THE UNIVERSITY OF WARWICK

Empirical Modelling for Educational Technology: Parser Construction and Construal

By

Hui Zhu

Thesis

Submitted to The University of Warwick
for the degree of Master of Science (Research)

Department of Computer Science

September 2013

Contents

List of Figures

Acknowledgments

I am deeply grateful to a number of people who have helped me in various ways during the writing of this thesis.

First and foremost, I would like to thank my supervisor Meurig Beynon for his unselfish support throughout the preparation of this thesis. I am very grateful for all the comments, discussions, encouragement and feedback. This thesis would not have been completed without his assistance.

I must also acknowledge the support and guidance provided by my supervisors Steve Russ and Mike Joy, particularly in helping me with the structure of the thesis and the literature.

I would also like to thank Nick Pope, Antony Harfield, Jonny Foss and Elizabeth Hudnott for some technical support and previous research students in the group, without whose contribution this thesis would not have been possible.

I would like to thank all of my friends for their support and encouragement.

Declarations

This thesis is presented in accordance with regulations for the degree of Master of Science by research. It has been composed solely by the author and has not been submitted in any previous application for any degree. All work in this thesis has been undertaken by the author unless otherwise stated.

Abstract

In this thesis, we demonstrate the qualities of Empirical Modelling (EM) for educational technology through studying EM construals of parsing. The design and implementation of our parsing models highlights both technical and theoretical aspects of definitive notations. We draw on experience and knowledge behind traditional parsing and EM in an experiment in novel definitive notation design and implementation with two aims: providing tool support for definitive notation development and illustrating EM's perspective on supporting domain learning. We prototype a learning environment for parsing in which users have the flexibility to change parsing models and to make their own construals. Our aim is to enable non-specialists to frame their own definitions and operators, and add new notations to the interpreter. We also identify connections between Tesnière's notion of dependency grammar and Agent Oriented Parsing with scope for interesting future applications.

Abbreviations

AOP Agent Oriented Parser

DoNaLD A Definitive Notation For Line Drawing

DG Dependency Grammar

DMT Dependency Modelling Tool

DN Definitive Notation

Eddi EDEN Database Definition Interpreter

EDEN The Evaluator of Definitive Notations

EM Empirical Modelling

IEM The *Introduction to Empirical Modelling* module

JS-EDEN The JavaScript-based Web-enabled Variant of EDEN

ODA Observable Dependency and Agency

PSG Phrase Structure Grammar

SCOUT A Notation for Screen Layout

Chapter 1 Introduction to EM and Learning

1.1 Introducing Empirical Modelling

This section is a brief introduction to the fundamental principles of Empirical Modelling (EM). Research into EM was initiated by Dr Meurig Beynon in 1983 at the University of Warwick with the design of the definitive notation ARCA [Bey83]. Over thirty years' development, EM has attracted students and researchers. Their models and research theses can be found on the EM website [EMW].

Empirical Modelling is an approach to constructing artefacts to support human thinking. The term "empirical" reflects the fact that the construction is based on experiments, observations and interactions. In contrast to conventional programming, EM models are not preconceived programs, but construals which evolve as the modeller's understanding evolves. A construal is a concrete artefact that is used to embody understanding of a phenomenon [Go90].

EM emphasizes modelling state-as-experienced rather than behaviour. The construction of a model is based on the modeller's own experience of its real-world referent. The modeller observes an external referent, and concurrently builds a computer model that metaphorically exhibits similar patterns of observables, dependency and agency [BS98]. State acts in an important role in the modeller's experience with referent and construal:
1 The interaction with referent and the observation could be meaningless if there is no expected state change perceived.
2 The modeller's understanding of a referent is based on the analysis of interaction with the referent's different states, as chosen by the modeller.
The analysis of state change reveals the inherent structure of the referent as defined by the intrinsic patterns of state change.

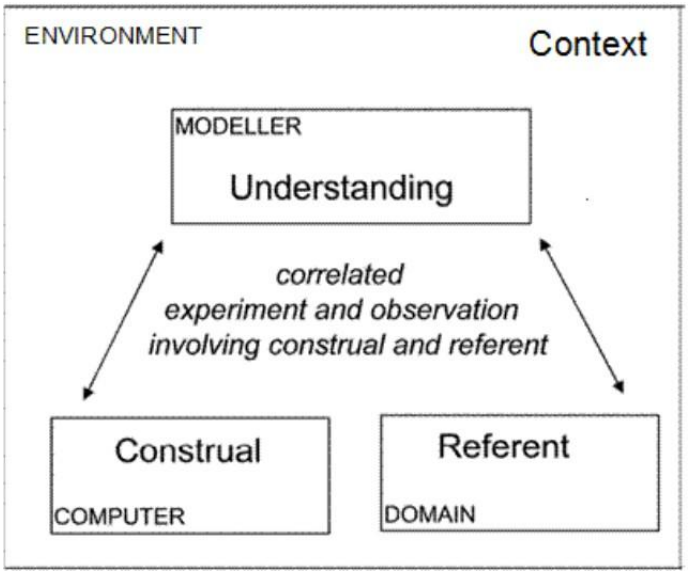


Figure 1.1 Modeller's Interaction with Construal and Referent

Figure 1.1 describes how the modeller gains his or her understanding by interacting with computer-based construal and referent. This kind of interaction is bidirectional and open: the modeller constructs and amends their own construal based on their experiment and observation on the referent; by interacting with the construal, the modeller may discover some connections and then associate the discovery with its counterpart in the referent.

1.2 Observables, Dependency and Agency

The key concepts in Empirical Modelling are *observables*, *dependency* and *agency*.

Observable
Literally, observables mean things you observe. The scope of observables in EM includes more than what can be seen. It encompasses things that are experienced in other ways, like sound, and what can be determined by scientific observations, for instance, an observable is a feature of the situation or domain that we are modelling to which we can attach an identity. It is represented in EM by a variable in a definitive script. "The plausible redefinitions for such a variable are those that have counterparts in interaction with the referent. This gives definitive variables in the script the characteristic qualities of observables rather than abstract programming variables [RO03] ".

Dependency
A dependency is a relationship amongst observables that expresses how they are linked in change. Unlike constraints, which express persistent relationships between values in a closed world, dependencies express the modeller's current expectation about how a change in one observable will affect the value of another [Bey01].

Agent
In EM, agents are sources of changes. An environment is static without interaction [Har03]. An agent is an independent entity that is capable of initiating state-change. For example, consider a room with a door. The door opens. The agent deemed responsible for opening the door may be a person or be the wind.

The observables, dependencies and agency (ODA) in a context form an integrated whole. A collection of the observables represents the state of the model. The network of dependencies among observables frames the intrinsic law for the state changing activity. And an agent acts as an initiator in changing states by changing the value or status of observables.

1.3 Constructionism and Computers for Learning

In his proposal to the National Science Foundation, Papert defined constructionism in the following way. "The word constructionism is a mnemonic for two aspects of the

theory of science education underlying this project. From constructivist theories of psychology we take a view of learning as a reconstruction rather than as a transmission of knowledge. Then we extend the idea of manipulative materials to the idea that learning is most effective when part of an activity the learner experiences as constructing is a meaningful product. " [Pap86]

Constructionism emphasizes the **process** of learning rather than the product [KaR96]. In constructionist learning, learners are active and subjective, they construct their own **mental model** to understand the world. The process of learning is a construction of knowledge from socializing, interaction and making contact with existing knowledge rather than passively receiving information. This part of constructionism is inspired by constructivist learning. Furthermore, constructionism emphasizes the role of activity and experience. Effective learning is based on making: building and sharing physical, virtual and intellectual structures.

Although Papert's definition of constructionism does not refer to computers, the LOGO programming language is used as a vehicle for his idea [Pap93]. Its modern application in education has been closely linked with computer use [BeR04]. In this section, we discuss the computer's role and features that are needed to cope with constructionism learning.

1.3.1 The computer concretizes the abstract

Seymour Papert said, "The computer can concretize and personalize the formal. The computer allows us to shift the boundary separating concrete and formal." This capacity of the computer is significant for learning especially in relation to topics such as mathematics and computing where abstraction has a key role. Some people may have less ability or not be interested in abstract thinking. For such people, it is essential to concretize the abstract. As Papert [Pap93] observes, imagine that in school all the students were required to draw dance steps on the paper every day, without any music or dance floors and that every week they had to pass tests in dancing physically. Would not our world be full of dancephobes?

At the beginning of a physics course students are typically taught the fundamentals of Newton's theorem: When viewed in an inertial frame of reference, an object is either at rest or moves at a constant velocity, unless acted upon by an external force. The students find it difficult to understand the theorem because they do not have experience of this kind of unconstrained motion. If we build a simulated world with little or no friction in the computer, and ask students to play with it, they will get their own experience. In a similar spirit, Mathematics courses in school can emphasize dissociated learning. The core resources are typically texts in books or lectures. To listen to explanations does not necessarily mean to understand. In school, Students study equations of curves and the Cartesian coordinate system by exercises in solving equations. Solving equations is a slow way to gain understanding of analytic geometry,

and does not contribute much in making sense of what the Cartesian coordinate system is. In the computer, we can build a simulated Cartesian system, where the students can program to move objects (cars, bicycles...) by changing their coordinates or setting their path by equations in the system. From the experience of interaction, students will discover that the use of Cartesian coordinates could precisely define the position of objects and that the equation of a curve can be used to describe the path of an object.

There is more to explore in the simulated Cartesian system. Students can create their own vectors, and enable an object to move according to a vector or sum of two vectors. This kind of exploration is self-directed. Through many experiments over a period of time, students will understand how it is that two vectors in different places can be viewed as the same vector, and it is a measure of displacement.

1.3.2 Experience

Although knowledge has been widely divided into different subjects, for a learner, he or she interacts the world with his or her own knowledge system as a whole. The learner compares his or her version of knowledge with the instructor's version, and concludes a new version of knowledge which forms part of the new knowledge system in the learner's mind. In the learner's knowledge system as an integrated whole, different objects and fields are related to each other. George Polya has argued that the general way of solving problems will be a list of heuristic questions: Can it be subdivided into smaller problems? Have I seen a similar problem before? Can this problem be related to a problem I already know how to solve? [Pol08] Learning is similar. When the learner encounters something new, the learner subdivides it into smaller pieces of knowledge and finds relationships between the new knowledge and the knowledge in the structure in his or her mind. In the Cartesian coordinate system example and Newton's theorem of motion example, the role of experience is significant. Because experience offers a chance for the learner to feel and to perceive connections between the new thing and his or her own knowledge. Scholars view learning as **active**, where learners should learn to discover principles, concepts and facts for themselves [Ack96]. There are many other ways to make the learner know the connection between the new thing and his or her own knowledge system, like doing course exercises, reading textbooks, etc.. The significance of experience is that experience makes an impression that can be immediately intelligible, even at a time when you may not understand what is said in the book and taught by the teacher. In the example of learning Newton's law of motion, many students find it difficult to understand. But for students who like ice-skating, they have experience that on ice there is an external force needed to stop moving. They will find a connection between Newton's law and ice-skating, and moreover find a connection between ice (slippery, small friction) and an inertial frame of reference. Vygotsky said that the greatest

moment in our mind's development occurs when speech and action converge. Before this convergence occurs, speech and action are seen as independent lines of development. Through actions a child constructs meaning for itself, while speech connects this meaning with the common world of his or her culture [Vyg78].

1.4 EM and Constructionist Learning

In section 1.3.1 and 1.3.2, we discussed two reasons why the computer can play an important role in constructionism. However, the use of the computer has not fulfilled the aspirations for constructionism in practice. "One reason why conventional program such as the LOGO program originally studied by Papert – fail to support constructionist learning as powerfully as we like is that they are built with quite different objectives in mind and according to inappropriate principles"[BeH10]. In a microworld which merely supports planned user actions and preconceived interpretations [BeR02], the learners have to adapt to the features of the program and its mode of responding to the user. In programming with LOGO, the child has to think in a procedural way, and to write a step-by-step algorithm. A typical approach is for the teacher to present some basic procedures and ask the pupils to write code to draw certain shapes as homework. This may prove stressful, because coding procedurally to draw a complicated shape is not easy to think about or necessarily well-motivated. In Papert's view, this kind of activity concretizes the formal through obliging one to think of oneself as the turtle. A circle is defined in LOGO by the turtle's repeating action: FORWARD a little, TURN a little. Papert has argued that through such activity, children develop a sense of differential geometry, and of the formal concept of 'point mass' in physics [Pap93]. Whilst this may be true, Logo still operates as a closed microworld which obliges children to adopt a particular perspective. No matter how the child may think of a circle, it has to be coded in a step-by-step algorithmic way. This seems to restrict the ways in which a shape can be viewed.

Constructing via Empirical Modelling, by contrast, is an open-ended story. EM supports a wide variety of learning activities. The EM approach to modelling is making construal. "A contrual is similar in character to the structure conceived by Papert, but with essential qualities that are not apparent when we interpret it as a conventional computer program [BeH10]". As illustrated in Figure 1.1, a construal is very personal, it is based on the modeller's own experience of important features of a situation, and how they are dependent on each other. In making a construal, the modelling is not goal-oriented or preplanned, and there are plenty of interactions among the modeller, the construal and the referent which are not preconceived. The construal changes as the modeller's understanding changes and as it is deployed by different modellers. There is not a strict result or ending for a construal. The EM approach is more about the process of interaction and construction, rather than the product and function of the model built.

As a modeller's real interest is in some aspect of the world, programming abstractly may distract him from his own interest in the process of constructing knowledge. This is a situation similar to that I encounter when I am expressing my idea in a foreign language, which has different principles of thinking from my own language. As Bret Victor has argued, a human should not think like a machine. As Papert has argued, the first step in learning is relating what is new and to be learned to something you already know. This is consistent with William James's idea that [Jam96], what is included in experience is the **conjunctive relations** by which we associate one element with another. This kind of activity is similar to identifying new observables and their relationship to existing observables. In contact with the world around, we observe and interact with the elements constituting the environment, and gradually construct a mental structure holding these elements. In parallel, the network of dependency in the construal is a vehicle for our mental structure. EM is in general more suitable than other approaches because model construction is more intimately linked to the development of domain understanding [Roe03].The definitive notations (DN) is more closely linked to human intelligence. One illustrative example was introduced by Edward Yung, the original author of the principal EM tool, the evaluator of definitive notations (EDEN) [Yun89]: Suppose that you want to place a lamp at the centre of a table. When we want to move the table, in our mind, the lamp naturally follows the table to the new position. Humans know something by its relationship to other things and "define" it in this way in their mental models.

1.5 Parsing and EM for Educational Technology

The theme of this thesis is using parsing to support EM for educational technology. The choice of parsing as the central topic is motivated by two research questions:
1 What is the most appropriate way to implement parsers for EM tools?
2 In what respects, can construals of parsing illustrate an EM perspective on learning?

In this thesis, we are going to study different construals relating parsing to EM. They link parsing to EM in several ways. Chapter 2 discusses an unconventional parser, the Agent Oriented Parser (AOP), developed by Chris Brown [Brn01] and Antony Harfield [Har03], that was designed for parsing in the spirit of EM. In chapter 2, we also discuss dependency grammar and its potential in processing natural language. Chapter 2 also highlights interesting parallels between Agent-Oriented-Parsing and dependency grammar. This idea is novel and merits much further investigation. Chapter 3 describes a construal of a traditional parser in EDEN, which implements all the activities in the traditional parser by explicit changes of state. It includes a description of parsing model construction, which shows how EM principles can be applied in traditional parsing construal. This can be a reference for future work in making parsing models of different grammars. Chapter 4 is a construal of a traditional parser in JS-EDEN. It makes a technical contribution by way of specifying new structures in JS-EDEN. Chapter 5 is an exploration of future directions for parsing in

an EM context. It illustrates EM's suitability in supporting constructionist learning by study of with reference to the potential use of parsing models in teaching.

The theme of Empirical Modelling for educational technology is well established. The first model JUGS [Bey88] that indicated the potential for applying EM principles to educational technology was developed in 1988. The first paper that elaborated this potential was "Empirical Modelling for Educational Technology" [Bey97], published in 1997. The paper discussed the connection between EM and the learning process, the connection between EM and constructionism, and an agenda for future applications of EM to education. Doctoral theses by Chris Roe [Roe03] and Antony Harfield [Har08] have consolidated the association between Empirical Modelling and learning. As explained on the EM website [EMWP], these studies particularly concern:

- the role that dependency has played in educational software;
- the importance of learning activities based on constructing interactive artefacts in technology-enhanced learning;
- the scope for applying EM principles and tools in blending the roles of the teacher, learner and developer in relation to educational software [EMW].

Further information about these studies can be found in the account of educational technology as an application area for EM at the EM website [EMWP].

Although many research reports have been written and many models built to illustrate the role of EM as an educational technology, the EM tools are not very mature. As the primary tool for EM, EDEN has a number of features that can pose problems for the novice learner:

(1) EDEN needs to be downloaded and installed.

(2) EDEN has its own distinctive syntax.

(3) EDEN incorporates many definitive notations (Eden, Scout and DoNaLD) as well as some other domain specific notations.

Web EDEN is a web-enabled version of the EDEN interpreter. Web EDEN was developed by Richard Myers in his final year computing systems project at the University of Warwick in 2007-8 [Web EDEN]. It solves the portability issues in downloading EDEN and models to a limited degree. The role of the clients in Web EDEN is restricted to visualization, which places a large computational load on the server.

As a new tool for Empirical Modelling, JS-EDEN was first developed by Tim Monks in his University of Warwick MSc dissertation [Mon11]. JS-EDEN allows definitive

notations to run in a modern browser. There are some advantages of JS-EDEN over EDEN and Web EDEN:

(1) The online nature of JS-EDEN is potentially important. There is no need to download the software and projects.

(2) JS-EDEN is easier to extend or modify. JavaScript code can be embedded into EDEN code, which makes it easier to connect EDEN to rich browser APIs.

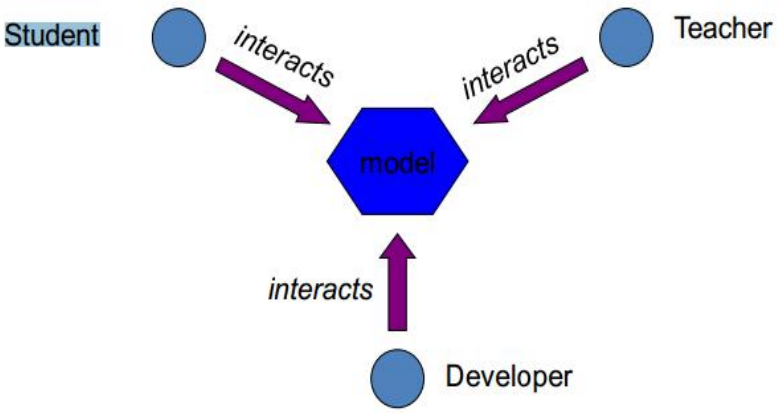


Figure 1.2 Model Building in Teacher, Student and Developer Roles

Our motivating hypothesis is that the limitations of EM tools are the primary reason why previous work discussing the application of EM principles and tools to educational technology has not led to wide adoption of the approach. The overall objective of this thesis is to examine how online application of EM based on JS-EDEN can be more effectively supported and how this can help to disseminate and evaluate EM principles for educational technology and promote wider adoption. Particular attention is paid to some specific objectives which reflect the characteristics of EM principles (cf. Chapter 5):

- exploring EM's role in supporting online learning in a constructionist spirit.
-illustrating how learning can be supported both through making and through exploring construals.
-integrating the developer, teacher and learner roles (cf. Figure 1.2). Whilst this has been shown to be possible in principle, better interfaces to EM tools are necessary to enable teachers and learners to engage in model-building.
-making the model-building role more accessible. In particular, developing a model-building environment that shows meaning clearly and inspires users to focus on thinking about high level concepts.

As the title of the thesis indicates, parser construction and construal is a key theme. The motivation for choosing this theme is that parser construction will help to

improve the quality of JS-EDEN as a replacement for EDEN and Web EDEN. The parsing theme is explored from two perspectives:

(1) to illustrate the effect of applying EM principles and tools for educational technology by studying parsing as a topic to be learnt.

(2) to enhance the expressive power of JS-EDEN by constructing parsers written directly in EDEN.

The contribution of this thesis is described in more detail in the concluding chapter. All the ideas underlying these contributions are my own original work. In some instances, where my ideas are complex, and have been first conceived in my own language, I have had help from my supervisor Meurig Beynon in expressing them in English.

Chapter 2 Reconceptualising Parsing in EM terms

2.0 Introduction

This chapter discusses Agent Oriented Parsing (AOP) as a basis for parsing using EM principles similar to a human's way to understand language. This chapter also introduces the concept of dependency grammar (DG), its modern development and its important role in parsing natural languages. In contrast with traditional parsing and phrase structure grammar, as originally introduced by Noam Chomsky [Ch57], AOP and dependency grammar emphasize the totality of a sentence or a string and the connections between the words or the constituents. The nature of these connections differ according to which theory of dependency grammar is adopted (cf. section 2.2.3), and can be based on both syntactic and semantic considerations.

2.1 Agent Oriented Parser

The Agent Oriented Parser (AOP) was first developed by Chris Brown in 2001 [Brn01] and then extended and improved by Antony Harfield in 2003 [Har03]. This novel idea liberated the development of definitive notations [EFS06]. The AOP allows new notations to be added to EDEN by interactively implementing a parser in EDEN itself instead of using the traditional tools and the recompilation of the EDEN platform. The parsing technique of the AOP is different from that traditionally used in computer science applications. It uses a hierarchy of user defined agents.

The way the AOP parses an input is similar to the way in which humans read languages. Traditional parsers read each character, one at a time, can only derive some meaning when the entire string is read. In contrast we humans read a sentence, we do not read each character or word one by one, we scan the sentence, and our mind will register the important words, from which we are able to derive meaning.

2.1.1 Agents

An agent is an independent entity in the domain being modelled that is capable of initiating state-change. In the AOP, agents respond to observed patterns by breaking up the given string into substrings which are allocated to other agents. Each agent will take an input string and a rule, and it will decide if the rule can be applied. If yes, then more agents will be generated to parse the substrings and the result will be passed to other agents. If all the agents are able to apply a rule, then the input is accepted, otherwise rejected.

Each agent is responsible for recognizing particular features of the input, such as instances of key characters. This observation allows the agent to understand something about the input [Har03] and to break the problem into sub-problems. These

sub-problems are allocated to new agents, called child agents, and the observation continues until all the characters of the input have been observed and the parsing finishes.

There are two behaviours of the agent: one is concerned with the generation of substrings and their allocation to agents, the other is concerned with the semantic actions.

2.1.2 Rules

A rule defines the action an agent takes to parse an input. Every rule has a pattern to match, and an operation to describe how the pattern is matched. The basic operations are:
Prefix – match the pattern at the beginning of the string
Suffix – match the pattern at the end of the string
Literal – match the pattern to the entire string
Pivot – match the pattern in the middle of the string (first occurrence)
Split – match the pattern at all occurrences in the input

The execution of a rule makes up a tree of agents each of which can perform semantic actions in two different ways: when the agent is initially invoked, or after all its child agents have finished their actions [Efs06]. These two different kinds of semantic action are named as *now-action* and *later-action* respectively.

A typical rule template is an EDEN list and consists of an operation, pattern, allocation to child agents, and actions.

```
rule1 is ["pivot", "=", ["agent2", "agent3"] ,
        ["action",["now", "$v is $p1-$p2;"]],
        ["fail",rule2]]
```

The *action* tag specifies that the list next to it is for action, and the *fail* tag specifies the next rule to try if this one could not be applied.

2.1.3 Blocks

There are situations in which we do not want an agent to match a pattern inside a block. For example, consider the input string (1+2)+3*4. We want the agent to first observe the "+" outside the brackets. Therefore a block is defined: the beginning of the block is marked by a left parenthesis, and the end of the block is marked by a right parenthesis.

2.1.4 Scripts

A script in the AOP defines what actions an agent performs in the environment. Without scripts an agent does not perform any actions. The script can be EDEN or any other notation. The scripting language is specific to each implementation and its behaviours cannot be generalized [Har03].

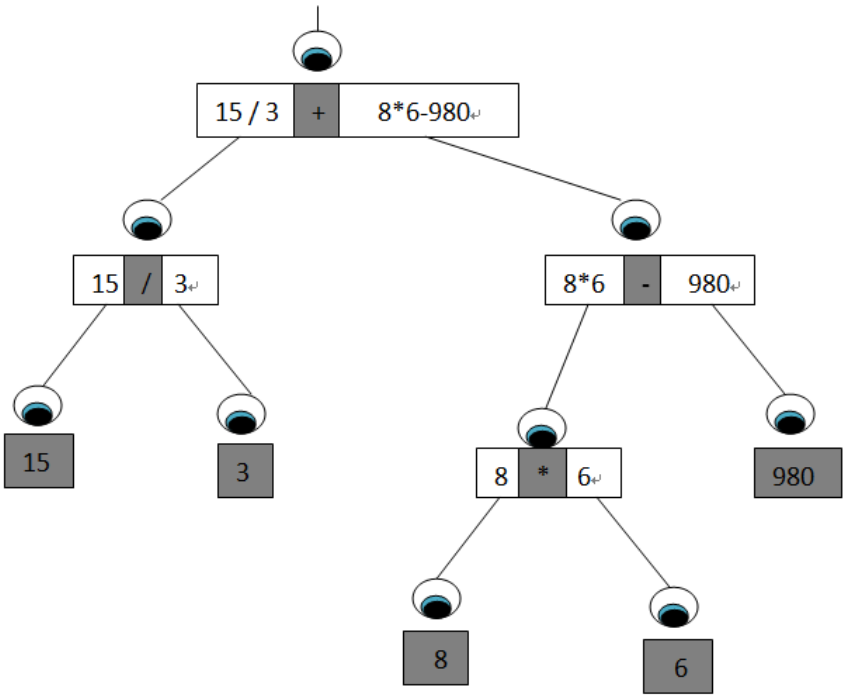


Figure 2.1 Parsing expression in AOP

The above diagram, reproduced from [Har03], describes how AOP parses the string 15/3+8*6-980. Each eye in the figure acts as an agent.

2.2 Dependency Grammar

The developers of the AOP suggested possible links with natural language processing. The author subsequently recognized potential links between the AOP and the well-established notion of dependency grammar. This section discusses the original concept of dependency grammar developed by Lucien Tesnière and its modern development. It points out that dependency grammar has advantages in natural language processing: the adequacy of dependency grammar as a way of describing language structure; and its ability to reveal the connections between elements that construct a sentence. This emerges from a comparison between dependency grammar and phrase structure grammar.

2.2.1 Lucien Tesnière and Dependency Grammar

Although, according to Wikipedia, the term dependency in the grammatical sense that we use it today may have been used first as early as the 12th century, modern dependency grammar begins with the work of Lucien Tesnière. His main thought is embodied in his major work *Éléments de syntaxe structurale*. This work is invaluable in the field of linguistics because many linguists generalize the widespread phenomenon of a language merely from being proficient in one language. In contrast, *Éléments de syntaxe structurale* is based on an extensive comparative study of human language. Being a polyglot, Lucien Tesnière developed many theories and discoveries with universal application in linguistics today. His intention was to establish a syntactic theory crossing the boundaries of different languages, potentially objectively revealing the inherent law of human language.

To Tesnière, all syntactic phenomena reduce to connection, junction and translation. In the article *Comment construer une syntaxes*, as cited by Nivre, he said "The sentence is an organized whole, the constituent elements of which are words. Every word that belongs to a sentence ceases by itself to be isolated as in the dictionary. Between the word and its neighbors, the mind perceives connections, the totality [Niv05]." Tesnière's primary interest was in the relationship between syntactic structure and understanding sentences rather than generating them. In Tesnière's view, the sense of organization of a sentence lies in the connection between the words. Although the connections may not be obvious, they must exist objectively in order for the sentence to be comprehensible to others. To make a sentence is to build a complete network of connections by organizing a set of words; vice versa, to understand a sentence is to identify the network of connections between the words in the sentence. According to Lucien Tesnière, connection is so salient that it offers meaning to a sentence and is its fundamental component. A dependency grammar is a means to specify the connections between words. Since the essence of the study and application of computational linguistics is to imitate human's language processing system and ability using computers, Lucien Tesnière's view about sentence making and sentence understanding ensured dependency grammar's important role in computational linguistics.

The syntactic connections set up the dependencies, which have governors and dependents. The dependency relation views the verb as the structural center of all clause structure. All other words are either directly or indirectly dependent on the verb. The verb is independent.

Tesnière uses stemma - graphical representations in the form of tree like diagrams - to represent the dependency relationships associated with the understanding of sentence structure and syntax [T159]. Figure 2.2 is an example of a dependency tree.

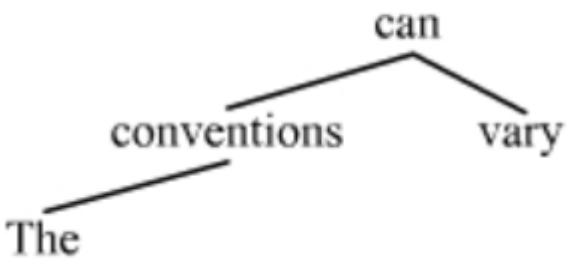


Figure 2.2 A Dependency Tree

In natural language processing, a tree representing a dependency relationship is called a dependency tree, to be distinguished from a tree representing phrase structure, which is called a phrase tree. In a dependency tree, all the nodes are words, not non-terminal symbols. The dependencies between words are manifested directly without using non-terminal symbols, which is more illustrative, concise and easier for the computer to process.

2.2.2 Introduction and Principles of Dependency Grammar

A dependency grammar is a set of rules that can be used to create a stemma for a given sentence. Dependency grammars are associated with parsing natural languages. There are many varieties of natural language, each requiring a different kind of rule-set. For instance, word-order is much stricter in some languages than others. In languages where word-order is strict, there is a close correspondence between the meaning of words and their locations in a sentence. Such languages lend themselves to representations by PSGs. It is when parsing natural languages that are comparatively word-order free that it is essential to take into account the network of connections between the words in a sentence as in a dependency grammar. In general, natural language understanding combines principles of phrase structure and dependency grammars.

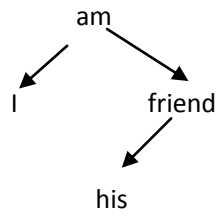
The concept of dependency grammar cannot be defined with reference to a monolithic universal set of rules. It emphasizes the *domination* and *being dominated*, *modification* and *being modified* relations between components of a sentence. However, it does not give precise prescriptions for different types of dependency, which can be informed by considerations that are conventionally seen as 'syntactic' and 'semantic'. Hence, a lot of research has been done on the analytical method (how to associate a stemma with a sentence) and translation principle (how to associate a sentence with a stemma) for dependency grammar.

Robinson formulated four axioms to govern the well-formedness of dependency structures in the article *Dependency Structures and Transformational Rules* in 1970. These four axioms, which laid the foundation for the application of dependency grammar in computational linguistics, are: [Rob70]

1. One and only one element is independent.
2. All others depend directly on some element.
3. No element depends directly on more than one other.
4. If A depends directly on B and some element C intervenes between them(in the linear order of the string), then C depends directly on A or B or some other intervening element.

Dependency grammar describes the connection between the words of a sentence. Assume W is the set of elements of a sentence. For $w_1, w_2 \in W$, $<w_1,w_2> \in R$ asserts that w_2 is a dependent of w_1. The network of dependency is represented by a directed graph (W,E): for each $w \in W$, w is an element of a sentence. If $<v,w> \in R$, then there is an edge from v to w in the directed graph. The relationship between *v* and *w* may reflect domination and modification. A word depends on another either if it is a complement or a modifier of the latter.

As a simple illustration, the dependency tree associated with the sentence "I am his friend." is as shown below.



The verb, 'am', is the centre of the structure, has two dependencies - on 'I' and on 'friend'. 'I' and 'friend' are complements of 'am'. 'his' is a modifier of 'friend'.

2.2.3 Dependency Grammar and Phrase Structure Grammar

In linguistics, phrase structure grammars (PSG) are all those grammars that are based on the constituency relation, as opposed to the dependency relation associated with dependency grammar.

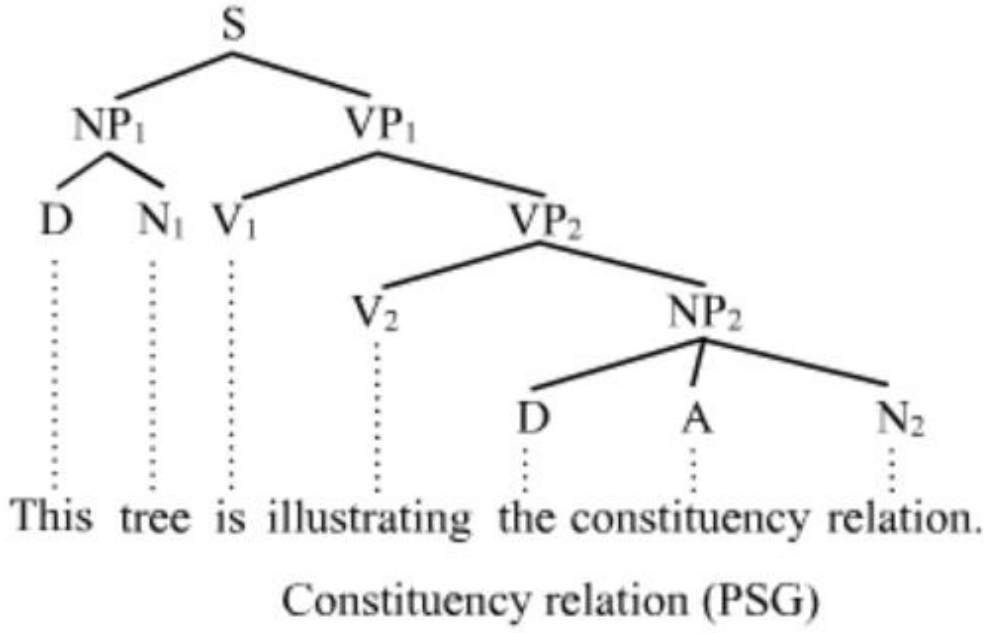


Figure 2.3 Constituency TREE

From figures 2.2 and 2.3, we can find three differences between a dependency tree and a phrase structure tree:

1. A dependency tree does not contain any non-terminal symbols: all of its nodes are terminal symbols-words. A phrase structure tree contains both non-terminal symbols and terminal symbols.

2. The parent-child relation has nothing to do with the order of the words. In phrase structure grammar, child nodes, being ordered, are constitutive parts of their parent node. Hence, a dependency tree emphasizes relational structure and a phrase structure tree emphasizes constitutional structure. A dependency grammar cannot reflect the linear order of a sentence directly.

3. There are fewer nodes in a dependency tree than in a phrase structure tree of the same sentence.

Furthermore, phrase structure grammar focuses on the characteristics of the sentence as actually observed. This means syntactic connections are implicit, making phrase structure grammar limited to analyzing language with relatively fixed order. In a dependency tree, edges are represented as triples (a,b,r) , where a and b are words, r is the directed edge from a to b, showing the dependency between a and b, where a is the *governor* and b is the *dependent*. *"This asymmetry reflects the actual situation in natural language."* said by Peter Hellwig [Hel86]. Unfortunately, phrase structure, which prevails in some grammar formalisms, is at odds with the asymmetric relation between elements. A logical consequence would be to choose dependency as the primary principle of representing syntactic structure [Hud84].

Dependency grammar is adequate for describing the language structure. The principle of grammar is to describe and reveal the connections between the elements which constitute language in whichever way these are ordered. Dependency grammar pays high attention to the relations which underpin language as a system for communication. This characteristic makes dependency grammar easy to extend from a sentences to higher level linguistic units, such as paragraphs and articles. By contrast, for phrase structural grammar, the extension will be difficult. Moreover, from a computational linguists' point of view, phrase structure is strong in sentence generation, but inadequate for sentence analysis.

Hellwig's theory of dependency grammar draws on convergent aspects of various approaches, such as Government-Binding Theory, Generalized Phrase Structure Grammar, Definite Clause Grammar, Lexical Functional Grammar, Functional Unification Grammar and others. These approaches share the following principles: [Hel86]

1. They take into account dependency relations, using notions such as "head" or "governor".
2. They pay attention to functional aspects. The representation of syntactic roles is seen to be a task.
3. They agree that syntax must be lexically restricted and thus place a large portion of the grammatical information in the lexicon.
4. They base their algorithms on the principle of unification, that is: complex categories are brought into agreement in the syntactic context.

Dependency grammar embodies these principles.

2.2.4 Hellwig's Formalisms of Dependency Grammar

Hellwig [Hel80] describes a language that is associated with a dependency grammar as a *dependency relation language*. The terms in a dependency tree are in a one-to-one relationship with the elements of a natural language. Since these elements have a wide variety of semantic features, Hellwig multi-labeled every term in a dependency relation language.

Each feature is labeled as an attribute-value pair, where the attribute is the type of the feature. Relationships are formulated on the level of the attributes. A complex category could have any number of attributes or attribute-value pairs. The key issue is to find a way to group the attributes such that what Hellwig refers to in [Hel80] as 'linguistic phenomena' are represented as fully as possible. In Hellwig's dependency unification grammar, three types of attributes are grouped together:

1. An elementary constituent segment of the rules of syntax for a language, which Hellwig called a syntagmatic role. For example, English uses determiner + adjective + noun, e.g. *the white snow*. In this case, determiner, adjective and noun

are syntagmatic roles.

2. An abstract unit of meaning, which Hellwig called a lexeme. For instance, *eats, eaten, ate* are different forms of the lexeme *eat*.
3. A description of the properties and syntactic category of the corresponding segment in the sentence, which Hellwig called the morpho-syntactic category.

For example, the sentence " *The cat eats fish.*" could be represented as:

```
(ILLOCUTION:    assertion:    clse typ<1>
        (PREDICATE:    eat:    verb fin<1> num<1> per<3>
                (SUBJECT:    cat:    noun num<1> per<3>
                        (DETERMINER:    the:    dete) )
                (OBJECT:    fish:    noun) ) );
```

Each term, printed on a separate line, corresponds to a word in the sentence. The first line corresponds to the period. The dependency structure is depicted by parentheses. The first attribute in each term is the syntagmatic role, the second is the lexeme. The third term consists of a main category, such as verb, noun, determiner, followed by a sequence which represent grammatical features such as finiteness, number, person. The values of person, finiteness and number of the word are indicated by the number attribute in angled brackets. For the word *eats*, its role is PREDICATE and its lexeme is *eat*. It is a transitive verb and is in the third person singular form.

Hellwig augments dependency grammar by adding positional phenomena in natural language via the third attribute, as morpho-syntactic features. Each term in a dependency representation corresponds to a word of the input string. Each subtree also corresponds to a segment which is composed of the words. Breaking down a dependency tree into subtrees creates an implicit constituent structure on the input string. According to Hellwig, any sequential order of constituents of a sentence which can be defined can be described in the set of attributes. Suppose that D is a string mapping to a subtree and H is the string that corresponds to the term dominating that subtree. The attribute "sequence" (seq) is defined as having the values 1: D precedes H, and 2: D follows H. The "adjacency (adj) is established with the values 1: D immediately precedes H, and 2 : D immediately follows H. Last but not least, "delimitation" (lim) with the values

1 : D is the leftmost of all of the strings corresponding to dependents of H, and
2 : D is the rightmost of all of the dependents of H.[Hel86]

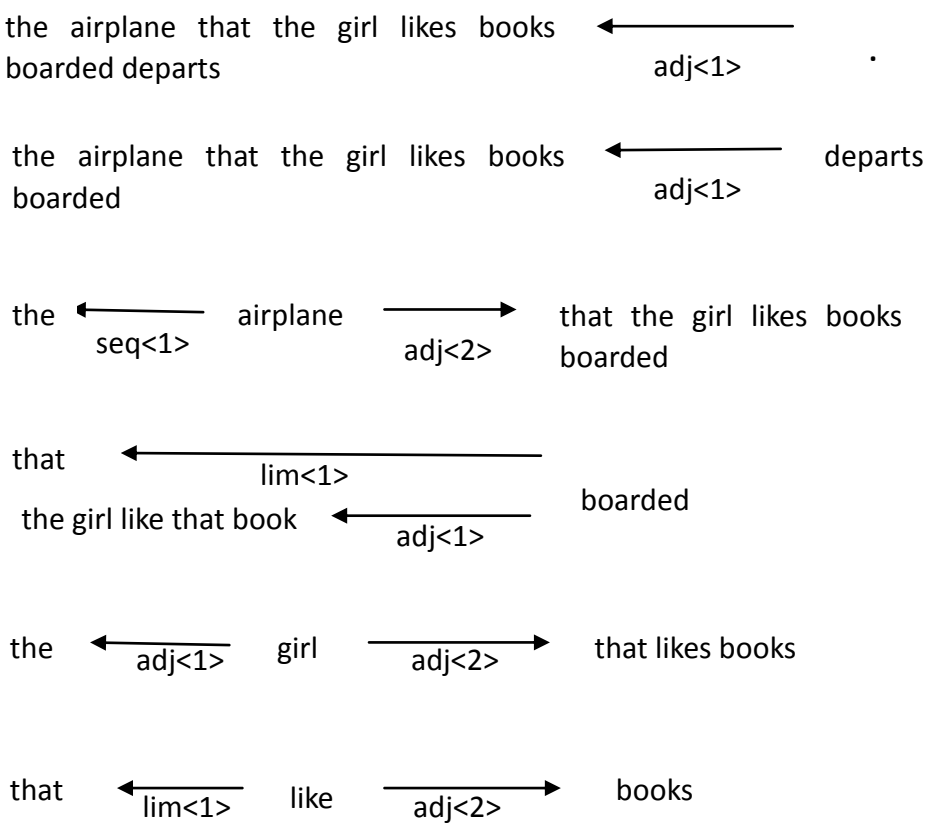
Below is an example:

The airplane that the girl that likes books boarded departs.

The following dependency relation tree depicts the dependencies and the word order

of this sentence.

(ILLOCUTION: assertion: adj<1>
(PREDICATE: depart: adj<1>
(SUBJECT: airplane: adj<1>
(DETERMINER: the: seq<1>)
(ATTRIBUTE: board: adj<2>
(OBJECT: that: lim<1>)
(SUBJECT: girl: adj<1>
(DETERMINER: the: adj<1>)
(ATTRIBUTE: like: adj<2>
(SUBJECT: that: lim<1>)
(OBJECT: book: adj<2>)))))));



There is exactly one sentence that is in agreement with all of the attribute-values in the tree. It is easier to augment dependency trees by coding constituency information than to process dependency features with phrase structure.

2.2.5 Dependency Grammar and Semantics

Tesnière believes that syntax and semantics should be distinguished: grammar is concerned with syntactic structure and semantics belongs to the field of psychology and logic. The independence of syntax might suggest that the theory of dependency

grammar is purely concerned with syntactic structure. But in fact, from a dependency grammar perspective, syntax and semantics are parallel. In parsing language symbols, form and content are concurrent dual interpretations. Viewing syntax as form, semantics as content, studying syntax is for better understanding and describing the content. This kind of parallelism is implicit in the connections within a sentence, and semantic connections are based on syntactic connections. According to Tesnière, dependency grammar expresses the meaning of a sentence by appending the meaning of the dependent to the meaning of its governor. In his theory, there is a natural relationship between semantic connections and syntactic connections, which enhances the scope for processing semantics in a dependency relation language.

Compared with syntax, semantics is hard to process objectively. Semantics is subtle and hard to quantify, this presents a particular challenge for computational approaches. The thesis behind dependency grammar is that, in studying and processing semantics, following the hierarchy of syntactic structure can guide understanding.

Machine translation investigates the use of computers to translate text from one language to another. It is the simulation of a human translation process.

Because of its connection with human interpretation, dependency grammar has advantages over other grammars for computers to process natural languages. In Tesnière's work, there are discussions of the conversion of structures in translations. Translation is not merely a mechanical substitution process, sentences must be considered as a totality. To realize machine translation, appropriate algorithms should be able to translate at a high level. Tesnière thought about translations when creating the basic theory of dependency grammar. The connection between dependency grammar and machine translation goes back to Tesnière's original thought.

2.2.6 Dependency in Dependency Grammar and Dependency in EM

Many intriguing parallels can be drawn between dependency grammar and EM.

In dependency grammar, the term “dependency” indicates that the relation between the words of a sentence is of an acyclic nature. This is consistent with the use of the term “dependency” in EM. In considering the syntactic structure of a sentence in dependency grammar, word and dependency relations are the only units of syntax. The dependency relations connect the elements into a structure. The term "structure" in dependency grammar is very different from "structure" in phrase structure grammar. In phrase structure grammar, child nodes, being ordered, are constitutive parts of their parent node. The structure in phrase structure consists of the basicrecursive structures of sentences, the way in which a sentence can be generated from the grammar [He11]. In dependency grammar, the syntactic structure consists of dependency relations between words that can only be determined by considering the sentence as a whole in its full context.

Generally, a dependency is the syntactic connection between a governor word and a dependent word: the dependent is the modifier, object or complement, while the governor plays a key role in determining the nature of the connection. As formalized by Basile and Caputo: A dependency between governor and dependent is represented as dep(governor, dependent), where dep is the syntactic connection between the dependent word and the governor word. For example,subj(drop, rain) means that "rain" is the subject of "drop". det(rain, the) means that "the" is the determiner of "rain" [BaC12].

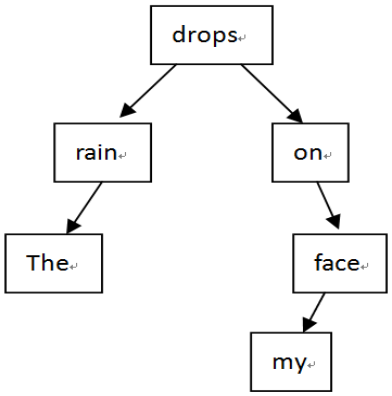


Figure 2.4 Dependency Tree of English Sentence

In the fundamental diagram of EM (cf. Figure 1.1), the connection between the construal (which is something directly experienced on the computer) and its referent (which is something directly experienced in the external world) is itself experienced. When we say we know something this is precisely what we mean – that we are experiencing a connection between one aspect of experience and another. In these respects, EM semantics follows the philosophical stance of William James [Jam96].

Reading a sentence and interpreting it as referring to some situation involves precisely this process of making a connection between one experience (viewing a string of symbols) and another (conceiving some configuration of things to which that experience can refer) [Bey13].

In EM, we think of the Jamesian connection between experiences as being established as a result of a correspondence between observables, dependencies and agency. The words in a sentence can be construed as referring to entities (and this correspondence between words and entities can be seen as connecting observables in the construal to observables in its referent). The way in which we view the dependency relation between words in a sentence in dependency grammar, as depicted in Figure2.3, is parallel to the way in which dependency relations are presented in Wong's Dependency Modelling Tool (DMT) model [Won03].

However when we want to look more deeply into the significance of relationships recorded in the DMT, we need to think explicitly about how *a* depends on *b* and *c* (as in e.g. *'a is b+c'*, rather than simply '*a* depends on *b* and *c*'). This level of detail is not explicitly included in the dependency grammar representation.

In dependency grammar, the sentence is an organized whole, the constituent elements of which are words, and the dependency relations between the words form the structure of the sentence.There is a direct link between syntactic structure and the meaning of the sentence; to understand its meaning is to find connections between words. The process of giving meaning to a sentence is what we refer to as "construal" in the linguistic sense. A natural way in which to construe a sentence is to perturb its structure and see how this affects its meaning. It seems plausible that such activities are used to identify the rules of a dependency grammar.

In EM, from a construal making perspective, the observables, dependencies and agency (ODA) in a context form an integrated whole. A construal, its referent and the surrounding environment are inseparably connected. The structure of a construal consists of dependencies among observables. The structure of a construal is a vehicle for the modeller's mental model of the referent (cf. section 1.4). The modeller's understanding of the referent is reflected in the construal (cf. Figure 1.1). Although the structure of the construal cannot fully represent the modeller's understanding of the referent, there is a direct link between the structure of the construal and the understanding of the referent. This can be disclosed and explored through interaction.

2.3 Parsing Principles of AOP and Dependency Grammar

Traditional parsing and phrase structure grammar have been studied in great detail in computer science. The tools available for compiler construction are mature and have a wide applicability.

The agent-oriented parser has little theoretical basis. It can be likened to how humans parse language, which also has little mathematical foundation (as not enough is understood about the brain) [Har03].

The agent oriented parser allows users to craft grammars for which the input is parsed in a similar way to which humans read languages. This links the AOP approach to parsing with dependency grammar because they are both studies based on a human's way of understanding natural language. In contrast with traditional parsing and phrase structure grammar, AOP and dependency grammar emphasize the totality of a sentence or a string and the connections between the elements.

The conventional parser parses a string by reading tokens from left to right and

matching them in that order. As in natural language, most meaningful strings have a salient feature that is the first to be recognized and which is the most significant when understanding the string [Brn01].

The crafting of a grammar in AOP involves arrangement of rules, which is a reflection of the way in which the language is configured in the modeller's mind. At each step of parsing, the AOP parser observes the most significant token defined by the modeller. The way that a modeller interprets a sentence or a string in the world is reflected in the idea of parsing tokens or words in order of significance. The principle of the AOP is to parse a string according to the modeller's mental configuration, and dependency grammar is a general study of the inherent structure of the sentence. In this way, using AOP as an approach to constructing parsers can be viewed as resembling using dependency grammar to interpret a string or a sentence.

The connection between tokens in AOP is the acyclic relation between the parent agent and the child agent. When dependency grammar is used to analyze natural language, the most important word is the verb of a sentence and the dependency relation between the words is syntactic connection between the governor word and the dependent word. In AOP, the connection is less explicitly defined than in dependency grammar, which depends on the sort of language to be parsed and the modeller's way of interpreting the language. In broad terms, the connection is a partial order. An agent’s input string is dependent on its parent agent because it is less significant than its parent agent.

If we take a look at the parsing tree of AOP and the dependency grammar, they have some similarities. Obviously they are both trees and whose construction is driven by a single linguistic unit at the top. In dependency grammar hierarchy, if *r* is a directed edge from *a* to b, then *b* is dependent on *a*; in AOP, a child agent is dependent on its parent agent.

As explained in section 2.2.4, in Hellwig's formalism of dependency grammar, each element is multi-labeled with three features: its syntagmatic role, lexeme, and morpho-syntactic category. Formally, at the lexical level, a syntagmatic role in a language is associated with the combination of words according to the rules of syntax for that language. For example determiner, adjective, noun in English. A lexeme is a unit of lexical meaning that exists regardless of the number of inflectional endings it may have or the number of words it contains. The morpho-syntactic category is the grammatical relationship between arguments. A parallel can be drawn between the rules of a dependency grammar as characterized by Hellwig and the rules in the AOP. In the AOP, rules set up the connections between parent agent and child agent. Every rule has a pattern to match, an operation that describes how the pattern is matched, and allocations to child agents and actions.

In the example we discussed above:

The airplane that the girl that likes books boarded departs.

the following dependency relation tree depicts the dependencies and the word order of this sentence.

(ILLOCUTION: assertion: adj<1>
(PREDICATE: depart: adj<1>
(SUBJECT: airplane: adj<1>
(DETERMINER: the: seq<1>)
(ATTRIBUTE: board: adj<2>
(OBJECT: that: lim<1>)
(SUBJECT: girl: adj<1>
(DETERMINER: the: adj<1>)
(ATTRIBUTE: like: adj<2>
(SUBJECT: that: lim<1>)
(OBJECT: book: adj<2>)))))));

This resembles what the rules for parsing the sentence in the AOP are like:

```
rule1 is ["suffix", "departs", ["rule2"]];
rule2 is ["pivot","airplane",["rule3","rule4"]];
rule3 is ["prefix","the",["fail","syntax error"]];
rule4 is ["suffix","boarded",["rule5","rule6"]];
rule5 is ["suffix","that",["fail","syntax error"]];
rule6 is ["pivot","girl",["rule7","rule8"]];
rule7 is ["prefix","the",["fail","syntax error"]];
rule8 is ["pivot","likes",["rule9","rule10"]];
rule9 is ["prefix","that",["fail","syntax error"]];
rule10 is ["suffix","book",["fail","syntax error"]];
```

2.4 Conclusions

This chapter studies the relationship between dependency grammar and AOP, which is the first piece of work bringing these two concepts together. It points out the similarities between parsing principles and semantic analysis of dependency grammar and the AOP. Possible future work and development could be the discussion of whether the AOP can be a dependency grammar parser and the study of the AOP and natural language processing.

The formalism of context-free grammars, also called phrase structure grammars, was developed by Noam Chomsky. In Chomsky's generative grammar, the syntax of natural language was described by context-free rules combined with transformation

rules. Phrase structure grammar has strong capability to generate a sentence but is weak in analyzing a sentence [Fe08]. Natural language is much more complicated than language generated by a phrase structure grammar:

- Natural language has a lot of ambiguities. Natural language is not context-free.
- The structure of natural language is complex and varied.
- In natural language, there is a complicated connection between syntactic structure and semantics. However the connection is not the direct correspondence between constituent parts of the sentence and their meanings that is associated with phrase structure grammar.

In *Can a Machine Think?* Alan Turing argued that, we could one day expect that a machine would compete with a human. Showing that the machine is able to process abstract activities, like playing chess, is not the primary point of such competition. Turing maintained that the best way to judge a machine's intelligence is to make it understand English and speak English [Tur56].

Natural language is an important intelligence possessed by humans. The study of natural language processing is not only for translation purposes, but also for communication between the human and the computer in the field of artificial intelligence. Humans use natural language to communicate with others and to think, to understand the world and to create. If in the future humans could communicate with the computer using a common language, the convergence to this common language will involve a seamless blending of artificial and natural languages. There are several questions relating to this possible convergence: Can we use an artificial language to think? Would such an artificial language be closely linked to our principles of thinking? Will we evolve to think as a computer? How can the computer finally process and understand something closer to natural language?

To conclude, it appears that the dependency grammar approach is more promising than the phrase structure approach for processing natural language. These two approaches are conceptually different. There are a lot of unsolved challenges and difficulties in using both approaches. The similarity between the AOP and dependency grammar suggests that the work relates to natural language parsing, but there is not yet enough evidence to prove any such claims. This leaves scope for future research in this area.

Chapter 3 Parsing Models in EDEN

3.0 Introduction

This chapter considers previous work on definitive notation (DN) development in Empirical Modelling (EM) and its limitations. It then discusses the process of constructing a parsing model in EDEN in accordance with construal of the traditional LR parsing method and establishes the difference between EM and traditional approaches.

3.1 Background and Motivations

The original motivation for building Parsing Models is the tool support for EM, to make the DN development easier. This is similar to the motivation for the AOP, as we discussed in Chapter 2. In EDEN (an engine for definitive notations), we have different notations to address different aspects of model-building, like DoNaLd for line-drawing [Bh86], Eddi for relations in a database [Tru96]. To write a model in a new notation, an external translator must be used. The process is cumbersome because it is necessary to write a script into a file and translate that file externally [Brn01]. So a parser written in EDEN directly is desirable.

AOP is a unique piece of research about using EM techniques in parsing and is the only work in the area. However the AOP is not powerful enough. As Efstathiou remarks, "From experience of C-Graph implementation, it is tempting to suggest that in future DN developement, the traditional tools, Yacc/Lex/Bison/Flex, should be preferred when implementing DNs which have rich underlying algebras and complex designs" [Efs06]. This is due to the fact that to develop an interpreter becomes a very difficult task when using AOP.

Similar considerations apply to JS-EDEN (cf. Chapter 4). Since JS-EDEN includes an environment in which to execute JavaScript, there are plentiful JavaScript resources that are ready to be exploited and routine ways of creating definitive notations in JS-EDEN is desirable. As mentioned above, AOP is not sophisticated enough and there is no generalized systematic way to write parsers in AOP. An approach based on deriving parsers for JS-EDEN from traditional LALR parsing techniques is more applicable.

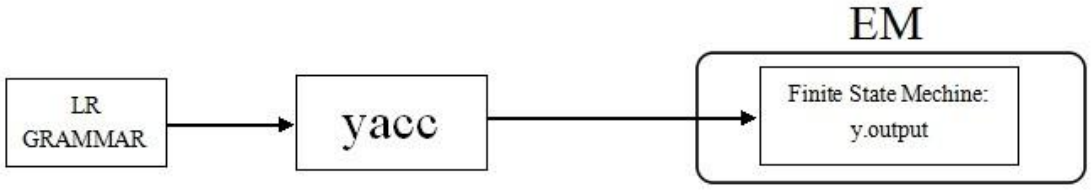


Figure 3.1 Deriving Parsers for JS-EDEN from Traditional Techniques

The approach adopted here is based on the UNIX utility *yacc*. *yacc* is a LALR parser generator. Invoking *yacc* in a suitable way creates a text file called *y.output*: an encoding of the finite state machine generated by *yacc* that shows all of the states in the parser and the transitions from one state to another. As depicted in Figure 3.1, the approach used in this thesis is to make an EM construal of LALR parsing using the core abstract ingredient *y.output* of a *yacc* parser.

3.2 Process of Construction

Developing parsers for definitive notations in JS-EDEN by making an EM construal of LALR parsing is quite unlike conventional parser development. The development of an EM construal is open-ended and exploratory rather than goal-oriented. Goals emerge from the interactions among the modeller, the construal and the referent rather than being preplanned. As will be discussed in Chapter 5, making an EM construal may serve many other goals. This section describes the activity of making a construal of LALR parsing in detail, referring where appropriate to personal experience that has a characteristic role in making EM construals.

The model I conceived at the beginning is a model of a traditional parser which is able to parse a given string and to translate. It should also be in the spirit of EM, so that we have more flexibility to change the parser and the state-changing activity is explicit, intelligible and controllable.

Most parsing methods fall into one of two classes, called top-down and bottom-up methods. These terms refer to the order in which nodes in the parse tree are constructed. In the former, construction starts at the root and proceeds towards the leaves, while, in the latter, construction starts at the leaves and proceeds towards the root. Bottom up parsing can handle a larger class of grammars , so software tools for generating parsers directly from grammars have tended to use bottom up methods. In particular, the parsers generated by *yacc* are using bottom-up syntax analysis [Als86].

3.2.1 Visualization of the Parser

The first ingredient of the construal is the visualization of the finite state machine. The figure above shows a screen shot from an early version of my parsing construal. The visualization of the finite state machine is important because it is a model of the mechanism of the parsing process. The user and the modeller can clearly see the state of the parsing model and available actions in different states. Before using the graph building tool in EDEN, I use DoNaLd to draw finite state machines. States are represented by circles and transitions are represented by arrowed lines.

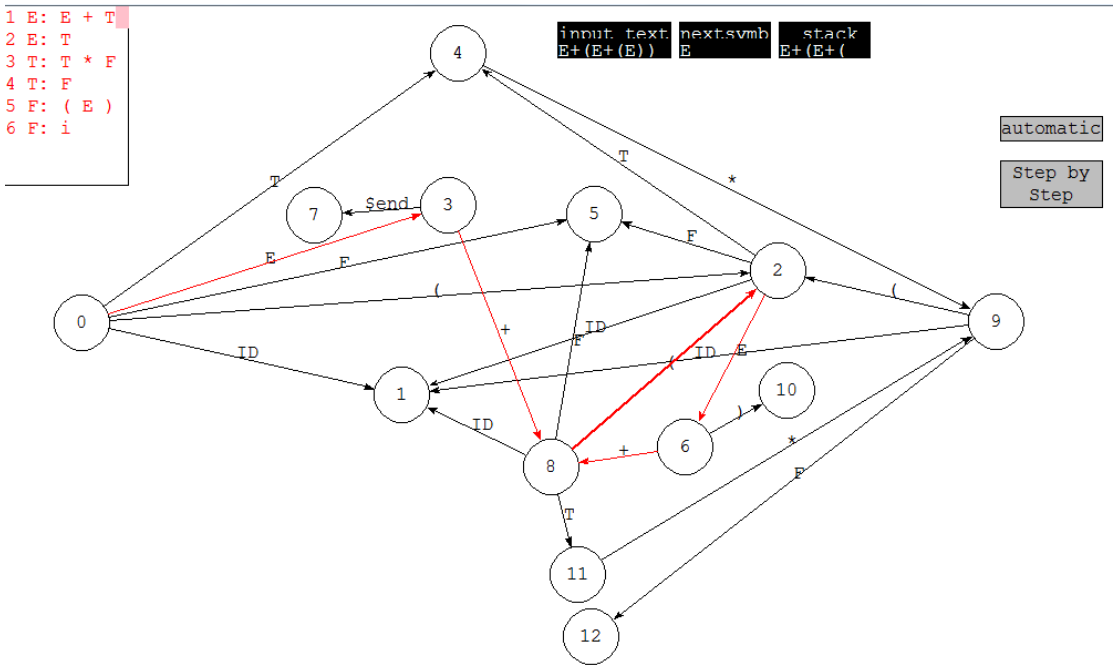


Figure 3.2 The Display of the Parsing Model: Expression Grammar

The left most window shows the grammar of the language and the pink highlight inside shows the position of the current symbol. The three black windows in the middle, from left to right, display the content of the input string, the next symbol to be read by the parser, and the content on the stack respectively. There are two buttons on the right. One is labeled "automatic", and the other "step-by-step". The parser will determine if the input string is acceptable immediately and automatically if the automatic button is pressed, otherwise it will read each symbol at a time and parse the input string step by step.

The attributes of directed edges are observables, the colours of the lines are dependent on whether the corresponding lines have been traversed (red if yes, black if not) and the widths are dependent on the number of times the corresponding lines have been traversed – the more the wider.

However, using DoNaLd to draw a finite state machine is cumbersome and slow if the finite state machine is complicated. One obvious thing is that, in addition to declaring every circle and line, the modeller also has to specify their positions. It is also not generic, which means the modeller has to do much work when he or she switches to a new finite state machine. Having studied the Empirical Modelling project archives, I found that the graph building tool by Charlie Care [Car06] is more concise and more easily transferred to another model. Hence the graph build part is done by using the graph building tool developed by Charlie Care in EDEN, which allows us to add a node to the graph:

```
name_addNode("a")
```

or manipulate the node list:

```
name_nodes = name_nodes // [[ "a", [50, 50], "label a", "green" ]];
```

to add a link to the graph:

```
name_addLink("a", "b");
```

or manipulate the link list:

```
name_links = name_links // [[ "a", "b", "black" ]];
```

One advantage of using the graph building tool is that the modeller does not have to think much about the positions when adding the nodes, because after the nodes are added, the modeller can move them to the ideal positions and save the definitions of all the nodes.

This is the latest interface for parsing models. In the middle, a button called "Error Recovery" allows the modeller to enter into a state to change the input to a correct form in the white textbox. The grey boxes with numbers represent production rules. The colour of the corresponding numbered box will become red if the model should reduce by this rule.

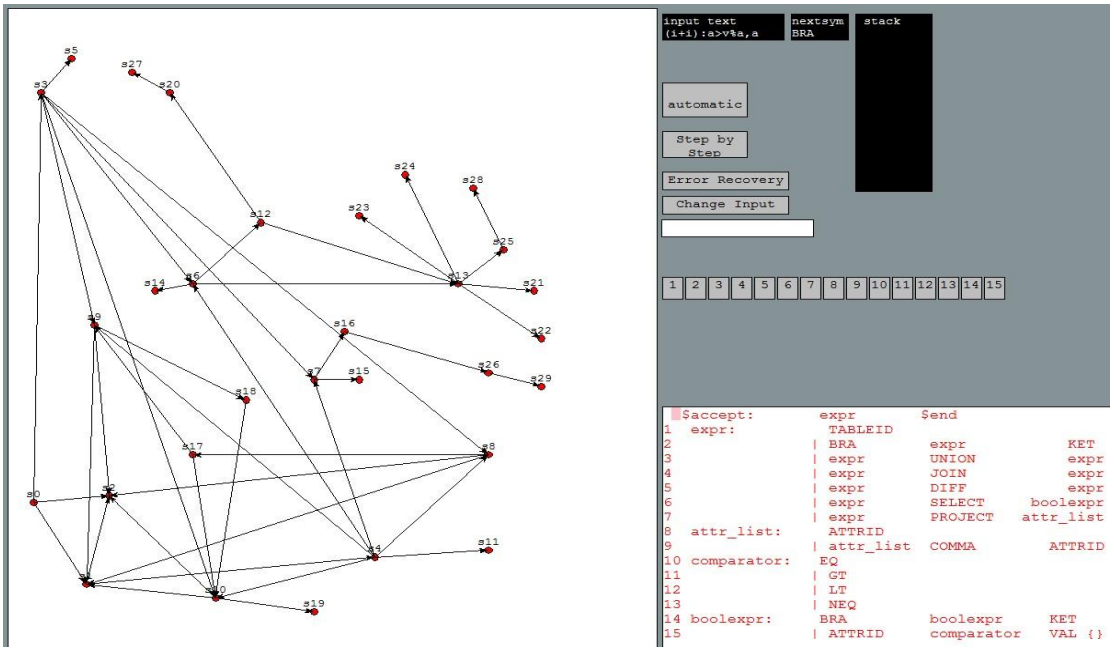


Figure 3.3 Display of the Parsing Model: Eddi Grammar

3.2.2 Information Stored

In our prelimary development we make use of the *y.output* file instead of building our own parsers. The next step is to input all the information in the *y.output* file to EDEN.

The parser produced by *yacc* is part of a compiler that analyzes syntactic structure.

The parser is also capable of reading and remembering the next input token, called the "look-ahead token". When *yacc* is invoked with the ***-v*** option, a file called *y.output* is produced with a human-readable description of the parser. Below is the *exprlist.output* file.

```
 Grammar
    0 $accept: expr_list $end
    1 expr_list: expr
    2          |
state 0
    0 $accept: . expr_list $end
    expr  shift, and go to state 1
    expr_list  go to state 2
state 1
    1 expr_list: expr .
    $default  reduce using rule 1 (expr_list)
state 2
    0 $accept: expr_list . $end
    2 expr_list: expr_list . COMMA expr
    $end   shift, and go to state 3
    COMMA  shift, and go to state 4
state 3
    0 $accept: expr_list $end .
    $default  accept
state 4
    2 expr_list: expr_list COMMA . expr
    expr  shift, and go to state 5
state 5
    2 expr_list: expr_list COMMA expr .
    $default  reduce using rule 2 (expr_list)
```

The machine has only five actions available: *shift, reduce, goto, accept*, and *error*. The *goto* is always performed as a component of a reduce action. The parser operates in the following manner:

1 Based on its current state, the parser decides whether it needs a look-ahead token to choose the action to be taken. If so, it calls a function to obtain the next token.

2 Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto or popped off the stack, and in the look-ahead token being processed or left alone.

All the information in the *y.output* file is stored in two Eddi tables, *transitions* and *reductions.* Each element in the *transitions* table is a 3-tuple, where the first is for current state, the second is for look-ahead token, and the third is for the next state (or the state being pushed onto the state stack). Each element in the *reductions* table is a

pair, where the first represents current state and the second represents the rule by which the parser reduces.

By way of illustration, the information associated with the following extract from the *exprlist.output* file:

```
state 4
    2 expr_list: expr_list COMMA . expr

    expr  shift, and go to state 5
state 5
    2 expr_list: expr_list COMMA expr .

    $default  reduce using rule 2 (expr_list)
```

is stored in the *transitions* table as a 3-tuple element [4,"expr",5], and in the *reductions* table as [5,2].

In fact, it is a laborious task to turn the content of *y.output* file to a file of two Eddi tables if typing manually. It is not convenient for a modeller to read the *y.output* file line by line, and turn it into two Eddi tables. I did some work using the *sed* utility, which parses text and implements a programming language which can apply transformations to such text. It reads input line by line , applying the operation which has been specified via a sed script [Gnu]. A script is written to do the pattern matching in the *y.output* file, and transform it to the format which we want. The following is a *sed* script for creating *transitions* tables:

```
#! /bin/sh
endix=`cat $1 | sed -n -e '/^state/p' | tail -n 1 | cut -d" " -f2`

for (( i = 0 ; i <=  $endix ; i++ ))
do
b=$(expr $i + 1)
a=$(cat $1 | sed -n -e '/^state '$i'$/,/^state '$b'$/p' | wc -l ) ;
cat $1 | sed -n -e '/^state '$i'$/,/^state '$b'$/p' | head -$( expr
$a - 1) > STATE.txt

fname=`echo $1 | cut -d"." -f1`

cat STATE.txt | sed -n -e 's/^\ *//p' | sed -n -e 's/\ .*state\ / ,
/p' |sed -n -e 's/^/\"/p' | sed -n -e 's/\ /"/p' | sed -n -e
's/^/transitions << ['$i', /p' | sed -n -e 's/$/];/p' >>
mktransitions$fname.txt
done
rm STATE.txt
```

This script first does the pattern matching and stores into the *state.txt*, then transforms the content in the *state.txt* into Eddi notation, saving the changes in a new file, and finally removes the *state.txt*.

3.2.3 Observable and Dependencies

Before composing the parsing models, I studied some features and actions of the traditional shift-reduce parser: The parser can read and remember the next input symbol, called the look-ahead. The current state is always on the top of the state stack. Initially, the machine is in state 0 (that is, the state stack contains only state 0) and no look-ahead has been read. The shift is the most common action the parser takes. A shift occurs when a token is recognized in the current state and whenever a shift action is taken, there is always a look-ahead. In a reduce action, all the states that were on the stack corresponding to the right-hand side of the rule are popped from the stack. Then a new state is put on the stack, based on the symbol to the left of the grammar rule. The accept appears only when the look-ahead is the end-marker (*$end*) .

Key Observables:

currpath - stores the content on the state stack as a list.

currstack – current content on the stack as a list.

nextsymbol – represents the look ahead token.

isreduction – reads the transformations table and decides if the parser should reduce in the current state. If there is a shift-reduce conflict, it always chooses to shift.

currrule – the number of the grammar rule the parser determines that all the items on the right-hand side of which have been seen (the production by which the parser reduces) .

currrstate – current state of the parser (the last element of the currpath, or the top element on the stack state) .

InputText – the input string, stays the same all the time.

k – the index of the current symbol in the InputText (how far we have read in the InputText) .

The observables *k* and *j* are used to control how many tokens the parser has read from the input. The values of *k* and *j* are increased by 1 every time the "step by step" button is clicked.

Network of Dependencies:

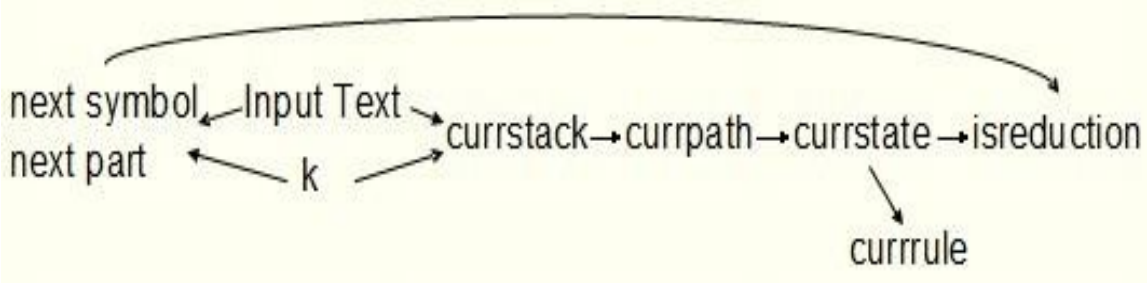


Figure 3.4 Network of Dependency in Parsing Model (1)

The observables whose values are subject to be changed directly by an agent action, rather than being defined by a dependency relation, are *InputText* and *k*. In this case, when reduction occurs, the content of *InputText* and the value of *k* are adjusted by substitution functions. There is a procedure triggered by the observable *isreduction*, and if *isreduction* is true, the procedure will invoke the corresponding reduction rule, which invokes the appropriate substitution rule.

This network of dependency is derived from the real traditional parsers: The input and how far the parser has read from the beginning defines the look ahead token; The shift action pops a token and a state onto the stack; Based on its current state, the model decides whether it needs a look ahead token to carry out the next action, or whether it should reduce.

There are still questions: Should we change the *InputText* when reduction occurs? Should the *InputText* stay the same? If yes, how could we make the model reduce?

There are two reasons why we should keep *InputText* the same. The first reason is to keep a record of the original input string. The second is that there are some definition conflicts when the model does substitution: sometimes *k* does not update immediately after the update of the observables *nextsymbol* and *nextpart*. For example, below is the function *substituteEplusT*. The function takes all the tokens except the top three on the *currstack*, concatenates it with symbol "E" and the *nextpart* of the input text. It implements the grammar rule E : E+T. When the model reduces by rule E : E+T, an agent changes *InputText* by calling function *substituteEplusT*. This action also changes the value of *k*. The change of *InputText* and the change of *k* both cause change to *nextpart* (Figure 3.3). In addition the value of *k* and the value of *InputText* may not be updated simultaneously. If the value of *k* has not been updated immediately (*InputText* has been updated before *k*)，the error message "index out of range" will be displayed. This is due to the fact that *nextpart* is defined to be the substring of *InputText* indexed from *k* to the length of *InputText*, and the delay in updating *k* may then cause the value of *k* to be greater than the length of *InputText*.

```
InputText = substituteEplusT(currstack);
```

```
func substituteEplusT {
    para  currstack;
    //this function takes currstack as parameter
    auto result;                //result is a local variable
    if(currstack#-3>0)
    //if the size of currstack is greater than 3
    result=substr(currstack,1,currstack#-3)//"E"//nextpart;
    else
    result="E"//nextpart;
    k=k-2;
    return result;
}
nextpart is substr(InputText,k+1,InputText#);
```

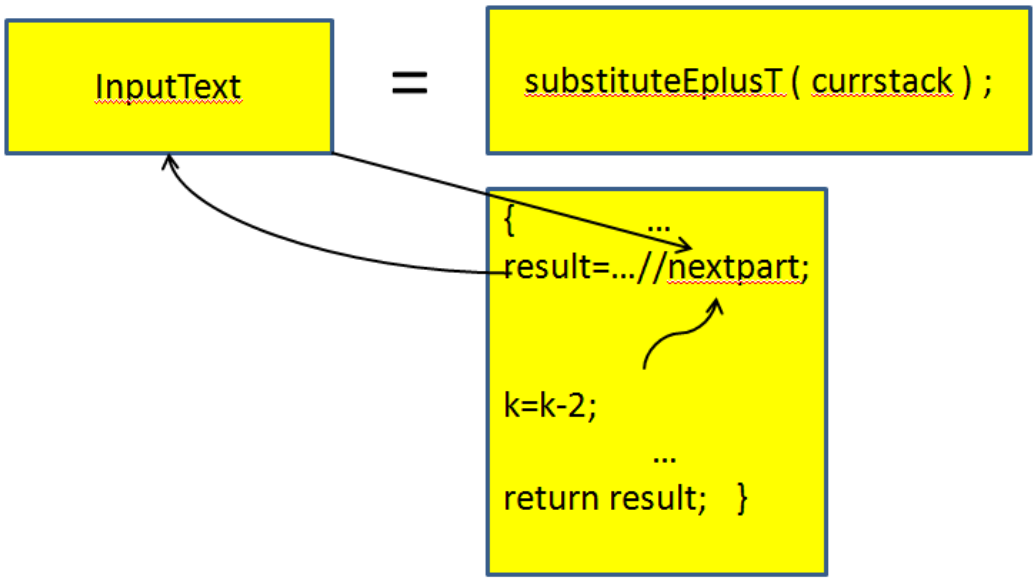


Figure 3.5 ODA in *substituteEplusT* function

Now the problem becomes if we should keep *InputText* the same, then how should the model reduce? We know that both the tokens on the stack and the state stack are dependent on the input, and if the parser reduces, the content on the stack should change and states should be pushed onto and popped off the stack, but now the input does not change any more.

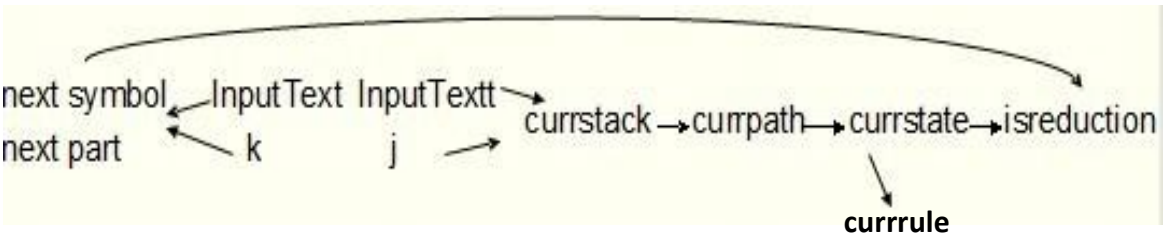


Figure 3.6 Network of Dependency in Parsing Model (2)

To solve this, a new observable *InputTextt* is included. It has the same initial value as *InputText* and changes when reduction occurs. An index *j* is associated to measure how far the parser has read in the *InputTextt*. Like *InputText***,** *InputTextt* is independent of other observables and the observables *currstack***,** *currpath***,** *currstate,etc* depend on it.

3.2.4 Syntax Directed Translation

At this stage of development the parser is ready to determine whether a given input string can be generated by the grammar or not. In order to add notations to EDEN, it should also be capable of doing the translation. A syntax directed definition is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes [Als86]. In addition to the state stack, a parallel stack, the value stack, holds the values returned by the lexical analyzer and the actions. In the parsing models, in parallel with the *currstack* and *InputTextt* , there are two observables : *stackvals* and *Inputtvals***,** with similar definition, which store information about the attributes of the tokens on the stack and in the *InputTextt*. In parallel with the functions and rules for the change of the *currstack* and *InputTextt*, there are substitution functions and rules written for changing the values in *stackvals* and *Inputtvals* according to actions.

Figure 3.7 Observables for Syntax Directed Translation

By way of illustration, consider the following example of syntax directed translation of the string "(A+B) : c>3 %d,e" using the parser model for the Eddi notation:

Lexer Output:

Input:	(i	+	i):	a>	v	%	a ,	a	$(end marker)
Attributes:	i=A		i=B	a=c	v=3		a=d	a=e	

Parser Observables:

```
InputTextt = "(i+i):a>v%a,a";
Inputtvals =["","A","","B","","","c","",3,"","d","","e",";"];
```

Translation into EDEN by the Parsing Model:

```
C is project(restrict(union(A,B),"c>3"),["d","e"]);
```

3.2.5 The Eddi Parser Model

As mentioned before, the information about the parser generated by Yacc is stored using two Eddi tables. The goal is to make a translator that is written purely in EDEN. It is then inappropriate to use Eddi to model the Eddi parser. We have to find another way to store the information. To resolve this issue, lists are used instead of Eddi tables to store all the information about the finite state machine. EDEN supports the nested list structure. There are two lists, one for transitions and the other for reductions. The length of the *transitions* list is the number of states in the finite state machine. The i^{th} entry in the *transitions* list consist of all the possible actions the parser could take when in state i.

In the original parsing model, the functions *makepath, makestack* and the observable *isreduction* were implemented by searching the Eddi table, and returning elements satisfying their conditions. It is not much work to do the pattern matching or condition searching since Eddi is a notation for manipulating relations in a database. When using an EDEN list representation, in contrast, the corresponding functions and observables should be rewritten and the elements should be checked one by one (see Appendix 1 and 2).

Based on the work done at this stage, we shall be able to add the EDDI notation to the JS-EDEN intepreter at a later stage.

3.3 Comparisons with Traditional Parsers and Conclusions

EM parsing models can be compared with models of traditional parsers from three perspectives:

1. An educational perspective:

- Understanding how a parser works. Conventional programming aims to automate everything, while Empirical Modeling makes the state-changing activity explicit, intelligible and controllable. Acting as an agent, the human can be involved in the process of state-changing, initiating state changes. Parsing is the process of determining if a string of tokens can be generated by a grammar. Traditional parsers receive a string, parse it if it is acceptable and otherwise return error, while the entire process of parsing is hidden.

- Understanding how a parser is designed. Making a parser for a language is a design task. We need to formulate the grammar in an appropriate way. And we also need to take the design of defaults into account. For example, when a parser faces a shift-reduce conflict, it should always shift where shift is possible. Empirical Modelling could promote better understanding in designing parsers due to the fact that grammars and default designs are more easily changed. For

example, if we want to change some production of the grammar when designing a parser in Empirical Modeling, we need only change the content of the observable related to this production, and also its connections with other observables. In contrast, in conventional programming, we could have to change many actions. As another example, when writing a language, we want to modify some productions. If after changing it, we give it to yacc, it will produce a new parser for this language. However we would not know any relationships between the old parser and the new one. But in Empirical Modeling we know the relationship between the old one and the new one, as the new parser is an adaptation of the older one.

2. A practical perspective:

- Making or growing a parser on the fly. When we apply Empirical Modelling, we consider parsing to be more related to crafting than specifying rules. Being able to craft a grammar, we can have a richer picture of what we have got and which state we are in. And building parsers on the fly means that we are able to take account of the current situation and to respond.

- Better error recovery and detection. In EM, an input string could be changed during the process of parsing when the modeller recognizes some error in the input or when the model detects an error.

3. A conceptual perspective:

Empirical Modelling is a different approach to computing. In Empirical Modelling, people build models based on a network of definitions and they focus on representing states of models using network of definitions not procedures or actions. "Conventional languages use variables and procedures to model the problem and have a set flow of control for solving it. The variables refer to storage locations that hold data about the objects being modeled and their values collectively represent the current state of the problem. Procedures describe what to do with the data stored in the variables and consist of sequences of instructions whose side effects change the state of the problem" [Bey01]. The computer has to be told explicitly when to change certain variables and how to compute them.

Networks of definitions consist of observables, which hold data representing different objects, and definitions that describe the dependencies between objects. A definition usually defines a target observable by an expression containing one or more source observables. These definitions are handled by the system, which ensures that whenever a observable is changed, the observables that depend on it are updated to reflect this change. The philosophical difference is also a reason why we have more flexibility to change the parsers in EM.

In summary:

Advantages of using Empirical Modelling to build a parser are:
1) flexibility of changing the parser
2) better error recovery and detection
3) scope for creating a parser on the fly

Disadvantages of using Empirical Modeling to build a parser are:
1) An EM parser is much less efficient than a traditional parser.
2) Parser development is not supported by mature tools.

3.4 Conclusions:

The AOP has been used to develop parsers for definitive notations (DN) . It generates parsers that are fundamentally in the spirit of EM. The process described in this chapter achieves the goal of supporting DN development, but does not produce unconventional parsers. The whole process is that we are using a traditional parser generator tool to make a parser, and then model the parser using EDEN. However it is more convenient for a developer to construct a parser based on the parsing model set up above in section 3.2 than to use the AOP, because the design of the parsing model imitates traditional parsers. The environment is also quite generic. It is easy to make the parsing model work on a new grammar.

Although the EM parsing model is not so efficient, efficiency is not the priority from EM perspective. The principal aim is that the parsing model should be controllable, for users to study parsing theory and to design parsers during interactions with model.

Chapter 4 Parsing Models in JS-EDEN

4.0 Introduction

JS-EDEN is a new tool for Empirical Modelling, first developed by Tim Monks in his MSc project [Mon11], which allows definitive notations to run in a modern browser. The name JS-EDEN reflects the fact that the underlying interpreter is for definitive notations, and the underlying implementation for EDEN is using JavaScript. This chapter discusses the process of converting parsing models in EDEN to parsing models in JS-EDEN, the process of extending EDEN code by embedding JavaScript, and prototyping structures in JS-EDEN using JavaScript.

4.1 Motivations

The standard tool for EM over many years has been the desktop interpreter EDEN. For instance, the EM project archive contains over 150 EDEN models, and over the period 2003-2012 almost all the coursework by students on the *Introduction to Empirical Modelling* (IEM) module was in EDEN. All the models submitted in coursework in 2012-13 were developed in JS-EDEN. JS-EDEN is becoming the main tool in Empirical Modelling, acting as a substitute for EDEN. There are some advantages of JS-EDEN over EDEN:

1 The "Modelling in the browser" feature makes JavaScript more accessible to users than EDEN. There is no need to download the software and projects, all of these can be run in browsers.

2 JS-EDEN is easier to extend or modify. JavaScript code can be embedded in to EDEN code, which connects EDEN to rich browser APIs (cf. section 4.3.1) .

3 The JS-EDEN codebase is smaller than the EDEN one.

4 JS-EDEN is more efficient as it performs Just In Time compilation [TM11] . In other respect, JS-EDEN is less expressive than EDEN. Some notations, such as Eddi, DoNaLD and Scout, have not been added to JS-EDEN yet. Tool support for DN development is desired.

In this chapter, we discuss how parsing models similar to that described in chapter 3 can be implemented in JS-EDEN.

4.2 Prototyping Structures in JS-EDEN

In EDEN, the visualization of the parsing model (described in Chapter 3) is implemented by applying the graph building tool developed by Charlie Care, the graph building tool can be accessed in the lab of the IEM course [Car06]. However

there is no corresponding graph building tool in JS-EDEN. Ideally we want to have a counterpart of the graph building tool in EDEN, a form of graph factory which produces and displays nodes and vertices on the canvas.

4.2.1 Design Concept and Implementation for a Graph Object

We are aiming to build a graph tool in JS-EDEN. The first step is to think about what we presently have in JS-EDEN. We can draw circles and lines on the canvas. And with development, we can draw arrowed lines. As discussed in section 3.2.1, drawing finite state machine with these primitives is cumbersome and not generic. Inspired by Douglas Crockford, "You start by making a useful object. You can then make many more objects that are like that one" [DC08], prototyping a structure in JavaScript can solve the problem. If we build a graph object in JavaScript, then inside the graph object, we can make a vertex object and edge object. Furthermore, we associate drawing methods with vertex and edge. This is generic since any vertex, edge or graph we create in JavaScript level will inherit the methods and attributes. The graph tool in JS-EDEN can be implemented in two stages:
1. Building the JavaScript graph prototype;
2.Connecting JavaScript objects to EDEN observables, and maintaining dependencies
The implementation of the graph tool in JS-EDEN elaborated in section 4.2 and 4.3 is novel, and can be a reference for future related work.

The main two constituents of a graph variable are nodes and edges, they are connected with each other, but should be treated separately. Thus it may be preferable to store them in separate data structures. For this kind of variable which contains two different components, we could use an aggregate which contains two separated arrays of nodes and edges (depicted in Figure 4.1 as an illustration). An example of a graph variable defined as an object in JavaScript is:

```
var g = {
   graphName: "_G",
   nodes : [],
   edges : [],
   nodeset: {},
   strNodes:"",
   strEdges:"",

  initialise : function() {
        this.nodes=[];
        this.edges=[];
       this.nodeset={};
   },
```

```
addNode: function(value){

   }

  addEdge: function(source,target){
   }
}
```

4.2.2 Vertex Object and Edge Object

The object *nodeset* is a collection of key value pairs. Once the *addNode* function is called, it will immediately make a new key equal to the value of the node to be added, and define an intermediate node to be nodeset[key]. If this intermediate node is defined, which means the node to be added already exists in the graph object, then nothing will happen. Otherwise, the function *Vertex( )* will create a new node and then the node will be added to the "nodes" array [DC08].

```
addNode : function(value) {
    var key=value;
    var node = this.nodeset[key];

    if(node == undefined)  {
    node = new Vertex(value);
    this.nodeset[key]=node;
    this.nodes.push(node);
            }
        return node;
   },
```

The function *addEdge* exploits a similar mechanism. It first calls *addNode* to create source and target nodes, then calls *Edge(s,t)* function to create and add a new edge to the edges array. *Vertex( )* and *Edge( )* are prototype objects(which makes it easy to attach common functionality to each vertex/edge object), from which we can generate nodes of type (name, x, y) and edges of type (source, target).

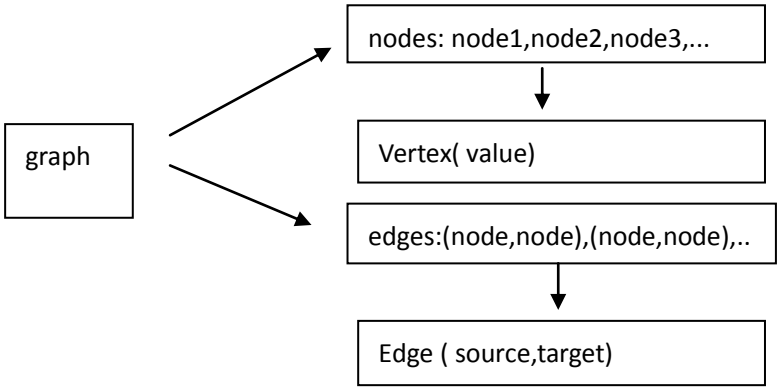


Figure 4.1 Graph Object in JS-EDEN

4.3 Connection of EDEN Observables and Dependencies with the Graph Object in JavaScript

This section discusses the technique used to bind objects and attributes in the graph object discussed in the section above with corresponding EDEN observables. First we need to understand the mechanisms for making dependencies in the underlying JavaScript level.

4.3.1 Definition Maintainer in JS-EDEN

From section 4.2.1, we know that the graph object can be implemented in the JavaScript level. However, users of JS-EDEN have no ways to access it, except by typing pure JavaScript in the console. In order to create a bridge between the graph object and observables in EDEN, a study of the JS-EDEN definition maintainer is needed. There are three issues: How are observables modelled, stored and accessed by name in JavaScript? How are definitions represented? How are dependencies maintained?

Observables:

The underlying implementation of JS-EDEN uses JavaScript. Observables defined in EDEN are modelled using some JavaScript data types. Observables should have a name associated with a value and a definition. This was implemented in JavaScript as prototype object. In JS-EDEN, observables are modelled as "symbol" objects and they all share the symbol prototype [TM11]:

```
symbol.prototype.value=function(){...}
```

Further to this, each symbol also stores other important information such as the symbols that are dependent on it, the agents that observe it and the symbols that it depends on.

Observables are stored in a table, which has identifiers as keys and values as symbols. A reference to an observable in EDEN is translated to a call to the lookup method on the object that represents the symbol table in JavaScript. As an illustration, consider how the observable *a* with definition *b+c* is modelled as a "symbol" in JavaScript as in Figure 4.2.

EDEN: a is b+c;

EDEN supports a number of computational operators for computations which are in common with most programming languages, so most operators in EDEN have directly corresponding JavaScript ones. For instance, addition in EDEN is translated into JavaScript addition as referred by the "+" in Figure 4.2.

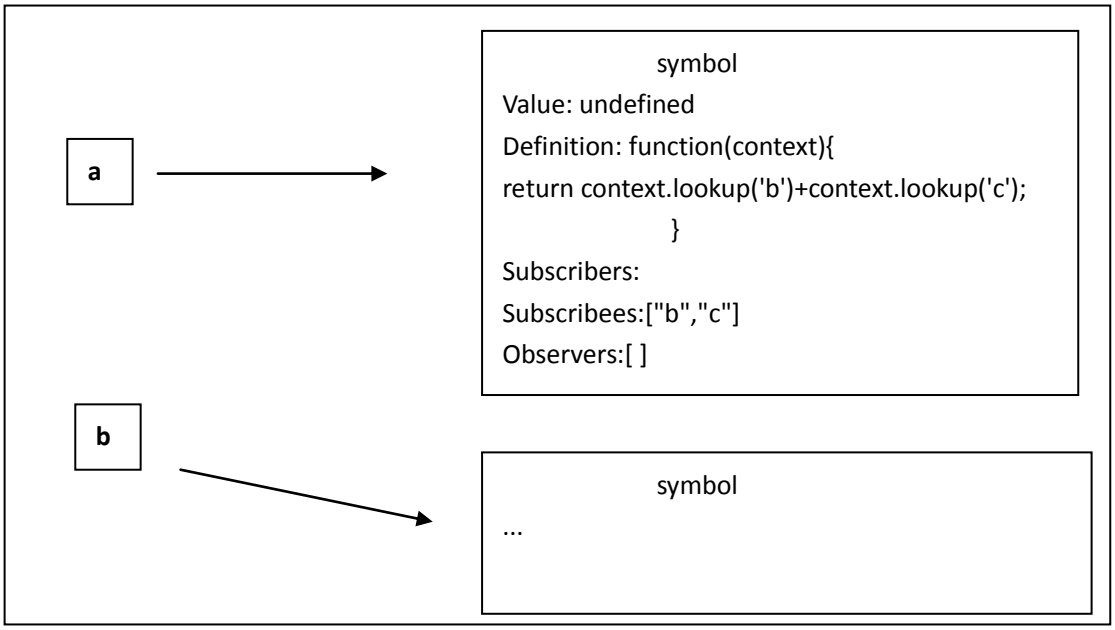


Figure 4.2 Observables maintained in JavaScript as symbols

Definitions:

Definitions are represented by functions, as in the last example, where the definition of *a* is implemented as a function:

```
function(context){
       return context.lookup('b')+context.lookup('c');
       }
```

that returns the sum of the current values of b and c. This kind of representation is associated with the objects "symbol" by the function "symbol.prototype.define".

Dependency:

The network of dependencies is maintained by the functions *subscribers* and *subscribees*. *Subscribers* contains the observables that the symbol depends on and *subscribees* contains the observables that depend on the symbol. Even without these two functions, we already have a structure mapping each observable to a function. However this is quite wasteful since the function is re-evaluated on each attempt to inspect it, and this makes EDEN trigger actions impossible to implement [TM11]. A EDEN procedure to trigger observable a is implemented in JavaScript:

```
context.lookup(“watcha”).assign(function() {
    alert(“a changed!”);
}).observe([“a”]);
```

In the code above, *watcha* is an observer of *a*. The trigger actions in EDEN only trigger when an observable changes. If we do not apply the functions *subscribers* and *subscribees*, the *define* function which associates observable with a formula will reevaluate the value of observable even there is no change to the observable. As a result, the EDEN procedure can print out "a changed！" falsely.

We want an dependency based model, which re-evaluates the values of observables only when the values of *subscribees* change. "*a is b+c*" could be maintained in JavaScript as:

```
context.lookup('a').define(function(context){
    return context.lookup('b')+context.lookup('c');
          }).subscribe(['b','c']);
```

The call to the subscribe function means that all the symbols subscribed will inform this symbol if any change has happened to them. In the code above, *b* and *c* will inform *a* of any changes, and the definition maintainer will update the cached value of *a* to be invalid.

Assignment:
The assignments of values to observables in EDEN is translated in JavaScript by the function "symbol.prototype.assign".

For example:
```
a=2;
```
is translated to JavaScript:
```
context.lookup(a).assign(2);
```

Figure 4.3 is an illustration of the mechanism:

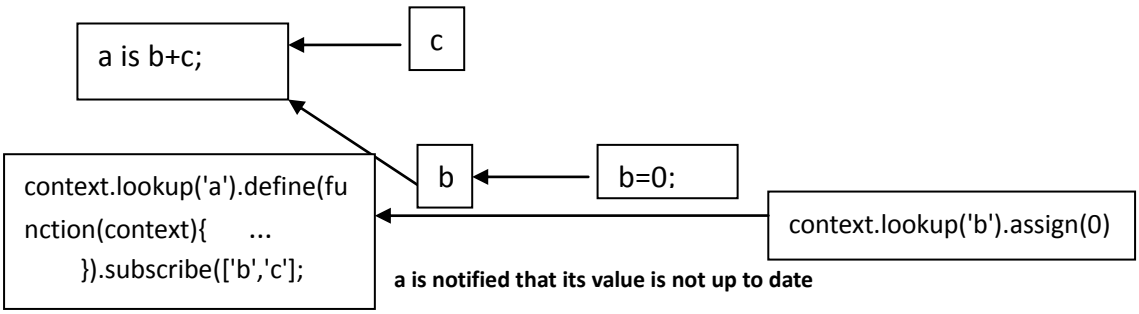


Figure 4.3 Mechanism for Assigning Values to Observable in JavaScript

4.3.2 Functions to draw Vertices and Edges

After the brief summary of how definitions, observables and dependencies are maintained in JS-EDEN, the next step is to connect a graph object and its components connected with EDEN observables, so that they can be accessed in the EDEN level.

Ideally, we want to achieve functionality similar to the graph building tool in EDEN by Charles Care [Car06], which means it should allow the users to easily add nodes and edges to the canvas by calling functions *addNode*( ), and *addEdge*( ). Furthermore, the attributes of node and edge objects, such as positions, colours and sizes, should be accessible from the EDEN level. We do not want all the changes to be made in the console with JavaScript.

This problem can be solved by adding the code which makes observables, definitions and network of dependencies in the JavaScript level of the graph object. Nodes in the graph object are modelled as vertex objects and they all share the same vertex prototype; Edges in the graph object are modelled as edge objects and they all share the same edge prototype; Each edge object is a pair of nodes (source node, target node).This code is implemented in two functions, *vertex.prototype.drawVertex( )* and *edge.prototype.drawEdge( ).*

A node object in the graph object has 3 attributes: x coordinate, y coordinate and name, and when displayed, it also has attributes colour and size. Unfortunately, EDEN does not support attributes grouped together as object-like observables. To solve this, each attribute of the node is assigned a corresponding EDEN observable, and all the attributes are aggregated in the same function. In section 4.2.1 we discussed how an observable in EDEN could be translated into JavaScript in this way:

EDEN:
```
a
```
JavaScript:
```
root.lookup('a')
```

Vice versa, the code *root.lookup('a')* in JavaScript will look for an observable *a* in EDEN, and create an observable *a* if it does not exist.

For example, in the function *vertex.phototype.drawVertex ( )*,

```
var o_Circle = context.lookup('Circle'); // a call to the Circle
var o_px = context.lookup('p'+ix+'x');//declare p+'ix'+x in EDEN
var o_py = context.lookup('p'+ix+'y'); //declare p+'ix'+y in EDEN
var o_circle = context.lookup('c'+ix); //declare c+'ix' in EDEN
var o_r = context.lookup('r'+ix); //declare r+'ix' in EDEN
var o_cl = context.lookup('col'+ix); //declare col+'ix' in EDEN
```

where *ix* is the name for the node.

We want the node to appear on the canvas as a circle. For node *a*, we already have observables *pax* for its x coordinate, *pay* for its y coordinate, *ca* for the name of circle, *cola* for the color of the circle and *ra* for the radius of the circle *a*. The EDEN definition for the circle *ca* is:

```
ca is Circle(pax,pay,ra,cola,cola);
```

In JavaScript definitions are modelled as functions *object.define*. The code in bold style below is the function corresponding to the EDEN definition for ca above. The subscribe function demonstrates the dependency between *ca* (name of the circle) and its attributes. The code in italic style below does the initial assignment of values to these observables.

```
Vertex.prototype.drawVertex = function() {
        (function(context,ix) {
                var o_Circle = context.lookup('Circle');
                var o_px = context.lookup('p'+ix+'x');

                var o_py = context.lookup('p'+ix+'y');
                var o_c = context.lookup('c'+ix);
                var o_r = context.lookup('r'+ix);

o_c.define(function(context) {
                return o_Circle.value().call(this, o_px.value(),
o_py.value(), o_r.value(), o_cl.value(), o_cl.value());
                }).subscribe(["Circle", "p"+ix+"x",
"p"+ix+"y","r"+ix,"col"+ix]));

        })(root,this.value);

        if((root.lookup("p"+this.value+"y")).cached_value===undefi
ned){(root.lookup("p"+this.value+"x")).assign(this.x);}
        if((root.lookup("p"+this.value+"y")).cached_value===undefi
ned){(root.lookup("p"+this.value+"y")).assign(this.y);}
        (root.lookup("r"+this.value)).assign(5);
        (root.lookup("col"+this.value)).assign("red");}
```

The function *Vertex.prototype.drawVertex* shows the basic mechanism for making observables and definitions in JavaScript. The function *Edge.prototype.drawEdge* is longer and more complicated. The interested reader can refer to appendix 3.

One point to note is that these two functions only provide a one–way connection between EDEN observables and JavaScript objects: changing a JavaScript object will affect the corresponding EDEN observable, but changing an EDEN observable will not affect its corresponding JavaScript object. However, if an update in the JavaScript

Object is needed, the following JavaScript function composed by Nick Pope can be used [Pop13] .

```
function declare_JSEDEN(object,name){
    object.__defineGetter__(name, function(){
        return (root.lookup(name)).cached_value;
    });
   }
```

4.4 Embedding JavaScript in EDEN:

4.4.1 addNode and addEdge functions in EDEN Level

Although the functions *prototype.drawVertex* make it possible to access the JavaScript objects in EDEN, they are still implemented by JavaScript code. The next thing to do is to make functions in JavaScript accessible by EDEN code. This is not difficult, because one advantage of JS-EDEN is that JS-EDEN can be hooked into rich JavaScript APIs by embedding JavaScript in the EDEN code.

sublist is a function in EDEN returning a sublist of a given list by index. Below is the code of *sublist* in JS-EDEN.

```
func sublist{
   ${{
   var src = arguments[0];
   var firstix = arguments[1];
   var lastix = arguments[2];

   return src.slice(firstix-1, lastix);
   }}$;

}
```

The code inside ${{ }}$ is JavaScript code. Generally, the EDEN code will first be translated to JavaScript code and then the JavaScript evaluator will evaluate the JavaScript code. Fortunately, in EDEN, the code inside ${{ }}$ will be executed directly as JavaScript code. This feature allows JavaScript to be embedded in EDEN.

When calling the *addNode* function in EDEN, which has two parameters, graph name and node name, we want the node to be added to the graph and displayed on the canvas. Hence we need to call the JavaScript *addNode* and *drawVertex* function developed in section 4.2.

```
func addNode {

   ${{
     var a = arguments[0];
     var value =arguments[1];
     a.addNode(value);
     a.nodeset[value].drawVertex();

    }}$;
};
```

Display of the nodes:

In JS-EDEN, the objects displayed on the canvas are included in a list, named *picture*. Suppose we have *circleA, circleB...* To display them on the canvas, *picture* should be defined as:

```
picture is [circleA,circleB,...];
```

Every node is associated with a circle in EDEN. When the *drawVertex* function is called, the corresponding EDEN observables for circles are created. So we need to attach the list of circle names to the definition of pictures.

```
picture is nodes//edges;
func addNode {

 ${{
      ...
     (root.lookup("nodestring")).assign(a.strNodes).value();

  }}$;
    execute("nodes is ["//nodestring//"];");
};
```

4.4.2 Lexical Analysis of Parsing Models in JS-EDEN

The lexical analysis part of parsing models in JS-EDEN is different from the lexical analysis of the traditional parser. This is done by using the function *subpatt*, which is another example of embedding JavaScript in EDEN:

```
func substpatt {
${{ var th = arguments[0];
        var repatt = arguments[1];
        var reattr = arguments[2];
```

```
        var repstr = arguments[3];
        return th.replace(RegExp(repatt, reattr), repstr);
}}$; }
```

This function recognizes regular expressions and the set of tokens are defined by regular expressions. Take the Eddi parser as an example.

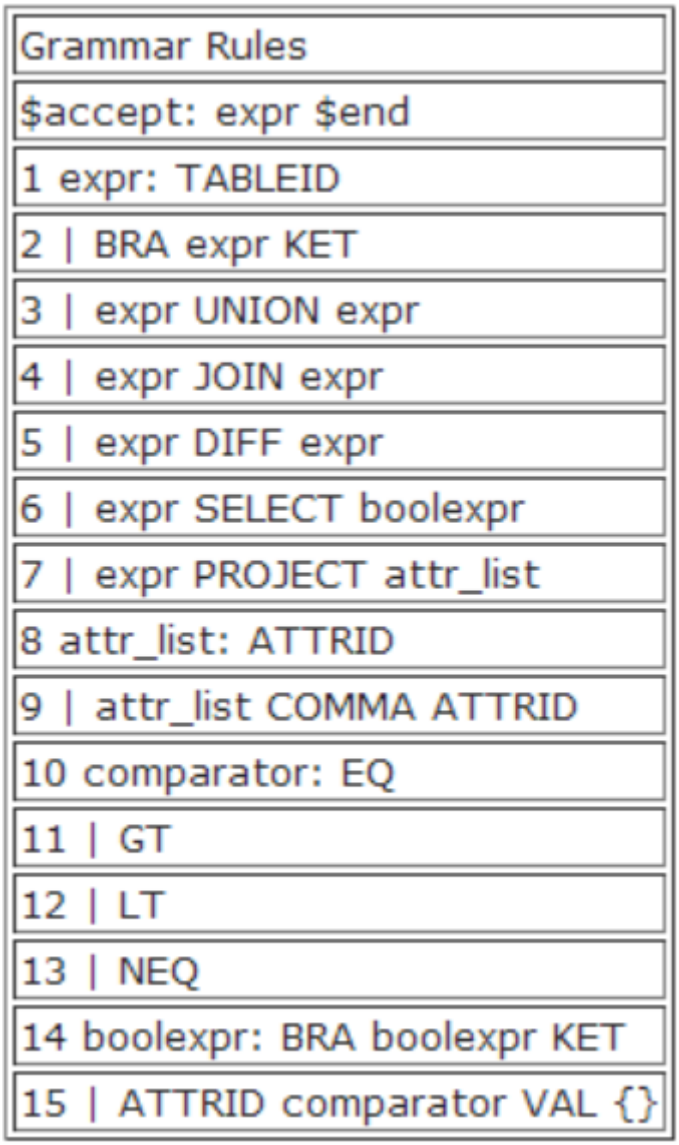


Figure 4.4 Grammar Information Of Eddi

In the grammar rules, we have TABLEID (table name) and ATTRID (attribute name), however in the lexical analysis, we do not want to differentiate between these two, they are all regarded as identifiers. They are differentiated later in the process of parsing according to the context. For instance, consider the input string:

```
inputstr="(A+BC):width>5 % dim,length";
```

In this input, we know that A, BC are names of tables, and width, dim (dimension) and length are names of attributes. However the model will treat all of them as

identifiers. Since white space is not significant in Eddi, the white space is eliminated first.

Tokens defined by regular expressions:

```
InputText=inputstr;   //InputText is qual to inputstr before lexical
                      //analysis
InputText = substpatt(inputstr," ","g","");
InputText = substpatt(InputText,"[A-Za-z][a-zA-Z0-9]*","g","x");
InputText = substpatt(InputText,"[0-9]+","g","v");
```

Here the method *substpatt* is called, which finds certain pattern in a given string and does the substitution of the pattern for the Eddi input above [MrJ12]. The InputText will have the value *"(x+x):a>5%a,a"*, where *x* stands for an identifier, and *a* stands for ATTRID (attribute) .

In chapter 3 we discussed that there is a value stack in parallel with the symbol stack. The lexical analysis should also store the values of tokens in a list in EDEN.

```
func findpatt {
   ${{
        var th = arguments[0];
        var repatt = arguments[1];
        var reattr = arguments[2];
        var result = th.match(RegExp(repatt, reattr));
        if(result==null)
        return "";
        else return result;
   }}$;
}
```

findpatt matches for a regular expression in a given string and returns the substring it finds. These substrings are stored in a list, *inputvals*. After that a function is called to transform the *inputvals* list so that the index of *inputvals* is consistent with the index of *InputText*: If *InputText[j]* is an identifier, the value of the identifier should be the jth element in *inputvals*.

As mentioned above, the tokens TABLEID and ATTRID should be differentiated during the process of parsing according to the context. Actually the parser does not know if an identifier is a TABLEID or an ATTRID, until it is in the state where the next symbol to be read is an identifier. As explained in chapter 3, section 3.2.3, the observable *nextsymbol* observes the next symbol to be read by the parser. Making *nextsymbol* depend on the context (in this case the current state of the parser) solves this problem. The function *findnext* will search the transitions table, and determine what symbol the parser could read in the current state.

4.5 Implementation of Parsing Models in JS-EDEN

4.5.1 Automation of Parsing Models in JS-EDEN

In Chapter 3 we discussed the observables and the network of dependencies of Parsing Models (cf section 3.2.3) .

Key Observables:

currpath – stores the content on the state stack as a list.

currstack – current content on the stack as a list.

nextsymbol – represents the look ahead token.

isreduction – reads the transformations table and decides if the parser should reduce in the current state. If there is a shift-reduce conflict, it always chooses to shift.

currrule – the number of the grammar rule the parser determines that all the items on the right-hand side of which have been seen(the production by which the parser reduces).

currrstate – current state of the parser (the last element of the currpath, or the top element on the stack state).

InputText – the input string, stays the same all the time.

InputTextt – same as the input string at the beginning, but changes according to the reduction.

k – the index of the current symbol in the InputText (how far we have read in the InputText).

j – the index of the current symbol in the InputTextt (how far we have read in the InputTextt).

The observables *k* and *j* are used to control how many tokens the parser has read from the input. The values of *k* and *j* are increased by 1 every time the "step by step" button is clicked.

Network of Dependencies:

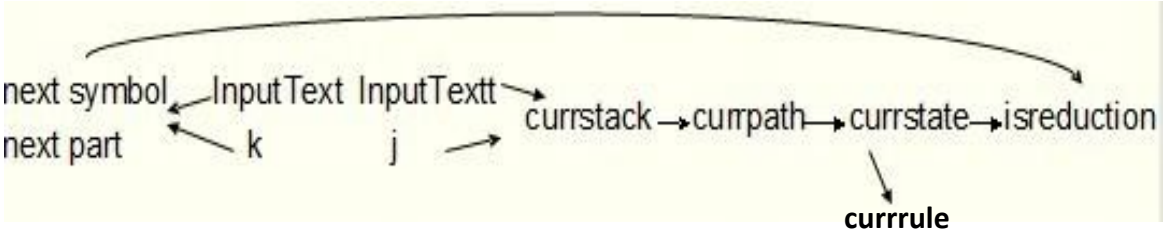


Figure 4.5 Network of Dependencies in parsing models in JS-EDEN

From the dependency diagram, we can find that the model is partly controlled by changing the value of *k, InputText* and *InputTextt*. All the other main observables are dependent on these three observables. The *InputText* is the string to be parsed and the *InputTextt* is set to be the same as *InputText* at the start of parsing. *k* is the index of the symbols in the *InputText*, which measures how many tokens have been read so far. Increasing the value of *k* adds more content on the current stack and current path. In some steps of parsing process, reduction should be carried out instead of shift. This is shown by the observable *currrule* and *nextsymbol*. If there is no reduction to be taken, the value of *currrule* will be empty, otherwise *currrule* shows the reduction rule which could be applied.

The model can be controlled manually, by increasing the value of *k*, and applying the reduction rule when necessary. However the function of the model is limited if it can only be run manually.

The automation of parsing uses the JS-EDEN actions, trigger procedures. In a procedure, a sequence of instructions will be invoked by the system whenever the values of some variables, specified explicitly in a list, are changed.
For example:

```
proc printa: a
{
       writeln(a);

}
```

The procedure *printa* will be invoked when the value of *a* is changed, and the value of *a* will be written.

In the parsing process, the *automatic* procedure is triggered by the observables *k, InputTextt* and *isautomatic*, where *isautomatic* is an observable which determines whether the model is in the manual state or automatic state. The user can switch between the manual and automatic state by setting boolean values to *isautomatic.*

Every time the *automatic* procedure (cf. code on next page) is called, it should first choose whether a shift or reduce should be carried out, based on the value of other observables. From the dependency network, we know that *currrule* is dependent on the model's state and the content on the stack. If *currrule* is empty, it means that there is no reduction rule available. If *currrule* is not empty, it means that there is a reduction rule that could be taken, however sometime the model faces a shift reduce conflict, or a reduce—reduce conflict. Hence the observable *nextsymbol* should also be taken into account to make the model decide the action to be taken when there is a conflict between shift and reduce. If the parsing model should shift, then the value of k is increased by 1. If the parsing model should reduce, then the *applyrule* function will be invoked, which changes the observable *InputTextt* according to different rules.

Since *currstack*, *currpath* and *currstate* are dependent on *InputTextt*, the stack, state and path will be updated automatically.

```
proc automatic: k, InputTextt, isautomatic {

   after (speed) {
     if ( (currrule!="") && (isautomatic==1) ) {

       if ( (currrule==3) && (nextsymbol=="JOIN") )
          k++;
       else if ...
       else
       {
          evalrule (currrule) ;
          applyrule (currrule) ;
       }
    }
  }

 after (speed) {
  if ( (currrule=="") && (k<=InputText#) && (isautomatic==1) )
          k++;
       }
}
```

4.5.2 Error Recovery of Parsing Models in JS-EDEN

One advantage of the Parsing Model in EM over a traditional parser is the flexibility. In a traditional parser, given a string, the parser will tell you if it is accepted or not, while in EM, parsing models can have better error detection and recovery.

The error recovery of parsing models is done in two aspects:
1 If an user makes a wrong decision when he/she trys to parse a string manually, the last state of the parser before the wrong action can be easily retrieved;
2 During the process of parsing, if there is an error in the input, the user can make a change to the input and continue to parse the input without restarting.

The first aspect is achieved by using a list, named *stacklist* to record the information of the key observables at each stage of the parsing process. The key observables which the list records are *InputTextt*, *inputvals* and *k*. Initially, the value of *InputTextt* is equal to the value of *InputText*. The difference between *InputTextt* and *InputText* is that *InputTextt* changes during the parsing process, it stores the changes of input during the parsing process. *InputText* does not change, together with *k* (the index of

InputText), they store information to let the model know what is the next symbol to read. A procedure is defined to trigger these three key observables, once any of them is changed, it adds a sublist composed of these three elements to the *stacklist*.

isreverse allows the model to switch between the parse state and the backtrack state. The user can control the model by clicking the "last step" button on the canvas to go into backtrack state, and by clicking any button on the manual panel to go back into parse state. If *isreverse* is one, which means the model is in backtrack state, the procedure will stop adding new sublists to stacklist and the model will get information from the stacklist, and assign the values to these three key observables.

```
stacklist=[[InputTextt,inputtvals,0]];
isreverse=0;
proc makestacklist: InputTextt,k {
  if(isreverse==0){
    stacklist=stacklist//[[InputTextt,inputtvals,k]];
    }
}


proc reverseback: laststep_clicked {
  if(laststep_clicked){
    isreverse=1;
    k=getlscomp (stacklist[stacklistindex-1],3) ;
    inputtvals=getlscomp(stacklist[stacklistindex-1],2);
    InputTextt=getlscomp(stacklist[stacklistindex-1],1);
    stacklistindex--;
  }
}
```

Chapter 5 Future Directions and Possible Implications

5.0 Introduction

This chapter illustrates how the online application of EM based on JS-EDEN can implement EM principles for educational technology and discusses the future directions and possible implications of the work described in previous chapters of the thesis. It illustrates how EM can support constructionist learning. An important aspect of supporting constructionist learning is making the model-building role more accessible. The study of parsing and EM aims to make a parsing environment in the spirit of EM, which provides an open environment in which learners and modellers can frame their scripts and create parsers for their own customized definitive notations. It also illustrates how a constructionist approach can be applied to learning about parsing itself: learning through making construals and learning through exploring others' construals. These agendas will be addressed by elaborating how an EM parsing model can be deployed to support constructionist learning.

5.1 Parsing with EM Spirit

Making EM construals of parsers is not merely intended to improve tool support for EM. From an educational perspective, it aims to provide an open learning environment and give users more freedom to make construals; it encourages ways of thinking, not ways of professionally programming. We want to achieve the situation in which non-specialists can frame their own definitions and operators, and add new notations to the interpreter. All relatively technical activities require precise formulation of text as in traditional programming.

In *Learnable Programming,* Bret Victor criticizes Alan Perlis's idea "To understand a program, you must become both the machine and the program". He argues that a programming environment, especially an environment for learning, should make meaning transparent, explain in context and make users focus on thinking about high-level concepts [Vic12]. One of the themes of William James's work is the inadequacy of words to convey connections that can be given in experience through using other media – like pictures or music [Jam96]. Visual programming environments such as Scratch reflect this spirit.

Scratch is a creative learning tool developed by MIT [SCW], which allows the users to create their own animation, program, music or stories easily. One important feature of Scratch is that you make your own script of the project by assembling blocks of code. In traditional programming, the programmer needs to think carefully before

compiling the code and seeing the effect. In contrast, programming is much easier in Scratch. Whenever something is added to the script, the user can choose to see the effect immediately, and the panel right beside the script gives options about what could be added to the script, so they can try to make something of which they do not have any idea at the beginning. This endorses Bret Victor's idea: "A canvas or sketchbook serves as an 'external imagination', where an artist can grow an idea from birth to maturity by continuously *reacting to what's in front of him*."

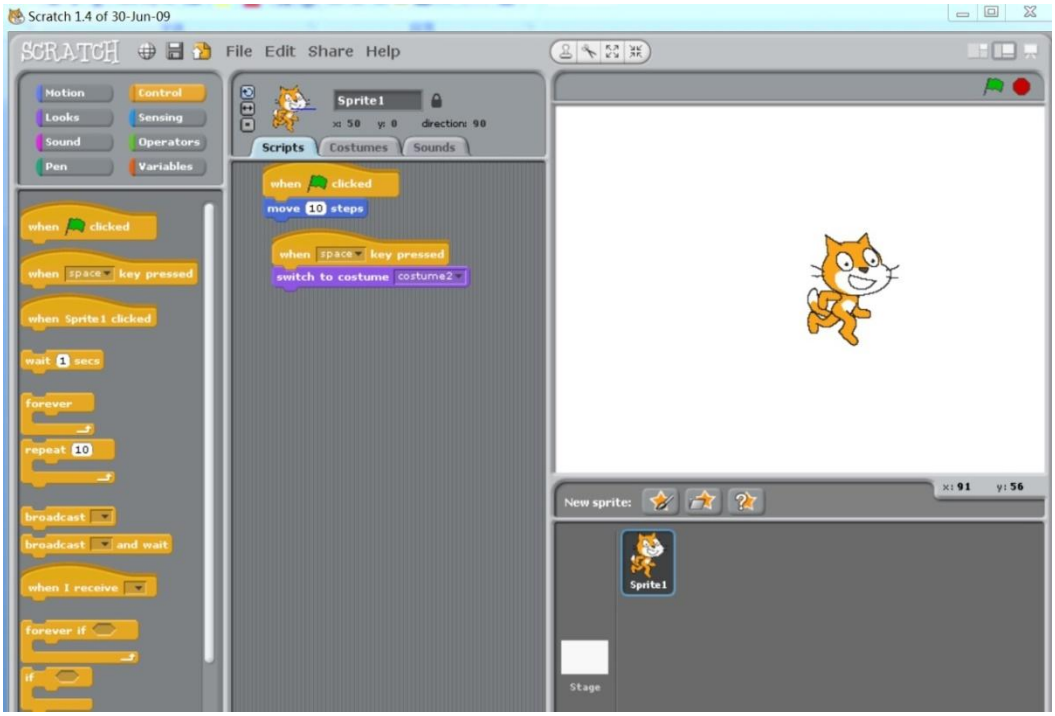


Figure 5.1 Screenshot of Scratch Interface

Scratch shifts the focus from constructing a string of symbols to assembling jigsaw-like pieces. The CADENCE tool in EM developed by Nick Pope has features that Pope later characterized as a "distributed-structure-base" [Pop11]. CADENCE provides a structure for organizing observables and cloning of objects to generate larger models. This kind of interface allows users to build structures of code rather than to write program text. In this way, the users' thoughts about programming are more visible than in a traditional programming environment (making the connection between the experience of structure and the experience of meaning, that was discussed in chapter 2).

From an EM perspective, the benefit of making this shift from the textual to the visual depends critically on the nature of artefacts being constructed. Bret Victor's concept of Learnable Programming is in the same spirit as EM thinking. However the implementation of his ideas and the Scratch interface are both about procedural programming, which sets both approaches apart from people's way of thinking and understanding. The "identify oneself as a turtle" feature in LOGO, and the "think of oneself as talking with other objects" feature in Smalltalk, which, according to Bret Victor, are designed to resonate with how people learn and understand, are only

representing abstract programming processes ("thinking like a machine") by concrete activities to which humans can relate (role-playing and conversation) [Vic12]. However, thinking about the process of programming in another way still distracts the novice programmer from his or her thinking about high level domain concepts. Victor's answer to "creating by reacting" is an environment which gets something on the canvas as soon as possible and makes all the information visible. In procedural programming in Scratch, the correspondence between the jigsaw of procedural code and the behaviour it represents is no easier to understand than its textual equivalent. A live-coding environment, editing the program and observing the result is not enough to support the programmer's exploration and imagination. This is because the correspondence between lines of programming code and their computational effect is typically difficult to understand. In EM, by comparison, there is a direct correspondence between observables and dependencies in the construal and observables and dependencies in the referent (cf. Figure 1.1).

A prototype environment for parsing the auxiliary definitive notation Eddi in JS-EDEN, is depicted in Figure 5.2 below. This has been developed by the author in order to explore the potential for exploiting Scratch like interfaces in EM. (For the code see Appendix 4.)

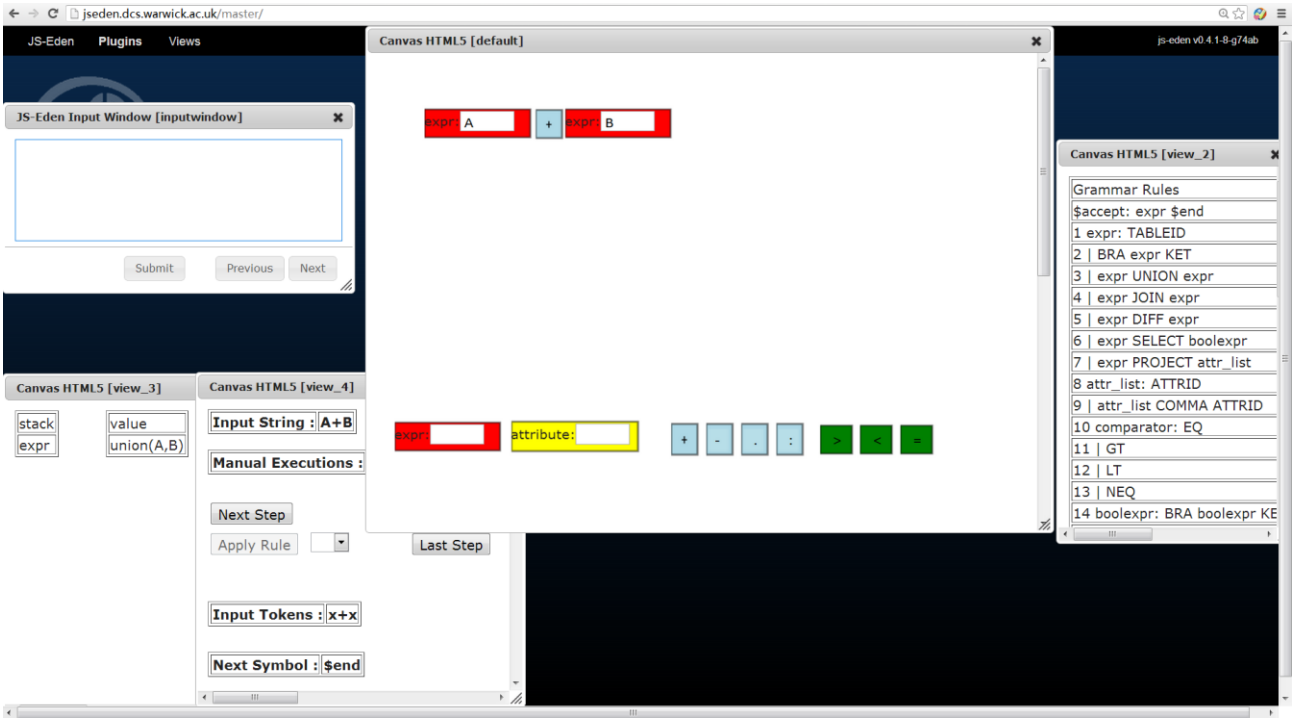


Figure 5.2 Scratch-like interfaces for parsing Eddi grammar

5.2 Deployment of Parsing Models in Teaching

In this section, we are going to discuss potential use of EM parsing models in learning from two aspects:

1) learning in the process of making construals, which is an individual activity that promotes creative, open and flexible learning

2) learning in the process of exploring construals made by others, which has a social dimension concerned with exploring the extent to which a modeller's understanding can be communicated and shared.[BeZ13]

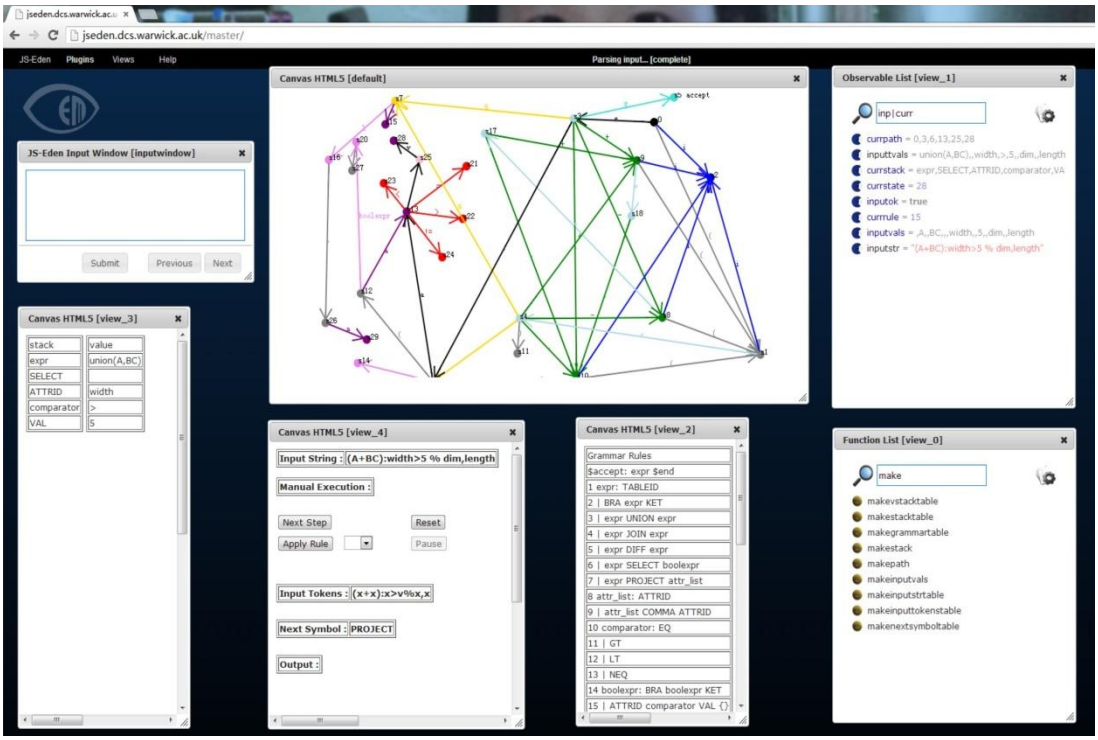


Figure 5.3 Screenshot of Parsing Model in JS-EDEN

Superficially, an EM parsing construal resembles a traditional computer program to parse a string in a certain grammar (cf. Figure 3.2 and Figure 5.3). The buttons on the interface allow the players to choose to read the next token and apply different reduction rules. However, an EM parsing construal is different from a traditional program in that it has the potential for open and flexible interactions.

The source for the parsing construal is a family of definitions of observables. The construal remains live to modification at all times – a new definition for an observable can be submitted via the EDEN Interpreter Window. The entire state of the construals displayed in Figures 5.3 and 3.2 is captured in the observables and dependencies. These observables include all the information required to display the parsing model (e.g. the edges and nodes in the finite state machine, the grammar information, the layout of the buttons and windows, the status of buttons, ...), and to determine the current state of the parsing model (e.g. the content on the stack, should the model shift or reduce, the current reduction rule).

All the meaningful activities that take place in simulating a traditional parser can be represented as changes of state that involve redefining one or more observables in the

construal. For instance, when a user initiates parsing by clicking the "step by step" button, the value of the observable *k* is set from 0 to 1. Concurrently, the observable *currstack* is changed by dependency so that it includes the first token of the *InputText* and the observable *currpath* is changed as it is dependent on *currstack* so that it includes the first edge has been traversed... (cf. section 3.2.3). There are a lot of activities for the user to explore. Some activities may not have an obvious meaning, like making the colour of edges depend on *nextsymbol*. But such a dependency may be of interest because it prompts the user to ask a question to himself or herself, such as: how is the path in the finite state machine related to the next symbol? What matters is that making a construal shows the state change explicitly and opens up a lot of possible interactions which involve changes to observables and dependencies that cannot be made in a traditional parsing model. In the process of learning, the user is not reading answers from the book or observing parsing flow flash by, they are exploring answers by engaging, initiating state change, redefining and observing state changes. "Making a construal opens up such a profusion of possible interpretations, stimulating the model-builder's imagination and creativity. Yet more important is the fact that the making of a construal is an open-ended activity that resembles organic growth rather than building to a specification" [Bez13].

5.2.1 The EMPE in JS-EDEN

JS-EDEN is a web-enabled variant of the EDEN interpreter that is still at an early stage of development. JS-EDEN differs from EDEN in that it can readily exploit features of pure JavaScript, but offers only very limited support for definitive notations. For instance, no counterparts of the definitive notations Scout and DoNaLd for screen layout and line drawing exist in JS-EDEN.

The Empirical Modelling Presentation Environment (EMPE), developed by Antony Harfield [Har07], is a tool for creating presentations. As is in illustrated in Figure 5.4, the left panel has a window for model display and a built-in-input box for users to interact with models. The right hand panel is the content of the presentation written in HTML, and it allows executable scripts to be embedded in the presentation. The EMPE has been ported to JS-EDEN Master version (cf. Figure 5.6).

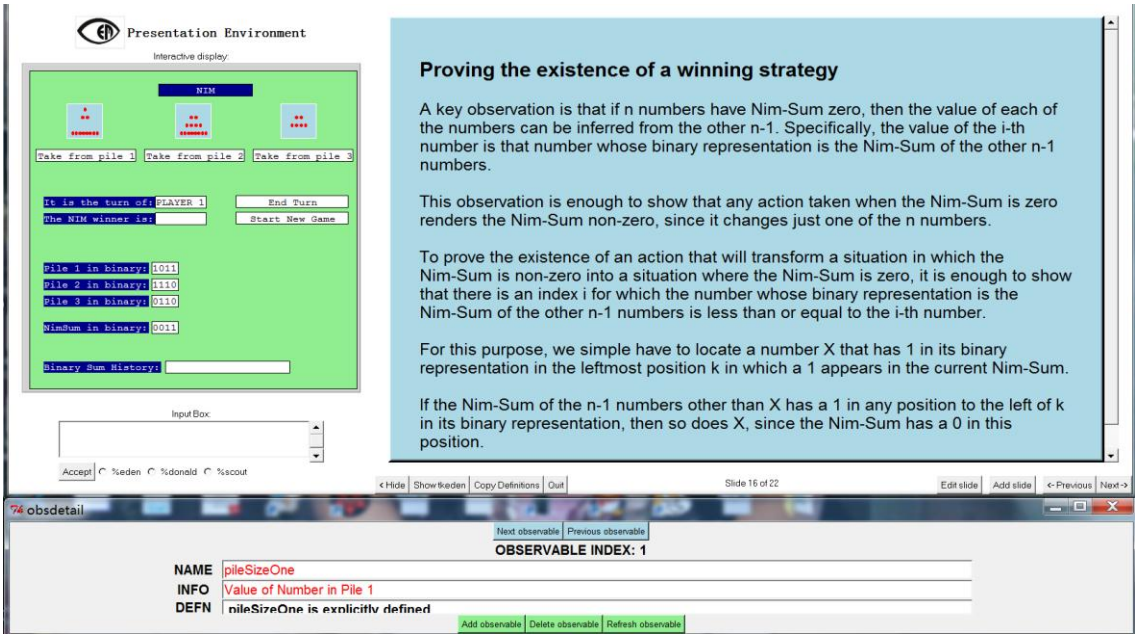


Figure 5.4 Screenshot of EMPE in EDEN

As discussed in section 5.1, Victor's concept of "learnable programming" is in the same spirit as EM. The implementation of his idea using a traditional style of programming does not do full justice to his idea of an environment which is designed to fit human's mind and to provide an external imagination for creating. Victor makes the case that the direct dependency between "editing the code" and "observing the changes to the webpage" encourages the learner to shift from "thinking like a machine" to "thinking like a human". This is not necessarily right. The fact that editing the code has a predictable effect on the display does not necessarily mean that this connection is intelligible with reference to human knowledge of the domain.

By way of illustration, Figure 5.5 is part of a demonstration from Learnable Programming [Vic12]. Imagine this example as a live coding environment but without the slider control on the top. As a non-specialist, in order to know how the shape on the right is connected with the code on the left, the first thing the novice programmer wants to understand is the meaning of the variable *shift*. The programmer will seek to understand this by trying to connect changes to the variable shift to the corresponding changes to the shape (cf. the notion of conjunctive relation introduced by William James, discussed in section 1.4). The programmer may start experimenting by setting the initial value of *shift* to 10, and see that the shape moves a bit to the right. The programmer may also change "shift+=14" to "shift+=30" and observes that the curvature changes. But what is the significance of shift remains ambiguous. This kind of interaction is of poor quality – connections are hardly "encountered in immediate experience". There is no direct correspondence between the code and the shape, only the mental "procedural thinking" model connects them. Adding a slider control is better because it allows the programmer to move around the loop at his or her own pace, and understand what is happening at each step. The programmer can study how the output is built up by experimenting and finally understands that *shift* does not have

a specific counterpart by way of a concrete observable, but is rather an abstract parameter for drawing the shape. In this illustrative example, the functionality of the slider control bar is similar to the functionality in a step-by-step traditional parsing model demonstration but without the additional scope for rich and open interactions.

Figure 5.5 Environment in Learnable Programming

In section 5.1, we discussed Victor's idea about the programming environment, which aims to make meaning transparent, show all the information of the source and display the immediate effect of code change. Where making construals is concerned, the EMPE interface realizes Victor's idea, and actually supports a richer environment in that it allows user-editable explanations to be presented as well as the source code. Furthermore, the code can be embedded in the explanations, so that users can both see the explanation and the effect of the code. In the JS-EDEN presentation environment (cf. Figure 5.6), a webpage is split into two panels so that all the source of the left-hand panel is displayed in the right-hand panel. A learner can interact with the model either by editing input in the *JS-EDEN Input Window* directly, or by selecting the observables in the *symbol list*. In both cases, the feedback is immediate.

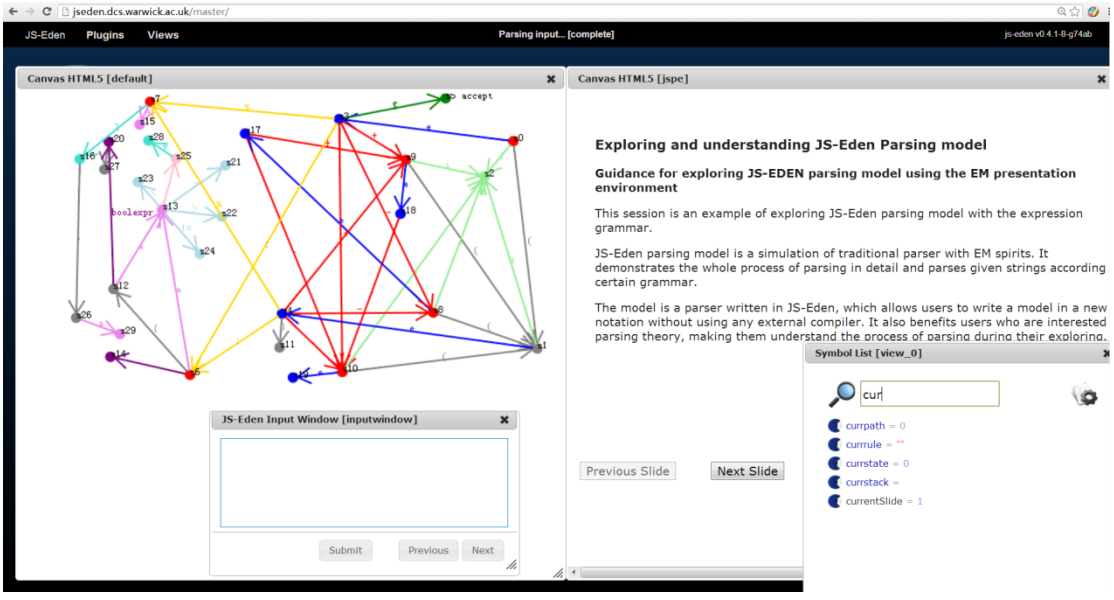


Figure 5.6 Screenshot of EMPE in JS-EDEN Master Version

5.2.2 Tutorials on making Parsing Models

This section discusses a tutorial on making parsing models, built in the JS-EDEN Presentation Environment, where parsing is illustrating the need for a constructionist framework in which the learner can build their own parser, and handle activities such as error detection and correction.

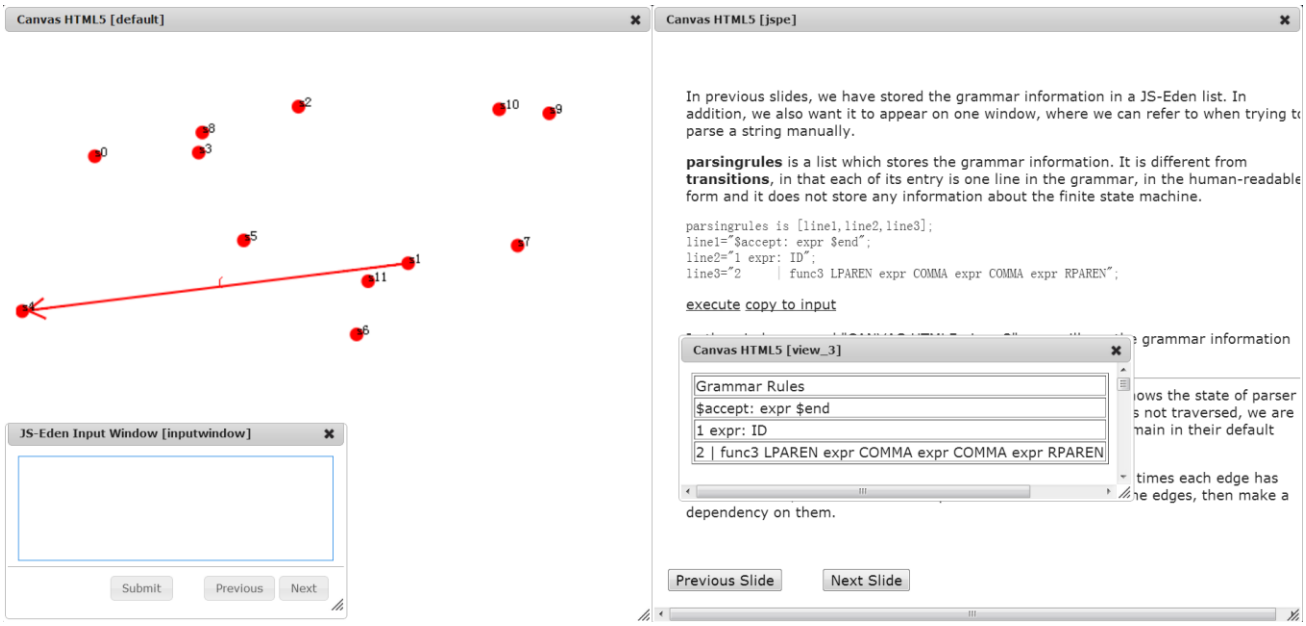


Figure 5.7 Screenshot of Tutorial in JSPE environment

From an educational perspective, this tutorial can be an example of using EM to support constructionist learning. The tutorial illustrates the method to apply in making parsing models for different grammars. This tutorial does not discuss the process of constructing parsing model in detail, but illustrates how the information about a particular grammar can be ported to the parsing model, and suggests how the dependencies can be applied to link key observables in the parsing model construal. Following the instructions, the learner can build a parser for a language that he or she wants to be translated to Eden using the parsing model. As illustrated in figure 5.7, this tutorial makes use of the EMPE environment. The canvas on the right is the content of the presentation written in HTML, and it allows executable scripts to be embedded in the presentation, with two options "execute" and "copy to input". The effect of the script is immediately displayed on the canvas after clicking the "execute" option.

Learning parsing theory in computer science can resemble the kind of learning of mathematics deprecated in section 1.3, and be dissociated learning. This tutorial aims to guide the users to construct some concrete artefacts or objects with which to think about parsing, and gives the users access to the experiences of both a learner and a developer. As illustrated in the fundamental diagram of EM (cf. Figure 1.1), the referent is the formalized theory of parsing and the learner gains his or her

understanding by interacting with the computer-based construal and referent. The learners give form to their ideas, and these forms, once built, in turn inform their ideas [Ack94]. The tutorial describes the key observables in the parsing model construal explicitly, and explains in detail how the dependencies can be applied to link the key observables. However, the explanation is merely a suggestion for learners – it does not restrict the learners' activity in constructing their own parsers and interacting with the parsing model. In constructionism, to learn is to relate. Knowledge is not information to be delivered. Instead knowledge is experience, in the sense that it is actively constructed and reconstructed through direct interaction with the environment [Ack96]. As a learner, the users cast themselves in the role of a traditional parser and become immersed in the phenomenon that they want to understand. The users parse a string manually using the parsing model and observe the changes in the diagram and the stack. In this process, they identify themselves as the parser trying to parse a given string and take shift or reduce actions step by step. This is similar to the activities in Papert's LOGO software and Victor's environment for learning about while loops (cf. Figure 5.4). Since the parsing model is based on EM principles, all the behaviours of the parser are represented as state changes and all the information of the key observables is displayed. The users metaphorically experience not only the behaviour of a traditional parser but also – in a way that is distinctive of EM – the state change patterns as defined by definitions. In the spirit of a developer, the users also detach themselves by projecting out from the experience. They stand at a distance and reconsider what they have "experienced" as a parser in different states. They try to link observables with different dependencies metaphorically according to their experience of being a parser and observing state changes, and their understanding of parsing theory from a book or a lecture (cf. Figure 1.1). This kind of activity offers the users a concrete experience which they can relate to. The shuttling between the perspectives of "the constructor of the parsing model" and "the learner of parsing theory" is consistent with Piaget's definition of intellectual development as *assimilation* and *accommodation* [Omr11], and with Ackermann's idea of two keys to constructionism as *perspectives taking* and *objects construction* [Ack96].

A construal is an object to think with in which the modeller's understanding and ideas can be tangible and shared with others. In using this tutorial, the users of the tutorial can switch between being a learner and being an instructor. They record their changes of the construal and key interactions in the presentation and invite others to follow their presentations. The evolution of these interactions and descriptions shows the development of the learners' understanding. The student’s task involves building an artifact in parallel with building an understanding that is embodied in interaction and interpretation with the construal. The teacher’s role in this process is at first that of a listener who poses questions that elicit further elaboration, and later that of an evaluator who assesses the authenticity of the student’s experience [BeZ13]. Contemporary natural language is not sufficiently rich when talking and thinking about structures and states. Making a construal, as a reflection of the modeller's mental model about the referent, has essential qualities which are not possessed by

other modes of assessments, like homework or exams, nor by other ways of communication, like conversation or reports. The teacher's assessment of the construal satisfies the requirement identified in the *Manifesto For Teaching Online* produced at the University of Edinburgh in 2011 [MTO] for feedback that "can be digested, worked with, created from". The interactions between teacher and learner through ongoing collaborative development of the construal are the vehicle for communicating their ideas and understanding.

Chapter 6 Conclusion

6.1 Summary

The content of the thesis can be summarized as follows:

Chapter 2 – Reconceptualising parsing in EM terms:

This chapter revisits a theme that was first explored by Chris Brown [Brn01] in his preliminary development of an agent-oriented parser ("the AOP"). The motivating idea behind the AOP is that EM principles, which provide the foundation for rich models of state, might be well-suited to representing the complex state-based activities involved in parsing. The development of the AOP was improved by Antony Harfield [Har03], and became the platform for parsers for a variety of auxiliary definitive notations (such as EDDI and Angel) for the traditional EDEN interpreter. This chapter highlights interesting parallels between work on the AOP and work of Lucien Tesnière [Tl59] on *dependency grammars* that has been applied in parsing natural language. The AOP aims to establish a direct link between the structure of a sentence and its meaning as does a dependency grammar. The process of construing natural language sentences through manipulating the structure of a sentence and seeing how this affects its interpretation resembles the activity represented in the fundamental diagram of EM (Figure 1.1).

Chapter 3 – Construing traditional LR-parsing using the EDEN interpreter:

Here the principles and tools are familiar and have been used extensively. The application is new, and mathematically more challenging than most previous modelling exercises. Making a construal of a conventional parser is instructive, as it involves construing an algorithmic procedure that is relatively sophisticated compared with any considered in previous work (cf. [Brn03]). It also sheds light on how making a construal and exercising the construal can potentially support learning about traditional parsing principles. This chapter includes a detailed account of making the parser construal that serves to illustrate how EM principles can be applied.

Chapter 4 – Construing traditional LR-parsing using the JS-EDEN interpreter:

This exercise is challenging because it makes use of a web-enabled variant of the EDEN interpreter that is still at an early stage of development. JS-EDEN differs from EDEN in that it can readily exploit features of pure JavaScript, but offers only very limited support for definitive notations. For instance, no counterparts of the definitive notations Scout and Donald for screen layout and line drawing exist in JS-EDEN, and these functions are respectively realised by using html and by rather *ad hoc* hybrid EDEN-JavaScript constructs. One motivation for carrying out this exercise is to explore the prospects for adding auxiliary definitive notations such as Scout and

Donald to JS-EDEN. With this in mind, we examine how syntax-directed translation can be most conveniently implemented.

Chapter 5 – Future directions and possible implications of the work in the thesis:

The work described in previous chapters has several implications and potential applications. The parsing model in EM provides tool support for an open learning and modelling environment in which the non-specialist can add new notations to EDEN and JS-EDEN. The Scratch-like interface to parsing model lays a foundation for an environment where the modellers can frame their definitions by assembling pieces of code rather than writing text. This has benefits in two aspects: making it easy to build structures; building structures that are easy to interpret. The deployment of parsing models in teaching supports a constructionist approach in which the learners can build their own parsers, handle activities such as error detection and correction, and explore others' parsing construals. Porting the EMPE to the web-enabled interpreter JS-EDEN makes it possible to devise tutorials that enable the parsing models to be deployed in open and distance learning. This in turn supports the construction of digital artefacts which promote many of the characteristics of an effective online course as identified in the *Manifesto for Teaching Online* [MTO]. The work on the parsing model in JS-EDEN also illustrates how EM principles make it possible to integrate model building on the web with configuration of the model building environment in a seamless manner. This is the basis for a "constructivist computing" approach to developing educational applications on the web [EBW].

6.2 Contributions and Further work

This thesis sets out to examine how online application of EM based on JS-EDEN could be more effectively supported and how this could help to disseminate and evaluate EM principles for educational technology (cf. section 1.5). The tutorial on making parsing models described in chapter 5 illustrates the potential for supporting online learning in a constructionist spirit. This tutorial was incorporated into a lab session in the annual IEM module in December 2013, and currently represents one of the most sophisticated EM exercises implemented in JS-EDEN. Further empirical studies, to include deployment in teaching automata theory, will be required to corroborate informal evidence that learning can be supported through making and exploring such construals. In the tutorial, the initial interaction is in the role of a learner, but subsequently traces the entire construction of the parsing construal with reference to the teacher and developer roles. The model building activity and the blending of the developer, teacher and learner roles is made more accessible in JS-EDEN because of the direct link to the underlying JavaScript and HTML representations. This gives the modeller a link to familiar ant conventional techniques that are absent in EDEN.

This thesis has also made a number of contributions to our understanding of parsing as a key ingredient in supporting EM for educational technology. By introducing dependency grammar, Chapter 2 has put the whole issue of parsing from an EM perspective – the connection between the experience of structure and the experience of meaning – in a new light. A full explanation of links with dependency grammar are beyond the scope of this dissertation. The AOP could be a way forward: it needs more dynamic contextual state. The criticisms of the AOP by Efstathiou [Efs06] may be influenced by the dominant ideas of phrase structure grammar, where the emphasis is on specifying parsing rules neatly and comprehensively once-and-for-all so as to be context independent. Chapter 3 illustrates how EM can be applied to make construals of relatively complex algorithmic procedures. We can now in principle add notations to JS-EDEN. Chapter 4 has contributed technically to this goal. It lays a foundation for empirical studies of teaching parsing theory in classical computer science. The work in this chapter also makes a significant technical contribution by way of specifying new structures in JS-EDEN that are implemented using the object-oriented prototyping mechanism in JavaScript. This will help future users to exploit prototyping structures in JS-EDEN.

References

[Ack94]Ackermann E. Construction and Transference of Meaning Through Form. In *Constructivist Education* (Steffe, L. & Gale, Eds.). Hillsdale, NJ: Lawrence Erlbaum Associates. In Press. (As cited in [Ack96].)

[Ack96] Ackermann E. Perspective-taking and object construction: two keys to learning. *Constructionism in practice: designing, thinking, and learning in a digital world*. Lawrence Erlbaum, Mahwah, NJ, 1996, 25-35.

[Als86]Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman *Compilers: Principles, Techniques, and Tools*, Pearson Education Inc 1986.

[BaC12] P Basile, A Caputo. Encoding Syntactic Dependencies using Random Indexing and Wikipedia as a Corpus IIR. 2012, 144-154..

[BeH10] W.M.Beynon and Antony Harfield. Constructionism through Construal by Computer. Constructionism 2010, The American University of Paris, August 2010.

[BeR02] Chris Roe and W.M.Beynon. Empirical Modelling principles to support learning in a cultural context. In Proceedings of 1st International Conference on Educational Technology in Cultural Context, University of Joensuu, Finland, 2002, pp 151-172.

[BeR04] W.M.Beynon and Chris Roe. Computer support for constructionism in context. Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference.(page 216-220).

[Bey83] W.M.Beynon. A definition of the ARCA notation. CS-RR-054, Department of Computer Science, University of Warwick, 1983.

[Bey88] W.M.Beynon. Jugs Model.
http://empublic.dcs.warwick.ac.uk/projects/jugsBeynon1988/. (access 10/03/2014)

[Bey01] W.M.Beynon. Definitive Notations- a Glossory for Empirical Modelling. University of Warwick, Online Resources. (access 10/10/2013).

[Bey13] W.M.Beynon. Personal discussion

[BeZ13] W.M.Beynon and Hui Zhu. Constructivist Computing - a New Paradigm for Open and Distance Learning?. Paper 382 in Proceedings of the 25th ICDE World Conference, Tianjin, China, 16 -18 October 2013.

[Bh86] W.M. Beynon, D. Angier, T. Bissell and S. Hunt.: DoNaLD: A Line-drawing

system based on definitive principles. CS-RR-086, Department of Computer Science, University of Warwick, 1986.

[Bha92] Namit Bhatia. *The Oxford Companion to the English Language*. Oxford University Press, Walton Street, Oxford,1992. (page 51-54)

[Brn01] Chris Brown. Agent-based Parsing System in Eden. Third Year Project, Computer Science, University of Warwick, May 2001.

[BS98] W.M.Beynon, P.H.Sun. Empirical Modelling: A new approach to understanding requirements. In Proceedings 11th International Conference on Software Engineering and its Applications, Vol 3,Paris, December 1998.

[Ch57] Noam Chomsky. *Syntactic structures*. Walter de Gruyter, 2002.

[Car06] Charlie Care. Graph Building Tool.
/dcs/emp/empublic/teaching/cs405-2006/lab5/ (accessed 10/10/2013)

[DC08] Douglas Crockford. *JavaScript: The Good Parts* O'Reilly Media, Inc., May 2008, 20-24, 50-52.

[EBW] Timothy Monks, Nicolas Pope, Richard Myers, Antony Harfield, Meurig Beynon and Hui Zhu. Web support for e-learning: a constructivist computing approach. Proceedings of the International Conference on E-Technologies and Business on the Web (EBW2013), University of the Thai Chamber of Commerce (UTCC), Thailand, May 7-9, 2013

[Efs06] George Efstathiou. C-GRAPH: A case study in the design, implementation and application of a definitive notation. MSc by Research thesis, Computer Science, University of Warwick, 2006.

[EMW] The Empirical Modelling Website.
http://www.dcs.warwick.ac.uk/modelling
(accessed 10/10/2013)

[EMWP] Online Resource from the Empirical Modelling Website.
http://www2.warwick.ac.uk/fac/sci/dcs/research/em/applications/educationaltech/
(accessed 10/10/2013)

[Fe08] Feng Zhiwei. *The Foundation of Computational Linguistics*. Beijing Commercial and Business Press, 2008.

[Gnu] Online Resource. http://www.gnu.org/software/sed/ (accessed 10/10/2013)

[Go90] D.Gooding. *Experiment and the making of meaning.* Kluwer Academic Publishers, 1990.

[Har03] Antony Harfield. Agent Oriented Parsing with Empirical Modelling. Third Year Project, Computer Science, University of Warwick, 2003.

[Har07] Antony Harfield. The Empirical Modelling Presentation Environment. http://empublic.dcs.warwick.ac.uk/projects/empeHarfield2007/

[Har08] Antony Harfield. Empirical Modelling as a New Paradigm for Educational Technology. PhD Thesis, Computer Science, University of Warwick, 2008

[He11] He Junjie. *Grammar Checking in a Combined Model: Phrase Structure and Dependency Grammars Reconciled.* Science Press, Beijing, November, 2011.

[Hel80] Hellwig, Peter (1980): PLAIN - A Program System for Dependency Analysis and for Simulating Natural Language Inference. In: *Bolc, Leonard (ed.) (1980): Representation and Processing of Natural Language*. München, Wien: Hanser; London: Macmillan, 271 - 376.

[Hel86] Hellwig, Peter (1986): Dependency Unification Grammar (DUG). In: *Proceedings of the 11th International Conference on Computational Linguistics (COLING 86)* , Bonn: Universität Bonn, 195-198.

[Hud84] Richard Hudson. *Word Grammar*, Oxford: Blackwell, 1984.

[Jam96] William James. *Essays in Radical Empiricism*, Bison Books 1996

[KaR96] Y.B.Kafai, M.Resnick. Introduction. In Y.B.Kafai, M.Resnick (eds). *Constructionism in practice: Designing, thinking, and learning in a digital world.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1996.

[Mon11] Tim Monks. A Definitive System For The Browser, MSc Project, Computer Science, University of Warwick, 2011

[MrH12] Mingrikeji. *HTML5：Dummy to Professional*, Tsinghua university press, September 2012.

[MrJ12] Mingrikeji. *JavaScript：Dummy to Professional*, Tsinghua university press, September 2012.

[MTO] Manifesto for Teaching Online. http://www.swop.education.ed.ac.uk/manifesto.html (accessed 10/10/2013).

[Niv05] Joakim Nivre. *Dependency Grammar and Dependency Parsing*, MSI report 5133 (1959): 1-32, 2005.

[Nor89] Fraser Norman. Parsing and Dependency Grammar, Carston, Robyn (ed.) UCL Work-ing Papers in Linguistics, 1989, 1: 29.

[Orm11] Jeanne Ellis Ormrod. *Essentials of Educational Psychology: Big Ideas to Guide Effective Teaching*. Boston, MA: Pearson Education Inc, 2011.

[Pap86] Papert S. *Constructionism: A new opportunity for elementary science education*. Massachusetts Institute of Technology, Media Laboratory, Epistemology and Learning Group, 1986.

[Pap93] Seymour Papert. *MindStorms*. Basic Books, A Member of The Perseus Books Group, Second edtion, New York, 1993.

[Pol08] George Polya. *How to solve it: A new aspect of mathematical method*. Princeton University Press, 2008.

[Pop11] Nicolas William Pope. Supporting the Migration from Construal to Program: Rethinking Software Development. PhD thesis, University of Warwick, December, 2011.

[Pop13] Nicolas William Pope. Personal Communication.

[Rob70] Jane J. Robinson. *Dependency structures and Transformational Rules*. IBM Watson Research Centre, New York, 1970.

[Roe03] Chris Roe. Computers for Learning: An Empirical Modelling Perspective. PhD Thesis, Computer Science, University of Warwick, November 2003

[SCW] Scratch website. http://scratch.mit.edu/. (accessed 10/10/2013).

[Tl59] Tesnière Lucien. *Eléments de syntaxe structurale*, 1959, Chinese translation by Deyi Fang, Renmin University of China (RUC,also known as the People's University of China) Press, 1987.

[Tru96] S.V.Truong. Interfacing EDEN with Oracle. Third year project, Computer Science, University of Warwick. 1996.

[Tur56] Turing A M. Can a machine think? *The World of Mathematics*, 1956, 4: 2099-2123.

[Vic12] Bret Victor. Learnable Programming.
http://worrydream.com/LearnableProgramming/ (accessed 10/10/2013)

[Vyg78] Lev Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*, Harvard University Press, 1978.

[Web EDEN] http://www2.warwick.ac.uk/fac/sci/dcs/research/em/software/webeden/

[Won03] Allan Wong. The Dependency Modelling Tool. Chapter 8 in PhD thesis, Computer Science, University of Warwick. January 2003.

[Wir76] Wirth Niklaus. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc. New Jersey,1976. 280-331.

Appendix 1:

Functions in the parsing model to search transition tables and construct currpath and currstack

```
func makepath {
   para inputlist;
   auto endix,result;
   endix =0;
   result=[0];
   while (inputlist !=[])
     {
     endix =     project(restrict(restrict(transitions,
     "state=="//str(endix)),"input=="//"\""//inputlist[1]
     //"\""),["nextstate"])[2][1];
     shift inputlist;
     result = result//[endix];
     }
   return result;
}
```

Appendix 2:

Functions rewritten for the Eddi parser to search the transitions list and construct the content stack and state stack.

```
func makestack {
   para inputlist;
   auto endix,result;
   endix=0;
   result=[];
   while ((inputlist !=[])&&(endix!=100))
      {
       writeln("input=="//"\""//inputlist[1]//"\"");
      if(endix==0)
          {
            for(i=1;i<=20;i++){
               if(transitions[30][i][2]==inputlist[1])
                   {
                      endix=transitions[30][i][3];
                      result=result//[inputlist[1]];
                      shift inputlist;
                      i=0;
                      break;
                   }
              }

           }
      else
           {
            for(i=1;i<=20;i++){
               if(transitions[endix][i][2]==inputlist[1])
                   {
                     endix=transitions[endix][i][3];
                     result = result//[inputlist[1]];
                     shift inputlist;
                     i=0;
                     break;
                    }
               }
            }
        }
   return result;
}
```

```
func makepath{
   para inputlist;
   auto endix,result;
   endix =0;
   result=[0];
   while ((inputlist !=[])&&(endix!=100))
     {
      if(endix==0){
         for(i=1;i<=20;i++)
            {
               if(transitions[30][i][2]==inputlist[1])
               {
                  endix = transitions[30][i][3];
                  shift inputlist;
                  result=result//[endix];
                  break;
               }

            }

         }
      else{
         for(i=1;i<=20;i++)
            {
               if(transitions[endix][i][2]==inputlist[1])
               {
                  endix = transitions[endix][i][3];
                  shift inputlist;
                  result=result//[endix];
                  break;
               }

             }
           }
      }
   return result;
}
```

Appendix 3:

```
Edge.prototype.drawEdge = function () {
    (function()context,ix1,ix2) {
        var o_Line = context.lookup('Line');
        var o_sqrt = context.lookup('sqrt');
        var o_pow = context.lookup('pow');
        var o_px = context.lookup('p'+ix1+'x');
        var o_py = context.lookup('p'+ix1+'y');
        var o_qx = context.lookup('p'+ix2+'x');
        var o_qy = context.lookup('p'+ix2+'y');
        var o_l = context.lookup('l'+ix1+ix2);
        var o_cl=context.lookup('col'+ix1+ix2);
        var o_mu = context.lookup('mu');
        var o_ptarr = context.lookup('pt'+ix1+ix2+'arr');
        var o_l1 = context.lookup('l'+ix1+ix2+'s');
        var o_l2 = context.lookup('l'+ix1+ix2+'t');
        var o_larr = context.lookup('l'+ix1+ix2+'arr');
        var o_rarr = context.lookup('r'+ix1+ix2+'arr');
        var o_x = context.lookup('l'+ix1+ix2+'dx');
        var o_y = context.lookup ('l'+ix1+ix2+'dy');
        var o_lambda = context.lookup('lambda');
        var o_mu = context.lookup('mu');
        var o_llength = context.lookup('llength');
        o_lambda.assign(9);
        o_mu.assign(15);

(o_x.eden_definition = "l"+ix1+ix2+"dx is 5/(sqrt(pow((p"+ix2+"x-p"+ix1+"x),2)
+pow((p"+ix2+"y-p"+ix1+"y),2)))*(p"+ix2+"x-p"+ix1+"x)", o_x.define(
    function(context)
    {
    return 5 /(o_sqrt.value().call(this,o_pow.value().call(this,(o_qx.value()-
    o_px.value( )), 2)+ o_pow.value( ).call(this,(o_qy.value()- o_py.value()),
    2)))*(o_qx.value()- o_px.value());
   }).subscribe(["sqrt", "pow", "p"+ix2+"x", "p"+ix1+"x", "p"+ix2+"y",
    "p"+ix1+"y"]));


(o_y.eden_definition="l"+ix1+ix2+"dy is 5/(sqrt(pow((p"+ix2+"x-p"+ix1+"x),2)
+pow(( p"+ix2+"y-p"+ix1+"y),2)))*(p"+ix2+"y-p"+ix1+"y)", o_y.define (
    function (context)
    {
     return 5/(o_sqrt.value().call(this,o_pow.value().call(this,(o_qx.value()-
    o_px.value()), 2)+ o_pow.value().call (this, (o_qy.value ()- o_py.value ()),
```

```
    2)))* (o_qy.value ()- o_py.value ());
   }).subscribe(["sqrt",   "pow",   "p"+ix2+"x",   "p"+ix1+"x",   "p"+ix2+"y",
    "p"+ix2+"y"]));


(o_l1.eden_definition="l"+ix1+ix2+"s    is    [p"+ix1+"x+l"+ix1+ix2+"dx,p"+ix1
+"y+l"+ix1+ix2+"dy]",   o_l1.define(
    function(context)
    {
     return [o_px.value ()+ o_x.value (), o_py.value ()+ o_y.value ()];
   }).subscribe(["p"+ix1+"x","l"+ix1+ix2+"dx","p"+ix1+"x","l"+ix1+ix2+"dy"])
);


(o_l2.eden_definition="l"+ix1+ix2+"t is [p"+ix2+"x-l"+ix1+ix2+"dx,p"+ix2+"y-l"
+ix1+ix2+"dy]",     o_l2.define(
    function(context)
    {
     return [o_qx.value ()- o_x.value (), o_qy.value ()- o_y.value ()];
   }).subscribe(["p"+ix2+"x","l"+ix1+ix2+"dx","p"+ix2+"y","l"+ix1+ix2+"dy"])
);


(o_l.eden_definition="l"+ix1+ix2+" is Line (l"+ix1+ix2+"s[1],l"+ix1+ix2+"s[2],
l"+ix1+ix2+"t[1],l"+ix1+ix2+"t[2],col"+ix1+ix2+")", o_l.define (
    function (context)
    {
       return o_Line.value ().call (this, o_l1.get (1 - 1).value (), o_l1.get (2
        - 1).value (), o_l2.get (1 - 1).value (), o_l2.get (2 - 1).value (),
        o_cl.value ());
   }).subscribe (["Line", "l"+ix1+ix2+"s", "l"+ix1+ix2+"t","col"+ix1+ix2]));


(o_llength.eden_definition="llength is sqrt (pow ((l"+ix1+ix2+"t[1]-l"+ix1+ix2+
"s[1]),2)+pow((l"+ix1+ix2+"t[2]-l"+ix1+ix2+"s[2]),2))",o_llength.define(
    function (context)
    {
       return o_sqrt.value ().call (this, o_pow.value ().call (this, (o_l2.get
        (1 - 1).value ()- o_l1.get (1 - 1).value ()), 2)+ o_pow.value ().call (this,
    (o_l2.get (2 - 1).value ()- o_l1.get (2 - 1).value()), 2));
   }).subscribe(["sqrt", "pow", "l"+ix1+ix2+"t", "l"+ix1+ix2+"s"]));

(o_ptarr.eden_definition="pt"+ix1+ix2+"arr  is[l"+ix1+ix2+"t[1]  -(l"+ix1+ix2+
"t[1]-l"+ix1+ix2+"s[1])/llength*mu,l"+ix1+ix2+"t[2]-(l"+ix1+ix2+"t[2]-l"+ix1
```

```
+ix2+"s[2])/llength * mu]", o_ptarr.define(
    function(context)
    {
       return [o_l2.get(1 - 1).value()-(o_l2.get(1 - 1).value()- o_l1.get(1 -
        1).value())/ o_llength.value()* o_mu.value(), o_l2.get(2 -
    1).value()-(o_l2.get(2 - 1).value()- o_l1.get(2 - 1).value())/
    o_llength.value()* o_mu.value() ];
   }).subscribe(["l"+ix1+ix2+"t", "l"+ix2+ix2+"s", "llength", "mu"]));


(o_larr.eden_definition  =  "l"+ix1+ix2+"arr  is  Line(  pt"+ix1+ix2+"arr[1]
+(l"+ix1+ix2+"s[2]-l"+ix1+ix2+"t[2])/llength  *(-lambda),  pt"+ix1+ix2+"arr[2]
+(l"+ix1+ix2+"t[1]-l"+ix1+ix2+"s[1])/llength*(-lambda),     l"+ix1+ix2+"t[1],
l"+ix1+ix2+"t[2], col"+ix1+ix2+")", o_larr.define(
function(context)
{
       return o_Line.value().call(this, o_ptarr.get(1 - 1).value()+(o_l1.get(2
- 1).value()- o_l2.get(2 - 1).value())/ o_llength.value()*(-o_lambda.value()),
o_ptarr.get(2 - 1).value()+(o_l2.get(1 - 1).value()- o_l1.get(1 - 1).value())/
o_llength.value()*(-o_lambda.value()), o_l2.get(1 - 1).value(), o_l2.get(2 -
1).value(), o_cl.value());
 }).subscribe(["Line", "pt"+ix1+ix2+"arr", "l"+ix1+ix2+"s", "l"+ix1+ix2+"t",
"llength", "lambda","col"+ix1+ix2]));


(o_rarr.eden_definition  =  "r"+ix1+ix2+"arr  is  Line(  pt"+ix1+ix2+"arr[1]
+(l"+ix1+ix2+"s[2]-l"+ix1+ix2+"t[2])/llength*lambda,      pt"+ix1+ix2+"arr[2]
+(l"+ix1+ix2+"t[1]-l"+ix1+ix2+"s[1])/llength*lambda,        l"+ix1+ix2+"t[1],
l"+ix1+ix2+"t[2], col"+ix1+ix2+")", o_rarr.define(
function(context)
{
        return o_Line.value().call(this, o_ptarr.get(1 - 1).value()+
(o_l1.get(2 - 1).value()- o_l2.get(2 - 1).value())/ o_llength.value()*
o_lambda.value(),o_ptarr.get(2 - 1).value()+(o_l2.get(1 - 1).value()-
o_l1.get(1 - 1).value())/ o_llength.value()* o_lambda.value(), o_l2.get(1 -
1).value(), o_l2.get(2 - 1).value(), o_cl.value());
}).subscribe(["Line", "pt"+ix1+ix2+"arr", "l"+ix1+ix2+"s", "l"+ix1+ix2+"t",
"lambda", "col"+ix1+ix2]));

})(root,this.source.value,this.target.value);

    if((root.lookup("p"+this.source.value+"x")).cached_value==undefined){(
    root.lookup("p"+this.source.value+"x")).assign(this.source.x);}
```

```
    if((root.lookup("p"+this.source.value+"y")).cached_value==undefined){(
    root.lookup("p"+this.source.value+"y")).assign(this.source.y);}

    if((root.lookup("p"+this.target.value+"x")).cached_value==undefined){(
    root.lookup("p"+this.target.value+"x")).assign(this.target.x);}

    if((root.lookup("p"+this.target.value+"y")).cached_value==undefined){(
    root.lookup("p"+this.target.value+"y")).assign(this.target.y);}

    (root.lookup("col"+this.source.value+this.target.value)).assign("red");

}
```

Appendix 4

```
${{
Textbox = function(name, titles, x, y) {
	this.name = name;
	this.titles = titles;
	this.x = x;
	this.y = y;

}
function maketitleshtml(titles,name) {
	var html = "";
	for(x in titles) {
		html = html + "\n"+ titles[x] + ":"+"<input
type=\"text\""+" id ="+name+" style="+"width:50px"+" onfocus =
root.lookup(\"focus\").assign("+"\""+name+"\""+");" + " onchange =
root.lookup("+"\""+name+"change"+"\""+").assign(1);"+">";
	}
	return html;
}

Textbox.prototype.draw = function(context,content) {
  var me = "canvas_" + this.name;
  var me2 = this.name;
  var box = $("#"+me).get(0);
  var me3 = maketitleshtml(this.titles,me2);
  if(box === undefined) {
	var dis = "";

	var can = $(content);
	var boxhtml = $("<form id=\"" + me + "\" " + dis + "
style=\"position: absolute; left: " + this.x + "px; top: " + this.y
+ "px;\"></form>");

	boxhtml.html(maketitleshtml(this.titles,me2));

	boxhtml.change(function() {

	root.lookup(me2 +"_value").assign($("#"+me2).get(0).value);
	}).appendTo(can);

	boxhtml.get(0).togarbage = false;

	root.lookup(me2 +"_value").assign($("#"+me2).get(0).value);
```

```
  } else {
        $(box).html(maketitleshtml(this.titles,me2));
        box.togarbage = false;
        box.style.left = "" + this.x + "px";
        box.style.top = "" + this.y + "px";
  }
};

Textbox.prototype.toString = function() {
  return "Textbox(" + this.name + ", " + this.titles + ", " + this.x
+ ", "+this.y+")";
};
}}$;
proc maintainforinput {
        ${{
                $("#input").get(0).value =
root.lookup("pp_value").cached_value;
        }}$;
};
```

```
unitoken.prototype.draw = function() {
    (function (content,name) {
       var o_Rect = context.lookup('Rectangle');
       var o_px = context.lookup('p'+name+'x');
       var o_py = context.lookup('p'+name+'y');
       var o_px2 = context.lookup('p'+name+'x2');
       var o_py2 = context.lookup('p'+name+'y2');
       var o_tname = context.lookup('t'+name);
       var o_ttext = context.lookup('t'+name+'text');
       var o_px3 = context.lookup('t'+name+'x');
       var o_py3 = context.lookup('t'+name+'y');
       var o_r = context.lookup('r'+name);
       var o_t = context.lookup('t'+name);
       var o_cl = context.lookup('col'+name);
(o_px2.eden_definition=     "p"+name+"x2     is     p"+name+"x     +
100",o_px2.define(function(context) {
              return o_px.value()+100;
          }).subscribe(["p"+name+"x"]));

(o_py2.eden_definition=     "p"+name+"y2     is     p"+name+"y     +
25",o_py2.define(function(context) {
              return o_py.value()+100;
          }).subscribe(["p"+name+"y"]));

(o_px3.eden_definition=     "t"+name+"x      is     p"+name+"x     +
15",o_px3.define(function(context) {
              return o_px.value()+15;
          }).subscribe(["p"+name+"x"]));

(o_py3.eden_definition=     "t"+name+"y      is     p"+name+"y     +
5",o_py3.define(function(context) {
              return o_py.value()+5;
          }).subscribe(["p"+name+"y"]));

(o_r.eden_definition           =           "r"+name+"           is
Rectangle(p"+name+"x,p"+name+"y,p"+name+"x2,p"+name+"y2,col"+name+",c
ol"+name+")", o_r.define(function(context) {
          return     o_Rectangle.value().call(this,     o_px.value(),
o_py.value(),       o_px2.value(),o_py2.value(),       o_cl.value(),
o_cl.value());
       }).subscribe(["Rectangle",                        "p"+name+"x",
"p"+name+"y","p"+name+"x2","p"+name+"y2","col"+name]));
```

```
(o_t.eden_definition = "t"+name+" is
Textbox(t"+name+",t"+name+"text,t"+name+"x,t"+name+"y)",
o_t.define(function(context) {
          return o_Textbox.value().call(this,
o_tname.value(),o_ttext.value(),o_px3.value(),o_py3.value());
       }).subscribe(["Textbox", "t"+name,
"t"+name+"text","t"+name+"x","t"+name+"y"]));

    })(root,this.name);

    (root.lookup("p"+this.name+"x")).assign(this.x);
    (root.lookup("p"+this.name+"y")).assign(this.y);
    (root.lookup(name+"Type")).assign(this.type);

}
```