The Acquisition of Programming Skills from Textbooks

Manfred Klenner
Computational Linguistics
Heidelberg University, Karlstr. 2, D-69117 Heidelberg, Germany
klenner@janus.gs.uni-heidelberg.de

Thomas Hanneforth
Computational Linguistics
Potsdam University, Am Neuen Palais 10, D-14415 Potsdam, Germany
hannefor@rz.uni-potsdam.de

Abstract

We present a computer model for the acquistion of programming languages from textbooks. Starting from a verbal description of the notational conventions that are used to describe the syntactic form of programming commands, a meta grammar is generated that parses concrete command descriptions and builds up grammar rules for that commands. These rules are realized as definite clause grammar rules that captures the syntax of these commands. They can be used to parse and generate syntactically correct examples of a command. However, to solve real programming problems also the semantics of a command and of its parameters needs to be acquired. This is accomplished by the natural language parsing of the explanations given in the text and the augmentation of the definite clause command grammars with semantic structures.

Introduction

The acquisition of a programming language from textbooks is a complex cognitive task that comprises two subtasks: the acquistion of the syntax of the formal expressions of the language and the acquistion of the corresponding (operational) semantics. Syntactic knowledge is necessary to generate well-formed, computer executable expressions. However, programming is a goal-directed task and expressions are generated to solve specific problems. Thus, the semantics of these expressions is crucial as well. A well-formed command expression only accomplishes its task if it realizes an appropriate form (syntax) for the intended action (semantics).

Accordingly, textbooks give a syntactic and semantic description for each command. This is done in a heterogeneous format: by a formal specification of the syntax and a verbal description of the semantics (the general function, the meaning and type of the parameters etc.). To understand the syntactic description the user needs to know the notational convention used in the textbook, e.g. *functions are enclosed in brackets with the first element denoting the function name* (Lisp), or *optional parameters are enclosed in square brackets* (MS-DOS command syntax). These conventions are given at the beginning of such textbooks, but often they are incomplete. However, such meta knowledge is necessary to understand (i.e. parse) the formal description of single commands, e.g. to recognize that *path* in *CD [device:][path]* is an optional parameter of the command *CD*. The result of this parsing process is a parse tree of the formal command expression which needs to be operationalized. In our model this is done by a rule constructor that generates a command grammar in form of a Definite Clause Grammar (DCG). Such a definite clause command grammar can be used to parse and generate instances of the command, e.g. *cd a:user*.

Nothing has been said so far about the acquisition of the meaning of commands, of which a verbal explanation is given in the accompanying text together with the command syntax. In this way, the general meaning of the command (e.g. *CD changes the current directory*) and its parameters is provided. The semantic interpretations must be related to the syntactic constructions they explain. Only such a coupling guarantees that semantic content and syntactic form will work together correctly in a problem solving environment. The semantic knowledge is necessary for the planning and reasoning process, the syntactic knowledge for the realization stage. Learning failures can occur at each of the following stages: the acquisition of the meta grammar, the acquisition of the command grammar and the acquistion of the syntactic-semantic interface.

We assume that the learner is a novice with respect to the programming language to be acquired. We confine ourselves to the acquisition of an operating system language (MS-DOS) which we think is easier to learn than a full-fledged programming language like Lisp or Prolog.

We distinguish four learning stages in the acquistion of a programming language:

- construction of a meta grammar from the notational conventions that are introduced (verbally) in the text.
- parsing of the formal command syntax with the meta grammar and construction of an initial command grammar from the derivation tree.
- natural language parsing of the command definitions and modification of the command grammar to incorporate the corresponding semantic information.
- parsing of command examples with the command grammar and parsing of the corresponding verbal explanations. This leads to further refinements of the command grammar.

Operationalization of Notational Conventions

Let the discussion become more concrete. In Figure 1, the notational conventions of a textbook that introduces the operating system MS-DOS and its commands are given. These sentences can be viewed as instructions to build up a (meta)

1. All DOS commands are given in upper case letters.
2. The parameters of a command are specified in lower case letters.
3. The name of a parameter often characterizes its function.
4. Optional parameters are enclosed in square brackets ([parameter]).
5. Multiple, but mutually exclusive parameters are enumerated in angle brackets.

Figure 1: Notational conventions from a textbook

grammar for MS-DOS commands. With the aid of this grammar, commands like CD or COPY can be understood, i.e. the learner can determine what the command name is, whether its parameters are optional or mutually exclusive, etc. Even information regarding the type of parameter is given (sentence 3). However, there is still some information left implicit and some necessary conventions are even missing. To give an example: The concept of a *parameter* is introduced without any clear distinction of what different types of parameters there are. For example, optional parameters are introduced, but does that imply the existence of mandatory parameters? And how will these parameters be marked? Also, no information about linearization is given: does the command name precede its parameters or vice versa? Additionally, special characters (including the white space characters like blank and tab) are not even mentioned.

The question is: How can these notational conventions be utilized to understand and acquire the syntax of commands? The answer is: It must be operationalized in form of a syntax recognizer (e.g., a context-free grammar and a parser) that can parse concrete system commands.

In Figure 2, we give a sketch of the meta knowledge necessary to build up a context-free grammar (CFG) from natural language input.

1. a CFG *rule* consists of a *head* and a *body* separated by an expansion symbol ($\rightarrow$)
2. the *body* consists of *terminals* and / or *nonterminals*
3. *nonterminals* are realized as CFG *rules*
4. *terminals* are single characters like "A" or "["
5. *optionality* of rules is expressed by at least two alternative rules, where one rule expands to [] (zero expansion); i.e. the input string is not decremented
6. *exclusiveness* of rules is realized by alternative rules without a zero expansion

Figure 2: Knowledge about the construction of a CFG

The module responsible for the translation from the instructional text, i.e., the notational conventions, to a CFG is called the *meta rule constructor*. The meta rule constructor is based on a Definite Clause Grammar for the English constructions which are used to specify the notational conventions. It uses a Montague style rule-by-rule approach to map the English sentences to context-free rules, the logical language we use. Every syntactic constituent receives a semantic value which is compositionally constructed from the semantic values of the immediate subconstituents (the details of how this is done will be given in a moment). The semantic value assigned to a syntactic constituent N is a pair <E,R> where

- E is an expression of the utilized logical language (a constant or a lambda term)
- R is the set of rules constructed for the constituent which consists of the union of the rule sets assigned to the subconstituents of N, plus the value of E in case N is the sentence symbol.

Consider the notational conventions from Figure 1 and the resulting CFG in Figure 3 as it was built by the meta rule constructor. Corresponding numbers between Figure 3 and Figure 1 indicate that the grammar rule was derived from the respective sentence.

1a) dos_command $\rightarrow$ upper_case_letters
1b) upper_case_letters $\rightarrow$ upper_case_letter
1c) upper_case_letters $\rightarrow$ upper_case_letter upper_case_letters

2a) parameter $\rightarrow$ lower_case_letters
2b) parameters $\rightarrow$ parameter
2c) parameters $\rightarrow$ parameter parameters
2d) lower_case_letters $\rightarrow$ lower_case_letter
2e) lower_case_letters $\rightarrow$ lower_case_letter lower_case_letters

4a) optional_parameter $\rightarrow$ '[' parameters ']'
4b) optional_parameters $\rightarrow$ optional_parameter
4c) optional_parameters $\rightarrow$ optional_parameter optional_parameters

5a) exclusive_parameters $\rightarrow$ '<' parameters '>'
5b) exclusive_parameters $\rightarrow$ exclusive_parameter
5c) exclusive_parameters $\rightarrow$ exclusive_parameter exclusive_parameters

Figure 3: Meta grammar

We cannot go through each rule, but will nevertheless comment on some interesting translations. Sentence 1 mainly gives rise to rule 1a, i.e.,

(Ex-1) *dos_command* $\rightarrow$ *upper_case_letters.*

But there are a number of other rules which were triggered by this sentence (1b and 1c), which we will now explain in more detail.

The semantic value of a singular noun like *upper case letter* is the pair

(Ex-2) <*upper_case_letter*, { *upper_case_letter* $\rightarrow$ *A*, *upper_case_letter* $\rightarrow$ *B*, ... }>,

i.e, the set of all lexical insertion rules for a single upper case letter. By contrary, nouns like "DOS command" have the semantic value <dos_command,$\emptyset$ >, that is the set of asscociated rules is empty. The reason for this difference lies in the different grammatical status of the corresponding nonterminals. *upper_case_letter* is a nonbranching preterminal, which dominates exactly one terminal symbol. Since the object language consists of sequences of single letters or symbols, not of preparsed tokens (finding these tokens is one of the goals of our approach), the operational meaning of such a noun is the set of rules used to recognize the corresponding terminal symbols. On the other hand, *DOS command* is translated into a nonterminal symbol for which we hope to find the relevant expansion rules in subsequent steps, therefore the associated rule set is empty. In other words: the human learner obviously

knows what upper case letters are but (s)he (as a novice) has no idea about the nature of DOS commands.

It is now interesting to see how the recursive rules 1b and 1c in Figure 3, which are used to recognize *sequences* of upper case letters, are built. We think the trigger condition for recursive rules is the *plural* of the nouns occurring in the notational conventions, in our example *upper case letters*. Therefore, we assume the following plural formation DCG-rule:

n(pl,NPl_Sem) → n(sg,NSg_Sem), [s],
{ plural_semantics(NSg_Sem,NPl_Sem) }

That is: the plural of a noun is formed by appending the letter *s* to the singular form. The predicate plural_semantics/2 is defined in Prolog[1] as follows:

```
plural_semantics(<C1,Rules1>,<C2,Rules2>):-
        concat(C1,s,C2)          % e.g., concat(letter,s,letters)
        NewRules = [C2 → C1, C2 → C1 C2],
        union(Rules1,NewRules,Rules2).
```

Figure 4: Plural formation rule in Prolog

An example will illustrate this matter. The semantic value of *upper_case_letter* was specified in (Ex-2) above. Plural_semantics/2 now introduces two new rules based on this value:

Rule 1. upper_case_letters → upper_case_letter
Rule 2. upper_case_letters → upper_case_letter
upper_case_letters

that is *upper_case_letters* is used to recognize the language {A, B, ..., Z }$^+$. The semantic value of the plural noun *upper case letters* then consists of the pair < *upper_case_letters*,{ Rule 1, Rule 2 } ∪ { all lexical insertion rules for *upper_case_letter* } >.

Let us now see how rule Ex-1 was constructed. The grammar for the notational conventions has two further rules (see fig. 5).

```
v(sg,< λyλx(x → y), ∅ >) → [are,given,in].
vp(Num,<VPSem,VPRules>) → v(Num,<VSem,VRules>),
                          np(_,<NPSem,NPRules>),
                          { apply(NPSem,VSem,VPSem),
                          union(VRules,NPRules,VPRules) }.
```

Figure 5: Two rules of the Meta Rule Constructor

We regard verb complexes such as "are given in" as manifestations of a general definition operator which is represented by the context-free arrow → . The first component of the semantic value of the VP *are given in upper case letters* is therefore reconstructed as λ x (x → *upper_case_letters*).

A final sentence rule (not shown here) applies the meaning of the subject NP to the meaning of the VP, takes the union of the rule sets of the NP and the VP and finally adds the rule *(Ex-1) dos_commands → upper_case_letters* to this set[2].

[1]We use a slightly modified Prolog syntax for the purpose of exposition

[2]The addition of the last rule is covered by the special condition in the definition of R regarding sentence symbols (see the definition of the 2-tuple <E,R>,above).

As is easily seen, this grammar fragment is a bit too general, but this does not matter because the meta grammar constructed in the first learning stage will later be refined by parsing the actual DOS commands which we assume to be correctly specified. In other words: in principle we do not have the problem of ungrammatical input, and therefore "wrong" grammar rules will never be used in the second learning stage and can be removed from the grammar after this stage.

Closing Knowledge Gaps in the Meta Grammar

After the derivation of the meta grammar from the notational conventions, the learner should be able to parse the formal command descriptions of the operating system language (e.g. *CD[device:][path]*). However, not only misconceptions of the learner but also incomplete explanations given in textbooks can impede this. In our example, a top level rule is missing, i.e., the rule that provides the information about the sequence of the *command name* and its *parameters*. Additionally, no rule for the interpretation of special characters (e.g. the colon in *device:*) has been introduced. We use a robust *bottom up chart parser* and a rule induction component to overcome such knowledge gaps. Knowledge gaps at the rule level (e.g. the missing top rule) are closed by investigating the resulting chart. The chart parser assigns partial parse trees to command expressions and a rule induction component generates all possible chart completions (see below for an example). Knowledge gaps at the character level (e.g. ':') are labeled *unknown* (e.g. *unknown(:)*) and integrated according to the following heuristic: any preterminal rule allows a category *unknown* to be integrated (gaps are closed on the deepest level of a parse tree).

The output of the parser for the MS-DOS command *CD[device:][path]* is given in Figure 6.

```
Edges  Partial Parse Trees
1-2    dos_command(upper_case_letters('CD')),
2-9    optional_parameters(
2-3      '['
3-4      optional_parameter(parameters(
                 parameter(lower_case_letters(device)),
4-5              parameter(unknown(:))),
5-7      ']' '['
7-8      optional_parameters(optional_parameter(parameters(
                 parameter(lower_case_letters(path))),
8-9      ']')
```

Figure 6: Trace of *CD[device:][path]*

This parse tree is slightly simplified for ease of readability (we removed the nodes for *lower_case_letter* and *upper_case_letter* and contracted the letters to words). The chart parser identifies two constituents: a simple constituent *dos_command* (CD), and a more complex constituent *optional_parameters* with the terminals *device*, *:* and *path*. No connection can be established between these two constituents due to the missing top rule. However, the *chart edges* provide linearization information, i.e., edge 1-2 covering *dos_command* precedes edge 2-9 covering the optional parameters. In this case, the completion of the parse tree

is a simple task, a new edge 1-9 called *command* is introduced that covers *dos_command* and *optional_parameters*. The corresponing top rule is *command* → *dos_command optional_parameters*. This rule completes the meta grammar. It is, however, too restrictive, since *exclusive_parameters* are not captured. In order to generalize it, a new expression, say, *command_parameters*, that replaces *optional_parameters* in the *command* rule, needs to be introduced. This will then expand to all known rules for parameters (e.g., *optional_parameters* and *exclusive_parameters*). We will not further elaborate on this.

Generating Grammars from Parse Trees

Given a derivation tree built from parsing a command expression with the meta grammar, the learner is able to construct a context-free Definite Clause Grammar representation of that command expression. In our example, *CD[device:][path]* was assigned the parse tree given in Figure 6 before the missing edge, 1-9, was induced by the system. In Figure 7 the completed parse tree is given.

```
Edges  Parse Tree
1-9    command(
         dos_command('CD'),
         optional_parameter(device),
         unknown(:),
         optional_parameter(path))
```

Figure 7: Completed and pruned trace of *CD[device:][path]*

This parse tree has been pruned (compared to fig. 6) according to the following operationalization criterion:

1. the root node is operational (here: *command*)
2. the heads of recursive rules are not operational
3. the most specific domain concepts are operational (here: *dos_command, optional_parameter*)
4. all leaves of non-branching preterminals are operational (here: *path*, *device*, but not "[")
5. the label *unknown* is operational

The notion of an operationalization criterion is adopted from Explanation-based Learning (EBL, Mitchell & Kedar-Cabelli (1986)). In general, only those node labels (i.e., rule heads) that are crucial for the definition of the command grammar are operational. This principle is best exemplified by criterion 3 which states that only the most specific domain concepts are operational. To give an example: Although *parameter* is a domain-specific concept, *optional parameter* is as a subconcept of *parameter* - more specific and thus *parameter* is not operational while *optional parameter* is. On the other hand, *lower_case_letter* is surely not domain specific at all and thus is not operational. Remember, however, that domain-specific concepts are not part of the background knowledge of a novice. So they need to be identified and acquired during text understanding. This is accomplished through the specialization of already existing concepts (e.g. *dos command* is a *command*, where *command* as a concept is known in advance) or through the integration of unknown words (concepts) without further semantic classification. Note, that our operationalization criterion is *structurally* defined, since the requirement to be the most specific concept is a taxonomic property.

Before going into the details of the learning algorithm, let us discuss the learning result, i.e., the resulting DCG of CD (fig. 8). Note that the generated DCG accepts list representations like ['CD',a,:,user] rather than a stream of characters like ['C','D',' ',a,:,u,s,e,r]. We assume that streams of characters are preprocessed and grouped (by humans) according to the following principles:

1. characters of the same type (e.g., lower case letters) are concatenated (e.g. [d,e,v,i,c,e,:] is transformed to [device,:])
2. characters of different types (e.g., 'a' and '[') indicate transitions, they separate character groups, thus ['C','D','[',d,e,v,i,c,e,:,']'] is transformed to ['CD','[',device,:,']']
3. blanks are separators and can be removed

The command grammar derived from the pruned explanation tree (fig. 7) is given in Figure 8.

```
command → dos_command, opt_param_1, opt_param_2.
dos_command → ['CD'].
opt_param_1 → [Var], {device(Var)}, [:].
opt_param_1 → [].
opt_param_2 → [Var], {path(Var)}.
opt_param_2 → [].
device(_).          path(_).
```

Figure 8: Command grammar of CD

The grammar reads as follows: A DOS command is realized as a rule *dos_command* (which expands to the terminal 'CD') followed by two nonterminials, i.e. the optional parameters. The first optional parameter is either a *device* (the first rule of *opt_param_1*) or is not realized at all (the second rule of *opt_param_1*). The second optional parameter is either a *path* (the first rule of *opt_param_2*) or is not realized at all (the second rule of *opt_param_2*).

Now, what a *device* is and what a *path* is are initially unknown to the learner, although (s)he may already have some background knowledge about such domain-specified concepts. In some textbooks the notion of a *path* is itself explained at some length, while in others it is only introduced in passing. We assume here that *device* and *path* are left unspecified initially, and thus assert two non-restrictive (i.e., always succeeding) type predicates (*device(_), path(_)*) to the knowledge base. As text understanding proceeds, these knowledge gaps will eventually be closed, for example from apposition phrases like *device a* in *start the program from device a.* As a result, *device(a)* is asserted to the knowledge base (a Prolog knowledge base, thus *asserta* is used).

The learning algorithm to generate a command grammar from an parse tree is given in Figure 9. It is restricted to those parts that are relevant to our example. Input is a parse tree such as the one from Figure 7, output is a DCG command grammar like the one given in Figure 8. Note that the knowledge specified in Figure 2 is also relevant to the construction a DCG grammar.

In general, each nonterminal node of the parse tree has one of two *rule types* attached to it: *optional* or *exclusive*,

cf. Figure 2. Currently, only *optional_parameter* is of type *optional*. This type information is learned during the construction of the meta grammar, especially while parsing phrases like "optional parameters" (cf. item 4 of fig. 1). All other rule labels are marked by default as *exclusive* (e.g. *dos_command*). Additionally, each preterminal node has a *leaf type* attached to it. There are two leaf types: *constant* and *variable*. For example in *dos_command(CD)* the type of CD is *constant* since *dos_command* is a *command* and commands are known to have unique names. Conversely, *optional_parameter* is a *parameter* and parameters denote variables. Given a tree with the tree structure *tree* = *node(..,subtree_i,..)*, where $i > 0$, and *subtree_i* is either a nonterminal with a tree structure or a terminal, i.e., a leaf:

1. transform(tree): each nonterminal node forms the head of a new rule. The rule skeleton is <*node_label*> → .. (*node_label* is a unique identifier, if *node* is unambigous in *tree*, *node_label* = *node*, e.g. *command* is unique with respect to the tree from fig. 7 thus *command* → .. and not e.g. *command_1* → ..)
2. if *node* is (also) a preterminal, i.e., *tree* = *node(subtree)* where *subtree* denotes a terminal, then
   (a) if the leaf type of *node* is *constant* (this is also true for the label *unknown*) then construct the body in the following manner: put the leaf entry in square brackets to be read off from the input string (e.g. *dos_command* → *['CD']*).
   (b) if the leaf type of *node* is *variable* then expand the body to *[Var], {Type(Var)}*, where Type is the name of *leaf*. Assert *Type(Var)* to the knowledge base as a predicate. For example: optional_parameter(path) is expanded to: *opt_param_2* → *[Var], {path(Var)}*
3. if the rule type of *node* is *optional* (additionally) generate a head with an empty body (i.e., < *node_label*> → *[]*)
4. if *node* is not a preterminal then for all $subtree_i$ of $node(.., subtree_i, ..)$ generate a unique label, attach it to the rule for *node_label* such that the label of $subtree_i$, $label_i$, appears in the rule body before the label of $subtree_{i+1}$, $label_{i+1}$, and call transform($subtree_i$).

Figure 9: A sketch of the learning algorithm

Acquiring the Semantics of System Commands

So far, a context-free grammar of the syntax of a command has been acquired. However, reasoning and planning processes depend on the semantics of commands as well. The semantics of a command such as CD or COPY normally cannot be explained in a brief sentence. The general function of the command, its preconditions and effects, the role of each parameter and the dependencies among them - all these things need to be introduced.

We capture this by augmenting the command DCGs with semantic and conceptual structures by means of a component called the DCG refiner. This module, however, is not yet a stable part of our model, we are still exploring and experimenting with various approaches and data formats. To give an impression of the learning tasks and the problems that need to be solved, we describe the model's present state.

In the following we distinguish the command parser (COM parser), which - given a command grammar like the CD grammar in Figure 8 - parses command examples like *['CD', a,:]* from the natural language parser (NL parser), which is a unification based, FUG-style parser (Kay, 1985) that parses the English explanation of the given DOS command e.g. *CD changes the ....* The COM parser assigns parse trees to command examples, while the NL parser assigns (semantic) feature structures to sentences. Both results are combined to construct augmented DCG rules of the form: rule(SemanticStructure) → ... . *SemanticStructure* is a common feature structure composed of attribute-value pairs (written: *attribute=value*) that capture morpho-syntactic information but are also used to incorporate the lexical semantics of words. Feature structures are manipulated with a single operation called *unification*. We use a two place unification procedure called *merge*: *merge(A,B)* combines A and B such that - if unification was successful - both denote exactly the same set of features[3].

Consider once again the CD command and its definition: *CD changes the current directory.* The COM parser yields the parse tree *dos_command(CD)*. The NL parser generates the following feature structure (only the relevant features are kept): [type=change_loc, source=[directory=Dir,property=current|_]|_]. This feature structure is derived from the semantics of the verb *change* and the nominal element *current directory*. *Change* denotes a *change_loc(ation)* situation with a *source* object that is a *directory*. We currently do not incorporate planning knowledge (e.g. the preconditions and effects of the *change_loc* situation) into these representations, although this is planned for the near future. Figure 10 shows the augmented DCG for CD. Each rule has received a semantic argument (*SemC, SemO1, SemO2*), which are merged to an overall semantic structure *SemC*.

```
dos_command(SemC) → ['CD'], ;;; replaces dos_command → ['CD']
      {merge(SemC,[type=change_loc,
                   source=[directory=Dir,property=current|_],
                   opt_param_1=SemO1, opt_param_2=SemO2|_]|_])},
      opt_param_1(SemO1),
      opt_param_2(SemO2).

opt_param_1([device=Var|_]) → [Var],{device(Var)},[:].
opt_param_1([device=unspec|_]) → [].
opt_param_2([path=Var|_]) → [Var], {path(Var)}.
opt_param_2([path=unspec|_]) → [].

device:(_).          path(_).
```

Figure 10: DCG with semantic structures

Note that the variable *Var* in the optional rules is not yet connected to any variable at the top level rule *command*. We can give only a brief and incomplete description of the DCG refiner.

Features structures are assigned to a rule level (top rule, embedded rules) according to the following principles:

[3] Feature structures are modeled in form of open prolog lists, i.e., [...|_]. Prolog variables are written in initial caps.

1. initialization stage (the first augmentation of a command DCG with semantic structures):
   (a) the top level rule is augmented with those features that come from the verb concept (e.g., *change_loc*).
   (b) The (optional) rules are assigned their test predicates (those enclosed in braces) as features. For example, rule *opt_param_1* receives *device=Var* as a feature.
2. refinement stage (subsequent modifications of a DCG):
   (a) *feature structure augmentation*: Features are attached to the least general level under the condition that all already encountered (stored) positive examples can still be captured by the command grammar.
   (b) *feature lifting and feature lowering*: Features can be lifted from deeper to higher rules and vice versa. If, for example, a new command instance is encountered that does not match the semantics at a given level, the mismatched parts are pushed down to compatible rules.
   (c) *grammar specialization and grammar generalization*: Examples might also trigger the assertion of new rules (rule order: more specific rules precede more general rules). This is a grammar specialization. Also, rule generalization is possible if two or more rules broadly share the same feature structures.

Related Work

In this paper, the acquisition of a formal language is focused on. However, our approach differs from other approaches from computational learning theory (e.g., Gold (1967)). In our case the grammar is not learned by way of induction from the processing of exhaustive examples. Instead, the grammar itself is provided according to some verbally introduced notational conventions and further explanations in the form of natural language texts.

Other approaches to knowledge acquisition from texts are concerned with the extraction of *declarative* knowledge necessary to define new concepts (e.g. Gomez (1995), Hahn et al. (1996)). In this paper, however, we are concerned with the acquisition of *procedural* knowledge as Norton (1983) is. In his work, a paragraph of an elementary textbook on programming was manually simplified and automatically translated by a parser into Prolog clauses. In his system, however, learning is reduced to natural language understanding, which is not sufficient in our learning scenario.

Another system concerned with the acquisition of procedural knowlegde is SIERRA (VanLehn, 1987) which models the acquisition of basic mathematical skills like subtraction. VanLehn demonstrates that the techniques of top down and bottom up parsing can be used to fix knowledge gaps in a procedural network (representing the subtraction procedure). He concludes from his experiments that those sequences of lessons that only provoke minimal gaps in the procedural network are best, since these gaps can then be easily identified and fixed (VanLehn's felicity conditions a term borrowed from Austin's speech act theory). In our system, the grammar to parse the training examples first needs to be learned from natural language input and formal command descriptions, a complex learning task in its own right which has not been considered relevant for SIERRA. Additionally, each training example in our case is accompanied by verbal descriptions which guide the learning process. A good verbal description might establish a kind of felicity condition in our learning environment.

Conclusion and Outlook

Our model is a multi-strategy approach which seeks to explain how programming novices acquire programming skills from textbooks. While we do not expect the learner to have any domain-specific knowledge, (s)he is considered to have the full natural language competence of an adult. Our model incorporates various analysis-based learning methods. Knowledge about the operationalization of knowledge (how to build CFGs from text) is used to induce meta grammars (from notational conventions). These meta grammars are often incomplete due to omissions in the explanatory text. In the sense of explanation-based learning, an incomplete domain theory has been acquired. Command grammars are then acquired by applying the learned meta grammars (bottom up) to formal command expressions in the textbook. An explanation structure is built while parsing the command expressions. Knowledge gaps in the meta grammar are closed by the induction of rules that complete the explanation trees. Next, these completed explanation trees are translated to command grammars in DCG format. Again, background knowledge about the construction of procedural knowlege is used. With the aid of command grammars, command instances and their textual explanations are parsed and used to further refine the command grammar. The *DCG rule refiner* augments the command DCG with semantic structures which are generalized and specialized as new examples are integrated. Eventually, the command grammar captures (programming language specific) syntactic, semantic and conceptual knowledge that can be used in understanding and planning processes in the field of programming. A lot of work remains to be done, however. Especially, an empirical evaluation of the model is pressing.

Acknowledgements: We would like to thank Bryan Jurish and Ina Bornkessel for valuable comments on a version of this paper.

References

Gold, E.M. (1967). Language identification in the limit. *Information and Control*, 10:447–474.

Gomez, F. (1995). Acquiring knowledge about the habitats of animals from encyclopedic texts. In *KAW'95 - Proc. Knowledge Acquisition Workshop*. Banff, Canada.

Hahn, U., M. Klenner & K. Schnattinger (1996). Learning from texts: a terminological metareasoning perspective. In S. Wermter, E. Riloff & G. Scheler (Eds.), *Connectionist, Statistical and Symbolic Approaches to Learning in Natural Language Processing*, pp. 453–468. Berlin: Springer.

Kay, M. (1985). Parsing in functional unification grammar. In L. Karttunen D.R. Dowty & A.M. Zwicky (Eds.), *Natural Language Parsing*, pp. 251–278. Cambridge University Press: Cambridge.

Mitchell, T.M.; Keller, R.M.; & S.T. Kedar-Cabelli (1986). Explanation-based generalization: a unifying view. *Machine Learning*, 1:47–80.

Norton, L.M. (1983). Automated analysis of instructional text. *Artificial Intelligence*, 20:307–344.

VanLehn, K. (1987). Learning One Subprocedure per Lesson. *Artificial Intelligence*, 31:1–40.