“Suppose someone to assert:
The gostak distims the doshes.
You do not know what this means; nor do I. But if we assume that it is English, we
know that the doshes are distimmed by the gostak. We know too that one distimmer
of doshes is a gostak. If, moreover, the doshes are galloons, we know that some
galloons are distimmed by the gostak. And so we may go on, and so we often do go
on.” (Richards and Ogden 1923)

The departure point for this chapter is a recursive routine for evaluating expres-
sions against an assignment function that stores accumulated binding information
for binding names (of which the classical interpretation of predicate logic in Sect. 1.2
is one example). An If conditional operation is added so that what is evaluated can
be automatically selected during the runtime of evaluation based on tests regarding
the state of the assignment function. This new operation is demonstrated to be an
essential component for enabling a robust interpretation of unknown lexical items
and for feeding an automated regulation of binding information to leave little need
for explicitly coding dependencies.

The chapter is organised as follows. Section 2.1 introduces sequence assignments.
Section 2.2 presents the ‘a Self.t language, with an evaluation routine for reach-
ing Lang.t expressions. The ‘a Self.t language is of interest because it includes
the conditional Self.If, as well as other operations that exploit the additional
structure present with a sequence assignment (notably Self.Lam and Self.Clean).
Section 2.3 links the ‘a Self.t language to natural language data. Section 2.4 offers
a summary. The purpose of creating and exploring the ‘a Self.t language and so
for this chapter is to prepare the ground for introducing the more capable ‘a Sct.t
language of Chap. 3.
2.1 Sequence Assignments

Having an `if` conditional such that what is evaluated can be selected based on the state of the assignment function is of utility with an assignment function that has sufficient properties to test. Typically assignment functions with the least amount of structure possible are favoured in order to limit assumptions. For example the assignment function for classical interpretations of predicate logic, as seen in Sect. 1.2, assigns to all variables a (possibly different) single value. This offers nothing to test, beyond assigned values being specific values.

In a series of papers Vermeulen (1993, 2000) and Hollenberg and Vermeulen (1996) propose an altogether richer assignment function in which (possibly empty) sequences (or stacks) of values are assigned to variables. Assignments with such additional structure are utilised in the evaluation systems of, for example, Visser and Vermeulen (1996), van Eijck (2001), Dekker (2002, 2012) and Butler (2007, 2010). With a sequence assignment, in addition to assigned values being specific sequences, it is possible to test for sequence length.

Having sequence assignments allows for operations to add (Assign.push), remove (Assign.pop and Assign.popLast) and manipulate (Assign.shiftLast and Assign.manage) assigned content. The following Standard ML implementation has the assignment type `‘a t assign` made polymorphic with regards to the type `‘a list` of values assigned to strings. Empty exceptions are raised if operations fail.

```ml
structure Assign =
  struct
  type ‘a t = string -> ‘a list
  fun push (d: ‘a, x: string, g: ‘a t): ‘a t =
    fn y =>
      if x = y then [d] @ g y else g y
  fun pop (x: string, g: ‘a t): ‘a t =
    fn y =>
      if x = y then tl (g x) else g y
  fun popLast (x: string, g: ‘a t): ‘a t =
    fn y =>
      if x = y then List.take (g x, length (g x)-1) else g y
  val shiftLast: string -> string -> ‘a t -> ‘a t =
    fn x => fn y => fn g =>
      popLast (x, push {List.last (g x), y, g})
```
fun manage (n: int, xs: string list, y: string, g: 'a t): 'a t =
  case xs of
    nil => g
  | x::r =>
    manage (n, r, y, iterate (shiftLast x y) (length (g x)-n) g)
end

The empty assignment is fn _ => nil, which maps every binding name to the empty list.

Assign.push returns a variant of g differing only in that d is added as the first element of the sequence assigned x. For example:

> (Assign.push (2, "arg0",
    Assign.push (1, "arg0", fn _ => nil))) "arg0";
val it = [2, 1]: int list

Assign.pop returns g except with the first sequence element assigned x removed. For example:

> (Assign.pop ("arg0",
    Assign.push (2, "arg0",
      Assign.push (1, "arg0", fn _ => nil))) ) "arg0";
val it = [1]: int list

Assign.popLast returns g except with the last sequence element assigned x removed. For example:

> (Assign.popLast ("arg0",
    Assign.push (2, "arg0",
      Assign.push (1, "arg0", fn _ => nil))) ) "arg0";
val it = [2]: int list

Assign.shiftLast returns g except with the last sequence element assigned x removed (by Assign.popLast) and made (by Assign.push) the new first sequence element assigned y. For example:

> val g = Assign.shiftLast "arg0" "c"
  (fn "arg0" => [4, 3, 2] | "c" => [1] | _ => nil);
val g = fn: int Assign.t

> g "arg0";
val it = [4, 3]: int list

> g "c";
val it = [2, 1]: int list

Assign.manage returns an assignment while taking as input:

1. integer n,
2. xs of type string list,
3. $y$ of type `string` and
4. assignment $g$.

The returned assignment is a variant of $g$ in having been possibly altered by (multiple applications of) `shiftLast`, which takes a string from $xs$ as the value for its first parameter and the string $y$ as the value for its second parameter. Exact applications of `shiftLast` are determined with `iterate` (re)applying `shiftLast` based on a number calculated by subtracting the integer value $n$ from the length of the sequence assigned to the string value from $xs$.

`Assign.manage` can be demonstrated as follows:

```plaintext
> val g = fn "arg0" => [2, 1] | "arg1" => [5, 4, 3] | _ => nil;
val g = fn: int Assign.t

> g "arg0";
val it = [2, 1]: int list

> g "arg1";
val it = [5, 4, 3]: int list

> g "c";
val it = []: int list

> val h = Assign.manage (1, ["arg0", "arg1"], "c", g);
val h = fn: int Assign.t

> h "arg0";
val it = [2]: int list

> h "arg1";
val it = [5]: int list

> h "c";
val it = [4, 3, 1]: int list
```

2.2 Self Language

The `a Self.t` datatype defines the `a Self.t` language. Function `Self.names` of type `a Self.t -> string list` extracts to a list the binding names of an `a Self.t` expression.

```plaintext
structure Self =
struct

datatype 'a t =
  T of string
```
2.2 Self Language

Some of string * 'a t
Rel of string * 'a t list
If of ('a Assign.t -> bool) * 'a t * 'a t
Lam of string * string * 'a t
Clean of int * string list * string * 'a t
Names of string list -> 'a t

fun names (f: 'a t): string list =
case f of
  T x => [x]
| Some (x, e) => uniq ([x] @ names e)
| Rel (_, es) => uniq (List.concat (map names es))
| If (_, e1, e2) => uniq (names e1 @ names e2)
| Lam (x, y, e) => uniq ([x, y] @ names e)
| Clean (_, xs, _, e) => uniq (xs @ names e)
| Names func => names (func nil)
end

Evaluation for the 'a Self.t language will be illustrated with the implementation of an evaluation routine, SelfToLang.eval, that is relativised against Lang.t Assign.t assignments and transforms Lang.t Self.t expressions into Lang.t expressions.

In transforming to Lang.t expressions, to ensure the Barendregt variable convention (see Sect. 1.1.3) is obeyed by the resulting Lang.t expression, it is important to ensure freshness for created variables of the Lang.t language that are added to sequences that serve as values assigned to Lang.t Self.t binding names. This is achieved with reference to SelfToLang.cnt, which has an initial state of 0.

structure SelfToLang =
struct
val cnt = ref 0;

fun env (x: string, g: Lang.t Assign.t): Lang.t Assign.t =
  (inc cnt ;
   Assign.push (Lang.X (!cnt, if x = "event" then x else "entity"), x, g))

fun eval (g: Lang.t Assign.t, f: Lang.t Self.t): Lang.t =
let
  fun evaluate (l, g, f) =
  case f of
    Self.T x => Lang.At (hd (g x), x)
  | Self.Some (x, e) =>
      let

val h = env (x, g)
in  
  Lang.QUANT ("\exists", [hd (h x)], evaluate (1, h, e))
end
| Self.Rel (s, es) => 
  Lang.REL (s, map (fn e => evaluate (l, g, e)) es)
| Self.If (func, e1, e2) => 
  if func g then evaluate (l, g, e1) else evaluate (l, g, e2)
| Self.Lam (x, y, e) => 
  evaluate  
  (1, Assign.pop (x, Assign.push (hd (g x), y, g)), e)
| Self.Clean (n, xs, y, e) => 
  evaluate (1, Assign.manage (n, xs, y, g), e)
| Self.Names func => evaluate (l, g, func l)
in  
evaluate (Self.names f, g, f)
end
end

Content is added to an assignment (that may be the empty assignment) with SelfToLang.env. Taking a string x and assignment g as parameters, SelfToLang.env treats x as a binding name, for which a fresh term is created as the new first assigned value. If the binding name is "event" then the added value is an event variable of Lang.t, otherwise the added value is an individual variable of Lang.t. For example:

> val g = SelfToLang.env ("event", 
  SelfToLang.env ("event", 
    SelfToLang.env ("arg0", fn _ => nil)));
val g = fn: Lang.t Assign.t

> g "event";
val it = \[X (3, "event"), X (2, "event")\]: Lang.t list

> g "arg0";
val it = \[X (1, "entity")\]: Lang.t list

Evaluation of term Self.T x returns a Lang.At role value construct (see Sect. 1.3.1) with the first element of the sequence assigned x as value and the name x to indicate grammatical role. An Empty exception is raised with failure. For example:

> SelfToLang.eval ( 
  fn "arg0" => [Lang.X (2, "entity"), Lang.X (1, "entity")]
  | _ => nil,
  Self.T "arg0")
val it = At (X (2, "entity"), "arg0"): Lang.t
> SelfToLang.eval (fn _ => nil, Self.T "arg0");

Exception- Empty raised

The quantifier `Self.Some (x, e)` adds one new value to the sequence assigned `x` and returns `Lang.QUANT` with `∃` to (i) bind as a variable the newly introduced value and (ii) scope over the evaluation of `e` against the adjusted assignment. For example:

```plaintext
> SelfToLang.eval (fn _ => nil,
   Self.Some ("arg0", Self.T "arg0"));
val it = QUANT ("∃", [X (1, "entity")], At (X (1, "entity"), "arg0")): Lang.t
```

Changes to the assignment that occur can be pictured as follows:

```
empty assignment

Self.Some ("arg0", _): Lang.QUANT("∃", [X (1, "entity")], _)
  ["arg0" -> [X (1, "entity")]]

| Self.T "arg0":
| Lang.At (X (1, "entity"), "arg0")
```

Beginning from the empty assignment a fresh value `Lang.X (1, "entity")` is entered as a value of the sequence assigned "arg0", and serves as the bound value for the role value construct returned with the evaluation of `Self.T "arg0"`.

Evaluation of `Self.Rel (s, es)` against assignment `g` returns a relation `s` that has the evaluation of the `n`th expression of `es` against `g` as the `n`th argument. For example:

```plaintext
> SelfToLang.eval (fn "arg0" => [Lang.X (1, "entity")] | _ => nil,
   Self.Rel ("", [Self.T "arg0", Self.T "arg0"]));
val it = REL ("", [At (X (1, "entity"), "arg0"), At (X (1, "entity"), "arg0")]): Lang.t
```

`Self.If` takes `Lang.t Assign.t -> bool` function `func` to test the current assignment state and two `Lang.t Self.t` expressions, `e1` and `e2`. If `func` applied to the assignment returns `true`, `e1` is evaluated, else `e2` is evaluated. For example:

```plaintext
> SelfToLang.eval (fn "arg0" => [Lang.X (1, "entity")] | _ => nil,
   Self.If (fn g: Lang.t Assign.t => null (g "arg1"),
     Self.T "arg0", Self.T "arg1"));
val it = At (X (1, "entity"), "arg0"): Lang.t
```

The test returns `true`, so `Self.T "arg0"` is evaluated.
Self-Selective Evaluation

"arg0" → [X (1, "entity")]

Self.If is true: _
Self.T "arg0":
Lang.At (X (1, "entity"), "arg0")

With a different assignment state the test can return false, so Self.T "arg1" is evaluated.

> SelfToLang.eval {
  fn "arg0" => [Lang.X (1, "entity")]
  | "arg1" => [Lang.X (2, "entity")]
  | _ => nil,
  Self.If (fn g: Lang.t Assign.t => null (g "arg1"),
            Self.T "arg0", Self.T "arg1"));
val it = At (X (2, "entity"), "arg1"): Lang.t

"arg0" → [X (1, "entity")]
"arg1" → [X (2, "entity")]

Self.If is false: _
Self.T "arg1":
Lang.At (X (2, "entity"), "arg1")

Self.Lam (x, y, e) returns the evaluation of e against assignment g modified with Assign.pop (x, Assign.push (hd (g x), y, g)). For example:

> SelfToLang.eval {
  fn "arg0" => [Lang.X (2, "entity"), Lang.X (3, "entity")]
  | "h" => [Lang.X (1, "entity")]
  | _ => nil,
  Self.Lam ("arg0", "h", Self.T "h"));
val it = At (X (2, "entity"), "h"): Lang.t
2.2 Self Language

That is, evaluation of `Self.T "h"` takes place against an assignment state where what had been the first element of the sequence assigned "arg0" is repositioned to be the first element of the sequence assigned "h".

`Self.Clean (n, xs, y, e)` modifies the assignment with `Assign.manage (n, xs, y)`, and returns the evaluation of `e` against the altered assignment. Consequently potentially multiple values from the sequences assigned to the names of `xs` are reallocated with `shiftLast` into the sequence assigned `y`, with the consequence that sequences with `n` elements remain assigned to each `xs` name. For example:

```plaintext
> SelfToLang.eval (
  fn "h" => [Lang.X (2, "entity"), Lang.X (3, "entity")]
  | _ => nil,
  Self.Clean (1, ["h"], "c", Self.T "h"));
val it = At (X (2, "entity"), "h"): Lang.t
```

The `Self.Clean` has the effect of an operation of 'unbinding' like in Berkling (1976) and still more like the 'end-of-scope' operator in Hendriks and van Oostrom (2003) since `Self.Clean` is not limited to the terminal level. A notable difference is that binding values are not destroyed but rather gathered as values of the sequence assigned to the name of the `y` parameter, which is employed in Chap. 3 to make antecedents accessible for pronouns.
Self NAMES is an operation to feed func of type string list -> \textquoteleft a Self.t a list of binding names gathered with Self.names applied when evaluation starts. For example:

\begin{verbatim}
> SelfToLang.eval {
  fn _ => nil,
  Self.Some ("arg0",
    Self.Some ("arg1",
      Self.Names (fn l =>
        Self.Rel ("", map (fn x => Self.T x) l)))));

val it = QUANT ("\∃", [X (1, "entity")],
  QUANT ("\∃", [X (2, "entity")],
    REL ("", [At (X (1, "entity"), "arg0"), At (X (2, "entity"), "arg1")])))
: Lang.t
\end{verbatim}

2.3 Applying the Self Language

The purpose of this section is to illustrate links of the \textquoteleft a Self.t language to natural language data. Section 2.3.1 applies the \textquoteleft a Self.t language to capture verbs with core arguments ("arg0" and "arg1" bindings). Section 2.3.2 adds encodings for nouns. Coverage is extended further in Sect. 2.3.3 to include non-core bindings.

2.3.1 Core Arguments

As a first application of the \textquoteleft a Self.t language, consider the task of simulating how differing presences of noun phrases influence processing the pseudoverb \textit{distims} in (1). For (1a) to be a well-formed sentence, \textit{distims} ought to be an intransitive verb; for (1b), a transitive verb; and for (1c), a verb without bound arguments (e.g., \textit{rains}).

(1) a. Someone distims.
  b. Someone distims someone.
  c. It distims.

The data of (1) can be captured with a method to handle the core grammatical roles of subject and object. Suppose grammatical subjects are established with "arg0" bindings, while grammatical objects involve "arg1" bindings. The noun phrase \textit{someone} should create either "arg0" or "arg1" bindings, while the verb \textit{distims} should bring about predicate encodings with appropriate "arg0" and "arg1" bound arguments.

Absence or presence of an "arg0" binding can be used as the basis for selecting whether \textit{someone} creates an "arg0" or "arg1" binding.
val SOMEONE = 
  fn e => 
    Self.If (fn g: Lang.t Assign.t => null (g "arg0"), 
      Self.Some ("arg0", e), 
      Self.Some ("arg1", e))

Consider verb1 as an initial attempt at an operation for constructing verbs from strings:

val verb1 = 
  fn s => 
    Self.Some ("event", 
      Self.If (fn g: Lang.t Assign.t => null (g "arg0"), 
        Self.Rel (s, [Self.T "event"]), 
        Self.If (fn g: Lang.t Assign.t => null (g "arg1"), 
          Self.Rel (s, [Self.T "event", Self.T "arg0"]), 
          Self.Rel
            (s, [Self.T "event", Self.T "arg0", Self.T "arg1"]))))

Encodings for the examples of (1) as Lang.t Self.t expressions can now be offered as follows:

val ex1 = SOMEONE (verb1 "distims")
val ex2 = SOMEONE (SOMEONE (verb1 "distims"))
val ex3 = verb1 "distims"

Evaluations result in Lang.t expressions, with generated bound arguments appearing alongside information about the grammatical role of the argument:

> SelfToLang.eval (fn _ => nil, ex1);
val it = 
QUANT ("∃", [X (1, "entity")], 
  QUANT ("∃", [X (2, "event")], 
    REL ("distims", [At (X (2, "event"), "event"), At (X (1, "entity"), "arg0")])))
: Lang.t

> SelfToLang.eval (fn _ => nil, ex2);
val it = 
QUANT ("∃", [X (1, "entity")], 
  QUANT ("∃", [X (2, "entity")], 
    QUANT ("∃", [X (3, "event")], 
      REL ("distims", [At (X (3, "event"), "event"), At (X (1, "entity"), "arg0"), 
                      At (X (2, "entity"), "arg1")]))))
: Lang.t

> SelfToLang.eval (fn _ => nil, ex3);
val it =
Three different forms for the verb "distims" arise, all of which have a bound argument with an "event" role while varying as to whether there are bound arguments with "arg0" and "arg1" roles. Notably, evaluation of ex1 produces an intransitive verb encoding for "distims", evaluation of ex2 produces a transitive verb encoding, and evaluation of ex3 produces a verb encoding without bound arguments.

To see why the results obtain, consider ex2 in detail. Executing ex2 at the Standard ML prompt results in the following reduced Lang.t Self.t expression:

```ml
> ex2;
val it =
  If (fn,
    Some ("arg0",
      If (fn,
        Some ("arg0",
          Some ("event",
            If (fn, Rel ("distims", [T "event"]),
              If (fn, Rel ("distims", [T "event", T "arg0"]),
                Rel ("distims", [T "event", T "arg0", T "arg1"])))),
          Some ("event",
            If (fn, Rel ("distims", [T "event"]),
              If (fn, Rel ("distims", [T "event", T "arg0"]),
                Rel ("distims", [T "event", T "arg0", T "arg1"]))))))),
    Some ("arg1",
      If (fn,
        Some ("argon",
          Some ("event",
            If (fn, Rel ("distims", [T "event"]),
              If (fn, Rel ("distims", [T "event", T "arg0"]),
                Rel ("distims", [T "event", T "arg0", T "arg1"]))))))));
```

The resulting Lang.t Self.t expression consists of a series of Self.If conditionals, with evaluation navigating a particular route through the conditionals to determine the form of the Lang.t expression that is returned, as the following chart illustrates.
The following picture depicts states of the assignment reached on the route taken by evaluation:

```
empty assignment

Self.If is true: _
Self.Some ("arg0", _): Lang.QUANT ("∃", [X (1, "entity")], _)
[arg0] → [X (1, "entity")]
Self.If is false: _
Self.If is false: _
Self.Rel ("distims", _): Lang.REL ("distims", _)
Self.T "event":
Lang.At (X (3, "event"), "event")
Self.T "arg0":
Lang.At (X (1, "entity"), "arg0")
Self.T "arg1":
Lang.At (X (2, "entity"), "arg1")
```
Self-Selective Evaluation

1. starts from the empty assignment,
2. finds the sequence assigned "arg0" is null with the first occurrence of SOMEONE and so creates an "arg0" binding with the addition of Lang.X (1, "entity") to the sequence assigned "arg0" and generation of the quantificational construct Lang.QUANT ("∃", [Lang.X (1, "entity")], _),
3. finds the sequence assigned "arg0" is not null with the second SOMEONE and so creates an "arg1" binding with the addition of Lang.X (2, "entity") and corresponding quantificational construct,
4. creates an "event" binding on entry to verb1 with the addition of Lang.X (3, "event") and corresponding quantificational construct,
5. finds the sequence assigned "arg0" is not null,
6. finds the sequence assigned "arg0" is not null, and finally
7. terminates with a transitive verb form that has "event", "arg0" and "arg1" bound arguments.

2.3.2 Nouns and Noun Phrases

Before considering formal encodings for nouns it is necessary to first create the possibility of noun phrases with restrictions as environments able to host nouns. The idea is that the restriction of a noun phrase should be insulated from the containing clause. This is accomplished with NP, which takes two parameters: x for the binding name that the noun phrase opens in the containing clause, and e to provide the content of the noun phrase restriction.

```
val NP = fn e => fn x =>
  Self.Names (fn lc =>
    Self.Lam (x, "h",
      Self.Clean (0, diff (lc, ["h"]), ",",
        Self.Clean (1, ["h"], ",", e))))
```

A call of NP:

1. repositions the first sequence element assigned x to be the first sequence element assigned "h",
2. shifts all other open bindings of the names taken from the call of Names minus "h" to "" bindings, and
3. shifts all sequence values assigned "h" to "" with the exception of the first value.

Shifting a binding to the empty name "" essentially removes the binding from further consideration to leave only the single "h" binding that it is the purpose of the noun phrase to introduce.

Having arguments with restrictions that bind with a given name is accomplished with PP:
2.3 Applying the Self Language

val PP = fn x => fn e1 => fn e2 =>
    Self.Some (x, Self.Rel ("∧", [e1 x, e2]))

As with someone in Sect. 2.3.1, the state of the assignment can determine the binding created by a bare noun phrase:

val NP1 = fn e1 => fn e2 =>
    Self.If (fn g: Lang.t Assign.t => null (g "arg0"),
        PP "arg0" (NP e1) e2,
        PP "arg1" (NP e1) e2)

Finally encodings for nouns can be considered, for which the simplest form is a predicate with a bound "h" argument:

val noun1 = fn s =>
    Self.Rel (s, [Self.T "h"])

The ingredients introduced in this section are brought together with the analysis of (2) as ex4.

(2) Gostak distims došes.

val ex4 =
  ( (NP1 (noun1 "gostak"))
   ( (NP1 (noun1 "došes"))
    (verb1 "distims")))

Here is an evaluation of ex4:

> SelfToLang.eval (fn _ => nil, ex4);
  val it = QUANT ("∃", [X (1, "entity")],
    REL ("∧", [REL ("gostak", [At (X (1, "entity"), "h")]),
        QUANT ("∃", [X (2, "entity")],
            REL ("∧", [REL ("došes", [At (X (2, "entity"), "h")]),
                QUANT ("∃", [X (3, "event")],
                    REL ("distims", [At (X (3, "event"), "event"), At (X (1, "entity"), "arg0"),
                        At (X (2, "entity"), "arg1")]))]))))

: Lang.t

The Lang.t expression that is the result of evaluation can be pretty printed as follows:

\[ \exists x_1 \text{gostak}(x_1) \land \exists x_2 \text{došes}(x_2) \land \exists e_3 \text{distims}(e_3, x_1, x_2) \]

Such a Davidsonian meaning representation retains grammatical role information with fixed arity positions, e.g., achieved with the Davidsonian.format routine of Sect. 1.3.2.
The evaluation of ex4 can be pictured as follows:

```
empty assignment

Self.If is true: _
Self.Some ("arg0", _): Lang.QUANT("∃", [X (1, "entity")], _)  
"arg0" → [X (1, "entity")]

Self.Rel ("∧", _): Lang.REL ("∧", _)

Self.Names: _
Self.Lam ("arg0", "h", _): _  
"h" → [X (1, "entity")]

Self.Clean (0, ["arg0", "event", "arg1"], "", _): _
Self.Clean (1, ["h"], "", _): _

Self.Rel ("gostak", _): Lang.REL ("gostak", _)
Self.Rel ("doshes", _): Lang.REL ("doshes", _)
Self.Rel ("distims", _): Lang.REL ("distims", _)

Self.If is false: _
Self.If is false: _
Self.If is false: _
```

The evaluation of ex4 can be pictured as follows:
2.3 Applying the Self Language

This shows how the binding created by the noun phrase gostak, while being opened as an "arg0" binding to bind the subject argument of the main predicate distims, shifts to "h" internally to its restriction to bind the nominal gostak, and furthermore shifts to "" to have no binding consequence inside the restriction of the subsequent noun phrase containing dothes.

The analysis of ex4 discriminates overtly between nouns (with noun1) and verbs (with verb1), but consider word1 which tests for the absence of an "h" binding such that with success a verb encoding is selected while failure selects a noun encoding.

val word1 = fn s => Self.If (fn g: Lang.t Assign.t => null (g "h"), verb1 s, noun1 s)

With word1 (2) can be analysed as ex5.

val ex5 = (NP1 (word1 "gostak")) (NP1 (word1 "doshes")) (word1 "distims"))

Evaluation of ex5 produces the same result as with the evaluation of ex4.

2.3.3 Adding Non Core Arguments

Examples considered so far had arguments with the privileged grammatical status of creating either "arg0", "arg1" or "h" bindings. Such limited data was covered with encodings that hard wired acceptable combinations of "arg0", "arg1" and "h" bindings. This is obviously inadequate as soon as the presence of other types of argument noun phrases are considered, as are created with preposition phrases in English, such as with as, following, including, on, but also more productively according to, compared with, out into, primarily because of, and so on. This section offers a method to incorporate such binding names.

Consider recursive addArgs.

val rec addArgs = fn l1 => fn l2 => fn pred => case l1 of nil => pred l2 | x::r => Self.If (fn g: Lang.t Assign.t => null (g x), addArgs r l2 pred, addArgs r (l2 @ [x]) pred)

This takes three parameters: l1 and l2 as sequences of binding names and pred that will itself have an open parameter to take a sequence of binding names. When l1
is nil the content of \texttt{l2} is applied to \texttt{pred}, otherwise an expression is created with \texttt{Self.If} that has:

1. a test for whether the first element of \texttt{l1} is assigned the empty sequence,
2. an expression to evaluate if the test succeeds assembled from a call to \texttt{addArgs} on the rest of \texttt{l1} with no change to \texttt{l2}, and
3. an expression to evaluate if the test fails assembled from a call to \texttt{addArgs} on the rest of \texttt{l1} with \texttt{l2} extended to include the first element of \texttt{l1}.

Predicates can be created with \texttt{predicate}, which calls \texttt{addArgs} as a wrapper around \texttt{Self.Rel} that creates at the very least arguments for the names of \texttt{args}, and possibly other arguments built from names taken from the call of \texttt{Self.Names} minus the names of \texttt{args} such that these added arguments will only have consequences when there is sufficient binding support from the assignment during evaluation.

\begin{verbatim}
val predicate = fn args => fn s =>
  Self.Names (fn lc =>
    addArgs (diff (lc, args)) args (
      fn l => Self.Rel (s, map (fn x => Self.T x) l)))
\end{verbatim}

Encodings for nouns and verbs can now be created.

\begin{verbatim}
val noun = fn s =>
  predicate ["h"] s
val verb = fn s => fn args =>
  Self.Some ("event", predicate (["event"] @ args) s)
\end{verbatim}

These differ in that the ever present argument for verbs is a locally created "event" rather than an inherited "h" binding. Also \texttt{verb} has an extra parameter \texttt{args} for taking a sequence of binding names that have to be arguments of the verb.

A generic coding for words, with sensitivity to the presence of an "h" binding, can be offered:

\begin{verbatim}
val word = fn s =>
  Self.If (fn g: Lang.t Assign.t => null (g "h"),
            verb s nil,
            noun s)
\end{verbatim}

The PP operation of Sect. 2.3.2 already gives a method to create arguments that bind with non-core binding names. Also, NP2 can be made to provide genitive bindings in nominal contexts, while otherwise calling the binding actions of NP1.
2.3 Applying the Self Language

val NP2 = fn e1 => fn e2 =>
  Self.If (fn g: Lang.t Assign.t => null (g "h"),
  NP1 e1 e2,
  PP "of" (NP e1) e2)

The ingredients introduced in this section are brought together with the analysis of (3) as `ex6`.

(3) For Americans Buffalo gostak distims doses.

val ex6 = ( (PP "for"
    (NP (word "Americans")))
  ( (NP2 ( (NP2 (word "Buffalo"))
        (word "gostak")))
   ( (NP2 (word "doshes"))
      (word "distims"))))

Here is an evaluation of `ex6`:

> SelfToLang.eval (fn _ => nil, ex6);
val it = QUANT ("∃", [X (1, "entity")],
REL ("∧", [REL ("Americans", [At (X (1, "entity"), "h")]),
QUANT ("∃", [X (2, "entity")]],
REL ("∧", [
QUANT ("∃", [X (3, "entity")],
REL ("∧", [REL ("Buffalo", [At (X (3, "entity"), "h")]),
REL ("gostak", [At (X (2, "entity"), "h"), At (X (3, "entity"), "of")])]),
QUANT ("∃", [X (4, "entity")],
REL ("∧", [REL ("doshes", [At (X (4, "entity"), "h")]),
QUANT ("∃", [X (5, "event")],
REL ("distims", [At (X (5, "event"), "event")],
 At (X (1, "entity"), "for"), At (X (2, "entity"), "arg0"),
 At (X (4, "entity"), "arg1")))]))])
: Lang.t

This shows creation of "event", "for", "arg0" and "arg1" bindings of the main predicate, `distims`. In addition to the "h" binding for the nominal `gostak` there is also an "of" binding created locally to the `gostak` noun phrase which is restricted by binding `Buffalo` under the "h" name. Pretty printing returns:

\[ \exists x_1 \text{Americans}(x_1) \land \\
  \exists x_2 (\exists x_3 \text{Buffalo}(x_3) \land \text{is}_\text{gostak}_\text{of}(x_2, x_3)) \land \\
  \exists x_4 \text{doshes}(x_4) \land \exists e_5 \text{distims}(e_5, x_2, x_4) \land \text{for}(e_5 = x_1) \] 

The pretty print involves fixing grammatical roles to arity positions. But this is not possible with the "for" binding of the main predicate, and so there is integration
of this adjunct binding as a modifier of the event ($e_5$) of the main predicate. This is achieved with Davidsonian.format of Sect. 1.3.2. Davidsonian.format is also responsible for changing the nominal gostak predicate to accommodate the genitive binding.

2.4 Summary

This chapter introduced conditional Self.If as part of the ‘a Self.t language, and included a Standard ML implementation of a recursive routine for evaluating Lang.t Self.t expressions against a sequence assignment function that stores accumulated binding information to return expressions of Lang.t. Having Self.If enabled an automated selection of expression content at the runtime of evaluation based on the state of the assignment function. This (i) allowed for single encodings of content that would otherwise require distinct expressions, and (ii) equipped expressions with ways to recover from situations that would otherwise lead to unwelcome results from evaluation. With these two properties evaluation was left to feed automatic regulation to enable coverage of unknown lexical items as well as novel binding names. This is especially notable for providing a means to get away with very little explicit coding of information about how binding dependencies should be established.

References

Berkling KJ (1976) A symmetric complement to the lambda-calculus. Interner Bericht ISF-76-7, GMD. St. Augustin, Germany
Linguistic Expressions and Semantic Processing
A Practical Approach
Butler, A.
2015, VIII, 172 p. 4 illus., Hardcover
ISBN: 978-3-319-18829-4