Technical Report

TWO SIGMA

TWO SIGMA INVESTMENTS
March 2, 2016

www.twosigma.com

NEW YORK HOUSTON
LONDON HONG KONG

TSTR-2016-1

Introduction to Compiler Generation Using HACS

Kristoffer H. Rose
Two Sigma Investments

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 as specified in HACS: lexical analysis, syntax analysis, semantic analysis, and code generation.

1. INTRODUCTION

Higher-order Attribute Contraction Schemes, or HACS, 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 [22], although HACS users need not be concerned with the details of higher-order rewriting (even if those are, in fact, most interesting).

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 follow common conventions [2]. Section 3 gives details of this notation.

Context-Free Grammars. HACS uses a form of BNF [18] *context-free grammars* but tweaks the notation to look more like *templates*, to allow for reuse of the notation in the subsequent rewrite rules. HACS includes simple mechanics to allow for the formalization of the transformation from token stream to abstract syntax. Details in Section 4.

Recursive Translation Schemes. Simple translation in general, and code generation in particular, is traditionally achieved by *recursive translation* from one abstract form to another. HACS includes special notations for defining such translations, described in Section 5, as well as a full programming language for defining auxiliary so-called "semantic" sorts and operators, detailed in Section 6.

Attribute Grammars. Analyses can often be conveniently described in the style of *syntax-directed definitions* [2], originally introduced as *attribute grammars* [14], which describe how properties propagate through the abstract syntax tree. Section 7 details how the basic propagation rules work for synthesized attributes. Section 8 explains how inherited attributes are integrated to enable the encoding of full syntax-directed definitions.

Higher-order Abstract Syntax. Most programming languages use *lexical scoping*, and compiler internals as well as target representations increasingly manipulate lexical scopes as part of their normal operation [16, 17]. HACS supports this by providing explicit support for *higher-order abstract syntax*, and integrates this with support for attributes with mappings from variables to values for modeling *symbol tables*.

A typical compiler written with HACS involves multiple instances of each of the above, as each used language, including intermediate and target languages, is specified using grammars and has language-specific analyses, with the transformations between them specified as translation schemes.

History. Many systems for writing programs that manipulate other programs, so-called *meta-programming*, have emerged over the years. These range from generic specification languages, where the goal is not to define how but only to declare the semantics of the program manipulation, all the way to tools that support specific aspects of program execution or compiler generation (a survey is beyond the scope of this document).

One such system was CRSX [23] developed for industrial use at IBM Research by a team led by the author [24, 26, 25, 22]. CRSX is a language based on *higher-order rewriting* [10] combined with *higher-order abstract syntax* (HOAS) [20], further extended for handling environments natively and using pluggable parsers. The programming of the IBM DataPower XQuery compiler [8] using CRSX proved that the approach can drastically reduce the development time of a compiler (the cited XQuery compiler was estimated to have been developed in a quarter of the traditional development time) as well as resulting in a rather more compact and high-level source program.

However, the CRSX notation, based on combinatory reduction systems [11, 13], which combines λ calculus [5, 3] and term rewriting systems [12], has proven to be unwieldy for several reasons, first of all by

being quite different from standard notations used in compiler construction reference works [2]. Adding a polymorphic sort system to CRSX following *contraction systems* [1] changed the system to be similar to Inductive Type Systems [4]), which helped, but did not make the system easy enough to, for example, teach compiler construction.

HACS is an attempt to remedy this situation by providing a front-end for CRSX that allows the use of standard notations and concepts of (formal) programming language descriptions to directly program compilers and other systems for manipulating code, as discussed above. HACS has been successfully used to teach the graduate computer science compiler construction class at New York University [21].

Acknowledgements. The author would like to thank his coteacher at NYU, Eva Rose, our grader, José Pablo Cambronero, as well as our students in the compiler construction class, for constructive comments to and insightful questions on HACS.[1] The implementation of HACS would not have been remotely possible without the CRSX team at IBM: Morris Matsa, Scott Boag, and especially Lionel Villard, who suffered through understanding and programming core fragments of the CRSX system that is used underneath HACS. HACS owes its sanity to a collaboration with Cynthia Kop, both as an intern with the Watson team in 2011, which created the polymorphic sort checker, in her thesis work [15], and in our continuing collaboration on keeping the formal basis for the system up to date. Finally, the author would like to thank Two Sigma for supporting this work, and in particular Eliot Walsh for corrections and advice on the use of English.

Outline of this report. The remainder of this document introduces the most important features of the HACS language by explaining the relevant parts of the included *First.hx* example (inspired by [2, Figure 1.7]) as well as several other minor examples. Section 2 shows how to install HACS and run the example, before proceeding to the writing of specifications. Section 3 explains lexical analysis; Section 4 syntax analysis; Section 5 basic recursive translation schemes; Section 6 semantic sorts, operations, and data; Section 7 bottom-up semantic analysis; Section 8 general syntax-directed definitions; and Section 9 higher order abstract syntax. Section 10 addresses the manipulation of primitive values, and Section 11 provides several examples of how everything can be combined. Finally, Appendix A has a reference manual, Appendix B explains some of the (still) cryptic error messages, and Appendix C lists some current limitations. Bibliographic references are collected last.

2. GETTING STARTED

This section walks through the steps for getting a functional HACS installation on your computer.[2]

2.1 REQUIREMENTS. To run the HACS examples presented 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, flex, and C99 and C++ compilers. In addition, the setup process needs internet access to retrieve the CRSX base system [22], JavaCC parser generator [27], and *icu* Unicode C libraries [9]. Finally, these instructions are written for and tested on Ubuntu and Debian GNU/Linux systems using the *bash* shell; if your system is different, some adaptation may be needed.

2.2 COMMANDS (install HACS)**.** If you have an old version of HACS installed, then it is best to remove it first with a command like

```
$ rm -fr ~/.hacs hacs
```

[1] A special shout-out goes to John Downs for starting the Emacs `M-x hacs-mode` syntax highlighting mode [7] and Tyler Palsulich for a HACS mode for Sublime Text [19].

[2] Please report any problems with this procedure to *hacs-bugs@crsx.org*.

Now retrieve the *hacs-1.1.20.zip* archive, extract it to a new directory, and install it, for example with the following commands:[3]

```
$ wget http://crsx.org/hacs-1.1.20.zip
$ unzip hacs-1.1.20.zip
$ cd hacs
energon1[hacs]$ make FROMZIP=1 install install-support
```

These commands will need Internet access to use the `wget` command to retrieve support libraries.[4] 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 may take several minutes the first time and should end without error.

Please check that your new installation works with these commands:

```
energon1[hacs]$ cd
$ mkdir myfirst
$ cd myfirst
$ cp $HOME/.hacs/share/doc/hacs/examples/First.hx .
$ $HOME/.hacs/bin/hacs First.hx
$ ./First.run --scheme=Compile \
              --term="{initial := 1; rate := 1.0; position := initial + rate * 60;}"
  LDF T,  #1
    STF initial, T
    LDF T_77,  #1.0
    STF rate, T_77
    LDF T_1,  initial
    LDF T_1_60,  rate
    LDF T_2,  #60
    MULF  T_2_21 ,  T_1_60 ,  T_2
    ADDF  T_96 ,  T_1 ,  T_2_21
    STF position, T_96
```

Congratulations—you just built your first compiler![5] The following assumes that you have issued the following command to make 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`.)

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

```
/* Our first compiler. */
module org.crsx.hacs.samples.First
{
  // Sections.
  Lexical Analysis (Section 3)
  Syntax Analysis (Section 4)
  Semantic Analysis (Sections 5, 7 and 8)
  Code Generator (Section 11)
```

[3] User input is like this.

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

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

```
    Main (Section 5)
}
```

Notice that HACS permits C/Java-style comments and that the final component of the module name is the same as the base of the filename.

2.4 NOTATION. The structure shown in Example 2.3 is formally explained in the appendix (Manual A.1), as is how to run HACS (Manual A.7).

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

Table 1: Unicode special characters used by HACS.

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

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. This section explains the most common rules; the appendix (Manual A.2) gives the formal rules.

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 this simple language:

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

// Basic terminals.                                           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.

3.2 NOTATION (lexical syntax).

1. Declarations generally start with a keyword or two and are terminated by a ; (semicolon).
2. **token** declarations in particular have the **token** keyword followed by the name of the token and a regular expression between a | (vertical bar) and a ; (semicolon). Each defines the token as a *terminal symbol* that can be used in other token declarations as well as the syntax productions described in the next section.
3. Token names must be words that start with an uppercase letter.
4. A regular expression is a sequence of units, corresponding to the concatenation of a sequence 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, in one of a sequence of individual characters and ranges), a *string* such as `"."` or `'foo'` (either kind of quotes is allowed), or a *reference* to a token or fragment such as ⟨Lower⟩, enclosed in the special Unicode mathematical angle brackets (see Table 1).
5. The declaration of a **token fragment** specifies that the token can only be used in other token declarations, not in syntax productions.
6. Every regular expression component can be followed by a repetition marker ?, +, or ∗, and regular expressions can be grouped with parentheses.
7. 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 to escape in character classes with usual C-style language escapes.

In addition, this manual follows the convention of naming proper grammar terminals with ALL-CAPS names, like INT, to make them easy to distinguish from nonterminals below. (Token declarations are not grammar productions: tokens cannot be recursively defined, and a token referenced from another token is merely an inlining of the character sequences allowed by what is referenced.)

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.

3.3 EXAMPLE (comments). One common question is what to do with comments. A common choice is to ignore them completely. This is achieved by including your comment syntax in your **space** declaration. For C/Java "modern" end of line comments, this is for example done as follows:

```
space [ \n\t]   |   '//' .* ;
```

(as in most regular expression formalisms, . (dot) abbreviates [^\n]).

In addition, HACS has special support for nested comments in the **space** declaration:

```
space [ \n\t]   |   nested '(*' '*)' ;
```

recognizes multi-line comments starting with (∗ and ending with ∗), which may nest, *i.e.*, the text (∗*a*(∗*b*∗)*c*∗) is skipped as one single space.

3.4 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 two arguments to the *First.run* command: a *token sort* and a *token term*.[6] Execution proceeds by parsing the string following the syntax of the token. For example, the following checks the lexical analysis of a number:

```
$ ./First.run --sort=FLOAT --term=34.56
34.56
```

Note that the term `34.56` could also have been provided enclosed in quotes, such as `"34.56"` or `'34.56'`. If there is an error, the lexical analyzer will inform us of this:

[6] The command has more options that will be introduced as needed.

```
$ ./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.

As the example illustrates, all the token declarations together define how the processed stream of characters is partitioned into terminal symbols with no need for consulting a grammar. The reference manual for tokens in the appendix (Manual A.2) gives further details, including some additional constructs.

4. SYNTAX ANALYSIS

Once tokens have been defined, it is possible to use HACS to program a syntax analysis with a grammar that specifies how to decompose the input text according to a *concrete syntax* and how to construct the desired *abstract syntax tree* (AST) 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 an AST is built. Grammars are structured following the *sorts* of AST nodes, with concrete syntax details managed through precedence annotations and "syntactic sugar" declarations. The complete specification for grammars is in the appendix (Manual A.3).

4.1 EXAMPLE. Here is the syntax analysis grammar from the *First.hx* example. This small example source language merely has blocks, assignment statements, and a few expression forms, like so:

```
main sort Stat  | ⟦ ⟨ID⟩ := ⟨Exp⟩ ; ⟨Stat⟩ ⟧                          1
                | ⟦ { ⟨Stat⟩ } ⟨Stat⟩ ⟧                              2
                | ⟦⟧ ;                                                3

sort Exp    | ⟦ ⟨Exp⟩ + ⟨Exp@1⟩ ⟧                                    5
            | ⟦ ⟨Exp@1⟩ * ⟨Exp@2⟩ ⟧@1                               6
            | ⟦ ⟨INT⟩ ⟧@2                                           7
            | ⟦ ⟨FLOAT⟩ ⟧@2                                         8
            | ⟦ ⟨ID⟩ ⟧@2                                            9
            | sugar ⟦ ( ⟨Exp#⟩ ) ⟧@2→ Exp# ;                        10
```

The grammar structures the generated AST with two sorts: Stat for statements and Exp for expressions.

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: spaces, literal character "words," and references to nonterminals and predefined tokens inside ⟨. . .⟩. In this way, literal syntax is similar to macro notation or "quasi-quotation" of other programming notations.

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

Precedence rules are represented by the @-annotations, which assign precedence and associativity to each operator. This example marks all references to the Exp nonterminal inside the productions for Exp with the lowest permitted precedence in each case. The first rule in line 5 says that the + operator is restricted on the right to only expressions with at least precedence 1 (but not restricted on the left, causing it to be left-recursive). The outer @1 in line 6 states that all * expressions have precedence 1, and the inner @-annotations allow left subexpressions of * with at least precedence 1 (* is also left recursive), whereas right subexpressions must have at least precedence 2. Notice that left (right) recursion is identified by the *leftmost (rightmost) unit in the production having the outer precedence.*

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

4.2 REMARK. Precedence is traditionally done with additional "helper" productions. It is possible, for example, to recognize the same Exp language as in Example 4.1 by something akin to the following concrete HACS specification (in which the sugar is also concrete):

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

However, this grammar generates a different result tree, where the nodes have the three 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 using standard techniques.

Also note that the notation is admittedly dense. This is intentional, as the notation can be generalized to serve all the formalisms of the following sections. Here are the formal rules.

4.3 NOTATION (syntax analysis)**.**

1. 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 BNF grammars.) All productions for a sort define cases for that sort as a nonterminal, called the "target" nonterminal.
2. Sort names must be words that start with an uppercase letter.
3. 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.
4. References to *terminals* (tokens) and *nonterminals* (other productions) are wrapped in ⟨ . . . ⟩ (angle brackets).
5. *Precedence* is indicated with @*n*, where higher numbers *n* designate higher (tighter) precedence. After every top-level ⟦⟧ and placed last inside every ⟨⟩-reference to a target nontermina,l there is a precedence, which defaults to @0. The precedence of self-references at the beginning and end of a production must be *greater than or equal to* the outer precedence; at most one of the ends can have precedence equal to the outer one.
6. The special **sugar** declaration expresses that the specified concrete syntax with a single disambiguated self-reference is *replaced* by what is written after the →. A reference is "disambiguated" by trailing it with a meta-variable starting with #.

Of all these rules, the one thing that is unique to parsing is the precedence notation with @. When specifying a grammar then *every* target nonterminal reference has a precedence, which determines how to parse ambiguous terms. So imagine that every ⟨⟩ contains a @ marker at the end, defaulting to @0, and that every ⟦⟧ is terminated with a @ marker, again defaulting to @0.

Notice that HACS will do three things automatically:

1. Split the productions into subproductions according to the precedence assignments.
2. Eliminate immediate left recursion, as in the example.
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, eliminating the need to be aware that this conversion happens.

4.4 COMMANDS. It is possible to 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 reinsertion of the sugared syntax.

```
$ ./First.run --sort=Exp --term="2 ** 3"
Exception in thread "main" java.lang.RuntimeException: net.sf.crsx.CRSException:
  Encountered " "*" "* "" at line 1, column 4.
Was expecting one of:
    "(" ...
    <T_INT> ...
    <T_FLOAT> ...
    <T_ID> ...
```

Similarly to the lexical analysis, the syntax analysis can return an error in the case that a term provided does not satisfy the grammar.

5. ABSTRACT SYNTAX AND RECURSIVE TRANSLATION SCHEMES

This section explains how to express recursive translation schemes over the abstract syntax implied by a grammar. The formal specification for this section is in the appendix (Manual A.4 and A.5).

5.1 EXAMPLE. Consider the following subset of the example expression grammar of Example 4.1:

```
sort Exp  | ⟦ ⟨Exp⟩ + ⟨Exp@1⟩ ⟧
          | ⟦ ⟨Exp@1⟩ * ⟨Exp@2⟩ ⟧@1
          | ⟦ ⟨INT⟩ ⟧@2
          | ⟦ ⟨FLOAT⟩ ⟧@2
          | sugar ⟦ ( ⟨Exp#⟩ ) ⟧@2→ Exp# ;
```

This grammar serves two purposes: to describe all token sequences that can be parsed and to describe the structures that are generated from them. Erasing all the information that is purely there for parsing leaves just

```
sort Exp  | ⟦ ⟨Exp⟩ + ⟨Exp  ⟩ ⟧
          | ⟦ ⟨Exp  ⟩ * ⟨Exp  ⟩ ⟧
          | ⟦ ⟨INT⟩ ⟧
          | ⟦ ⟨FLOAT⟩ ⟧ ;
```

This is dubbed the *abstract syntax* for the Exp sort, because all the helper information for the parser has been removed, leaving only the essential, structural information.

The abstract syntax, illustrated by the example, is relevant because the output of a HACS-generated parser is an *abstract syntax tree*, or *AST*, thus all subsequent processing with HACS is based on this simplified structure.

Formally, the abstract syntax is obtained as follows:

- Erase all @*n* precedence markers.
- Remove **sugar** productions.

What remains are minimal productions with the essential information describing the AST.

5.2 REMARK. The dragon book [2] achieves a similar but even bigger leap by associating the grammar with explicit precedence, written

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (\ E\ ) \mid \textbf{int} \mid \textbf{float}$$

with the abstract syntax

$$E \to E + E \mid E * E \mid \textbf{int} \mid \textbf{float}$$

which additionally "folds" the T and F productions into E, as was effectively done in the previous section.

With such a description of the AST, it is possible to write code that operates on the AST, implementing the notion of *syntax-directed translation* (although it would perhaps be even better called "abstract syntax-directed translation").

5.3 DEFINITION. A scheme is *syntax-directed* if it has one case per abstract syntax production.

In practice, syntax-directed translation schemes defined in HACS have one *rule* per abstract syntax production.

5.4 EXAMPLE (scheme over syntax)**.** Consider the abstract expression grammar from Example 5.1, and define a new scheme called Leftmost, which for an expression returns the leftmost number of the expression. To achieve this, first declare the scheme as follows:

```
sort Exp | scheme Leftmost(Exp) ;
```

The declaration states that the scheme takes one parameter of sort Exp in ()s and delivers a result that is also of sort Exp. (Translations that convert from one sort to another will come into play as needed going forward.)

To have Leftmost actually do something requires a set of rules of the form

```
Leftmost(. . . ) →    . . . ;
```

where the two . . . in each case are replaced with a *pattern* and a *result*, in this case both of sort Exp. The patterns are obtained directly from the abstract syntax productions simply by marking every ⟨⟩-embedded token or nonterminal with a "disambiguation" mark, #*n*, giving the following:

```
Leftmost(⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧)      →   . . . ;
Leftmost(⟦⟨Exp#1⟩ * ⟨Exp#2⟩⟧)      →   . . . ;
Leftmost(⟦⟨INT#⟩⟧)                 →   . . . ;
Leftmost(⟦⟨FLOAT#⟩⟧)               →   . . . ;
```

It is now possible to express the right side of each translation rule using the now named fragments of the pattern, keeping in mind that the result should always be of Exp sort:

```
Leftmost(⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧)      →   Leftmost(Exp#1) ;
Leftmost(⟦⟨Exp#1⟩ * ⟨Exp#2⟩⟧)      →   Leftmost(Exp#1) ;
Leftmost(⟦⟨INT#⟩⟧)                 →   ⟦⟨INT#⟩⟧ ;
Leftmost(⟦⟨FLOAT#⟩⟧)               →   ⟦⟨FLOAT#⟩⟧ ;
```

The first two rules pass the #1 fragment, which is of Exp sort, into Leftmost, which is guaranteed (by the declaration) to return something of Exp sort. The last two rules explicitly return the argument form of sort Exp. Notice that in the last two rules, the pattern sets up # to be of sort INT and FLOAT, respectively, disallowing their use directly as the result, as they have the wrong sort. Instead it uses syntax construction to obtain the correct sort by making use of an appropriate production.

A syntax-directed scheme like Leftmost is called a *semantic* translation scheme because it is declared without any new syntax, *i.e.*, without any use of ⟦⟧.

5.5 COMMANDS (invoke scheme). The Leftmost scheme above is also included in *First.hx*. Since it operates on a single syntactic expression, it is possible to invoke the Leftmost scheme from the command line as follows:

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

It is necessary to specify the sort of the input expression here because Leftmost takes an Exp argument, which is different from the main Stat sort.

The notation for defining a syntax-directed scheme is as follows.

5.6 NOTATION (syntax-directed schemes).

1. Set the result **sort** and add a declaration for the scheme. A scheme S of result sort R with argument sorts A and B is declared by

   sort R | **scheme** S(A, B);

   A scheme is named with a capitalized word (the same as sort names), optionally followed by some "arguments" in ()s, where the declaration gives the sort for each argument.

2. To make the scheme S syntax-directed in, say, A, create a separate rule for each "pattern case" of A, which is just each abstract syntax production for A with a #n marker after the token or nonterminal name to identify the subexpression of that token or nonterminal for use on the right side of the →. Thus,

   sort A | ⟦ *a* ⟨E⟩ *b* ⟧ | ⟦ ⟨G⟩ *c* ⟨H⟩ ⟧ ;

   should have

   sort R;
   S(⟦ *a* ⟨E#1⟩ *b* ⟧, #2) → ...;
   S(⟦ ⟨G#1⟩ *c* ⟨H#2⟩ ⟧, #3) → ...;

   The #-markers are called *meta-variables* and are tied to one specific sort in a rule, whether by its marker inside syntax or position as an argument. In the example this means that in the first rule, #1 is of sort E and #2 of sort B, whereas in the second rule, #1 is of sort G, #2 of sort H, and here #3 is of sort B. (Rules are very like **sugar** productions except a given construct can have more than one rule; **sugar**, however, is limited to a single rule that cannot depend on the inner structure of the construct.)

3. Each rule should be equipped with a result (right of the →) of the result sort, which can use the "meta-variable" #-named fragments using any (syntactic or semantic) mechanism for building something of the result sort. For example, adding

   sort R | ⟦ *x* ⟨E⟩ ⟧ | **scheme** Other(B);

   allows writing

   sort R;
   S(⟦ *a* ⟨E#1⟩ *b* ⟧, #2) → ⟦ *x* ⟨E#1⟩ ⟧ ;
   S(⟦ ⟨G#1⟩ *c* ⟨H#2⟩ ⟧, #3) → Other(#3) ;

5.7 EXAMPLE (default rules). Since the last two cases of the Leftmost system in Example 5.4 above really just return the entire term again, the system can instead be written as follows:

```
sort Exp | scheme Leftmost(Exp) ;
Leftmost(⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧)    → Leftmost(#1) ;
Leftmost(⟦⟨Exp#1⟩ * ⟨Exp#2⟩⟧)    → Leftmost(#1) ;
default Leftmost(#)   → # ;
```

which explicitly calls out that if neither of the two regular cases apply, then the scheme just returns the contained expression.

Thus far, the syntactic brackets ⟦. . . ⟧ have been consistently used for "stuff in the input," essentially *input data*, and semantic constructors (like Leftmost above) for "things that can be computed," or *functions*. HACS does not, in fact, insist on this separation. In particular, it is permissible to define "syntactic schemes," which introduce new syntax that has simplification rules associated with it.

Specifically, syntactic schemes are schemes that are defined using ⟦. . . ⟧ notation. They are very similar to **sugar** declarations, except that they can have multiple rules with pattern matching to select which to apply, in contrast to sugar declarations, which can only have one generic case with a single unconstrained meta-variable.

5.8 EXAMPLE (syntactic scheme). Consider the syntactic list (data) sort

```
sort List | ⟦ ⟨Elem⟩ ⟨List⟩ ⟧ | ⟦⟧ ;
```

There is sometimes a need to *flatten* nested lists when working with complex expressions. This can, of course, be done with a usual syntax-directed semantic scheme and a lot of nested ⟦⟨⟩⟧ constructions. An alternative is to define a *syntactic scheme*, which is a scheme expressed in syntactic form but in fact defined by rewrite rules. Flattening of lists can, for example, be defined as follows:

```
sort List | scheme ⟦{ ⟨List⟩ } ⟨List⟩ ⟧ ;
⟦ { ⟨Elem#1⟩ ⟨List#2⟩ } ⟨List#3⟩ ⟧   → ⟦ ⟨Elem#1⟩ { ⟨List#2⟩ } ⟨List#3⟩ ⟧ ;
⟦ { } ⟨List#⟩ ⟧ → # ;
```

This creates a scheme that can be informally written as ⟦{_}_⟧ with the understanding that the two occurrences of "_" should be filled with lists as prescribed by the syntax specification in the **scheme** declaration. Notice that the two rules differ on the *content* of the braces and are clearly designed to fully eliminate all possible contents of the braces. This is essential; the scheme should be *complete*. To be precise: the first _ position in the ⟦{_}_⟧ function definition is filled differently in the two rules, namely once with each of the data shapes of lists – indeed the rules are syntax-directed in the first list argument.

Syntactic schemes are very useful for working with output structures; for example, the flattening scheme of Example 5.8 makes it much easier to manipulate sequences of assembler instructions.

5.9 EXAMPLE (stack code compiler). Figure 1 shows the HACS script *Stack.hx*, which contains a compiler from simple expressions to stack code. Lines 3–13 contain a simple expression grammar, as already discussed. Lines 15–24 contain a separate grammar, this time for the output stack Code (the special ¶ marks indicate where to insert newlines when printing code, and are not part of the syntax). Lines 26–30 contain a flattening syntax scheme (as in Example 5.8) for sequences of instructions. Finally, lines 32–38 contain the syntax-directed translation from expressions to stack code. It is used it in the usual way,

```
$ hacs Stack.hx
HACS 1.1.20
...
$ ./Stack.run --scheme=Compile --term="(1+2)*(3+4)"
  PUSH 1
  PUSH 2
  ADD
  PUSH 3
  PUSH 4
  ADD
  MULT
```

Figure 1: *examples/Stack.hx.*

```
module org.crsx.hacs.samples.Stack {

// Grammar.
space [ \t\n] ;
token INT     | [0−9]+ ;
token ID      | [a−z] [a−z0−9_]* ;

main sort Exp
  | ⟦ ⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1
  | ⟦ ⟨Exp@2⟩ * ⟨Exp@3⟩ ⟧@2
  | ⟦ ⟨INT⟩ ⟧@3
  | sugar ⟦ (⟨Exp#⟩) ⟧@3→ Exp#
  ;  //

// Stack code.
sort Code
  | ⟦ ⟨Instruction⟩ ⟨Code⟩ ⟧
  | ⟦ ⟧
  ;
sort Instruction
  | ⟦ PUSH ⟨INT⟩ ¶ ⟧
  | ⟦ ADD ¶ ⟧
  | ⟦ MULT ¶ ⟧
  ; //

// Flattening helper.
 sort Code | scheme ⟦{ ⟨Code⟩ } ⟨Code⟩ ⟧;
⟦ { ⟨Instruction#1⟩ ⟨Code#2⟩ } ⟨Code#3⟩ ⟧
  → ⟦ ⟨Instruction#1⟩ { ⟨Code#2⟩ } ⟨Code#3⟩ ⟧  ;
⟦ { } ⟨Code#⟩ ⟧ →  Code# ; //

// Compiler.
sort Code | scheme Compile(Exp) ;
Compile(⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧)
  → ⟦ { ⟨Code Compile(#1)⟩ } { ⟨Code Compile(#2)⟩ } ADD ⟧ ;
Compile(⟦⟨Exp#1⟩ * ⟨Exp#2⟩⟧)
  → ⟦ { ⟨Code Compile(#1)⟩ } { ⟨Code Compile(#2)⟩ } MULT ⟧ ;
Compile(⟦⟨INT#⟩⟧) →  ⟦PUSH ⟨INT#⟩⟧ ; //

}
```

Simple code generation tasks can be handled with recursive schemes, such as this one. However, serious compilation tasks will require proper attributes and semantic helper structures, detailed in the following sections.

6. SEMANTIC DATA, OPERATORS, AND EVALUATION

For most compilers, simple recursive translations, such as presented above, do not suffice. More complex programming tools are necessary to support more sophisticated analyses and transformations.

Specifically, note that rules for syntax-directed schemes are similar to definitions by case in functional programming. However, there are important differences to be detailed here that sometimes make HACS pick rules differently than functional programming would.

As discussed in the previous section, it is possible to have functions written with syntactic brackets, and similarly it is possible to have data written with semantic constructors, called "semantic data." Semantic data forms are introduced as non-syntax (so-called "raw") notations that can be used in patterns.

6.1 EXAMPLE (semantic data constants)**.** Another fragment of the *First.hx* example has the semantic sorts and operations that are used. For the 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. Term structures are written with a leading "constructor," which should be a capitalized word (the same as sort and scheme names), optionally followed by some arguments in ()s, where the declaration gives the sort for each argument (here there are none).

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

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

Note that overlaps are permissible, but it is important to verify determinacy, *i.e.*, if a particular combination of arguments can be subjected to two rules, then they should give the same result! This example satisfies this requirement because the term Unif(Float,Float) can be rewritten by both the rule in line 7 and the rule in line 8, but it does not matter, because the result is the same. Also note that *the order of the rules does not matter*: the three rules in lines 6–8 above can be given in any order.

6.2 EXAMPLE. Consider the complete HACS script in Figure 2. Lines 3–12 define a syntax of expressions with lists, sums, external references, and the Peano convention that 0 stands for itself and *s n* stands for $n + 1$. Line 15 defines a new semantic data sort, Value, which represents the same information but outside of syntax brackets. Line 18 defines a scheme as before, except now it is defined in lines 19–23 also over the data forms with arguments. Lines 25–31 provide a traditional syntax directed translation from the syntactic Exp format to the semantic Value form. However, an attempt to run the Load scheme of the script on a single example like this:

```
$ hacs SZ.hx
...
```

Figure 2: Peano numbers with addition and lists (*examples/SZ.hx*).

```
module org.crsx.hacs.samples.SZ { //−∗−hacs−∗−

  // Input syntax.
  token ID | [a−z]+ [0−9]∗ ;
  main sort Exp
    | ⟦ ⟨Exp@1⟩ : ⟨Exp@2⟩ ⟧@1
    | ⟦ ⟨Exp@2⟩ + ⟨Exp@3⟩ ⟧@2
    | ⟦ ⟨ID⟩ ⟧@3
    | ⟦ 0 ⟧@3
    | ⟦ s ⟨Exp@3⟩ ⟧@3
    | sugar ⟦ (⟨Exp#⟩) ⟧@3 → #
    ;

  // Semantic Values.
  sort Value | Pair(Value, Value) | Plus(Value, Value) | Ref(ID) | Zero | Succ(Value) ;

  // Semantic Operations.
  | scheme Add(Value, Value) ;
  Add(Ref(#id), #2) → Plus(Ref(#id), #2) ;
  Add(Zero, #2) → #2 ;
  Add(Succ(#1), #2) → Succ(Add(#1, #2)) ;
  Add(Pair(#11, #12), Pair(#21, #22)) → Pair(Add(#11, #21), Add(#12, #22)) ;
  Add(Plus(#11, #12), #2) → Plus(#11, Add(#12, #2)) ;

  // Loading input into internal form.
  | scheme Load(Exp) ;
  Load(⟦ ⟨Exp#1⟩ : ⟨Exp#2⟩ ⟧) → Pair(Load(#1), Load(#2)) ;
  Load(⟦ ⟨Exp#1⟩ + ⟨Exp#2⟩ ⟧) → Add(Load(#1), Load(#2)) ;
  Load(⟦ s ⟨Exp#⟩ ⟧) → Succ(Load(#)) ;
  Load(⟦ 0 ⟧) → Zero ;
  Load(⟦ ⟨ID#id⟩ ⟧) → Ref(#id) ;

  // External translation.
  sort Exp | scheme Calc(Exp) ;
  Calc(#) → Unload(Load(#)) ;
  | scheme Unload(Value) ;
  Unload(Zero) → ⟦ 0 ⟧ ;
  Unload(Succ(#)) → ⟦ s ⟨Exp Unload(#)⟩ ⟧ ;
  Unload(Plus(#1, #2)) → ⟦ ⟨Exp Unload(#1)⟩ + ⟨Exp Unload(#2)⟩ ⟧ ;
  Unload(Pair(#1, #2)) → ⟦ ⟨Exp Unload(#1)⟩ : ⟨Exp Unload(#2)⟩ ⟧ ;
  Unload(Ref(#)) → ⟦ ⟨ID#⟩ ⟧ ;
  Unload(Succ(#)) → ⟦ s ⟨Exp Unload(#)⟩ ⟧ ;
}
```

```
$ ./SZ.run --scheme=Load --term="s 0+s 0"
« $Print2-SZ$Value[SZ$Value_Succ[SZ$Value_Succ[SZ$Value_Zero]], 0] »
```

delivers the right result, which is the Peano numeral for 2, however, *it is not printed properly*: the Succ constructors show up in internal form. This is because *semantic data structures have no syntax and therefore cannot be printed*. The proper way is to use the Calc script, which translates back to external form, like this:

```
$ hacs SZ.hx
...
$ ./SZ.run --scheme=Calc --term="s 0+s 0"
 s  s  0
```

which generates the correct result. This uses the Calc scheme defined in lines 34–35 and the Unload scheme from line 37 of Figure 2.

However, while HACS definitions for schemes and data sorts look like function definitions and algebraic data types, they are evaluated differently than those would be in functional programming. Specifically, HACS allows rules to match and be applied *before* the matched arguments have been evaluated, which can give results that are surprising to the uninitiated. Specifically, HACS is neither "eager" nor "lazy," although it is closer to the latter.

Consider this context:

sort Bool | True | False | **scheme** Or(Bool, Bool);

Defining the Or scheme as follows

Or(False, False) → False ; //1
default Or(#1, #2) → True ; //2 problematic

is only correct under one condition: that the arguments to Or are never computed by schemes. Consider, for example, the term

Or(Or(False, False), False)

For this term, HACS is allowed to decide that //1 cannot be immediately applied, because *at this time* the first parameter is different from False, and HACS may then decide to instead use //2. This is probably not what is intended.

The best way to avoid this is to fully expand the observed parameters for Or, which leads to the classic definition,

Or(False, #2) → #2 ;
Or(True, #2) → True ;

With rules that check for equality between several parameters, this requires further care. Consider, for example, the following additions that search a list for a specific boolean value:

sort Bools | Cons(Bool, Bools) | Nil ;
sort Bool | **scheme** Find(Bools, Bool) ;
Find(Cons(#*b*, #*bs*), #*b*) → True ; //3
Find(Cons(#*b*, #*bs*), #*b2*) → Find(#*bs*, #*b2*) ; //4 problematic
default Find(#*bs*, #*b*) → False ; //5 problematic

These rules are problematic for two reasons. First, HACS can *always* pick rule //4 over rule //3, and in this way give a false negative, as will be discussed below. Second, if the list is computed, then the **default** rule can be accidentally used. Consider an expression like

Find(Append (...), True)

(with some suitable definition of Append). Because this does not immediately match //3+4, HACS can decide *at that time* to apply rule //5, which of course is wrong.

A fix for the second issue is to instead use

```
Find(Nil, #b) → False ;                                    //6
Find(Cons(#b, #bs), #b) →  True ;                          //7 careful
default Find(Cons(#b, #bs), #b2) →  Find(#bs, #b2) ;       //8
```

This avoids issues where the list parameter is unevaluated: they will not match any rule, so no wrong simplification is done.

However, it still requires care to ensure that *both* of the compared pieces in //7 are *always* in data form. To see why, consider an expression like

```
Find(Cons(Or (...), ...), True)
```

Again, this does not immediately fit //7 without evaluating the Or(...) subterm, so HACS may instead use //8, leading to a false negative result.

The fix to this is to guarantee that elements of a list are always data, and not unevaluated. One way to achieve this is to use a special constructor when building such lists. If the expression is

```
Find(EvalCons(Or (...), ...), True)
```

with

```
sort Bools | scheme EvalCons(Bool, Bools) ;
EvalCons(T, #bs) →  Cons(T, #bs) ;
EvalCons(F, #bs) →  Cons(F, #bs) ;
```

then the problem does not occur, because the Or is forced to be evaluated before the value is stored in the list. With the use of EvalCons instead of Cons everywhere, //6–8 are safe to use. Note that //3–5 are still not safe because they may observe an unevaluated EvalCons, which would still allow picking the wrong rule. The EvalCons approach has the advantage of stating explictly what is forced.

Finally, the above is so common that it is supported with a special declaration that can be added to achieve the same effect: if the rules are written as

```
Find(Nil, #b) → False ;
[data #b] Find(Cons(#b, #bs), #b) →  True ;
default Find(Cons(#b, #bs), #b2) →  Find(#bs, #b2) ;
```

with the *option prefix* [data #b], then the rule will force complete evaluation of the #b component before it is determined that the second rule does not apply and thus that the default rule may be used. Indeed it is seen as good practice to use the data option for all instances where two subterms are compared for equality.

7. SYNTHESIZING INFORMATION

HACS has special support for assembling information in a "bottom-up" manner, corresponding to the use of *synthesized attributes* in compiler specifications written as syntax-directed definitions (SDD), also known as attribute grammars. This section explains *how* to convert any SDD synthetic attribute definition into one appropriate for HACS, introducing the necessary HACS formalisms along the way. You can find further details in the appendix Manual A.6.

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

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

(E1)

The rule is what the Dragon book calls "S-attributed" because it exclusively relies on synthesized attributes. This allows expression of the rule 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 of sort Type. Attributes are always named with lowercase names.

2. Second, create patterns from the abstract syntax production, as in the previous section: the pattern for the single production is

   ⟦ ⟨E#1⟩ + ⟨E#2⟩ ⟧

   using the subscripts from (E1) as #-disambiguation marks.

3. Next add in *synthesis patterns* for the attributes to be read. 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 this example,

   ⟦ ⟨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 ;), to give

   ⟦ ⟨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 (abstract syntax) 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*."

Assuming that Unif refers to the semantic scheme defined in Example 6.1, the process is complete.

7.2 NOTATION (value synthesis rules)**.**

1. Synthesized simple attributes are declared with declarations like **attribute** ↑ $a(S)$; with a a lowercase attribute name and S the sort name.
2. The synthesized attribute a is associated with a sort by adding the pseudo-production | ↑ a to the sort declaration.
3. Synthesis rules as discussed here have the form $p \uparrow a(r)$, where p is like a pattern in a rule, but with the requirement that p be a *data* instance (not a scheme); a is an attribute name; and r should be a replacement of the value sort of a.

7.3 EXAMPLE. Example 4.1 presented the abstract syntax of the small language processed by *First.hx*. A type analysis of the expressions of the language (for now excluding variables) might look as follows as a standard SDD (syntax-directed definition), where E is the Exp nonterminal and is associated with one attribute: $E.t$ is the synthesized Type of the expression E. In the notations of [2], the SDD can be specified something like this:

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

where it is again assumed that Unif is defined as discussed in Example 6.1. Convert this SDD to the following HACS (using the proper names for the sorts as actually found in *First.hx*):

Figure 3: Synthesizing the value of a Boolean expression (*examples/Bool.hx*).

```
// Boolean algebra.                                                          1
module org.crsx.hacs.samples.Bool {                                          2

  // Syntax.                                                                 4
  sort B                                                                     5
    | ⟦ t ⟧@4| ⟦ f ⟧@4                          // true and false constants     6
    | ⟦ ¬ ⟨B@3⟩ ⟧@3                             // negation                    7
    | ⟦ ⟨B@3⟩ & ⟨B@2⟩ ⟧@2                  // conjunction                 8
    | ⟦ ⟨B@2⟩ | ⟨B@1⟩ ⟧@1                  // disjunction                 9
    | sugar ⟦ ( ⟨B#⟩ ) ⟧@4→  B#                 // parenthesis             10
    ;                                                                        11

  // Main: evaluate Boolean expression.                                      13
  main sort B  | scheme Eval(B) ;                                            14
  Eval(# ↑b(#b)) →  #b ;                                                     15

  // Actual evaluation is a synthesized attribute.                           17
  attribute ↑b(B);                                                           18
  sort B | ↑b ;                                                              19

  // Constants.                                                              21
  ⟦t⟧ ↑b(⟦t⟧) ;                                                              22
  ⟦f⟧ ↑b(⟦f⟧) ;                                                              23

  // Disjunction.                                                            25
  ⟦ ⟨B#1 ↑b(#b1)⟩  | ⟨B#2 ↑b(#b2)⟩ ⟧  ↑b(Or(#b1, #b2)) ;                   26
  | scheme Or(B, B) ;  Or(⟦t⟧, #2) →  ⟦t⟧ ; Or(⟦f⟧, #2) →  #2 ;              27

  // Conjunction.                                                            29
  ⟦ ⟨B#1 ↑b(#b1)⟩  & ⟨B#2 ↑b(#b2)⟩ ⟧  ↑b(And(#b1, #b2)) ;                   30
  | scheme And(B, B) ;  And(⟦t⟧, #2) →  #2 ; And(⟦f⟧, #2) →  ⟦f⟧ ;           31

  // Negation.                                                               33
  ⟦ ¬ ⟨B# ↑b(#b)⟩ ⟧  ↑b(Not(#b)) ;                                           34
  | scheme Not(B) ;  Not(⟦t⟧) →  ⟦f⟧ ;  Not(⟦f⟧) →  ⟦t⟧ ;                    35

}                                                                            37
```

```
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 from the abstract syntax patterns and synthesis semantic rules, as discussed above.

7.4 EXAMPLE. Figure 3 shows an implementation of Boolean algebra implemented with synthesized attributes. Notice how the main Eval scheme is defined to "request" the value of the synthesized attribute, which then triggers the evaluation of the Boolean expression. An example run would be

```
$ hacs Bool.hx
...
$ ./Bool.run --scheme=Eval --term='t|(¬f)'
 t
```

Finally, note that in the case of multiple synthetic attributes, 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 (see below for examples of such systems).

8. FULL SYNTAX-DIRECTED DEFINITIONS WITH ENVIRONMENTS

Descriptions to this point have used "top-down" recursive schemes with positional parameters and "bottom-up" synthesis of named attributes with simple values. The last component used in traditional compiler specification is the use of *inherited* named attributes, which are distributed top-down, like parameters. HACS supports this with a hybrid combination of implicit named parameters for recursive schemes. The section introduces features of HACS covered in the appendix Manual A.6.

One of the main uses of inherited attributes in formal compiler specifications is *symbol management*. The focus here is on the HACS notion of *environment*, which fills this niche and cannot be easily achieved with usual positional arguments to recursive schemes.

8.1 EXAMPLE. Compiler construction formalization expresses the use of a symbol table using an SDD with an inherited attribute that for each node in the AST associates the appropriate symbol table for that node. Consider the following three simple semantic rules, which demonstrate this approach:

PRODUCTION	SEMANTIC RULES	
$S \to T_1\ \mathbf{id}_2 = E_3; S_4$	$E_3.e = S.e; S_4.e = \text{Extend}(S.e, \mathbf{id}_2.sym, T_1)$	(1)
$E \to E_1 + E_2$	$E_1.e = E.e; E_2.e = E.e$	(2)
$\mid \mathbf{id}_1$	$E.t = \text{Lookup}(E.e, \mathbf{id}_1.sym)$	(3)
$\mid \mathbf{num}_1$		(4)

Rule (1) handles declarations in the toy language: it creates a new environment (or symbol table) that *extends* the environment from the context, the *inherited* environment $S.e$, with a new coupling from the symbol of $\mathbf{id}_2$

to the type T_1: the notation means that the new environment $S_4.e$ contains the new mapping as well as all the bindings from $S.e$ (*except* any previous mapping of the symbol $\mathbf{id}_2$). Rule (2) merely expresses that the e attribute is inherited from the context to both subexpressions of addition expresions. Rule (3) uses the environment to attach a synthesized attribute t onto an E that contains an **id** token (but is not otherwise used in these rules). Specifically, the notation is meant to suggest that the type value is obtained by a *lookup* of the mapping for the symbol of $\mathbf{id}_1$. Finally, rule (4) is included to exemplify what to do with rules that do not propagate the environment.

Here are the steps to translate the above SDD fragment to HACS.

1. First, encode the grammar (assuming an existing T sort of types and ID tokens as in the previous section):

   ```
   sort S  | ⟦ ⟨T⟩ ⟨ID⟩ = ⟨E⟩ ; ⟨S⟩ ⟧ ;
   sort E  | ⟦ ⟨E⟩ + ⟨E@1⟩ ⟧  | ⟦ ⟨ID⟩ ⟧@1;
   ```

   (Assume that the T sort is defined elsewhere.)

2. Having defined the grammar, declare the new attribute:

   ```
   attribute ↓e{ID : T} ;
   ```

   Like synthesized attributes, inherited attributes are always given lowercase names. The {ID:T} part declares the value of the attribute to be a *mapping* from values (token strings) of ID sort to values of T sort. Such mappings take the role of symbol tables from traditional compilers.

3. Second, associate the inherited attribute to one or more *recursive schemes*, which will be responsible for propagating that inherited attribute over values of a certain sort. This has to be done by inventing a separate scheme for each combination of a sort and an inherited attribute:

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

   As can be seen, the scheme generates results of the S and E sorts, in each case taking a single argument of the same sort, and for each indicating with a ↓ that the scheme *carries* the associated inherited attribute. Notice that, unlike for synthesized declarations, there is no | in front of the attribute, because the attribute itself is specific to the scheme, not a declaration for the entire sort.

4. Next, encode the simplest rule, (2). As in the previous section, observe that (2) operates on sums, which leads to a pattern like the following, using the subscripts from (2):

   ```
   ⟦ ⟨E#1⟩ + ⟨E#2⟩ ⟧
   ```

   Now insert the pattern into the scheme, as was done for recursive syntax-directed schemes:

   ```
   Ee(⟦⟨E#1⟩ + ⟨E#2⟩⟧)
   ```

   This clearly respects the sort constraints defined above, with Ee being applied to an E expression. Since there are no complicated dependencies in (2), the process is almost complete. All that is left to do is create a rule that, on the right side of the → applies the Ee scheme recursively to the subexpressions that should inherit the *e* attribute, which are the two marked subexpression, *inside the syntax markers*:

   ```
   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).

   Also notice that for inherited attributes *all* the equations are handled by *one* rule, unlike for synthesized, where there is a synthesis rule for every equation.

5. Next, consider rule (3), which defines the synthesized attribute t on **id**-expressions using a "Lookup" meta-function that is meant to suggest extracting the mapping from the $E.e$ environment inherited from the context. Begin with a template like this:

   $\mathrm{Ee}(\llbracket\langle \mathrm{ID\#1}\rangle\rrbracket) \;\to\; \llbracket\langle \mathrm{ID\#1}\rangle\rrbracket$;

   which just states that the e attribute is inherited "into" a name subterm (through Ee), essentially making $E.e$ available, but note that the rule does not use the inherited attribute. In particular, this rule fails to actually set the t attribute. Fixing this entails first capturing the "Lookup" in the inherited environment. HACS has a special notation for this. Write the pattern as follows:

   $\mathrm{Ee}(\llbracket\langle \mathrm{ID\#1}\rangle\rrbracket) \downarrow e\{\#1 : \#t\} \;\to\; \ldots$

   The pattern of this rule is equipped with a *mapping constraint* on the *e* inherited attribute, which corresponds to the Lookup notation of (3) above. It will match when Ee is applied to a name, called #1, which is *also* mapped by the associated *e* attribute to a type, denoted #*t*. After capturing the mapped type this way, complete the rule by explicitly associating it to the *t* attribute associated with ID expressions.

   $\mathrm{Ee}(\llbracket\langle \mathrm{ID\#1}\rangle\rrbracket) \downarrow e\{\#1 : \#t\} \;\to\; \llbracket\langle \mathrm{ID\#1}\rangle\rrbracket \uparrow t(\#t)$;

   Notice how mappings like *e* use {} notation for both declarations and constraints, and simply valued attributes like *t* use ().

6. The semantic rules of the SDD can now be completed by encoding (1). Begin with

   $\mathrm{Se}(\llbracket\langle \mathrm{T\#1}\rangle\ \langle \mathrm{ID\#2}\rangle = \langle \mathrm{E\#3}\rangle;\ \langle \mathrm{S\#4}\rangle\rrbracket) \;\to\; \llbracket\langle \mathrm{T\#1}\rangle\ \langle \mathrm{ID\#2}\rangle = \langle \mathrm{E}\ \mathrm{Ee(\#3)}\rangle;\ \langle \mathrm{S}\ \mathrm{Se(\#4)}\rangle\rrbracket$;

   which merely establishes that the basic rules that the $S.e$ attribute inherited from the context (through the left hand Se) are passed to both E_3 and S_4 through the right Ee and Se, respectively. This captures everything except the "Extend" notation. HACS supports this by allowing mapping constraints also on the right side of the $\to$ acting as *extensions* of the map.

   $\mathrm{Se}(\llbracket\langle \mathrm{T\#1}\rangle\ \langle \mathrm{ID\#2}\rangle = \langle \mathrm{E\#3}\rangle;\ \langle \mathrm{S\#4}\rangle\rrbracket)$
   $\to\; \llbracket\langle \mathrm{T\#1}\rangle\ \langle \mathrm{ID\#2}\rangle = \langle \mathrm{E}\ \mathrm{Ee(\#3)}\rangle;\ \langle \mathrm{S}\ \mathrm{Se(\#4)}\ \downarrow e\{\#2 : \#1\}\rangle\rrbracket$;

   Consider carefully how the $\llbracket \ldots \rrbracket$ and $\langle \ldots \rangle$ nest: the first set wraps syntax fragments, and the second wraps raw (non-syntax) fragments inside syntax. Attribute constraints are always in the raw fragments.

7. Finally, (4) does not contain any semantic rules but a case must still be provided for the scheme:

   $\mathrm{Ee}(\llbracket\langle \mathrm{NUM\#1}\rangle\rrbracket) \;\to\; \llbracket\langle \mathrm{NUM\#1}\rangle\rrbracket$;

Furthermore, if the SDD in question has more semantic rules with synthetic attributes, then all of the above rules should be equipped with a special ↑#*syn* marker to preserve *all* synthesized attributes through the inheritance scheme, which makes the whole system look as follows.

$\mathrm{Se}(\llbracket\langle \mathrm{T\#1}\rangle\ \langle \mathrm{ID\#2}\rangle = \langle \mathrm{E\#3}\rangle;\ \langle \mathrm{S\#4}\rangle\rrbracket \uparrow \#syn)$
$\to\; \llbracket\langle \mathrm{T\#1}\rangle\ \langle \mathrm{ID\#2}\rangle = \langle \mathrm{E}\ \mathrm{Ee(\#3)}\rangle;\ \langle \mathrm{S}\ \mathrm{Se(\#4)}\ \downarrow e\{\#2 : \#1\}\rangle\rrbracket \uparrow \#syn$;

$\mathrm{Ee}(\llbracket\langle \mathrm{E\#1}\rangle + \langle \mathrm{E\#2}\rangle\rrbracket \uparrow \#syn) \;\to\; \llbracket\langle \mathrm{E}\ \mathrm{Ee(\#1)}\rangle + \langle \mathrm{E}\ \mathrm{Ee(\#2)}\rangle\rrbracket \uparrow \#syn$;
$\mathrm{Ee}(\llbracket\langle \mathrm{ID\#1}\rangle\rrbracket \uparrow \#syn) \downarrow e\{\#1 : \#t\} \;\to\; \llbracket\langle \mathrm{ID\#1}\rangle\rrbracket \uparrow \#syn \uparrow t(\#t)$;
$\mathrm{Ee}(\llbracket\langle \mathrm{NUM\#1}\rangle\rrbracket \uparrow \#syn) \;\to\; \llbracket\langle \mathrm{NUM\#1}\rangle\rrbracket \uparrow \#syn$;

One important constraint for inherited attributes is that there is *precisely one* rule for the carrier scheme of the attribute per syntax case. This has an interesting consequence: *if a rule mixes several inherited attributes, those inherited attributes must share the same carrier.*

The addition of inherited attributes completes the list of possible attributes forms.

Table 2: Attribute constraints.

NOTATION	IN PATTERN	IN RESULT
$D{\uparrow}a(\#)$	D must synthesize a-value in #	D will synthesize a-value in #
$D{\uparrow}a\{:\#\}$	— a-environment in :#	— a-environment in :#
$D{\uparrow}a\{\#k : \#v\}$	— — which includes binding #k to #v	— — which includes binding #k to #v
$D{\uparrow}a\{\neg\#k\}$	— — which does not have binding for #k	n/a
$D{\uparrow}a\{\}$	n/a	D synthesizes empty environment in a
$D{\uparrow}\#$	*All* D-synthesized attributes in #	*All* attributes in # D-synthesized
$F{\downarrow}a(\#)$	F must inherit a-value in #	F will inherit a-value in #
$F{\downarrow}a\{:\#\}$	— a-environment in :#	— a-environment in :#
$F{\downarrow}a\{\#k : \#v\}$	— — which includes binding #k to #v	— — which includes binding #k to #v
$F{\downarrow}a\{\neg\#k\}$	— — which does not have binding for #k	n/a
$F{\downarrow}a\{\}$	n/a	F inherits empty environment in a

8.2 NOTATION (attributes)**.**

1. Attributes are defined with attribute declarations:

$$\textbf{attribute}\ \left\{\begin{matrix}\uparrow\\ \downarrow\end{matrix}\right\}\ a\ \left\{\begin{matrix}(\,S\,)\\ \{\,S'\,\}\\ \{\,S' : S\,\}\end{matrix}\right\}\ ;$$

   where each braced unit represents a choice:

   - The arrow determines whether to define a synthetic (↑) or inherited (↓) attribute.
   - a is the attribute name, a lowercase identifier.
   - The last unit gives the category and sort(s) of the attribute: (S) is a simply valued attribute of sort S, $\{S'\}$ is an attribute with a set of S'-members, and $\{S' : S\}$ is an attribute with a map from S'-values to S-values. For sets and maps, the S' sort must be a token sort (or a sort with a single **symbol** case, as explained in the next section).

2. With D denoting data terms and F "function" terms, *i.e.*, applied schemes, yields the conventions of Table 2 in HACS rules. Note that one can have several constraints on a single (data or function) term, as long as they are all properly declared for the appropriate sort or scheme, respectively.

8.3 EXAMPLE. Figure 4 illustrates how an environment can be synthesized and then inherited. The HACS script implements variable substitutions of mutually recursive bindings, which is essentially achieved by two "passes," one for synthesizing an environment that contains the collected bindings, and a second pass that actually distributes and applies the collected (now inherited) environment to the target variable. The key rule is in line 22, where the environment that has been synthesized in *b* is then copied over to the inherited attribute *e* to the Apply scheme.

The following code tests that the lookup mechanism is truly recursive:

```
$ hacs LetrecMap.hx
...
$ ./LetrecMap.run --scheme=Reduce --term="a:b b:c in a"
c
$ ./LetrecMap.run --scheme=Reduce --term="b:c a:b in a"
c
```

Figure 4: Synthesizing and then inheriting an environment (*examples/LetrecMap.hx*).

```
module org.crsx.hacs.samples.LetrecMap {                                    1
  // Syntax.                                                                2
  token ID | [a−z][a−z0−9_]* ;                                              3
  sort B | ⟦ ⟨ID⟩ : ⟨ID⟩ ⟨B⟩ ⟧ | ⟦⟧ ;                                      4
  main sort P | ⟦ ⟨B⟩ in ⟨ID⟩ ⟧ ;                                          5
  sort Out | ⟦⟨ID⟩⟧;                                                        6

  // Synthesize environment.                                                8
  attribute ↑b{ID:ID} ;                                                     9
  sort B | ↑b ;                                                             10
  ⟦ ⟨ID#v1⟩ : ⟨ID#v2⟩ ⟨B#B ↑b{:#b}⟩ ⟧ ↑b{:#b} ↑b{#v1 : #v2} ;              11
  ⟦ ⟧ ↑b{} ;                                                                12

  // Environment and application on variable.                               14
  attribute ↓e{ID:ID} ;                                                     15
  sort Out | scheme Apply(ID) ↓e ;                                          16
  Apply(#v) ↓e{#v : #v2} → Apply(#v2) ;                                     17
  Apply(#v) ↓e{¬#v} → ⟦ ⟨ID#v⟩ ⟧ ;                                          18

  // Main makes sure list is synthesized and passes control to conversion.  20
  sort Out | scheme Reduce(P) ;                                             21
  Reduce(⟦ ⟨B#B ↑b{:#b}⟩ in ⟨ID#v⟩ ⟧) → Apply(#v) ↓e{:#b} ;                  22
}                                                                           23
```

Figure 5: SDD for type checking.

Production	Semantic Rules	
$S \to \textbf{id} := E_1; S_2$	$E_1.e = S.e; S_2.e = \text{Extend}(S.e, \textbf{id}.sym, E_1.t)$	(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)
$\mid$ **id**	$E.t =$ **if** $\text{Defined}(E.e, \textbf{id}.sym)$ **then** $\text{Lookup}(E.e, \textbf{id}.sym)$ **else** TypeError	(E5)

Figure 6: HACS code for type analysis.

```
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{ID:Type}; // inherited type environment                                   18
sort Exp | scheme Ee(Exp) ↓e ; // propagates e over Exp                                19

// These rules associate t attribute with variables (missing case above).              21
Ee(⟦⟨ID#v⟩⟧) ↓e{#v : #t} → ⟦⟨ID#v⟩⟧ ↑t(#t);                                          22
Ee(⟦⟨ID#v⟩⟧) ↓e{¬#v} → error⟦Undefined identifier⟧;                                    23

Ee(⟦⟨Exp#1⟩ + ⟨Exp#2⟩⟧ ↑#syn) → ⟦ ⟨Exp Ee(#1)⟩ + ⟨Exp Ee(#2)⟩ ⟧ ↑#syn ;              25
Ee(⟦⟨Exp#1⟩ * ⟨Exp#2⟩⟧ ↑#syn) → ⟦ ⟨Exp Ee(#1)⟩ * ⟨Exp Ee(#2)⟩ ⟧ ↑#syn ;              26
Ee(⟦⟨INT#⟩⟧ ↑#syn) → ⟦⟨INT#⟩⟧ ↑#syn ;                                                  27
Ee(⟦⟨FLOAT#⟩⟧ ↑#syn) → ⟦⟨FLOAT#⟩⟧ ↑#syn ;                                              28

sort Stat | scheme Se(Stat) ↓e ; // propagates e over Stat                             30

Se(⟦⟨ID#v⟩ := ⟨Exp#1⟩; ⟨Stat#2⟩⟧ ↑#syn)                                               32
  → SeB(⟦⟨ID#v⟩ := ⟨Exp Ee(#1)⟩; ⟨Stat#2⟩⟧ ↑#syn);                                    33
{                                                                                      34
  | scheme SeB(Stat) ↓e; // helper scheme for assignment after expression type analysis 35
  SeB(⟦⟨ID#v⟩ := ⟨Exp#1 ↑t(#t1)⟩; ⟨Stat#2⟩ ⟧ ↑#syn)                                   36
   → ⟦⟨ID#v⟩ := ⟨Exp#1⟩; ⟨Stat Se(#2) ↓e{#v : #t1}⟩⟧ ↑#syn ;                          37
}                                                                                      38

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

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

The examples presented thus far have very simple attribute dependencies. A final example demonstrates more complicated attribute dependencies.

8.4 EXAMPLE. Figure 5 shows a more realistic SDD for type checking, and Figure 6 shows the corresponding HACS. Here are the steps followed to obtain this script:

1. Lines 1–7 define the Type semantic data sort along with the helper Unif semantic function as previously, except here extended with a TypeError case.

2. Lines 9 and 10 define the synthesized type t on expression data, and lines 12–15 give the type synthesis rules except for typing variables, to be described later. This makes it clear that *type synthesis cannot happen until variable occurrences are typed.* In the SDD, this corresponds to all the t-assignments except the one in (E5), which depends on $E.e$.

3. Line 18 declares the inherited environment attribute, and line 19 associates it with the recursive scheme Ee on expressions.

4. Lines 21–28 give the environment propagation rules for expressions. Specifically notice how there are two cases for identifier, line 22 and 23, corresponding to whether the identifier is defined in the environment or not, with the latter resulting in an **error** message in special HACS form. Also notice how the recursive rules in lines 25–28 take care to preserve all synthesized attributes on the terms that are traversed by using a catch-all ↑*#syn* constraint.

5. In total, lines 9–28 fully capture rules (E1–E5). The HACS specification adds a condition on evaluation: first apply the environment, and then synthesize the type.

6. Line 30 declares a recursive scheme carrying the *e* attribute over statements, and lines 40 and 42 are simple recursive rules for the propagation of *e* corresponding to (S2–S3).

7. Rule (S1) is slightly more complicated, because the inherited attribute has non-trivial dependencies. It is essential to know the dependency relationship of the attributes to devise a *recursive strategy* for the attribute evaluation. Recall 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, one has to evaluate (S1) 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 $S_3.e = \mathrm{Extend}(S.e, \mathbf{id}_1.sym, E_2.t)$.

   These two steps are achieved by having an extra carrier scheme, SeB, which is locally declared, so that Se and SeB can handle the two steps above: first Se, in lines 32–33, copies *e just* to E_1 (from (S1)), and then *chains* to SeB.

8. The helper scheme, SeB declared in line 35, is set up to *wait* for the synthesized t attribute (line 36) to be computed for E_1 and only *then* replaces the term with a new one to compute the S_2 part with an extended environment (line 37).

The environment helpers become translated into the native HACS environment patterns from Table 2 as follows:

- A "$\mathrm{Defined}(N.e, x)$" test is encoded by having two rules: one for the "true" branch with the constraint ↓*e*{*#v*} in a pattern, and one for the "false" case with the constraint ↓*e*{¬*#v*} in the pattern.
- "$\mathrm{Lookup}(N.e, x)$" is encoded by adding a constraint ↓*e*{*#v*:#} in a pattern, which then binds the meta-variable # to the result of the lookup. (This will imply the "defined" pattern discussed above.)
- "$\mathrm{Extend}(N.e, x, V)$" is encoded by adding a constraint ↓*e*{*#v*:V} in the replacement.

Figure 7: λ calculus (*examples/Lambda.hx*).

```
module org.crsx.hacs.samples.Lambda {                                          1
  space [ \t\n] ;                                                              2
  token ID | [a−z] [_0−9]* ;                                                   3

  main sort T                                                                  5
  | ⟦λ ⟨ID binds x⟩ . ⟨T[x as T]⟩⟧                  // abstraction            6
  | scheme ⟦⟨T@1⟩ ⟨T@2⟩⟧@1                          // application             7
  | symbol ⟦⟨ID⟩⟧@2                                 // variable occurrence     8
  | sugar ⟦(⟨T#⟩)⟧@2 → # ;                                                     9

  ⟦(λ x . ⟨T#1[T⟦x⟧]⟩) ⟨T#2⟩⟧ → #1[T#2] ;                                      11
}                                                                              12
```

9. HIGHER ORDER ABSTRACT SYNTAX

The examples thus far have analyzed and translated abstract syntax trees with structure and symbols. The next order of business is to construct *new* structures with new scopes, or copied fragments with scopes inside. These are the features where the "H" of HACS matter.

9.1 EXAMPLE (untyped λ calculus)**.** A simple example of a system that does rewriting of "higher order" terms is the classic λ calculus. The λ calculus is specified in HACS as shown in Figure 7.

- The specification has properties like others presented above: it is a *module* (lines 1 and 12); it has a lexical specification of *spacing* (line 2) and *identifiers* (line 3); it has a *main sort* (line 5) of terms T; the term grammar uses @ precedence markers (lines 7–9) and syntactic sugar for grouping (line 9); the term has one *scheme*, which means that there can be rules for application (line 7), which is so basic in λ calculus that it is denoted by simple concatenation.
- The first syntax production (line 6) is different: "| ⟦λ⟨ID **binds** *x*⟩. ⟨T[*x* **as** T]⟩⟧" has several new constructs inside the syntax brackets:
  - The first, ⟨ID **binds** *x*⟩, is just like ⟨ID⟩ by itself but marks this particular identifier as being a *binder*, which is used in a special way, and for the scope of the production is named *x* (in case there is more than one).
  - The second, ⟨T[*x* **as** T]⟩, is just like ⟨T⟩ by itself but marks this particular instance of T as the *scope* for the binder named *x*, with the added information that all occurrences of the bound variable should (also) be considered to be of sort T.
- The third production (line 8) is also special: it declares that if a term is a simple ID, then in fact it is a *symbol*, which is required for identifiers to occur bound as sort T.
- Finally, the defined scoping in the classic β rewrite rule is used for our one scheme in line 11: it expresses that a λ term which is an application (of the scheme declared in line 7) where the first subterm is a λ-abstraction (declared in line 6) can be rewritten.

The sample can be run the usual way:

```
$ hacs Lambda.hx
...
```

```
$ ./Lambda.run --sort=T --term='(λx.x)y'
 y
```

More formally, HACS provides support for manipulating scoped terms using *higher-order abstract syntax* (HOAS) [20] through some new constructs:

- Outside syntax, HACS recognizes identifiers starting with a lowercase letter as *variables*.
- In concrete syntax grammar productions, for a token T and a HACS variable x, one can use the special reference $\langle T$ **binds** $x\rangle$ to define that this particular instance of T is a *binder*, with the variable x as the label to indicate the scoping.
- In a grammar production with a $\langle T$ **binds** $x\rangle$ reference, there can be one reference of the form $\langle S[x$ **as** $S']\rangle$. This is like a reference $\langle S\rangle$ but with the added information that all occurrences of the binder labeled x *must* occur inside the S subterm, and furthermore that the occurrences will have the sort S'. This implies that there must in addition be a grammar production like **sort** S' | **symbol** $[\![\langle T\rangle]\!]$, to allow the occurrences to be properly parsed.
- Rules *must* use the native variable form (no $\langle\rangle$s) for binders.
- Patterns (left of $\rightarrow$) should include references to scoped subterms using a special notation where scoped meta-variables are "applied" to the bound variables in [ ]s, looking something like this: $[\![\ldots\langle S\#[S'[\![x]\!]]\rangle\ldots]\!]$.
- Replacements (right of $\rightarrow$) should always have scoped metavariables applied to the same number of arguments in [ ]s as the corresponding pattern sets up. The result of such an application is to replace all bound occurrences of the variable that was indicated in the pattern with whatever is used in the replacement.

The formal rules are rather dense, however, as will now become apparent with some examples, their use is really just a compact way of encoding the usual notions of binders and parameter substitution that are familiar from ordinary programming.

9.2 EXAMPLE. Consider the following variation of the grammar in Example 4.1, which makes the scoping rules of this little assignment language explicit.

```
main sort Stat   | [[ ⟨ID binds v⟩ := ⟨Exp⟩ ; ⟨Stat[x as Exp]⟩ ]]                1
                 | [[]] ;                                                       2

sort Exp   | [[ ⟨Exp⟩ + ⟨Exp@1⟩ ]]                                             4
           | [[ ⟨Exp@1⟩ * ⟨Exp@2⟩ ]]@1                                         5
           | [[ ⟨INT⟩ ]]@2                                                     6
           | [[ ⟨FLOAT⟩ ]]@2                                                   7
           | symbol [[⟨ID⟩ ]]@2                                                8
           | sugar [[ ( ⟨Exp#⟩ ) ]]@2→ Exp# ;                                  9
```

The HOAS constructs are only present in lines 1 and 8 of the grammar, and the difference may seem irrelevant. However, consider these simple rules that *duplicate* and *append* statements (such a rule may be useful in loop hoisting code, for example):

```
sort Stat | scheme Duplicate(Stat) | scheme Append(Stat,Stat) ;                10
Duplicate(#S) → Append(#S, #S) ;                                               11
Append([[]], #S2) → #S2 ;                                                      12
Append([[ old := ⟨Exp#2⟩; ⟨Stat#3[old]⟩ ]], #S2)                               13
  → [[ new := ⟨Exp#2⟩; ⟨Stat Append(#3[new], #S2⟩ ]] ;                          14
```

Figure 8: Definitions of Call-by-Value Continuation Passing Style (*examples/CPS.hx*).

```
// samples/CPS.hx: Continuation-passing for 2-level λ-calculus in -*-hacs-*-                1
module org.crsx.hacs.samples.CPS {                                                          2

  // Tokens.                                                                                4
  space [ \t\n] ;                                                                           5
  token ID | [a−z] [_0−9]* ;  // single-letter identifiers                                  6

  // λ Calculus Grammar.                                                                    8
  main sort E                                                                               9
  | ⟦ λ ⟨ID binds x⟩ . ⟨E[x as E]⟩ ⟧ | ⟦ ⟨E@1⟩ ⟨E@2⟩ ⟧@1                                   10
  | symbol ⟦ ⟨ID⟩ ⟧@2| sugar ⟦ ( ⟨E#⟩ ) ⟧@2→# ;                                             11

  // One-pass CBV CPS.                                                                      13
  | scheme CPS(E) ;                                                                         14
  CPS(#) → ⟦ λk.{⟨E#⟩ | m.k m} ⟧ ;                                                          15
  | scheme ⟦{⟨E⟩ | ⟨ID binds m⟩ . ⟨E[m as E]⟩} ⟧@2;                                        16
  ⟦ {v | m.⟨E#F[m]⟩} ⟧ → #F[v] ;                                                            17
  ⟦ {λx.⟨E#[x]⟩ | m.⟨E#F[m]⟩} ⟧→#F[E⟦ λx.λk.{⟨E#[x]⟩ | m.k m} ⟧] ;                         18
  ⟦ {⟨E#0⟩ ⟨E#1⟩ | m.⟨E#F[m]⟩} ⟧→⟦{⟨E#0⟩ | m.{ ⟨E#1⟩ | n.m n (λa.⟨E#F[a]⟩ )}} ⟧;          19
}                                                                                           20
```

Notice how the pattern of the last rule in lines 13–14 explicitly calls out the used binding: the actual variable that is used in the program is referenced in the pattern as *old*, and the pattern matching should also keep track of all the occurrences inside the scope, which is #3, by writing it as #3[*old*]. (This exploits the aforementioned HACS hack, where HACS variables that are also "raw" variables can be written in this short form instead of the full #3[Exp⟦*old*⟧].) This setup makes it possible to now *systematically replace the symbol* with a new and fresh one, referred to as *new* in the replacement in line 16, but which the system will in fact replace with a new and unique name, both the binder *and all the in-scope occurrences* inside #3. This ensures that there is no so-called "variable capture," *i.e.*, that occurrences of the *old* name inside #S2 accidentally become double-bound.

Interestingly, HOAS also makes it impossible to have variables "escape" from their scope. A rule in the old grammar from Example 4.1 might look like

Flip(⟦ ⟨ID#*v1*⟩ := ⟨Exp#2⟩; ⟨Stat#3⟩ ⟧) → ⟦ { ⟨Stat#3⟩ } ⟨ID#*v1*⟩ := ⟨Exp#2⟩; ⟧ ;

which will take uses of #*v1* inside #3 and move them outside of their scope. This will give a syntax error possible with HOAS without explicitly substituting the occurrences of the bound variable with something else in the copy, which would expose the error. (You may think that this is an obvious mistake that no one would make, but this sort of variable escaping is a real problem that leads to bugs in compilers.) The use of HOAS in HACS in fact allows the full range of methods used in higher-order rewriting [10, 13], a very powerful mechanism.

9.3 EXAMPLE. Next consider a real example from the literature [6]. Figure 8 shows a system for transforming λ-terms to continuation-passing style with call-by-value evaluation semantics. Lines 14–19 specify a *one-pass* variant of continuation-style conversion. Notice how the transformation defines a syntactic helper scheme to make the recursive composition easier to express.

The script compiles and generates code as follows:

```
$ hacs CPS.hx
...
$ ./CPS.run --scheme=CPS --term="(λx.x x)"
  λ k .  k (  λ x .   λ k_85 .   x x  (  λ m_40 .  k_85 m_40  )  )
```

Finally, sometimes symbols serve as *labels* or for some other purpose, when there is a need for globally unique symbols. This creates a dilemma: writing ⟦*x*⟧ for some symbol sort, how to distinguish between the actual variable written *x* and a variable that is automatically renamed to be globally unique? By default, HACS follows the following rules:

- Every token that is parsed to be of a **symbol** sort is *automatically renamed* to a *fresh* variable name. (If used in a pattern, the fresh name will match the actual symbol that is present in the term, and thus in essence really be a placeholder for an existing symbol.)
- If a rule is prefixed with the option [**global** *x*], then the symbol *x* will not be subject to this automatic conversion.

10. COMPILE-TIME COMPUTATIONS

There is sometimes a need to compute helper values, most commonly for counting. HACS supports this through a dedicated sort, Computed, which has special syntax for operations on primitive values.

The mechanism is quite simple: in rule replacements, arguments of sort Computed can contain expressions in ⟦⟧ of the shape summarized in Table 3. In the table, *E* in general stands for "any expression above the nearest line," however, there are exceptions: the binary operators in the multiplicative and additive groups in the middle have the usual left recursive syntax, so ⟦1+2+3⟧ is the same as ⟦(1+2)+3⟧, and any *E* inside some kind of parenthesis can, in fact, come from the entire table.

Notice that for Computed, meta-variables are part of the syntax: essentially #*x* is used instead of "⟨Computed#*x*⟩," which is not permitted, inside the special syntax.

Also, since *all* computed parts of a rule are evaluated as soon as the rule is expanded, only use this for calculations that can be safely computed in any case where the *pattern* of the rule matches.

10.1 EXAMPLE (count). Consider the list from Example 5.8. The following computes the length of the list, using a helper.

```
sort Computed | scheme ListLength(List) | scheme ListLength2(List, Computed) ;
ListLength(#) → ListLength2(#, ⟦0⟧) ;
ListLength2(⟦ ⟨Elem#1⟩ ⟨List#2⟩ ⟧, #n) → ListLength2(#2, ⟦ #n + 1 ⟧) ;
ListLength2(⟦ ⟧, #n) → #n ;
```

Note how the declaration of the helper sets up the use of the Computed mechanism.

10.2 EXAMPLE (string operations). Figure 9 illustrates working with strings. Notice the following:

- There is a strict distinction between the token WORD and the sort Word. Something of sort Word can be written in syntax brackets—like ⟦X⟧ in the Test rule.
- A map attribute—called *dup* in the example—can have a token sort as the key sort, here WORD.
- The Test2 rules have two cases for processing a word: one for a word that is previously unmapped, and one for a word that was already mapped.
- A *new token is constructed* for each word. This is achieved with a variant of the ⟨WORD. . . ⟩ notation: normally, in such a construction, the . . . must be something of token sort WORD; however, as a special case, it is permissible to use an expression of Computed sort, as in lines 18 and 21 here.

Table 3: Summary of Computed syntax.

Expression	*Explanation*
#x	refer to Computed value or token string *#x*
`$` *#x*	extract integer value from token string *#x*
`%` *#x*	convert integer Computed value *#x* to string value
`"foo"` `42` `0x2A`	literal strings and integers
`(`*E*`)`	grouping
E `[`*E*`:`*E*`]`	substring (optional indices are `[`*first*`:`*after-last*`]` with `0`-based indexing)
`+`*E* `-`*E*	unary integer plus and minus
`~`*E*	bitwise not
`length` *E*	length of (Unicode) string
`up-case` *E*	convert string to upper case
`down-case` *E*	convert string to lower case
`escape` *E*	convert all special characters to their Java escape equivalent
`unescape` *E*	convert all Java escape sequences to special characters
`trim` *E*	remove leading and trailing spacing
T E	generate constants in user's type *T*
E`*`*E* *E*`/`*E* *E*`%`*E*	integer multiplication, division, modulo
E`&`*E*	bitwise and
E`<<`*E* *E*`>>`*E*	bitwise shift left and right
E`+`*E* *E*`-`*E*	integer sum and difference
E`|`*E* *E*`^`*E* *E*`\`*E*	bitwise or, exclusive or, subtraction
E`@`*E*	string concatenation
E`=`*E* *E*`/=`*E*	integer equality and inequality
E`<`*E* *E*`<=`*E*	integer less than and less than or equal
E`>`*E* *E*`>=`*E*	integer greater than and greater than or equal
E `same-as` *E*	string equality
E `contains` *E*	right string is contained as a substring of left
E `starts-with` *E*	right string is a prefix of left
E `ends-with` *E*	right string is a suffix of left
E`?`*E*`:`*E*	ternary test

- Lines 26–27 define the "word concatenation helper" of sort Computed with an expression that concatenates the two tokens as strings. This only works with arguments of token sort.

Here is a run illustrating the effect:

```
$ hacs MakeToken.hx
...
$ ./MakeToken.run --scheme=Test --term="Hello Hello Hello John"
X_Hello_x_x_John
```

Notice that it is *not* presently possible to nest computations, *i.e.*, insert a value in a Computed expression that is computed by user functions (this will change in future versions of HACS). However, it is still possible to use helpers to achieve nested computations, as illustrated by the simple desk calculator in the following example.

10.3 EXAMPLE. Figure 10 implements a simple desk calculator. Notice how it uses the `$` marker to import a token as an integer value of sort Computed. Here is a run:

Figure 9: *examples/MakeToken.hx.*

```
module org.crsx.hacs.tests.MakeToken {
  // Input grammar.
  token WORD | [A-Za-z.!?_]+;
  sort Word | ⟦⟨WORD⟩⟧ ;
  main sort Words | ⟦⟨Word⟩ ⟨Words⟩⟧ | ⟦⟧;

  // Main scheme: concatenate words but replace repetitions with "x".
  sort Word | scheme Test(Words);
  Test(#ws) → Test2(#ws, ⟦X⟧);

  // Map encountered words to "x".
  attribute ↓dup{WORD : WORD};

  // The aggregator.
  sort Word | scheme Test2(Words, Word) ↓dup ;

  Test2(⟦ ⟨WORD#w1⟩ ⟨Words#ws⟩ ⟧, ⟦⟨WORD#w⟩⟧) ↓dup{¬#w1}
  → Test2(#ws, ⟦⟨WORD Concat(#w, #w1)⟩⟧) ↓dup{#w1 : ⟦x⟧} ;

  Test2(⟦ ⟨WORD#w1⟩ ⟨Words#ws⟩ ⟧, ⟦⟨WORD#w⟩⟧) ↓dup{#w1 : #w2}
  → Test2(#ws, ⟦⟨WORD Concat(#w, #w2)⟩⟧) ;

  Test2(⟦ ⟧, #w) → #w ;

  // Helper to concatenate two words.
  sort Computed | scheme Concat(WORD, WORD) ;
  Concat(#w1, #w2) → ⟦ #w1 @"_" @#w2 ⟧ ;
}
```

```
$ hacs Desk.hx
...
$ ./Desk.run --scheme=Eval --term="1+2*3/(2-1)"
7.0
```

10.4 EXAMPLE. It is possible to select between structures with the Computed ternary choice operator and special "pass-through" parameters. Consider the following definition:

```
sort S | scheme IfPositive(Computed, S, S) | scheme ComputedToS(Computed);
IfPositive(#n, #1, #2) → ComputedToS(⟦ #n >= 0 ? #1 : #2 ⟧);
ComputedToS(#s) → #s;
```

Note the use of the extra "cast-like" function ComputedToS, which ensures that our ternary expression is computed (because the argument is Computed) yet is returned as something of sort S. This would appear to not type check, but is explicitly allowed by HACS to permit passing term structures through Computed expressions.

Figure 10: *examples/Desk.hx.*

```
module edu.nyu.cs.cc.Desk {

  // Syntax.

  token NUM | [0−9]+ ;

  sort E
    | ⟦ ⟨NUM⟩ ⟧@5
    | sugar ⟦ ( ⟨E#⟩ ) ⟧@5→ #

    | ⟦ ⟨E@2⟩ / ⟨E@3⟩ ⟧@2
    | ⟦ ⟨E@2⟩ * ⟨E@3⟩ ⟧@2

    | ⟦ ⟨E@1⟩ − ⟨E@2⟩ ⟧@1
    | ⟦ ⟨E@1⟩ + ⟨E@2⟩ ⟧@1
    ;

  // Evaluation.

  sort Computed | scheme Eval(E) ;
  Eval(⟦ ⟨NUM#⟩ ⟧) → ⟦ $# ⟧ ;

  Eval(⟦ ⟨E#1⟩ + ⟨E#2⟩ ⟧) → Plus(Eval(#1), Eval(#2)) ;
  | scheme Plus(Computed, Computed) ;
  Plus(#1, #2) → ⟦ #1 + #2 ⟧ ;

  Eval(⟦ ⟨E#1⟩ − ⟨E#2⟩ ⟧) → Minus(Eval(#1), Eval(#2)) ;
  | scheme Minus(Computed, Computed) ;
  Minus(#1, #2) → ⟦ #1 − #2 ⟧ ;

  Eval(⟦ ⟨E#1⟩ * ⟨E#2⟩ ⟧) → Times(Eval(#1), Eval(#2)) ;
  | scheme Times(Computed, Computed) ;
  Times(#1, #2) → ⟦ #1 * #2 ⟧ ;

  Eval(⟦ ⟨E#1⟩ / ⟨E#2⟩ ⟧) → Divide(Eval(#1), Eval(#2)) ;
  | scheme Divide(Computed, Computed) ;
  Divide(#1, #2) → ⟦ #1 / #2 ⟧ ;
}
```

11. EXAMPLES

Once the structure of a specification is clear, it is possible to start analyzing and manipulating the internal representation. This section presents some examples of this.

Figure 11: *examples/IsCat.hx*: Finding cats.

```
module org.crsx.hacs.samples.IsCat {                                    1

token WORD | [A−Za−z]+ ;                                                3
main sort Word | ⟦⟨WORD⟩⟧ ;                                             4
main sort Var | symbol ⟦⟨WORD⟩⟧ ;                                       5

sort Boolean | ⟦True⟧ | ⟦False⟧ ;                                       7

sort Boolean | scheme IsCat(Word) ;                                     9
IsCat(#word) → IsSameWord(#word, ⟦cat⟧) ;                               10

sort Boolean | scheme IsSameWord(Word, Word) ;                          12
IsSameWord(#, #) → ⟦True⟧ ;                                             13
default IsSameWord(#1, #2) → ⟦False⟧ ;                                  14
}                                                                       15
```

11.1 EXAMPLE (finding cats). The small example in Figure 11 illustrates how to test for equality using a non-linear rule in line 12 combined with a "catch-all" default rule in line 13.

It is not permissible to use ⟦*cat*⟧ directly in a pattern; patterns are restricted to *syntactic cases* of the grammar. Also, note the definition of the Boolean sort with syntactic values rather than just constructors: this allows them to be printed.

Here is a possible run using this command:

```
$ hacs IsCat.hx
$ ./IsCat.run --scheme=IsCat --term="dog"
False
$ ./IsCat.run --scheme=IsCat --term="cat"
True
```

11.2 EXAMPLE. Figure 12 illustrates the different conventions for using plain tokens—here uppercase words—and using symbols—here lowercase words. In the comments, notice the difference in use in rule cases and as map keys and set members.

```
$ hacs Symbols.hx
$ ./Symbols.run --scheme=Test --term="A A a a * A * a"
A A 2 a a * s A 3 s_39 a * END END END s_46 s_46 s_46
```

Notice the following:

- Input tokens and symbols passed through, *e.g.*, `A` and `a`.
- Repeated input tokens followed by count, *e.g.*, `A 2` and `A 3`.
- Repeated input symbols followed by star, *e.g.*, `a *`.

Figure 12: *examples/Symbols.hx*: Variations on symbols.

```
module org.crsx.hacs.tests.Symbols { //-*-hacs-*-

  token UP | [A−Z]+ ;  // upper case words are tokens
  token LO | [a−z]+ [0−9_]* ;  // lower case words with optional index
  token INT | [0−9]+ ; // integer

  sort S | symbol ⟦⟨LO⟩⟧ ;  // symbol sort

  // Input and output is list of words.
  sort W | ⟦⟨UP⟩⟧ | ⟦⟨S⟩⟧ | ⟦*⟧ | ⟦⟨INT⟩⟧ ;
  main sort L | ⟦⟨W⟩ ⟨L⟩⟧ | ⟦⟧ ;

  // Main scheme!
  sort L | scheme Test(L) ; Test(#) → Emit(#) ;

  // Inherited attribute to pass map of next token counts.
  attribute ↓ups{UP : Computed} ;

  // How to initialize counters.
  sort Computed | scheme Two ; Two → ⟦2⟧ ;

  // Inherited attribute with set of seen symbols.
  attribute ↓los{S} ;

  // Helper scheme, passing counts of tokens and symbols!
  sort L | scheme Emit(L) ↓ups ↓los ;

  // Rules for tokens (not seen before and already seen).
  Emit(⟦ ⟨UP#id⟩ ⟨L#⟩ ⟧) ↓ups{¬#id}
    → ⟦ ⟨UP#id⟩ ⟨L Emit(#) ↓ups{#id : Two}⟩ ⟧ ;
  Emit(⟦ ⟨UP#id⟩ ⟨L#⟩ ⟧) ↓ups{#id : #n}
    → ⟦ ⟨UP#id⟩ ⟨INT#n⟩ ⟨L Emit(#) ↓ups{#id : ⟦ #n + 1 ⟧}⟩ ⟧ ;

  // Rule for symbols (not and already seen) - note how an exemplar symbol is used.
  Emit(⟦ s ⟨L#⟩ ⟧) ↓los{¬⟦s⟧} → ⟦ s ⟨L Emit(#) ↓los{⟦s⟧}⟩ ⟧ ;
  Emit(⟦ s ⟨L#⟩ ⟧) ↓los{⟦s⟧} → ⟦ s * ⟨L Emit(#)⟩ ⟧ ;

  // Rule to generate a random symbol.
  Emit(⟦ * ⟨L#⟩ ⟧) → ⟦ s ⟨L Emit(#)⟩ ⟧ ;

  // Rule to skip existing counts.
  Emit(⟦ ⟨INT#n⟩ ⟨L#⟩ ⟧) → Emit(#) ;

  // Rule to finish off with thrice END and a symbol.
  Emit(⟦⟧) → ⟦ END END END s s s ⟧ ;
}
```

- Rule symbols generated fresh for each use of a rule replacement, *e.g.*, `s` ≠ `s_30` ≠ `s_46`.
- Generated symbols consistent within each replacement, *e.g.*, `s_46`.

Figure 13: *examples/WordSet.hx*: Sets of Words.

```
module org.crsx.hacs.samples.WordSet {                                          1

// Simple word membership query.                                                3
main sort Query | ⟦ ⟨WORD⟩ in ⟨List⟩ ⟧ ;                                        4
sort List | ⟦ ⟨WORD⟩, ⟨List⟩ ⟧ | ⟦ ⟨WORD⟩ ⟧ ;                                  5
token WORD | [A−Za−z0−9]+ ;                                                     6

// Collect set of words.                                                        8
attribute ↑z{WORD} ;                                                            9
sort List | ↑z ;                                                                10
⟦ ⟨WORD#w⟩, ⟨List#rest ↑z{:#ws}⟩ ⟧ ↑z{:#ws} ↑z{#w} ;                            11
⟦ ⟨WORD#w⟩ ⟧ ↑z{#w} ;                                                           12

// We'll provide the answer in clear text.                                      14
sort Answer                                                                     15
| ⟦Yes, the list has ⟨WORD⟩.⟧                                                   16
| ⟦No, the list does not have ⟨WORD⟩.⟧                                          17
;                                                                               18

// Check is main query scheme, which gives an Answer.                           20
sort Answer | scheme Check(Query) ;                                             21

// The main program needs the synthesized list before it can check membership.  23
Check( ⟦ ⟨WORD#w⟩ in ⟨List#rest ↑z{#w}⟩ ⟧ ) → ⟦Yes, the list has ⟨WORD#w⟩.⟧ ;    24
Check( ⟦ ⟨WORD#w⟩ in ⟨List#rest ↑z{¬#w}⟩ ⟧ )                                    25
   → ⟦No, the list does not have ⟨WORD#w⟩.⟧ ;                                    26
}                                                                               27
```

11.3 EXAMPLE (set of words). One common task is to synthesize a set from some syntactic construct and subsequently search the set. Figure 13 shows a small toy syntax that allows simple queries of word set membership.

The example uses some new mechanisms for synthesizing the set:

- A helper *z* synthetic attribute contains a *set* of word tokens, which is indicated by the attribute declaration ↑z{WORD} in line 9.
- Line 10 associates a *z* set with all values of the syntactic sort List.
- Lines 11 and 12 capture the synthesis of the set. Line 12 captures the simple case where a singleton list synthesizes a singleton set.
- Line 11 has a few more notations in play. First, the *pattern* part of the rule includes the inner pattern ↑z{:#ws}. This specifies that the special meta-variable ":#ws" captures all the existing members of the *z* set. Second, the result of the rule is to add *two* new things to the top level of the rule: ↑z{:#ws} ↑z{#w}. This adds *both* the existing members (just matched) *and* the one new member #w to the result set.

- Lines 24–26 are almost the same: the one difference is that 24 matches sets that contain the *#w* word, whereas 25–26 matches sets that do not because of the ¬ logical negation sign.

The example runs as follows:

```
$ hacs WordSet.hx
$ ./WordSet.run --scheme=Check --term="a in a,b,b,a"
Yes, the list has a.
$ ./WordSet.run --scheme=Check --term="Foo in Bar"
No, the list does not have Foo.
```

11.4 EXAMPLE (map of words). Figure 14 shows how a map can be synthesized and then used as an environment. The pattern is similar to the set example, except this case not only synthesizes the map attribute *m* but also serves to "copy" it over to an inherited map—an environment—*e*. Notice these extras:

- The map attribute is synthesized in lines 12–13, just like the set attribute was in the previous example. The only difference is that the map of course includes both a key and value.
- Line 23 simply captures all the "mappings" of the *m* attribute with the special *:#ms* pattern, which is then *reused* to populate the *e* environment.
- Lines 26–34 combine the distribution of the inherited map with a recursive transformation that replaces words. The two rules for an initial WORD are mutually exclusive because the pattern in line 26 requires the word to be present with a mapping in the *e* attribute, whereas the pattern in line 31 requires that the word not be present.

Here is a run demonstrating the program:

```
$ hacs WordMap.hx
$ ./WordMap.run --scheme=Substitute --term="a:b in a b b a"
b b b b
$ ./WordMap.run --scheme=Substitute --term="k:v in a b c"
a b c
```

11.5 EXAMPLE (word substitution). Figure 15 shows a HACS program to collect substitutions from a document and apply them to the entire document. Notice the following:

- This example uses a typical two-pass strategy: first, one pass to collect the substitutions into a synthesized attribute, then a second pass where the full list of substitutions is applied everywhere.
- A choice has been made to synthesize the map as a *data structure* instead of a native HACS map (as in the previous Example 11.4) because of the need here to *append* two maps (in line 42), which is not supported for the native maps. The synthesis happens in lines 41–49.
- The synthesized map is translated in list form into a native HACS map before starting the second pass. Notice how Run2 starts by recursing over the list of substitutions, inserting each into the carried inherited *env* map. Since the map is consumed from left to right, the *latest* substitution for any variable is always used.
- Since the inheritance schemes for *env* in lines 53–63 are doing a recursive traversal of the term, a benefit accrues from building the actual substitutions into the traversal.
- The inheritance rules carefully preserve the synthesized attributes only when the term does not change. In the present case, this is manifest by just the rule in line 59, not including the ↑*#s* marker to capture and copy the synthesized attributes; in general, this should be considered for every situation.

Figure 14: *examples/WordMap.hx*: Apply Word Substitution as Map.

```
module org.crsx.hacs.samples.WordMap {

// Simple word map over list.
main sort Query | ⟦ ⟨Map⟩ in ⟨List⟩ ⟧ ;
sort List | ⟦ ⟨WORD⟩ ⟨List⟩ ⟧ | ⟦ ⟧ ;
sort Map | ⟦ ⟨WORD⟩ : ⟨WORD⟩ , ⟨Map⟩ ⟧ | ⟦ ⟨WORD⟩ : ⟨WORD⟩ ⟧ ;
token WORD | [A−Za−z0−9]+ ;

// Collect word mapping.
attribute ↑m{WORD:WORD} ;
sort Map | ↑m ;
⟦ ⟨WORD#key⟩ : ⟨WORD#value⟩, ⟨Map#map ↑m{:#ms}⟩ ⟧ ↑m{:#ms} ↑m{#key:#value} ;
⟦ ⟨WORD#key⟩ : ⟨WORD#value⟩ ⟧ ↑m{#key:#value} ;

// Main program takes a Query and gives a List.
sort List | scheme Substitute(Query) ;

// Environment for mappings during List processing.
attribute ↓e{WORD:WORD} ;
sort List | scheme ListE(List) ↓e ;

// The main program needs the synthesized map before it can substitute.
Substitute( ⟦ ⟨Map#map ↑m{:#ms}⟩ in ⟨List#list⟩ ⟧ ) → ListE( #list ) ↓e{:#ms} ;

// Replace any mapped words.
ListE( ⟦ ⟨WORD#word⟩ ⟨List#words⟩ ⟧ ↑#syn ) ↓e{#word : #replacement}
→
⟦ ⟨WORD#replacement⟩ ⟨List ListE(#words)⟩ ⟧↑#syn
;

ListE( ⟦ ⟨WORD#word⟩ ⟨List#words⟩ ⟧ ↑#syn ) ↓e{¬#word}
→
⟦ ⟨WORD#word⟩ ⟨List ListE(#words)⟩ ⟧↑#syn
;

ListE( ⟦ ⟧ ↑#syn ) → ⟦ ⟧ ↑#syn ;
}
```

Here is a run with this system:

```
$ hacs WordSubst.hx
$ ./WordSubst.run --scheme=Run --term="a=1 a"
 a=1  1
$ ./WordSubst.run --scheme=Run --term="b a {a=1 b=2}"
 2  1     a=1  b=2
$ ./WordSubst.run --scheme=Run --term="{a=1 b=2 c=3} a b c {a=4} a b c"
  {  a=1  b=2  c=3     }   4  2  3   {  a=4   }   4  2  3
```

Figure 15: *examples/WordSubst.hx*: Combining list, maps, and transformation.

```
module org.crsx.hacs.samples.WordSubst {

// Grammar.
sort Units | ⟦ ⟨Unit⟩ ⟨Units⟩ ⟧ | ⟦⟧ ;
sort Unit | ⟦⟨Variable⟩=⟨NAT⟩⟧ | ⟦⟨Variable⟩⟧ | ⟦⟨NAT⟩⟧ | ⟦ { ⟨Units⟩ } ⟧ ;
sort Variable | symbol ⟦⟨ID⟩⟧ ;

token ID | [A−Za−z]+ ;
token NAT | [0−9]+ ;
space [\ \t\n\r] ;

// Helper Subst structure: lists of variable-NAT pairs.
sort Subst | MoreSubst(Variable, NAT, Subst) | NoSubst ;

// Append operation for Subst structures.
| scheme SubstAppend(Subst, Subst) ;
SubstAppend(MoreSubst(#var, #nat, #subst1), #subst2) → MoreSubst(#var, #nat, SubstAppend(#subst1, #subst2)) ;
SubstAppend(NoSubst, #subst2) → #subst2 ;

// Attributes.
attribute ↑subst(Subst) ;          // collected Subst structure
attribute ↓env{Variable:NAT} ;     // mappings to apply

// Top scheme.
main sort Units | scheme Run(Units) ;
Run(#units) → Run1(#units) ;

// Strategy: two passes.
// 1. force synthesis of subst attribute.
// 2. convert subst attribute to inherited environment (which forces replacement).

| scheme Run1(Units) ;
Run1(#units ↑subst(#subst)) → Run2(#units, #subst) ;

| scheme Run2(Units, Subst) ↓env ;
Run2(#units, MoreSubst(#var, #nat, #subst)) → Run2(#units, #subst) ↓env{#var : #nat} ;
Run2(#units, NoSubst) → Unitsenv(#units) ;

// Synthesis of subst.

sort Units | ↑subst ;
⟦ ⟨Unit #1 ↑subst(#subst1) ⟩ ⟨Units #2 ↑subst(#subst2)⟩ ⟧ ↑subst(SubstAppend(#subst1, #subst2)) ;
⟦ ⟧ ↑subst(NoSubst) ;

sort Unit | ↑subst ;
⟦v=⟨NAT#n⟩⟧ ↑subst(MoreSubst(⟦v⟧, #n, NoSubst)) ;
⟦v⟧ ↑subst(NoSubst) ;
⟦⟨NAT#n⟩⟧ ↑subst(NoSubst) ;
⟦ { ⟨Units#units ↑subst(#subst)⟩ } ⟧ ↑subst(#subst) ;

// Inheritance of env combined with substitution.

sort Units | scheme Unitsenv(Units) ↓env ;
Unitsenv( ⟦ ⟨Unit#1⟩ ⟨Units#2⟩ ⟧↑#s ) → ⟦ ⟨Unit Unitenv(#1)⟩ ⟨Units Unitsenv(#2)⟩ ⟧↑#s ;
Unitsenv( ⟦ ⟧↑#s ) → ⟦ ⟧↑#s ;

sort Unit | scheme Unitenv(Unit) ↓env ;
Unitenv( ⟦v=⟨NAT#n⟩ ⟧↑#s) → ⟦v=⟨NAT#n⟩⟧↑#s ;
Unitenv( ⟦v⟧ ) ↓env{⟦v⟧:#n} → ⟦⟨NAT#n⟩⟧ ;
Unitenv( ⟦v⟧↑#s ) ↓env{¬⟦v⟧} → ⟦v⟧↑#s ;
Unitenv( ⟦⟨NAT#n⟩⟧↑#s ) → ⟦⟨NAT#n⟩⟧↑#s ;
Unitenv( ⟦ { ⟨Units#units⟩ } ⟧↑#s ) → ⟦ { ⟨Units Unitsenv(#units)⟩ } ⟧↑#s ;
}
```

The last example shows how the latest substitution for *a* "wins."

A. MANUAL

This appendix is an evolving attempt at giving a systematic description of HACS.

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

```
module modulename
{
  Declarations
}
```

where the *modulename* should be a Java-style fully qualified class name with the last component capitalized and the same as the file base name, *e.g.*, `org.crsx.hacs.samples.First` is an allowed module name for a specification stored as *First.hx*. The individual sections specify the compiler, and the possible contents are documented in the manual blocks below.

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 (in order of increasing 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 represents itself.
5. A string in single or double quotes represents the contents of the string except that \ introduces an *escape code* that represents the encoded character in the string (see next item).
6. A stand-alone \ followed by an *escape code* represents 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 for the contents of the brackets:
   (a) If the first character is ^ then the character class is negated.
   (b) If the first character (after ^) is] then that character is (not) permitted.
   (c) A \ followed by an *escape code* represents the encoded character.
   (d) The characters \ ' " should always be escaped (this is a bug).
   (e) Two characters connected with a − (dash) represent a single character in the indicated (inclusive) *range*.

   Note that a character class cannot be empty. However, [^] is permitted and represents all characters.
8. The . (period) character represents the character class [^\n].
9. A nested regular expression can be given in ( ).
10. An entire defined 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. For example, 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 to be described later). For example, ⟦⟨Exp⟩+⟨Exp⟩⟧ describes the form where two expressions are separated by a + sign. Occurrences of tokens are referenced in the same way.
4. Concrete syntax specification can include ¶ characters (Unicode U+00b6) to indicate where *newlines* should be inserted in the printed output.
5. A @*p* for some natural number *p* is a *precedence indicator*, with higher numbers indicating higher precedence, *i.e.*, tighter association. A precedence indicator can be added to a production reference (*i.e.*, ⟨Exp@2⟩) or an entire concrete syntax production (*i.e.*, ⟦⟨Exp⟩+⟨Exp⟩⟧@2), indicating that either the appropriate subexpression or the entire alternative (as appropriate) is restricted to occur only at (at least) the specified precedence level. (For details on the limitations of how the precedence and left-recursion mechanisms are implemented, see Appendix C.)
6. A **sugar** ⟦. . . ⟧→. . . alternative specifies an equivalent form 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 a reference to a token, where furthermore the token is defined such that it can end with a sequence of _*n* units (an underscore followed by a count), then the sort case can be qualified as a **symbol** case, which implies:
   - instances of the token can be used with **binds** (see below),
   - the sort with the case can be used as the **as** sort of scopes (see below), and
   - the sort with the case can be used as a key sort of map attributes (actual keys must be variables).

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 using 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 (see below) 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 ⟨. . . ⟩. Note that **error** terms will be evaluated *when the rule is expanded*, thus only use **error** when matching the *pattern* of the rule is sufficient to produce the error.

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 lowercase 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 parentheses.
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 parenthesized ,-separated list of *scope arguments*, which each consist of a term optionally preceded by an optional *binder list* of variables enclosed in [] (dot). So in the most general case, a term looks like this:

   $$C\ (\ [\ x_{11}\ ,\ \ldots\ ,x_{1n_1}\ ]\ t_1\ ,\ \ldots\ ,\ [\ x_{m1}\ ,\ \ldots\ ,x_{mn_m}\ ]\ t_m\ )$$

   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 occurrences of x in t are said to be *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 term can include embedded parsed terms. However, these must in general have a sort prefix, except when they are arguments to defined constructors.
8. A *rewrite rule* is a pair of terms separated by $\rightarrow$ (arrow, U+2192), with a few additional constraints on 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 distinct bound variables.
   - t must be a *contraction*, which means that 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.
9. One rule per scheme can be prefixed with the qualifier **default**. If so, the pattern cannot have any 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.
10. 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 [10, 13].

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*{KeySort} defines that the synthesized attribute *name* is a set of KeySort values;

(c) ↑*name*{KeySort:ValueSort} defines that the synthesized attribute *name* is a map from KeySort to ValueSort values;

(d) ↓*name*(ValueSort), ↓*name*{KeySort}, and ↓*name*{KeySort:ValueSort} similarly for inherited attributes;

2. One can add a simple *synthesized attribute* after a raw data term as ↑*id*(*value*), where the *id* is an attribute name and the *value* can be any term of the appropriate sort.
3. Simple *inherited attributes* are added similarly after a raw scheme term as ↓*id*(*value*).
4. An *inherited symbol table attribute extension* is added to a raw scheme term as ↓*id*{*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 ↑*id*; and declares that the current sort synthesizes *id* attributes.
6. A scheme declaration can include *inherited attribute references* of the form ↓*id*, which declares that the scheme inherits the *id* 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 *id* attributes.
8. In *rules*, one can use the special forms ↑*#m*, which captures *all* synthesized attribute values; ↑*t*{:*#ms*} (↓*t*{:*#ms*}), which captures the full set of keys or key-value mappings of the *t* synthesized (inherited) attribute.

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* uses the *text* as the input.
4. `--input=`*file* (or just the *file*) reads the input from *file*.
5. `--output=`*file* sends the input to *file* (the default is the standard output).
6. `--errors` (or `-e`) reports details of errors found by subprocesses.
7. `--keep` (or `-k`) does not remove temporary generated files.
8. `--interpret` uses the much slower *interpreted* version of HACS and activates the following options.
9. `--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).
10. `--parse-verbose` activates (very!) 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

This appendix lists some of the more common of what can be called the "error messages" of HACS. Note that most of these only occur when HACS is run with the `--error` option.

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> ...
```

This error message from the `hacs` command 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 is not immediately helpful (this is a known bug). In error messages a sort is typically referenced as a lowercase 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 (precedence error)**.**

```
Java Compiler Compiler Version 6.0_1 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file OrParser.jj . . .
Error: Line 170, Column 1: Left recursion detected: "N_Exp1... --> N_Exp2... --> N_Exp1..."
Detected 1 errors and 0 warnings.
```

This suggests that a production breaks the precedence rule that all subterm precedence markers must be at least as high as the entire production's precedence marker, in this case between the Exp@1 and Exp@2 prededence markers, so presumably one of the rules for Exp with @2 allows an Exp with @1 as a first subterm.

B.4 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. Yes, the choice conflict is annoying but is in fact safe.

B.5 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.6 ERROR (meta-variable mistake)**.**

```
Error in rule Tiger-Ty99_9148-1: contractum uses undefined meta-variable (#es)
Errors prevent normalization.
make: *** [pr3.crs-installed] Error 1
```

A rule uses the meta-variable *#es* in the replacement without defining it in the corresponding pattern.

B.7 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.

B.8 ERROR.

```
// $Sortify
// $[Load, ".../build/edu/nyu/csci/cc/fall14/Pr2Solution.hx", "pr2solutionMeta_HxModule"]
Exception in thread "main" edu.nyu.csci.cc.fall14.TokenMgrError:
   Lexical error at line 184, column 31.  Encountered: "t" (116), after : "Call"
```

This indicates an undefined symbol of sort error in the *.hx* file: the symbol starting with CallI is either undefined or used in a location where it does not match the required sort.

B.9 ERROR.

```
// $Sortify
// $[Load, ".../build/edu/nyu/csci/cc/fall14/Pr2Solution.hx", "pr2solutionMeta_HxModule"]
Exception in thread "main" java.lang.RuntimeException: net.sf.crsx.CRSException:
   Encountered " ")" ") "" at line 255, column 112.
Was expecting one of:
    "," ...
```

This indicates an incorrect number of arguments in the *.hx* file: here insufficient arguments (encountering a parenthesis instead of comma); a similar but opposite error is given when excess arguments are present.

B.10 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.

B.11 ERROR.

```
« $Print-Check[
...
»
```

This *.run* script error indicates a request for a `--scheme` Check, which is not in fact declared as a **scheme** in the *.hx* file.

C. LIMITATIONS

- At most one **nested** choice is permitted per **token** declaration.
- It is not possible to use binders and left recursion in the same production with the same precedence.
- Only *immediate* left recursion is currently supported, *i.e.*, left recursion should be within a single production. Specifically,

  sort A | ⟦ ⟨A⟩ *stuff* ⟧ | ⟦ *other−stuff* ⟧ ;

  and

  sort A | ⟦ ⟨A@1⟩ *stuff* ⟧@1| ⟦ *other−stuff* ⟧@2;

  are allowed, but

  sort A | ⟦ ⟨B⟩ *a−stuff* ⟧ | ⟦ *other−a−stuff* ⟧;
  sort B | ⟦ ⟨A⟩ *b−stuff* ⟧ | ⟦ *other−b−stuff* ⟧;

  and

  sort A | ⟦ ⟨A@1⟩ *stuff* ⟧@2| ⟦ *other* ⟧ ;

  are not: prohibited cases involve indirect recursion (the latter case, where the inner left recursive precedence @1 is less than the outer precedence @2).
- Productions can share a prefix but only within productions for the same sort, and the prefix must be precisely identical unit for 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 (making it impossible for multiple binding constructs to have the same binder prefix).
- Variables embedded in **error**⟦. . . ⟧ instructions must start with a lowercase letter.
- When using the **symbol** qualifier on a reference to a token, then the token *must* be defined such that it allows ending with a sequence of *_n* for *n* any natural number.
- Symbols inside of ⟦. . . ⟧ and raw variables outside the ⟦⟧ must still have different names or they will be identified.
- Special terms like **error**⟦. . . ⟧ cannot be used as raw subterms.
- Synthesized attribute patterns with pattern-matching in the attributes may not always work.
- The Computed sort only works if there is at least one data or scheme constructor that *returns* a value of Computed sort.

REFERENCES

[1] Peter Aczel. A general Church-Rosser theorem. http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf, July 1978. Corrections at http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGRT_corrections.pdf.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, Inc, second edition, 2006. URL: http://dragonbook.stanford.edu/.

[3] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[4] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive-data-type systems. *Theor. Computer Science*, 272(1-2):41–68, 2002. Corrected version in http://arxiv.org/abs/cs/0610063. doi:10.1016/S0304-3975(00)00347-9.

[5] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, N. J., 1941.

[6] Olivier Danvy and Kristoffer H. Rose. Higher-order rewriting and partial evaluation. In Tobias Nipkow, editor, *RTA '98—Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 124–140, Tsukuba, Japan, March 1998. Springer. Extended version available as the technical report BRICS-RS-97-46 (http://www.brics.dk/RS/97/46/). doi:10.1007/BFb0052377.

[7] John Downs. hacsel Emacs mode for HACS. Github, February 2015. URL: https://github.com/jdowns/hacsel.

[8] IBM WebSphere DataPower appliances firmware V6.0 announcement. IBM United States Software Announcement 213-172, April 2013. URL: http://www-01.ibm.com/common/ssi/rep_ca/2/897/ENUS213-172/index.html.

[9] ICU Project Management Committee. *ICU – International Components for Unicode*, 54 edition, October 2014. URL: http://site.icu-project.org/home.

[10] Jean-Pierre Jouannaud. Higher-order rewriting: Framework, confluence and termination. In *Processes, Terms and Cycles: Steps on the road to infinity—Essays Dedicated to Jan Willem Klop on the occasion of his 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 224–250. Springer Verlag, 2005. URL: http://www.lix.polytechnique.fr/~jouannaud/articles/hor-fct.pdf.

[11] Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Also available as Mathematical Centre Tracts 127.

[12] Jan Willem Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–116. Oxford University Press, 1992.

[13] 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.

[14] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. doi:10.1007/BF01692511.

[15] Cynthia L. M. Kop. *Higher Order Termination*. PhD thesis, Institute for Programming Research and Algorithmics, Vrije Universiteit Amsterdam, 2012. IPA Dissertation Series 2012-14. URL: http://hdl.handle.net/1871/39346.

[16] Simon Marlow and Simon Peyton-Jones. The Glasgow Haskell Compiler. In *Structure, Scale, and a Few More Fearless Hacks*, volume II of *The Architecture of Open Source Applications*, chapter 5. Lulu.com, May 2012. URL: http://www.aosabook.org/en/ghc.html.

[17] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego, CA, USA 1998.

[18] P. Naur et al. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3:299–314, 1960. doi:10.1145/367236.367262.

[19] Tyler Palsulich. Sublime text mode for HACS. GitHub, December 2014. URL: https://github.com/tpalsulich/hacs_sublime_text_theme.

[20] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM. doi:10.1145/53990.54010.

[21] Eva Rose and Kristoffer H. Rose. Compiler construction. The graduate school computer science Compiler Construction class (CSCI-GA.2130) at the Courant Institute for the Mathematical Sciences, New York University, 2016. URL: http://cs.nyu.edu/courses/pring16/CSCI-GA.2130-001/.

[22] Kristoffer Rose. Combinatory reduction systems with extensions. GitHub, 2014. URL: https://github.com/crsx.

[23] Kristoffer H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 København Ø, February 1996. http://krisrose.net/thesis.pdf.

[24] Kristoffer H. Rose. CRSX – an open source platform for experimenting with higher order rewriting. Presented at HOR 2007, June 2007. URL: http://www.irit.fr/~Ralph.Matthes/HOR/ProceedingsHOR2007.pdf.

[25] Kristoffer H. Rose. CRSX – combinatory reduction systems with extensions. In Manfred Schmidt-Schauß, editor, *RTA '11—22nd International Conference on Rewriting Techniques and Applications*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 81–90, Novi Sad, Serbia, June 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.RTA.2011.81.

[26] Kristoffer H. Rose. Higher-order rewriting for executable compiler specifications. In Eduardo Bonelli, editor, *HOR '10—Proceedings of the 5th International Workshop on Higher-Order Rewriting*, Edinburgh, Scotland, July 14, 2010, volume 49 of *Electronic Proceedings in Theoretical Computer Science*, pages 31–45. Open Publishing Association, 2011. doi:10.4204/EPTCS.49.3.

[27] Sreeni Viswanadha, Sriram Sankar, et al. *Java Compiler Compiler (JavaCC) - The Java Parser Generator*. Sun, 4.0 edition, January 2006. URL: https://javacc.java.net/.