⟦ CG name = T ; ⟨I_Progr#⟩ ⟧
16   → ⟦ STF name, T CG ⟨I_Progr#⟩ ⟧ ;

17 sort A_Arg | scheme ⟦ [⟨I_Arg⟩] ⟧ ;
18 ⟦ [T] ⟧ → ⟦ T ⟧ ;
19 ⟦ [name] ⟧ → ⟦ name ⟧ ;
20 ⟦ [⟨Float#1⟩] ⟧ → ⟦ #⟨Float#1⟩ ⟧ ;
21 ⟦ [⟨Int#1⟩] ⟧ → ⟦ #⟨Int#1⟩ ⟧ ;
```

Figure 2: Code Generation.

It is also possible to define wrapper schemes that can be invoked with an option, this is useful when the input term is read from a file where it is not obvious to include the compiler instructions.

```
3  | scheme Compile(Stat);
4  Compile(#stat) → ⟦ Compile ⟨Stat#stat⟩ ⟧ ;
```

This is the `--action` we invoked back in Com. 1.2. Such wrapper raw schemes must have a single argument.

A Reference Manual

This appendix contains the concise details of the HACS system in the form of a number of "manual" sections followed by some hints on the notation and current limitations.

A.1 Manual (grammar structure)**.** A HACS compiler is specified as a single *.hx* module file with the following structure:

```
module modulename
{
  // Lexical Analysis
  // Syntax Analysis
  // Semantic Analysis
  // Code Generator
  // Main
}
```

the *modulename* should be a string with a Java style fully qualified class name, where the last component is capitalized, like `"net.sf.crsx.samples.gentle.First"` in the example above. The individual sections specify the compiler, and the possible contents is documented in the manual blocks throughout this document.

A.2 Manual (lexical declarations)**.** A token is declared with the keyword `token` followed by the token (sort) name, a `|` (vertical bar), and a *regular expression*, which has one of the following forms (with increasing order of precedence):

1. Several alternative regular expressions can be combined with further `|` characters.
2. Concatenation denotes the regular expression recognizing concatenations of what matches the subexpressions.
3. A regular expression (of the forms following this one) can be followed by a *repetition marker*: `?` for zero or one, `+` for one or more, and `*` for zero or more.
4. A simple word without special characters stands for itself.
5. A string in single or double quotes stands for the contents of the string except that `\` introduces an *escape code* that stands for the encoded character in the string.
6. A stand-alone `\` followed by an *escape code* stands for that character: escape codes include the usual C and Java escapes: `\n`, `\r`, `\a`, `\f`, `\t`, octal escapes like `\177`, special character escapes like `\\`, `\'`, `\"`, and Unicode hexadecimal escapes like `\u27e9`.
7. A *character class* is given in `[]`, with these rules:
   (a) if the first character is `^` then the character class is negated;
   (b) if the first (after `^`) character is `]` then that character is (not) permitted;
   (c) if a `\` followed by an *escape code* is encountered then it stands for the encoded character;
   (d) if two characters are connected with a `-` (dash) then all characters in the indicated *range* (inclusive) are permitted (or excluded).

   Note that a character class cannot be empty.
8. The `.` (period) character stands for the character class `[^\n]`.
9. A nested regular expression can be given in `()`.
10. An entire other token `T` can be included (by literal substitution, so recursion is not allowed) by writing ⟨`T`⟩ (the angle brackets are unicode characters U+27E8 and U+27E9). As a special convenience, tokens declared with `token fragment` can *only* be used this way.

11. The special declaration `space` defines what constitutes white space for the generated grammar. (Note that this does not influence what is considered space in the specification itself.) A spacing declaration permits the special alternative `nested` declaration for nested comments, the following defines usual C/Java style spacing with comments, for example:

    ```
    space [ \t\f\r\n] | nested "/*" "*/" | "//" .* ;
    ```

Notice that spacing is not significant in regular expressions, except (1) in character classes, (2) in literal strings, (3) if escaped (as in `\ `).

A.3 Manual (syntactic sorts)**.** Formally, HACS uses the following notations for specifying the syntax to use for terms.

1. HACS *production names* are capitalized words, so we can for example use `Exp` for the production of expressions. The name of a production also serves as the name of its *sort*, *i.e.*, the semantic category that is used internally for abstract syntax trees with that root production. If particular instances of a sort need to be referenced later they can be *disambiguated* with a `#`i suffix, *e.g.*, `Exp#2`, where i is an optional number or other simple word.
2. A sort is declared by one or more `sort` declarations of the name optionally followed by a number of *abstract syntax production* alternatives, each starting with a `|`. A sort declaration sets the *current sort* for subsequent declarations and in particular any stand-alone production alternatives. All sort declarations for a sort are cumulative.
3. Double square brackets ⟦...⟧ (unicode U+27E6 and U+27E7) are used for *concrete syntax* but can contain nested angle brackets ⟨...⟩ (unicode U+27E8 and U+27E9) with *production references* like ⟨`Exp`⟩ for an expression (as well as several other things that we will come to later). We for example write ⟦⟨`Exp`⟩+⟨`Exp`⟩⟧ to describe the form where two expressions are separated by a + sign.
4. Concrete syntax specification can include ¶ characters to indicate where *newlines* should be inserted in the printed output. (The system can also control indentation but that is not enabled yet.)
5. A trailing `@`p for some precedence integer p indicates that either the subexpression or the entire alternative (as appropriate) should be considered to have the indicated precedence, with higher numbers indicating higher precedence, *i.e.*, tighter association. (For details on the limitations of how the precedence and left recursion mechanisms are implemented, see Appendix A.12.)
6. `sugar` ⟦...⟧→... alternatives specify equivalent forms for existing syntax: anything matching the left alternative will be interpreted the same as the right one (which must have been previously defined); references must be disambiguated.
7. A simple sort which contains only a reference to a token, where furthermore the token is defined such that it can end with `_`n (an underscore followed by a count), then the sort can be qualified as a `symbol` sort, which can be used for variables and binders.

A.4 Manual (raw terms, schemes, and rules)**.** "Raw" declarations consist of the following elements:

1. A *constructor* is a capitalized word (similar to a sort name but in a separate name space).
2. A *variable* is a lower case word (subject to scoping, described below).
3. A sort can be given a *semantic production* as a `|` (bar) followed by a *form*, which consists of a constructor name, optionally followed by a `()`ed `,`-separated list of *scope forms*, which each consist of a *sort* optionally preceded by a *binder form*, which is a list of sorts followed by a `.` (dot). Thus in the most general case, a semantic production has the form

   $$\texttt{|}\ C\ \texttt{(}\ S_{11}\cdots S_{1n_1}\ \texttt{.}\ S_1\ \texttt{,}\ \ldots\ \texttt{,}\ S_{m1}\cdots S_{mn_m}\ \texttt{.}\ S_m\texttt{)}$$

   with C a constructor name and all S_i and S_{ij} sort names. The S_i declares the *argument sort* for the ith argument of the construction term, and the S_{ij} is the *binder sort* of the jth binder for the ith argument; m is the *arity* of the construction and n_i the *rank* of the ith argument.
4. A semantic production can be qualified as a `scheme`, which marks the declared construction as a candidate for rewrite rules (defined below).
5. A *raw term* is either a *construction*, a *variable use*, or a *meta-application*, as follows

   (a) A *construction* term is a constructor name followed by an optional `()`ed `,`-separated list of *scope arguments*, which each consist of a term optionally preceded by a *binder list*, which is a list of variables followed by a `.` (dot). So in the most general case, a term looks like this:

   $$C\ \texttt{(}\ x_{11}\cdots x_{1n_1}\ \texttt{.}\ t_1\ \texttt{,}\ \ldots\texttt{,}\ x_{m1}\cdots x_{mn_m}\ \texttt{.}\ t_m\texttt{)}$$

The "C-construction" is said to have the *subterms* $t_1, \ldots, t_m$, and the arity m and ranks $n_1 \ldots n_m$ must correspond to a semantic production. If present, the binder prefix of each introduces the specified variables *only* for the appropriate subterm modulo usual renaming, *i.e.*, writing `A(x y.x, x y.y)` and `A(a b.a, a b.b)` and even `A(s t.s, t s.s)` all denote the same term following the conventions of α*-equivalence*. In a scope argument $x.t$ we say that occurrences of x in t are *bound* by the binder.

(b) A *variable use* term is a variable, subject to the usual lexical scoping rules.

(c) A *meta-application* term is a *meta-variable*, consisting of a `#` (hash) followed by a number or word and optionally by a meta-argument list of `,`-separated terms enclosed in `[]`. Examples include `#t1` (with no arguments), `#[a,b,c]`, and `#1[OK,#]`.

6. A term can have a *sort prefix*. So the term `Type Unif(Type #t1, Type Float)` is the same as `Unif(#t1,Float)` provided `Unif` was declared with the raw production `|Unif(Type,Type)`.
7. A *rewrite rule* is a pair of terms separatede by $\rightarrow$ (arrow, U+2192), with a few additional constraints: in the rule $p \rightarrow t$, p must be a *pattern*, which means it must be a construction term that has been declared as a `scheme` (syntactic or raw) and with the restriction that all contained arguments to meta-applications must be bound variables, and all meta-applications in t must have meta-variables that also occur in p with the same number of meta-arguments.

   Rule declarations must either occur with the appropriate current sort or have a pattern with a sort prefix.
8. One rule per scheme can be prefixed with the qualifier `default`. If so then the pattern can have no structure: all subterms of the pattern scheme construction must be plain meta-applications. Such a default rule is applied *after* it has been ensured that all other rules fail for the scheme.
9. Finally, a rule can be prefixed with the word `rule` for clarity, which can be followed by a raw term and a `:` (TODO: document possible name/option choices).

Rules are used for *rewriting*, a definition of which is beyond the scope of this document; please refer to the literature on higher order rewriting for details [2].

A.5 Manual (attributes and synthesis rules)**.**

1. Attributes are declared by `attribute` declarations followed by an *attribute form* of one of the following shapes:

   (a) ↑`Name(ValueSort)` defines that the synthesized attribute `Name` has `ValueSort` values;

   (b) ↓`Name(ValueSort)` similarly for a simple inherited attribute;

   (c) ↓`Name{SymbolSort:ValueSort}` defines the inherited symbol table attribute `Name` which for each constant or variable of `SymbolSort` has a distinct `ValueSort` value.
2. One can add a simple *synthesized attributes* after a raw data term as ↑*name*`(`*value*`)`, where the *name* is an attribute name and the *value* can be any term.
3. Simple *inherited attributes* are added similarly after a raw scheme term as ↓*name*`(`*value*`)`.
4. An *inherited symbol table attribute extension* is added to a raw scheme term as ↓*name*`{`*symbol*`:`*value*`}`, where the *symbol* is either a variable or a constant (of the appropriate sort).
5. A *synthesized attribute reference* has the simple form ↑*name*`;` and declares that the current sort synthesizes *name* attributes.
6. A scheme declaration can include *inherited attribute references* of the form ↓*name*, which declares that the scheme inherits the *name* attributes.
7. A *synthesis rule* is a special rule of the form $t \uparrow name(t')$, where the term t may contain subterms with attribute constraints. The rule specifies how terms of the current sort and shape t synthesize *name* attributes.

Inherited attributes are managed with regular rules (for schemes) with inherited attribute constraints and extensions.

A.6 Manual (parsed terms)**.** Man. A.4 documented terms without concrete syntax. The full term model combines this with *parsed terms*.

1. Double square brackets ⟦. . .⟧ (unicode U+27E6 and U+27E7) can be used for *concrete terms*, provided the *sort* is clear, either

   (a) by immediately prefixing with the sort (as in `Exp⟦1+2⟧`), or

   (b) by using as the argument of a defined constructor (as `IsType(⟦mytype⟧)`), or

   (c) by usng as an attribute value, or

(d) by using as a top level rule pattern or replacement term with a defined current sort.

2. Concrete terms can contain nested raw terms in $\langle \ldots \rangle$ (unicode U+27E8 and U+27E9). Such nested raw terms *must* have an explicit sort prefix.
3. The special term `error`$[\![\ldots]\!]$ will print the error message embedded in $[\![\ldots]\!]$, where one is permitted to embed `symbol`-declared variables in $\langle \ldots \rangle$.

A.7 Manual (attributes and synthesis rules)**.**

1. Attributes are declared by `attribute` declarations followed by an *attribute form* of one of the following shapes:
   (a) ↑`Name(ValueSort)` defines that the synthesized attribute `Name` has `ValueSort` values;
   (b) ↓`Name(ValueSort)` similarly for a simple inherited attribute;
   (c) ↓`Name{SymbolSort:ValueSort}` defines the inherited symbol table attribute `Name` which for each constant or variable of `SymbolSort` has a distinct `ValueSort` value.
2. One can add a simple *synthesized attributes* after a raw data term as ↑*name*`(`*value*`)`, where the *name* is an attribute name and the *value* can be any term.
3. Simple *inherited attributes* are added similarly after a raw scheme term as ↓*name*`(`*value*`)`.
4. An *inherited symbol table attribute extension* is added to a raw scheme term as ↓*name*`{`*symbol*`:`*value*`}`, where the *symbol* is either a variable or a constant (of the appropriate sort).
5. A *synthesized attribute reference* has the simple form ↑*name*`;` and declares that the current sort synthesizes *name* attributes.
6. A scheme declaration can include *inherited attribute references* of the form ↓*name*, which declares that the scheme inherits the *name* attributes.
7. A *synthesis rule* is a special rule of the form $t \uparrow name(t')$, where the term t may contain subterms with attribute constraints. The rule specifies how terms of the current sort and shape t synthesize *name* attributes.

Inherited attributes are managed with regular rules (for schemes) with inherited attribute constraints and extensions.

A.8 Manual (special replacement terms)**.** The special term `error`$[\![\ldots]\!]$ will print the error message embedded in $[\![\ldots]\!]$, where one is permitted to embed `symbol`-declared variables in $\langle \ldots \rangle$.

A.9 Manual (building and running)**.** To use HACS you need a copy of the *hacs* directory somewhere. If you are working on a HACS specification, say *mycompiler.hx*, then your working directory should have a *Makefile* containing (at least) the following:

```
1 # Makefile for mycompiler.

2 # HACS configuration.
3 HACS = $(abspath hacs)
4 include $(HACS)/Makefile-hx

5 # Dependencies.
6 mycompiler.run: mycompiler.hx
```

where you have replaced `hacs` with the path to the HACS directory on your system (as written the system expects to find *hacs* as a local subdirectory of your working directory). With this setup, and a suitable *mycompiler.hx*, you have the following options:

1. `make mycompiler.run` will generate the script *mycompiler.run*, which implements the compiler you specify in *mycompiler.hx*. Generation creates a number of support files, specifically
   (a) *build* subdirectory has runtime resources needed by the script.
   (b) *src* subdirectory contains auxiliary Java resources used by the build process.
   (c) *mycompiler.crs-installed* and *mycompiler.pg-installed* record when the generated parser and rewrite system were installed.

   Note that the first time, some utility programs are compiled from C to ensure that HACS is fully enabled on your system.
2. `make clean` will remove all temporaries not needed for running the compiler script.

3. `make realclean` will remove all traces of the generated compiler.
4. `make distclean` will remove all traces of the generated compiler as well as all generated HACS tooling files. Do not do this unless you really mean it: recovering a useable HACS system requires getting a fresh copy of *hacs.zip* or half an hour of CPU time on nontrivial hardware.

The generated script refers absolutely to files under the *hacs* directory, so the generated *mycompiler.run* script itself can be moved but the *hacs* directory cannot.

The script accepts a number of options:

1. `--sort=`*Sort* sets the expected sort (and thus parser productions) for the input to *Sort*. The input is read, normalized, and printed.
2. `--action=`*Constructor* sets the computation for the compiler to *Constructor*, which must be a unary raw scheme; the argument sort of *Constructor* defines the parser productions to use. The input is read, wrapped in the action, normalized, and printed.
3. `--term=`*term* use the *term* as the input.
4. `--input=`*file* reads the input from *file*.
5. `--output=`*file* sends the input to *file* (the default is the standard output).
6. `--verbose=`n sets the verbosity of the underlying CRSX rewrite engine to n. The default is 0 (quiet) but 1–3 are useful (above 3 you get a lot of low level diagnostic output).
7. `--parse-verbose` activates verbose output from JavaCC of the parsing.

You must provide one of `--sort` or `--action`, and one of `--term` and `--input`.

HACS will eventually contain a convention for defining "main" sorts and schemes such that defaults can be provided for the configuration options.

A.10 Notation (grammar). This is a *work in progress* "HACS in HACS" self-grammar.

```
// HACS SYNTAX
module "net.sf.crsx.hacs.Hacs" {


// MODULES.

sort Module
| ⟦ module ⟨ModuleName⟩ { ⟨Declaration*⟩ } ⟧
;
sort ModuleName | ⟦⟨ProperName⟩⟧ | ⟦⟨String⟩⟧ ;

sort Declaration
| ⟦⟨Module⟩⟧
| ⟦ import ⟨ModuleName⟩ ⟨SortNameList?⟩ ; ⟧
| ⟦⟨LexicalDeclaration⟩⟧
| ⟦⟨SyntaxDeclaration⟩⟧
| ⟦⟨RuleDeclaration⟩⟧
;


// LEXICAL ANALYSIS.

sort LexicalDeclaration
| ⟦ space ⟨RE⟩ ; ⟧
| ⟦ token ⟨Name⟩ | ⟨RE⟩ ; ⟧
| ⟦ fragment ⟨Name⟩ | ⟨RE⟩ ; ⟧
| sugar ⟦token fragment ⟨Name#1⟩ | ⟨RE#2⟩ ; ⟧→⟦fragment ⟨Name#1⟩ | ⟨RE#2⟩ ; ⟧
;
```

```
sort RE
| ⟦ ⟨RE@1⟩ ⟨Repeat⟩ ⟧@1
| ⟦ ⟨LAng⟩ ⟨Name⟩ ⟨RAng⟩ ⟧@1
| ⟦ ⟨String⟩ ⟧@1
| ⟦ ⟨CC⟩ ⟧@1
| ⟦ ⟨Char⟩ ⟧@1
| ⟦ ( ⟨RE⟩ ) ⟧@1
| ⟦ ⟨RE@1+⟩ ⟧
;

sort Repeat | ⟦+⟧ | ⟦?⟧ | ⟦*⟧ | ⟦⟧;

token CC | \[ \^? (\− | \] ⟨CCRange⟩)? ( ([^\]\\−] | ⟨Escape⟩) ⟨CCRange⟩ )* \−? \] ;
fragment CCRange | \− ([^\]\\] | ⟨Escape⟩) ;

token Char | [A−Za−z0−9] | ⟨Escape⟩ ;

token ProperName | ⟨UpperEtc⟩ ⟨AlphaNum⟩* ;
token LowerName | ⟨Lower⟩ ⟨AlphaNum⟩* ;
token AnyName | (⟨Lower⟩ | ⟨UpperEtc⟩) ⟨AlphaNum⟩* ;
fragment UpperEtc | [A−Z$_] ;
fragment Lower | [a−z] ;
fragment AlphaNum | [A−Za−z0−9] ;


// SYNTAX ANALYSIS.

sort SyntaxDeclaration
| ⟦ sort ⟨Name⟩ ⟨SortAlternatives⟩ ; ⟧
| ⟦ attribute ⟨UpDown⟩ ⟨AttributeName⟩ ⟨AttributeSort⟩ ; ⟧



sort AttributeName | ⟦⟨ProperName⟩⟧ | ⟦⟨VariableName⟩⟧ ;



// UNICODE CHARACTERS.

token Negate  | "¬" ;      // \u00AC
token Paragraph | "¶" ;     // \u00B6
token Up      | "↑" ;      // \u2191
token Right   | "→" ;      // \u2192
token Down    | "↓" ;      // \u2193
token LWhite  | "⟦" ;      // \u27E6
token RWhite  | "⟧" ;      // \u27E7
token LAngle  | "⟨" ;      // \u27E8
token RAngle  | "⟩" ;      // \u27E9

}
```

A.11 Notation (used unicode characters)**.**

Glyph	*Code Point*	*Character*
¬	U+00AC	logical negation sign
¶	U+00B6	paragraph sign
↑	U+2191	upwards arrow
→	U+2192	rightwards arrow
↓	U+2193	downwards arrow
⟦	U+27E6	mathematical left white square bracket
⟧	U+27E7	mathematical right white square bracket
⟨	U+27E8	mathematical left angle bracket
⟩	U+27E9	mathematical right angle bracket

If you are using a Linux computer (or any where the display is controlled by the X Window System), then you can create a file called *.XCompose*, which has your special characters. A minimal one looks something like this (adjust for your locale, *etc.*):

```
include "/usr/share/X11/locale/en_US.UTF-8/Compose"
<Multi_key> <P> <P>                       : "¶" U00B6 # PARAGRAPH SIGN
<Multi_key> <u> <greater>                 : "↑" U2191 # UPWARDS ARROW
<Multi_key> <minus> <greater>             : "→" U2192 # RIGHTWARDS ARROW
<Multi_key> <d> <greater>                 : "↓" U2193 # DOWNWARDS ARROW
<Multi_key> <bracketleft> <bracketleft>   : "⟦" U27E6 # MATHEMATICAL LEFT WHITE SQUARE BRACKET
<Multi_key> <bracketright> <bracketright> : "⟧" U27E7 # MATHEMATICAL RIGHT WHITE SQUARE BRACKET
<Multi_key> <less> <period>               : "⟨" U27E8 # MATHEMATICAL LEFT ANGLE BRACKET
<Multi_key> <greater> <period>            : "⟩" U27E9 # MATHEMATICAL RIGHT ANGLE BRACKET
```

(We have included a slightly more extensive one in the *gentle* directory.) You should then map the *MultiKey* key to some key on your keyboard in the keyboard options (this is done in the keyboard settings), and then you are ready to enter the fancy characters as the indicated three-key combinations. In some programs (like Eclipse) you need to select the "Input Method" to be the "X Input Method" for this to work.

A.12 Manual (limitations)**.**

- At most one `nested` declaration per `token`.
- Precedence can only be used on self references, *i.e.*, ⟨`E@2`⟩ can only occur inside productions for the sort `E`.
- It is not possible to use binders and left recursion in the same production with the same precedence.
- Only *direct* left recursion is currently supported, *i.e.*, the left recursion should be within a single production.
- Productions can share a prefix but only within productions for the same sort, and the prefix has to be literally identical unit by unit, *i.e.*,

  ```
  sort S | ⟦ ⟨A⟩ then ⟨B⟩ then C ⟧
         | ⟦ ⟨A⟩ then ⟨B⟩ or else D ⟧ ;
  ```

  is fine but

  ```
  sort S | ⟦ ⟨A⟩ then ⟨B⟩ then C ⟧
         | ⟦ ⟨A⟩ ⟨ThenB⟩ or else D ⟧ ;
  sort ThenB | ⟦ then ⟨B⟩ ⟧;
  ```

  is not.
- It is not possible to left-factor a binder (so multiple binding constructs cannot have the same binder prefix).
- Binders must occur to the left of all their occurrences.
- Repeated production references cannot be used in sorts that also use precedence, *i.e.*, you cannot mix ⟨`E*`⟩ and ⟨`E@2`⟩ inside the productions for the sort `E`.

- Variables embedded in `error`⟦...⟧ instructions must start with a lower case letter.
- When using the `symbol` qualifier on a reference to a token then the token *must* allow ending in $\_n$ for n any natural number.
- When using the same name for a symbol inside of ⟦...⟧ and the corresponding raw variable outside of the ⟦⟧, then the common symbol and variable name must be a plain word starting with a lower case letter.
- Special terms like `error`⟦...⟧ cannot be used as raw subterms.
- The `default` rule qualifier is rather fragile and does not yet always work.

B Bonus Examples

Some additional examples. *Note: still need to be worked through.*

B.1 Example (*crsx/samples/gentle/bool.hx*).

```
module "net.sf.crsx.samples.gentle.Bool" {

// Boolean sort.
sort B | ⟦ t ⟧@4 | ⟦ f ⟧@4                    // constants
       | sugar ⟦( ⟨B#⟩ ) ⟧@4 →⟦⟨B#⟩⟧ // parenthesis
       ;

// Disjunction.
| scheme ⟦⟨B@2⟩ ∨⟨B@1⟩ ⟧@1 ;
⟦t∨⟨B#⟩⟧ →⟦t⟧ ;
⟦f∨⟨B#⟩⟧ →B# ;

// Conjunction.
| scheme ⟦⟨B@3⟩ ∧⟨B@2⟩ ⟧@2 ;
⟦t∧⟨B#⟩⟧ →B# ;
⟦f∧⟨B#⟩⟧ →⟦f⟧ ;

// Negation.
| scheme ⟦¬⟨B@3⟩ ⟧@3 ;
⟦¬t⟧ → ⟦f⟧ ;
⟦¬f⟧ → ⟦t⟧ ;

}
```

B.2 Example (*crsx/samples/gentle/scopes.hx*).

```
module "net.sf.crsx.samples.gentle.Scopes" {
space [ \t\n] ;

token IDENTIFIER | [a−z] [A−Za−z_0−9]* ;

sort Id    | symbol ⟦⟨IDENTIFIER⟩⟧ ;

sort Type | ⟦ A ⟧ | ⟦ B ⟧ ;

sort Stat | ⟦ def ⟨Type⟩ ⟨Id⟩; ⟨Stat⟩ ⟧
          | ⟦ use ⟨Type⟩ ⟨Id⟩; ⟨Stat⟩ ⟧
          | ⟦ ⟧ ;
```

```
// MAIN: Check that every 'use' uses same type as nearest in−scope 'def'.

sort Stat | scheme Check(Stat) ;

Check(#s) →Check2(#s) ;

// 1. SYNTHESIZE ATTRIBUTE WITH ALL TOP LEVEL DEFINITIONS IN A PLAIN LIST.

attribute ↑ds(Defs) ;
sort Defs | ⟦ ⟨Type⟩ ⟨Id⟩ ; ⟨Defs⟩ ⟧| [[]] ;

sort Stat | ↑ds ;
⟦ def ⟨Type#t⟩ id; ⟨Stat#s↑ds(#ds)⟩ ⟧↑ds(⟦ ⟨Type#t⟩ id; ⟨Defs#ds⟩ ⟧) ;
⟦ use ⟨Type#t⟩ id; ⟨Stat#s↑ds(#ds)⟩ ⟧↑ds(#ds) ;
[[]] ↑ds([[]]) ;

// 2. CONVERT DEFINITIONS RECURSIVELY TO ENVIRONMENT.

attribute ↓e{Id:Type} ;

sort Stat | scheme Check2(Stat)
          | scheme Check2_(Defs, Stat) ↓e ;

Check2(#s↑ds(#ds)) →Check2_(#ds, #s) ;

Check2_(⟦ ⟨Type#t⟩ id; ⟨Defs#ds⟩ ⟧, #s) →Check2_(#ds, #s) ↓e{id : #t} ;

Check2_([[]], #s) →Check3(#s) ;

// 3. CHECK USING ENVIRONMENT.

sort Stat | scheme Check3(Stat) ↓e ;

Check3(⟦ use ⟨Type#t⟩ id; ⟨Stat#s⟩ ⟧)
  ↓e{id:#t}
  // − we found a mapping from id to same type \#t
  →⟦ use ⟨Type#t⟩ id; ⟨Stat Check3(#s)⟩ ⟧;

Check3(⟦ use ⟨Type#t⟩ id; ⟨Stat#s⟩ ⟧)
  ↓e{¬id}
  // − we did not find a mapping from id
  →error⟦ Undefined ⟨id⟩ ⟧;

default Check3(⟦ use ⟨Type#t⟩ id; ⟨Stat#s⟩ ⟧)
  ↓e{id:#t2}
  // − we found a mapping from id to some type \#t so it has the wrong type
  →error⟦ Bad type for ⟨id⟩ ⟧;

Check3(⟦ def ⟨Type#t⟩ id; ⟨Stat#s⟩ ⟧) →⟦def ⟨Type#t⟩ id; ⟨Stat Check3(#s)⟩ ⟧;

Check3([[]]) →[[]];

}
```

B.3 Example (*crsx/samples/gentle/deriv.hx*).

```
module "net.sf.crsx.samples.gentle.Deriv" {

import "bool.hx"(B);

//// SYNTAX.

// Arithmetic and basic functions.

sort Exp | ⟦⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1
         | ⟦ ⟨Exp@1⟩ − ⟨Exp@2⟩ ⟧@1
         | ⟦ ⟨Exp@2⟩ * ⟨Exp@3⟩ ⟧@2
         | ⟦ ⟨Exp@2⟩ / ⟨Exp@3⟩ ⟧@2
         | ⟦ ⟨Fun⟩ ⟨Exp@4⟩ ⟧@3
         | ⟦ ⟨Int⟩ ⟧@4

         | sugar ⟦( ⟨Exp#⟩ ) ⟧@4 →Exp#
         | sugar ⟦+ ⟨Exp#1@2⟩ ⟧@1 →⟦0 + ⟨Exp#1⟩ ⟧
         | sugar ⟦− ⟨Exp#1@2⟩ ⟧@1 →⟦0 − ⟨Exp#1⟩ ⟧
         ;

sort Fun | ⟦sin⟧@2 | ⟦cos⟧@2 | ⟦ln⟧@2 | ⟦exp⟧@2 ;

token Int | [0−9]+ ;

// Functionals.

sort Exp | symbol ⟦⟨Var⟩ ⟧@4 ;
token Var | [a−z] [A−Za−z0−9]* ;
sort Fun | ⟦[ ⟨[Var#1]⟩ ↦⟨Exp[Var#1:Exp]⟩ ] ⟧@2 ;

//// SCHEMES.

sort Fun | scheme ⟦d⟨Fun@1⟩⟧@1 ;

⟦d sin⟧ →⟦cos⟧ ;
⟦d cos⟧ →⟦[a ↦ − sin a]⟧ ;
⟦d ln⟧  →⟦[z ↦ 1/z]⟧ ;
⟦d exp⟧ →⟦exp⟧ ;

⟦d[x ↦⟨Exp#1[x]⟩]⟧ →⟦[y ↦D(y)[z↦⟨Exp#1[z]⟩]]⟧ ;

sort Exp | scheme ⟦D ⟨Exp⟩ [⟨[Var#1]⟩↦⟨Exp[Var#1:Exp]⟩] ⟧@3 ;

⟦ D⟨Exp#1⟩[x↦⟨Int#2⟩] ⟧→⟦0⟧ ;

⟦ D⟨Exp#1⟩[x↦x] ⟧→⟦1⟧ ;
⟦ D⟨Exp#1⟩[x↦y] ⟧→⟦0⟧ ;

⟦ D⟨Exp#0⟩[x↦⟨Exp#1[x]⟩+⟨Exp#2[x]⟩] ⟧→⟦D⟨Exp#0⟩[y↦⟨Exp#1[y]⟩] + D⟨Exp#0⟩[z↦⟨Exp#2[z]⟩]⟧ ;
⟦ D⟨Exp#0⟩[x↦⟨Exp#1[x]⟩−⟨Exp#2[x]⟩] ⟧→⟦D⟨Exp#0⟩[y↦⟨Exp#1[y]⟩] − D⟨Exp#0⟩[z↦⟨Exp#2[z]⟩]⟧ ;

⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩ * ⟨Exp#2[x]⟩] ⟧
```

```
 →⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩] * ⟨Exp#2[#]⟩ + ⟨Exp#1[#]⟩ * D⟨Exp#⟩[x↦⟨Exp#2[x]⟩] ⟧;

⟦ D⟨Exp#⟩[x↦⟨Exp#1[x]⟩ / ⟨Exp#2[x]⟩] ⟧
 →⟦ ( D⟨Exp#⟩[x↦⟨Exp#1[x]⟩] * ⟨Exp#2[#]⟩ − ⟨Exp#1[#]⟩ * D⟨Exp#⟩[x↦⟨Exp#2[x]⟩] )
      / (⟨Exp#2[#]⟩ * ⟨Exp#2[#]⟩) ⟧;

⟦ D⟨Exp#⟩[x↦⟨Fun#f⟩⟨Exp#2[x]⟩] ⟧→⟦d⟨Fun#f⟩ ⟨Exp#2[#]⟩ * D⟨Exp#⟩[x↦⟨Exp#2[x]⟩] ⟧;

//// STATIC REDUCTIONS.

sort Exp;

⟦ 0 + ⟨Exp#⟩ ⟧→# ;
⟦ ⟨Exp#⟩ + 0 ⟧→# ;
⟦ ⟨Exp#⟩ − 0 ⟧→# ;
⟦ 1 * ⟨Exp#⟩ ⟧→# ;
⟦ ⟨Exp#⟩ * 1 ⟧→# ;

⟦ 0 * ⟨Exp#⟩ ⟧→⟦0⟧ ;
⟦ ⟨Exp#⟩ * 0 ⟧→⟦0⟧ ;

⟦ ⟨Exp#1⟩ * (1 / ⟨Exp#2⟩) ⟧→⟦⟨Exp#1⟩ / ⟨Exp#2⟩ ⟧;

⟦ [x ↦ ⟨Exp#[x]⟩] ⟨Exp#2⟩ ⟧→#[#2] ;

//// MISCELLANEOUS

space [ \t\n\r] | "//" [^\n\r]* | nested "/*" "*/" ;

}
```

C Common Errors

C.1 Error (HACS syntax).

```
Exception in thread "main" java.lang.RuntimeException: net.sf.crsx.CRSException:
 Encountered " "." ". "" at line 35, column 6.
Was expecting one of:
    <MT_Repeat> ...
    "%Repeat" ...
    <MT_Attributes> ...
```

Indicates a simple syntax errors in the *.hx* file.

C.2 Error (user syntax).

```
Exception in thread "main" java.lang.RuntimeException:
  net.sf.crsx.CRSException: net.sf.crsx.parser.ParseException:
mycompiler.crs: Parse error in embedded myDecSome term at line 867, column 42:
 ⟦ $TA_Let2b ⟨Dec (#d)⟩{ ⟨DecSome (#ds)⟩} ⟧ at line 867, column 42
 Encountered " "\u27e9" "\u27e8Dec (#d)\u27e9 "" at line 867, column 53
...
```

This indicates a concrete syntax error in some parsed syntax—inside ⟦...⟧—in the *.hx* file. The offending fragment is given in double angles in the message. Check that it is correctly entered in the HACS specification in a way that corresponds to a syntax production. Note that the line/column numbers refer to the generated *build/...Rules.crs* file, which us not immediately helpful (this is a known bug). In error messages a sort is typically referenced as a lower case prefix followed by the sort name—here `myDecSome` indicates that the problem is with parsing the `DecSome` sort of the `My` parser.

C.3 Error (JavaCC noise)**.**

```
Java Compiler Compiler Version ??.??_?? (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file net/sf/crsx/samples/gentle/FirstParser.jj . . .
Warning: Line 769, Column 51: Non-ASCII characters used in regular expression.
Please make sure you use the correct Reader when you create the parser,
 one that can handle your character set.
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated with 0 errors and 1 warnings.
Note: net/sf/crsx/samples/gentle/FirstParser.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

These are "normal" messages from JavaCC.

C.4 Error (missing library)**.**

```
gcc -std=c99 -g    -c -o crsx_scan.o crsx_scan.c
crsx.c:11:30: fatal error: unicode/umachine.h: No such file or directory
```

The HACS tools only use one library in C: ICU. You should get the *libicu-dev* package (or similar) for your system.

C.5 Error (meta-variable mistake)**.**

```
Error in rule Tiger-Ty2222222222111_9148-1: contractum uses undefined meta-variable (#es)
Errors prevent normalization.
make: *** [pr3.crs-installed] Error 1
```

A rule uses the metavariable `#es` in the replacement without defining it in the corresponding pattern.

C.6 Error.

```
/home/krisrose/Desktop/teaching/.../hacs/cookmain PG pr3.hxt > pr3.pg
cookmain: crsx.c:528: bufferEnd: Assertion
   '(((childTerm)->descriptor == ((void *)0)) ? 0 :
        (childTerm)->descriptor->arity) == bufferTop(buffer)->index' failed.
/bin/sh: line 1: 14278 Aborted
  (core dumped) /home/krisrose/Desktop/teaching/.../hacs/cookmain PG pr3.hxt > pr3.pg
```

This indicates an arity error: a raw term in the *.hx* file does not have the right number of arguments.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, , and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, Inc, 2006.

[2] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Computer Science*, 121:279–308, 1993. doi:10.1016/0304-3975(93)90091-7.

[3] Sreeni Viswanadha, Sriram Sankar, et al. *Java Compiler Compiler (JavaCC) - The Java Parser Generator*, 6.0 edition, 2013. URL: http://javacc.java.net/.