Compiler Generation Using HACS*

Kristoffer H. Rose
Two Sigma Investments/New York University

November 18, 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, and walks through the code of a simple example with each of the main stages of a compiler in HACS: lexical analysis, syntax analysis, semantic analysis, intermediate code generation with optimization, and code generation.

1 Introduction

HACS abbreviates *Higher-order Attribute Contraction Schemes*, which is a formal system for symbolic rewriting extended with programming idioms commonly used when coding compilers. HACS is developed as a front-end to the CRSX higher-order rewriting engine [**?**].

A compiler written in HACS consists of a single *specification file* with a series of formal sections, each corresponding to a stage of the compiler. Each section is written in a formal style suitable for that stage of the compiler. Specifically, HACS supports the following notations:

Regular Expressions. Used to describe how an input text is partitioned into *tokens*. The regular expressions of HACS follows common conventions [**?**]. Section 3 gives details of this notation.

Context Free Grammars. HACS uses a form of BNF [**?**] with common extensions to describe *context free grammars*. HACS includes simple mechanisms for automatic resolution of operator precedence and productions using immediate recursion such that the transformation from token stream to abstract syntax can be formalized. Details in Section 4.

Recursive Translation Schemes. Simple translations in general, and code generation in particular, are traditionally achieved by *recursive translation* from one abstract form to another. HACS includes special notations for defining such translations, as well as a mechanism for defining auxiliary so-called "semantic sorts," detailed in Section 5.

Attribute Grammars. Analyses can be described with attribute grammars in the style of *Syntax-Directed Definitions* [**?**], originally introduced as *attribute grammars* [**?**], which describe how

*This version describes HACS β version 0.9.1 released for use at NYU.

properties propagate through the abstract syntax tree. Section 6 details how the basic propagation rules work for synthesized attributes, Section 7 explains how inherited atributes are described, and

In the remainder of this document we introduce the most important features of the HACS language by explaining the relevant parts of the included *First.hx* example, inspired by [**?**, Figure 1.7], as well as several other minor examples. We first show in Section 2 how to install HACS and run the example, before we go on to how to write specifications. We explain lexical analysis in Section 3, syntax analysis in Section 4, basic semantic sorts and recursive translation schemes in Section 5, bottom-up semantic analysis in Section 6. and general syntax-directed definitions in Section 7, Appendix A has a reference manual, Appendix B explains some cryptic error messages, and Appendix C lists some current limitations.

2 Getting Started

In this section we walk through the steps for getting a functional HACS installation on your computer.[1]

2.1 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); and a standard *nix development setup including GNU Make and a C99 compiler. In addition, the setup process needs internet access to retrieve the CRSX base system [**?**], JavaCC parser generator [**?**], and *icu* Unicode C libraries [**?**].

2.2 Commands (install HACS)**.** Retrieve the *hacs-0.9.1.zip* archive, extract it to a new directory, and install it, for example with the following commands:[2]

```
energon1[~]$ wget http://crsx.org/hacs-0.9.1.zip
energon1[~]$ unzip hacs-0.9.1.zip
energon1[~]$ cd hacs
energon1[hacs]$ make
energon1[hacs]$ make install install-support
```

These commands will need Internet access, using the wget command to retrieve support libraries.[3] The above command will install HACS in a *.hacs* subdirectory of your home directory. You can change this with the option `prefix=...` to (all uses of) *make*; if so then make sure to replace occurrences of `$HOME/.hacs` everywhere below with your chosen directory.

The main *make* command will take some time (of the order of 10 minutes) but should end without error.

The following command makes the main *hacs* command available for use:

```
energon1[hacs]$ alias hacs=$HOME/.hacs/bin/hacs
```

(It may be worth including this command in your setup, or including the `$HOME/.hacs/bin` directory in your `$PATH`.)

Please check that your new installation works with these commands:

[1] HACS is still a β release so please report any problems with this procedure to *hacs-bugs@crsx.org*.

[2] User input is blue.

[3] Specifically, HACS needs the CRSX system, JavaCC parser generator, and ICU4C Unicode library. The setup is documented in *src/Env.mk*.

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

Table 1: Unicode special characters used by HACS.

```
energon1[hacs]$ cd
energon1[~]$ mkdir myfirst
energon1[~]$ cd myfirst
energon1[~]$ cp $HOME/.hacs/share/doc/hacs/examples/First.hx .
energon1[~]$ $HOME/.hacs/bin/hacs First.hx
energon1[~]$ ./First.run --scheme=Compile \
           --term="{initial := 1; rate := 1.0; position := initial + rate * 60;}"
  LDF T_2,  #1     STF initial, T_2    LDF T_2,  #1.0     STF rate, T_2
  LDF T_3,  initial     LDF T_3,  rate     LDF T_4,  #60
  MULF  T_4 ,  T_3 ,  T_4     ADDF  T_2 ,  T_3 ,  T_4     STF position, T_2
```

Congratulations—you just built your first compiler![4]

2.3 Example (module wrapper)**.** The source file for the *First.hx* file used in the example above has the structure

```
/* Top comment. */
module org.crsx.hacs.samples.First
{
  // Remark.
  Lexical Analysis (Section 3)
  Syntax Analysis (Section 4)
  Semantic Analysis (Sections 6 and 7)
  Code Generator (Section 5)
  Main (Section 5)
}
```

Notice that HACS permits C/Java style comments.

2.4 Notation (special Unicode characters)**.** HACS uses a number of special symbols from the standard Unicode repertoire of characters, shown in Table 1

3 Lexical Analysis

Lexical analysis is the process of splitting the input text into tokens. HACS uses a rather standard variation of *regular expressions* for this.

[4]Please do not mind the spacing – that is how HACS prints in its present state.

3.1 Example (tokens and white space)**.** Here is a HACS fragment for setting up the concrete syntax of integers, basic floating point numbers, identifiers, and white space, for use by a simple language:

```
// White space convention.                                          1
space [ \t\n] ;                                                     2

// Basic nonterminals.                                              4
token INT      | ⟨DIGIT⟩+ ;                                         5
token FLOAT    | ⟨DIGIT⟩* "." ⟨DIGIT⟩+ ;                           6
token ID       | ⟨LOWER⟩+ ('_'? ⟨INT⟩)? ;                          7

// Special categories of letters.                                   9
token fragment DIGIT | [0−9] ;                                      10
token fragment LOWER | [a−z] ;                                      11
```

The example illustrates the following particulars of HACS lexical expressions:

- Declarations generally start with a keyword or two and are terminated by a ; (semicolon).
- token declarations in particular have the token keyword followed by a regular expression between a | (vertical bar) and a ; (semicolon). It defines the token as a *non-terminal* that can be used in syntax productions described in the next section.
- A regular expressions is a sequence of units, corresponding to the concatenation of sequences of characters that match each one. Each unit can be a *character class* such as [a−z], which matches a single character in the indicated range (or, more generally, a sequence of individual characters and ranges), a *string* such as ".", or a *reference* to a token or fragment such as ⟨Lower⟩, enclosed in the special Unicode mathematical angle brackets (see Table 1).
- A token fragment declaration means that the defined token can only be used in other token declarations, and not as grammar non-terminal in syntax productions.
- Every regular expression component can be followed by a repetition marker ?, +, or ∗, and regular expressions can be *grouped* with parentheses.
- 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.

In addition, we have followed the convention of naming proper grammar terminals with ALL-CAPS names, like INT, so they are easy to distinguish from non-terminals below.

Notice that while it is possible to make every keyword of your language into a named token in this way, this is not necessary, as keywords can be given as literals in syntax productions, covered in the next section.

3.2 Commands (lexical analysis)**.** The fragment above is part of *First.run* from Section 1, which can thus be used as a lexical analyzer. This is achieved by passing the *First.run* command two arguments: a *token sort* and a *token term*.[5] Execution proceeds by parsing the string following the syntax of the token. We can, for example, check the lexical analysis of a number:

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

```
$ ./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:
  Encountered " <T_FLOAT> "34.56 "" at line 1, column 1.
Was expecting:
    <T_INT> ...
```

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

4 Syntax Analysis

Once we have tokens, we can use HACS to program a 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 is built. Grammars are structured following the *sorts* of AST nodes, with concrete syntax details managed through annotations and "syntactic sugar" declarations.

4.1 Example. Here is another extract from our *First.hx* example, with a small syntax analysis, or grammar. Our small example source language merely has blocks, assignment statements, and a few forms of expression, like so:

```
// Main program construct.                                                    1
main sort Stat  | ⟦ ⟨Name⟩ := ⟨Exp⟩ ; ⟨Stat⟩ ⟧                               2
                | ⟦ { ⟨Stat⟩ } ⟨Stat⟩ ⟧                                      3
                | ⟦⟧ ;                                                       4

sort Exp | ⟦ ⟨Exp⟩ + ⟨Exp@1⟩ ⟧                                               6
         | ⟦ ⟨Exp@1⟩ * ⟨Exp@2⟩ ⟧@1                                           7
         | ⟦ ⟨INT⟩ ⟧@2                                                       8
         | ⟦ ⟨FLOAT⟩ ⟧@2                                                     9
         | ⟦ ⟨Name⟩ ⟧@2                                                      10
         | sugar ⟦ ( ⟨Exp#⟩ ) ⟧@2 →Exp# ;                                    11

sort Name | symbol ⟦⟨ID⟩⟧ ;                                                  13
```

The grammar structures the input as three sorts: Stat for statements, Exp for expressions, and Name for names (which we shall need later for symbol tables).

The example grammar above captures the HACS version of several standard parsing notions:

Literal syntax is indicated by the double "syntax brackets," ⟦...⟧. Text inside ⟦...⟧ consists of three things only: space, literal character "words," and references to non-terminals and predefined tokens inside (nested) ⟨...⟩. In this way, literal syntax is similar to macro notation or "quasi-quotation" of other programming languages.

Syntactic sugar is represented by the sugar part of the Exp sort declaration, which states that the parser should accept an Exp in parenthesis, identified as #, and replace it with just that same Exp, indicated by the →Exp# part. This avoids any need to think of parentheses in the generated AST.

Precedence rules are represented by the @-annotations, which assign precedence and associativity to each operator. Thus the same Exp language would be recognized by something like the following concrete HACS specification (where we have also made the sugar concrete):

```
sort Exp  | ⟦ ⟨Exp0⟩ ⟧;
sort Exp0 | ⟦ ⟨Exp0⟩ + ⟨Exp1⟩ ⟧| ⟦⟨Exp1⟩ ⟧;
sort Exp1 | ⟦ ⟨Exp1⟩ * ⟨Exp2⟩ ⟧| ⟦ ⟨Exp2⟩ ⟧;
sort Exp2 | ⟦ ⟨Int⟩ ⟧ | ⟦ ⟨Float⟩ ⟧ | ⟦ ⟨Name⟩ ⟧| ⟦ ( ⟨Exp⟩ ) ⟧ ;
```

However, this grammar generates a different result tree, where the nodes have the four different sorts used instead of all being of the single Exp sort that the precedence annotations make possible. The transformed system also illustrates how HACS deals with left recursion with @-annotations: each becomes an instance of *immediate left recursion*, which is eliminated automatically.

The precedence notation allows us to define one sort per "sort of abstract syntax tree node." This allows us to use a single kind of AST node to represent all the "levels" of expression, which helps the subsequent steps.

The notation is admittedly dense: this is intentional, as we generalize this very same notation to serve all the formalisms of the following sections. Here are the formal rules:

- 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 ⟦...⟧ ("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 references in ⟨...⟩ (angle brackets), which are special.
- References to *nonterminals* (other productions) are wrapped in ⟨...⟩ (angle brackets).
- *Precedence* is indicated with @n, where higher numbers n designate higher (tighter) precedence. After every top-level ⟦⟧ and inside every ⟨⟩ there is a precedence, which defaults to @0.
- The special sugar declaration expresses that the concrete syntax can use parentheses to raise the precedence of the enclosed expression to 2: 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 Exp# to the right of the → to indicate that the result of simplifying 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.)
- As a special case, A sort can be defined with a single symbol declaration. If so, then the actual syntax must refer to a single token, and that token must allow that a trailing _n (underscore and count) is added to any instance (this permits automatic symbol generation).

Of all these rules, the one thing that is unique to parsing is the precedence notation with @. When specifying a grammar then *every* subterm has a precedence, which determines how ambiguous terms should be parsed. So imagine that every ⟨⟩ contains a @ marker, defaulting to @0, and that every [] is terminated with a @ marker, again defaulting to @0.

Notice that HACS will do two things automatically:

1. Eliminate immediate left recursion, such as found in the example.
2. Split the productions into subproductions according to the precedence assignments.
3. Left factor the grammar, which means that productions within a sort may start with a common prefix.

However, this is *not* reflected in the generated trees: they will follow the grammar as specified, so you do not need to be aware that this conversion happens.

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

```
$ ./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.

5 Sorts and Recursive Translation Schemes

In this section we explain how basic algebraic structures and transformations are expressed in HACS.

5.1 Example. Another fragment of the *First.hx* example has the semantic sorts and operations that are used. For our toy language that just means the notion of a *type* with the way that types are "unified" to construct new types.

```
// Types to associate to AST nodes.                    1
sort Type | Int | Float ;                              2

// The Type sort includes a scheme for unifying two types.   4
| scheme Unif(Type,Type) ;                             5
Unif(Int, Int) → Int;                                  6
Unif(#1, Float) →Float;                                7
Unif(Float, #2) →Float;                                8
```

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 []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 (here there are none).

The semantic sort also includes a scheme declaration for the Unif constructor, which must be followed by an argument list with two Type arguments. The scheme declaration is "instantiated" 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 *parameters* in the form

of "meta-variables" starting with # (hash), like #1, to designate "function arguments" that should be copied from the pattern to the replacement.

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

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

Note how overlaps are allowed but please do verify determinacy, *i.e.*, if a particular combination of arguments can be subjected to two rules then they should give the same result! In this example i happens because the term Unif(Float,Float) can be rewritten by both the rule in line 7 and 8, but it does not matter, because the result is the same.

5.2 Example (Syntactic scheme)**.** The Unif scheme defined in the example is a simple example of a recursive translation scheme, defined by rewrite rules. We are permitted to define such schemes over the syntactic sorts, as well. Here is, for example, code to extract the leftmost leaf expression from an Exp tree from Example 4.1:

```
sort Exp | scheme Leftmost(Exp) ;
Leftmost(⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧) →Leftmost(Exp#1) ;
Leftmost(⟦⟨Exp#2⟩ * ⟨Exp#3⟩⟧) →Leftmost(Exp#1) ;
Leftmost(⟦⟨INT#⟩⟧) →⟦⟨INT#⟩⟧ ;
Leftmost(⟦⟨FLOAT#⟩⟧) →⟦⟨FLOAT#⟩⟧ ;
Leftmost(⟦⟨Name#⟩⟧) →⟦⟨Name#⟩⟧ ;
```

Notice how:

- We first set the current sort to Exp and then add a scheme called Leftmost that takes one argument of Exp sort.
- There is precisely one rule with a pattern applying Leftmost to each non-sugar production for Exp from Example 4.1.
- Each non-terminal reference has the non-terminal name followed by a $\#n$ marker to identify the subexpression of that non-terminal sort for use on the right side of the →.
- Each rule rewrites an Exp expression to another Exp expression, either by recursively invoking the defined Leftmost scheme on a smaller part of the term or by returning the term itself.
- We *have* to write ⟦⟨Name#⟩⟧ rather than just Name# or # in the last rules because the form Name# describes something of Name sort, not Exp sort. Indeed in the last rule, Name# stands for the subterm of the Exp expression that is a Name.

5.3 Commands (invoke scheme)**.** The Leftmost scheme above is also included in *First.hx*. Since it operates on a syntactic expression, we can invoke the Leftmost scheme from the command line as follows:

```
$ ./First.run --scheme=Leftmost --sort=Exp --term="(2*3)+4"
 2
```

We specify the sort of our expression here because Leftmost takes an Exp argument, which is different from the main Stat sort.

Note that we cannot meaningfully invoke the Unif scheme from the command line because there is no user syntax for types in our example!

5.4 Example. The *First.hx* example defines the Compile scheme as a top level function to be used from the command line. This looks as follows:

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

The Compile scheme refelcts how our compiler is structured, and in particular that the input is a Stat and the output an AProgr, which stands for "assembly program." (You will see later what the right side of the $\rightarrow$ here means.) This is the reason for the `--scheme=Compile` option we invoked back in the getting started section. Such wrapper raw schemes must have a single argument.

6 Collecting Information

HACS has special support for assembling information in a "bottom-up" manner, corresponding to how *synthetic attributes* are used in syntax-directed definitions (or attribute grammars). In this section we explain *how* you convert any SDD synthetic attribute definition into a HACS one.

Consider the following single definition of the synthesized attribute t for expressions E:

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 + E_2$	$E.t = \mathrm{Unif}(E_1.t, E_2.t)$

(E1)

The rule is "S-attributed" because it exclusively relies on synthesized attributes. This allows us to express it directly in HACS as follows.

1. The first thing to do is declare the attribute and associate it with the E sort.

   ```
   attribute ↑t(Type);
   sort E | ↑t ;
   ```

   the ↑ indicates "synthesized" because the attribute moves "up" in the tree. The declaration of the attribute indicates with (Type) that the *value* of the synthesized attribute is a Type. Attributes are always named with lower case names.

2. The second thing to do is copy the corresponding syntax production but omit any @-markings and add a unique $\#n$ disambiguation mark to each production, essentially "upgrading" each nonterminal reference to a meta-variable. Since (E1) is based on the alternative

   ```
   sort E  | ⟦ ⟨E@1⟩ + ⟨E@2⟩ ⟧@1
   ```

   similarly to Example 4.1, we start with

   ```
   ⟦ ⟨E#1⟩ + ⟨E#2⟩ ⟧
   ```

   where we have used the subscripts from (E1) as #-disambiguation marks.

3. Next add in *synthesis patterns* for the attributes we are reading. Each attribute reference like $E_1.t$ becomes a pattern like ⟨E#1 ↑t(#t1)⟩, where the meta-variables like #t1 should each be unique. For our example, this gives us

   ```
   ⟦ ⟨E#1 ↑t(#t1)⟩ + ⟨E#2 ↑t(#t2)⟩ ⟧
   ```

   which sets up #t1 and #t2 as synonyms for $E_1.t$ and $E_2.t$, respectively.

4. Finally, add in the actual synthesized attribute, using the same kind of pattern at the *end* of the rule (and add a ;), and we get

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

   This is read "When considering an E (the current sort) which has the shape $[\![\langle E\rangle + \langle E\rangle]\!]$ 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 should be assigned the value Unif(#t1,#t2) for the synthesized attribute t."

6.1 Example. In Example 4.1, we presented the abstract syntax of the small language processed by *First.hx*. A type analysis of the expressions of the language excluding variables might look as follows as a standard SDD (syntax directed definition), where we use E for the Exp non-terminal, and one attribute: $E.t$ is the synthesized Type of the expression E. In the notations of [**?**], the SDD can be specified something like this:

PRODUCTION	SEMANTIC RULES
$E \to E_1 + E_2$	$E.t = \text{Unif}(E_1.t, E_2.t)$
$\mid E_1 * E_2$	$E.t = \text{Unif}(E_1.t, E_2.t)$
$\mid$ **int**	$E.t = \text{Int}$
$\mid$ **float**	$E.t = \text{Float}$

where we assume that Unif is defined as discussed in Example 5.1. We can convert this SDD to the following HACS (using the proper names for the sorts as actually found in *First.hx*):

```
attribute ↑t(Type);          // synthesized type                                  1

sort Exp | ↑t ;              // expressions have an associated synthesized type, E.t  3

// Synthesis rules for E.t.                                                       5
⟦ ⟨Exp#1 ↑t(#t1)⟩ + ⟨Exp#2 ↑t(#t2)⟩ ⟧↑t(Unif(#t1,#t2));                          6
⟦ ⟨Exp#1 ↑t(#t1)⟩ * ⟨Exp#2 ↑t(#t2)⟩ ⟧↑t(Unif(#t1,#t2));                          7
⟦ ⟨INT#⟩ ⟧↑t(Int);                                                               8
⟦ ⟨FLOAT#⟩ ⟧↑t(Float);                                                           9
```

Line 1 declares the value of the synthesized t attribute to be a Type. Line 3 associates the synthetic attribute t to the Exp sort: all synthetic attributes are associated with one or more abstract syntax sorts. The remaining lines 5–9 are *synthesis rules* that show for each form of Exp what the value should be, based on the values passed "up" from the subexpressions; these are generated mechanically as discussed above from the synthesis semantic rules.

Finally, note that if you have multiple synthetic attributes then a synthesis rule *only* adds one new attribute to the program construct in question, it does not remove any other attributes already set for it.

7 Full Syntax-Directed Definitions

In general, however, we wish to implement analyses that are more general than what can be achieved with S-attributed syntax-directed definitions: we also want to use *inherited* attributes. This sectionexplains how inherited attributes are implemented in HACS.

Consider the following two simple semantic rules:

PRODUCTION	SEMANTIC RULES	
$S \to \mathbf{name}_1 := E_2; S_3$	$E_2.e = S.e; S_3.e = \mathrm{Extend}(S.e, \mathbf{name}_1.sym, E_2.t)$	(1)
$E \to E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e$	(2)

where we furthermore have the property that th4 $E.t$ property cannot be synthesized until *after* the inherited attribute $E.e$ has been "spread" into the term (we shall see later why this is typical).

Here are the steps to follow to translate the inheritance to HACS.

1. The first thing to do is, again, to declare the attribute. Since $S.e$ and $E.e$ are a *map* from names to types, this is written as follows:

   ```
   attribute ↓e{Name:Type} ;
   ```

   The ↓ indicates an inherited attribute, and the {Name:Type} part declares the value of the attribute to be a mapping from values of Name sort to values of Type sort. (We'll assume that we still have the $E.t$ synthesized attribute defined as previously.) As above, attributes are always given lower case names.

   Note that only sorts with a symbol declaration can be used for keys of mappings: the mappings take the rôle of *symbol tables* from traditional compilers.

2. The second thing to do is to associate the inherited attribute to a *recursive scheme*, which will be responsible for propagating that inherited attribute over a values of a certain sort. This has to be done separately for each sort that e propagates over. Rule (2) is the simplest, so we take that first. The rule propagates the e attribute over E subexpressions, so we invent a scheme, Ee, which does that.

   ```
   sort E | scheme Ee(E) ↓e ;
   ```

   As can be seen, the scheme generates results of the E sort and also takes parameters of the E sort; in addition it *carries* the inherited attribute e.

3. We first observe that (2) operates on sums, which in our case have a syntax like this:

   ```
   sort E | ⟦ ⟨E@1⟩ + ⟨E@2⟩ ⟧@1 ;
   ```

   As before, we create a *pattern* that is equivalent to this, using the subscripts from (2):

   ```
   ⟦ ⟨E#1⟩ + ⟨E#2⟩ ⟧
   ```

4. Now *insert* the pattern into the scheme:

   ```
   Ee(⟦⟨E#1⟩ + ⟨E#2⟩⟧)
   ```

   This clearly respects the sort constraints we defined above, with Ee being applied to an E expression.

5. Since there are no complicated dependencies in (2), we are almost done: we just have to create a rule where we on the right side of the → *apply* the Ee scheme recursively to the subexpressions that should inherit the e attribute:

   ```
   Ee(⟦⟨E#1⟩ + ⟨E#2⟩⟧) →⟦⟨E Ee(#1)⟩ + ⟨E Ee(#2)⟩⟧ ;
   ```

Notice that there is no explicit mention of the e attribute, only the *implicit* copying that follows from the use of the Ee scheme. The recursive arrangement of the Ee wrappers implies the two attribute equations $E_1.e = E.e$ and $E_2.e = E.e$ from (2).

6. The above rule implements (2), with one caveat: every time the e attribute is propagated through, a new expression is created, which thus loses all the synthesized properties it may have had. If we know that the transformation does not invalidate any of the already synthesized attributes, then we express that by augmenting the rule to

   Ee(⟦⟨E#1⟩ + ⟨E#2⟩⟧ ↑#syn) →⟦⟨E Ee(#1)⟩ + ⟨E Ee(#2)⟩⟧ ↑#syn ;

   which now explicitly declares that the new expression should keep all synthetic attributes (again named with a meta-variable so several sets can be preserved by a single rule).

7. Rule (1) is slightly more complicated, because the inherited attribute has non-trivial dependencies. We must know the dependency relationship of the attributes to devise a *recursive strategy* for the attribute evaluation. Recall that we have the following (realistic) dependency for (1): "The $E_2.t$ attribute cannot be computed until *after* $E_2.e$ has been instantiated (and recursively propagated)." In that case we have to evaluate (1) in two steps:

   (a) Do $E_2.e = S.e$, establishing the precondition for allowing the system to compute $E_2.t$.

   (b) When the system has computed $E_2.t$ then do $S_3.e = \mathrm{Extend}(S.e, \mathbf{name}_1.sym, E_2.t)$.

   These two steps are achieved by having an extra carrier schemes:

   sort S | scheme Se(S) ↓e | scheme SeB(S) ↓e ;

   The first propagates into statements in the same way as for (2), except only into the first subterm, and *chains to the next stage*:

   Se(⟦x := ⟨E#2⟩; ⟨S#3⟩⟧ ↑#syn) → SeB(⟦x := ⟨E Ee(#2)⟩; ⟨S#3⟩⟧ ↑#syn) ;

   Notice how we invoke the Ee scheme to pass $E_2.e$ (the system understands that $S.e$ and $E_2.e$ refer to the same attribute), and how we do *not* wrap #3 in anything as nothing should be passed there; instead we just leave the extra top wrapper SeB to wait for when it can process.

   Once the synthetic is satisfied, *i.e.*, once $E_2.t$ is computed, then the second stage can finishes the job, using the notation of the previous section, and proceed recursively on the appropriate subterm. However, it also needs to compute the Extend result, which is achieved with the following rule:

   SeB(⟦x := ⟨E#2 ↑t(#t2)⟩; ⟨S#3⟩⟧ ↑#syn) →⟦x := ⟨E#2⟩; ⟨S Se(#3)↓e{⟦x⟧:#t2} ⟩⟧↑#syn ;

   The last bit of notation is the way we call Se with an *extended* $S_3.e$ attribute: the notation Se(#3)↓e{⟦x⟧:#t2} means "call Se (passing e) on the statement #3 but use an e which has been extended with the mapping from x to #t2," which corresponds to the notation $S_3.e = \mathrm{Extend}(S.e, \mathbf{name}_1.sym, E_2.t)$ of the SDD.

   Also note that because the Name sort is a symbol sort, we should *directly* use the variable x (which is a legal ID token) in the rules instead of ⟨Name#x⟩ or such. However, we still take care to distinguish a symbol that is part of the syntax, like x, from the other symbols that are part of the formalism, like S and Se: x is always enclosed in ⟦⟧s

PRODUCTION	SEMANTIC RULES	
$S \to \textbf{name} := E_1; S_2$	$E_1.e = S.e; S_2.e = \text{Extend}(S.e, \textbf{name}.sym, E_1.t)$	(S1)
$\mid \{\ S_1\ \}\ S_2$	$S_1.e = S.e; S_2.e = S.e$	(S2)
$\mid \epsilon$		(S3)
$E \to E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \text{Unif}(E_1.t, E_2.t)$	(E1)
$\mid E_1 * E_2$	$E_1.e = E.e; E_2.e = E.e; E.t = \text{Unif}(E_1.t, E_2.t)$	(E2)
$\mid$ **int**	$E.t = \text{Int}$	(E3)
$\mid$ **float**	$E.t = \text{Float}$	(E4)
$\mid$ **name**	$E.t = \text{if Defined}(E.e, \textbf{name}.sym)$ then $\text{Lookup}(E.e, \textbf{name}.sym)$ else TypeError	(E5)

Figure 1: SDD for type checking.

7.1 Example. An SDD for a simple type analysis can be implemented with two attributes (and using the usual convention that the SDD uses E and S where the HACS grammar has `Exp` and `Stat`):

- The inherited *environment* attribute e, which is a map from variables to types on both the statement and expression non-terminals S and E.
- The synthesized *type* attribute t on expressions E, which contains the type of the expression.

With those, the SDD can be expressed as shown in Figure 1, where we use the helpers Extend, Defined, and Lookup to build an extended environment with an additional type declaration, check for existence, and look one up, respectively, and Unif to find the type of an arithmetic operation with operands of two specific types.

If we use the translation mechanism, then we obtain the HACS fragment in Figure 2.

For reference, we conclude this section with a summary of how symbol tables are encoded in a production for a nonterminal N with an inherited environment attribute e:

- "$\text{Defined}(N.e, x)$" is encoded as true with the additional constraint ↓e{⟦x⟧} in a pattern.
- "$\text{Lookup}(N.e, x)$" is encoded as the additional constraint ↓e{⟦x⟧:#} in a pattern, which then binds the meta-variable # to the result of the lookup.
- "$\text{Extend}(N.e, x, V)$" is encoded as the additional constraint ↓e{⟦x⟧:V} in the result.

A Manual

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

```
module modulename
{
  Declarations
}
```

```
sort Type | Int | Float | TypeError                                                          1
         | scheme Unif(Type,Type) ;                                                        2

Unif(Int, Int) → Int;                                                                     4
Unif(#t1, Float) →Float;                                                                  5
Unif(Float, #t2) →Float;                                                                  6
default Unif(#1,#2) →TypeError; // fall-back                                              7

attribute↑t(Type); // synthesized expression type                                         9
sort Exp | ↑t;                                                                            10

⟦ (⟨Exp#1 ↑t(#t1)⟩ + ⟨Exp#2 ↑t(#t2)⟩) ⟧↑t(Unif(#t1,#t2));                                 12
⟦ (⟨Exp#1 ↑t(#t1)⟩ * ⟨Exp#2 ↑t(#t2)⟩) ⟧↑t(Unif(#t1,#t2));                                 13
⟦ ⟨INT#⟩ ⟧↑t(Int);                                                                        14
⟦ ⟨FLOAT#⟩ ⟧↑t(Float);                                                                    15
// Missing case: variables – handled by Ee below.                                         16

attribute↓e{Name:Type}; // inherited type environment                                     18

sort Exp | scheme Ee(Exp) ↓e ; // propagates e over Exp                                   20

// These rules associate t attribute with variables (missing case above).                 22
Ee(⟦id⟧) ↓e{⟦id⟧ : #t} →⟦ id ⟧ ↑t(#t);                                                    23
Ee(⟦id⟧) ↓e{¬⟦id⟧} → error⟦Undefined identifier ⟨id⟩⟧;                                    24

Ee(⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧ ↑#syn) →⟦⟨Exp Ee(#1)⟩ + ⟨Exp Ee(#2)⟩ ⟧↑#syn ;                    26
Ee(⟦⟨Exp#1⟩ * ⟨Exp#2⟩⟧ ↑#syn) →⟦⟨Exp Ee(#1)⟩ * ⟨Exp Ee(#2)⟩ ⟧↑#syn ;                    27
Ee(⟦⟨INT#⟩⟧ ↑#syn) →⟦⟨INT#⟩⟧ ↑#syn ;                                                      28
Ee(⟦⟨FLOAT#⟩⟧ ↑#syn) →⟦⟨FLOAT#⟩⟧ ↑#syn ;                                                  29

sort Stat | scheme Se(Stat) ↓e ; // propagates e over Stat                                31

Se(⟦id := ⟨Exp#1⟩; ⟨Stat#2⟩⟧ ↑#syn) →SeB(⟦id := ⟨Exp Ee(#1)⟩; ⟨Stat#2⟩⟧ ↑#syn);            33
{                                                                                         34
  | scheme SeB(Stat) ↓e; // helper scheme for assignment after expression typeanalysis    35
  SeB(⟦id := ⟨Exp#1 ↑t(#t1)⟩; ⟨Stat#2⟩ ⟧↑#syn)                                           36
   → ⟦id := ⟨Exp#1⟩; ⟨Stat Se(#2) ↓e{⟦id⟧:#t1}⟩⟧ ↑#syn ;                                37
}                                                                                         38

Se (⟦ { ⟨Stat#1⟩ } ⟨Stat#2⟩ ⟧↑#syn) →⟦{ ⟨Stat Se(#1)⟩ } ⟨Stat Se(#2)⟩ ⟧↑#syn ;             40

Se (⟦ ⟧ ↑#syn) →⟦ ⟧ ↑#syn ;                                                               42
```

Figure 2: HACS code for type analysis.

where the *modulename* should be a Java style fully qualified class name with the last component is capitalized, like `org.crsx.hacs.samples.First`. 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 (see next item).
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) a \ followed by an *escape code* is encountered then it stands for the encoded character;
   (d) two characters connected with a − (dash) stands for a single character in the indicated (inclusive) *range*.

   Note that a character class cannot be empty, however, [^] is permitted and stands for all characters.
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). 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, even inside syntax productions.) A spacing declaration permits the special alternative `nested` declaration for nested comments, illustrated by the following, which defines usual C/Java style spacing with comments as used by HACS itself:

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

Notice that spacing is not significant in regular expressions, except (1) in character classes, (2) in literal strings, and (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 C.)
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. If a production 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 (parsed terms)**.** The term model includes *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 ⟨...⟩ (unicode U+27E8 and U+27E9). Such nested raw terms *must* have an explicit sort prefix.
3. The special term error⟦...⟧ will print the error message embedded in ⟦...⟧, where one is permitted to embed symbol-declared variables in ⟨...⟩.

A.5 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 list of the subexpression sorts in parenthesis.
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 subterms.
   (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.

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 [**?**].

A.6 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.7 Manual (building and running)**.** To translate a HACS script to an executable, run the *hacs* command, which generates a number of files under a *build* subdirectory, as well as the main script with a *.run* extension. 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. `--scheme=`*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=`*text* use the *text* 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 `--scheme`, and one of `--term` and `--input`.

Notice that the *.run* script has absolute references to the files in the *build* directory, so the latter should be moved with care.

B Common Errors

B.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.

B.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.

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

```
Java Compiler Compiler Version ??.??_?? (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file FirstHx.jj . . .
Warning: Choice conflict involving two expansions at
         line 3030, column 34 and line 3033, column 8 respectively.
         A common prefix is: "{" <T_HX_VAR>
         Consider using a lookahead of 3 or more for earlier expansion.
Warning: Line 4680, Column 18: 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.

B.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.

B.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.

B.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.

C 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).
- 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.