Adaptive automata for context-dependent languages

João José Neto
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, trav. 3, n. 158. Cidade Universitária
CEP 05508 - São Paulo - Brasil
e-mail: JJNETO@PEC001.USP.ANSP.BR

Abstract - The present paper introduces the concept of **adaptive automata** as an alternative formal tool for describing context-dependent languages. This formal framework has the advantage of allowing easy mapping of a language description into an efficient parser for that language. Such a good performance is due to the potential hierarchical structure adaptive automata may exhibit, allowing natural construction of acceptors no more complex than strictly needed by each particular language. Efficiency is also due to the way adaptive automata operate, by changing according to its input, including and discarding transitions as needed to parse the particular input text - adaptive automata start from an initial self-modifying version, and evolve through a path of intermediate configurations until a final configuration is reached, when the source text is exhausted. The evolution from an automaton's configuration to the next one may be designed to occur strictly when a construct is found which is not recognized by the current configuration of the automaton. So, one may view the acceptance of a particular sentence as a sequence of recognitions of its substrings, each operated by the corresponding configuration of the adaptive automaton. That offers a practical way for efficiently accepting context-dependent languages in a purely syntactical way, allowing full treatment for syntactical aspects of the language such as dynamic syntax and the so-called static semantics. Then, the use of adaptive automata brings the possibility of handling in a purelly syntactical way several autentically syntactical concepts, such as predefined words, symbol-tables, scoping, type-checking, argument-to-parameter matching, macro definitions and expansions, syntax macros for defining new language constructs, and many others, usually treated semantically, or resolved outside the parser.

1. Introduction

This work introduces a formal model designed to describe context-dependent languages by means of a recognition device whose operation may be split into a sequence of partial recognitions of successive substrings of the input text by corresponding finite-state or pushdown-automata.

In a well-designed such a device, starting from a fixed initial configuration, each intermediate recognizer is obtained from the earlier one by means of input-driven self-transformations which gradually modify the shape of the automaton according to its needs for parsing the current input string.

This behavior is achieved by dynamically inserting and removing states and transitions, which are themselves allowed to further modify the resulting automaton.

Each intermediate recognizer may be viewed as a set of finite-state submachines, executing internal transitions in order to consume regular substrings of the language and external transitions to make calls and returns to submachines.

This organization was inspired on ideas from an early work by Conway [6] and from papers by Barnes [1] and Lomet [7].

In such a scheme, regular languages may be recognized by one of these submachines alone, with no help of any auxiliary storage.

Context-free languages may be accepted by using an additional stack to hold return states, in the same way sequential computers handle return addresses to perform procedure calls and returns.

So, each nonterminal may be implemented by a different submachine, activated through calls from any submachine implementing the current intermediate recognizer.

After recognizing the nonterminal associated to a submachine, its caller is resumed at a state retrieved from the top of the stack.

This structural simplicity allow the device to achieve an excellent speed by working essentially as a finite-state machine.

This approach is also found in several recent independent works, such as those by Cabasino [4] and Burshteyn [2,3], and the ones surveyed by Christiansen [5], although most of these tend to use grammatical formalisms, while our approach explore devices based on automata.

2. Structured Pushdown Automata

Before presenting adaptive automata, its basic underlying mechanism is informally described as being implemented by specially organized context-free recognition devices called structured pushdown automata.

Structured pushdown automata may be viewed as sets of finite-state devices with internal and external transitions, called submachines.

Internal transitions are those responsible by consuming input text and by state changes within a particular submachine.

External transitions provide means for calling a submachine from another one, and for returning back to the caller from final states of called submachines.

A structured pushdown automaton is a 8-tuple

$$M = (Q , A , \Sigma , \Gamma , P , Z_0 , q_0 , F)$$

where Q is the set of states of M, Σ is its input alphabet, Γ is its stack alphabet, to which the empty-stack marker Z_0 belongs, q_0 is a member of Q, denoting the initial state of M, and F is a subset of Q, representing the set of final states of M, all these elements having the standard meaning.

Set A is the collection of submachines a_i , i=1,...,n, implementing M, each of the form

$$a_i = (Q_i , \Sigma , P_i , q_{0,i} , F_i)$$

where $\{ Q_1 , ... , Q_n \}$ and $\{ P_1 , ... , P_n \}$ are respectively partitions of Q and P, F is a subset of Q , representing the set of return states of submachine a_i , and q_i is its entry state.

Each production p in set P has the general form

$$(\gamma\ g , e , s\ \alpha) : \rightarrow (\gamma\ g' , e' , s'\ \alpha)$$

where the left 3-tuple represents the current situation of M, and the right one denotes its situation after applying production p: γ and g' are the contents of the top of the stack, e and e' are states, s and s' are atoms form set S. Meta-symbols γ and a denote respectively parts of the stack and of the input string that do not affect the application of the production.

Particular cases of the above general form are relevant:

- when e and e' belong to the same submachine, production p is said to represent an internal transition of that submachine. In this case γ and g' must be empty.
- when γ is not empty, g' , s and s' have to be empty, and e' must be an entry state of some submachine. In this case, γ must represent a return state to which control is to be passed when the operation of the called submachine is over. In this case, production p denotes an empty transition that implements a submachine call.
- when g' is not empty, g, s and s' must be empty, and production p denotes an empty transition that provides a return to the calling submachine. State e' must be the return state previously pushed onto the stack when the current submachine was called.

By maintaining coherence between states and stack contents, one may obtain automata that use the stack strictly when it finds self-embedding constructions in the input string.

When a string is submitted to such an automaton to be accepted, the automaton must be first set to its initial situation (empty stack, initial state, full input string).

Next, a sequence of internal transitions are performed, consuming atoms from the input string.

Eventually, a call to some submachine occurs, control is passed to it, then, after its operation, a return state is reached with no possibility of further internal transitions, and the stack is popped allowing a proper return to the calling submachine.

This procedure is repeated while needed until a final situation of the automaton is reached (empty stack, some final state, input string exhausted).

The language accepted by such a device is the set of all strings that lead the automaton to any final situation.

The equivalence between structured pushdown automata and the traditional formulation for pushdown automata may be easily demonstrated by showing how to build a structured pushdown automaton that simulate a given standard one, and vice-versa.

3. Adaptive Automata

This section describes the ideas behind the formal structure of adaptive automata, by showing first the intuition of its operation, and then stating more rigorously some relations among its components.

An adaptive automaton M consists of the following components:

- ω - input string to be accepted by the device
- E - state machine implementing M at the start of its operation
- E - state machine implementing M just after accepting w (m>0)
- E - state machine implementing M just after executing i adaptive transitions (0<i<n)

w = a a ... a

by M may be macroscopically stated as the recognition of a sequence of substrings a w by the corresponding state machines E_j (0<j<m).

In other words, M describes a recognition path

$$<E , a > \rightarrow <E , a > \rightarrow ... \rightarrow <E , a >$$

where each element $<E_j , a >$ represents the recognition of the input substring a by the corresponding state machine E_j.

Let's refine the description by studying in more detail the role each of the state machines E plays while consuming its corresponding input string a .

Let w = w; Let w be the portion of w initially given as input to state machine E, either at the beginning of the process or immediately after an execution of any adaptive transition by state machine E.

Let a be the prefix of w to be effectively consumed by E the recognition of w by M may be considered as a process executing the following sequence of steps:

- E receives as input the string w =a ß
- E consumes the prefix a of w
- At this point, either w is exhausted (in this case, ß =e and M finishes its operation), or E executes an adaptive transition, evolving to E , and then activating it from an adequate state with w = ß as its input string

An adaptive transition P is described by a quadruple

$$P = (t , A , u , B)$$

where t and u denote the configurations that M assumes before and after the application of P , respectively.

A and B represent adaptive actions to be performed just before and after M changes its configuration to u , respectively (0<j<n). Adaptive actions are implemented as calls to adaptive functions, defined below

An adaptive function may be defined as a 9-tuple of the form:

$$(F , P , V , G , C , E , I , A , B)$$

where:

- F is the adaptive function's name
- P is the list (r , r , ...) of its formal parameters
- V is the set of identifiers of its variables
- G is the set of identifiers of its generators
- C is a set of production patterns to be searched and inspected
- E is a set of production patterns to be searched and removed
- I is a set of production patterns to be inserted
- A is an adaptive action to be performed before F is executed
- B is an adaptive action to be performed after F is executed An adaptive action represents a call to an adaptive function, and it is described by an ordered pair (F, P), where
- F is the adaptive function's name
- P is the sequence of arguments (p , p , ...) to be passed to F

The parameter-passing mechanism automatically assigns to each formal parameter r the current value associated to the positionally correspondent argument p when F is called.

All parameters are write-once (at parameter-passing time), then read-only, and no output parameters are allowed.

In order to state how an adaptive automaton defines a language, some definitions are needed.

Define a configuration t of the adaptive automaton at instant k to be the quadruple (E , γ , q , w), where

- E is the state machine that implements M at instant k
- γ is the contents of the stack at instant k
- q is the current state of E at instant k
- w is the input string to be processed by E at instant k

For k=0, we have the initial configuration of M, with γ =Z (empty stack), q the initial state of M, E the initial state machine implementing M, and w =w its full input string

Define also a final configuration t = (E , γ , q , w) where

- E is the state machine that implements M when w is exhausted
- γ = Z denotes an empty stack at that instant
- q is one of E 's final states if and only if w is a sentence
- w = e represents that w has been entirely consumed by M

M evoves from a configuration t to the next one t each time it executes one of its transitions.

Transitions of an adaptive automaton may be grouped in two categories: normal transitions, which are similar to the transitions of a finite-state or a pushdown automaton, and adaptive transitions, which are able to implement the dynamic behavior of the adaptive automaton by changing its topology.

One says that an adaptive automaton M defines a language L(M) in the following way:

Let w = a a ... a , a e Σ , i=1, ..., m, and let <E , a >→<E , a >→...→<E , a > be its recognition path, as defined above.

The configurations assumed by M at the start and just after the end of each step on its recognition path are, respectively:

$$t = (E , Z , q , w)$$

and

$$t' = t = (E , \gamma , q , w)$$

for (E , a) ;

$$t = (E , \gamma , q , w)$$

and

$$t'= t = (E , \gamma , q , w)$$

for (E , a) , 0<k<n;

$$t = (E , \gamma , q , w)$$

and

$$t'= t = (E , Z , q , e)$$

for (E , a).

Assuming that E reaches q , a final state of M, after exhausting its input string a , then w will be a sentence of the language described by M, and so we can define L(M) as being the set of all such strings:

$$L(M) = \{ w = e \Sigma \mid t \rightarrow t \rightarrow ... \rightarrow t \}$$

4. Notation

The following algebraic notation for specifying adaptive automata extends that chosen for structured pushdown automata by including adaptive actions, used to specify adaptive transitions.

Each of the transitions forming the adaptive automaton is denoted by a production of the form:

$(\gamma\ g , e , s\ \alpha : A , \rightarrow (\gamma\ g' , e' , s'\ \alpha , B$

which may be abbreviated to

$(e , s) : A , \rightarrow e' , B$

whenever $\gamma = g' = s' = e$, the empty string.

In this notation, γ and a are meta-symbols denoting respectively the contents of the stack and the part of the input string not yet considered by the automaton, both irrelevant to the application of the transition.

The current state of the transition is denoted by e , and its next state, by e' . The explicit top of the pushdown store is denoted by γ and g' , before and after the transition, respectively. These four symbols are optional, and, when omitted, they correspond to the empty string.

A and B are optional, and correspond to lists of calls to adaptive functions, to be performed respectively before and after the change of state determined by the production, and assume the general form

{ v , v , ... , v }

where each v represents a call to some adaptive function F .

Abbreviations are allowed in some particular cases: null actions, denoted by {}, may simply be omitted, and singletons of the form { v } may be written v , by dropping the brackets.

Calls to an n -parameter adaptive function F assume the form

F (t , t , ... , t)

where t are arguments that may assume any coherent value within the function's body.

A function declaration of F with n parameters q consists of a header

F (q , q , ... , q) =

and a body of the form

{ declaration of names (optional) :
declaration of actions (optional) }

where

declaration of names is a list of names chosen to represent objects in the scope of the function's body,

and

declaration of actions is a list of elementary adaptive actions, preceded and followed (optionally) by calls to adaptive functions.

Each declaration of names assumes the following form:

v , v , ... , v , g * , g * , ... , g *

where names followed by an asterisk denote generators, and the remaining names denote variables.

Values are assigned only once to variables by elementary adaptive inspection- and elimination-type actions (explained below), then those objects become read-only.

Generators receive unique values also once at the start of function execution and remain read-only until its exit.

Parameters are also assigned values by the parameter-passing scheme before executing the function. Each parameter assumes the current value of the positionally corresponding argument in the function call, remaining unchanged thereafter.

Declaration of actions is a list (eventually empty) of elementary adaptive actions, preceded and followed optionally by a call to some adaptive function (named initial and final adaptive action, respectively).

Initial and final adaptive actions correspond to adaptive actions designed to be applied just before and after the execution of the specified list of elementary adaptive actions.

Initial adaptive actions are executed just after parameter are passed to the function, and just before its execution is started, being allowed to inspect and use formal parameter values, but having no access to variables and generators, which are still undefined.

Final adaptive actions, which are executed just before exiting the function, may read and use values associated to any of its parameters, variables and generators.

Elementary adaptive actions assume the general form.

prefix [production pattern]

where

prefix chooses the type of elementary adaptive action to be performed:

"?" (inspection action),

"-" (elimination action) and

"+" (insertion action),

while

production pattern represents a parameterized production to which actions are to be applied.

Inspection-type elementary adaptive actions refer to adaptive production forms having the same format of adaptive productions, except for elements to be inspected, that are replaced by variable names.

These variables must be unique, being filled by the inspection mechanism with the current value of the corresponding inspected unknown elements, becoming read-only thereafter.

A production pattern assumes one of the already known forms

$(\gamma\ g\ ,\ e\ ,\ s\ \alpha : A\ ,\rightarrow(\gamma\ g'\ ,\ e'\ ,\ s'\ \alpha\ ,\ B$

or

$(\ e\ ,\ s\ ) : A\ ,\rightarrow e'\ ,\ B$

where g , e , s , g' , e', s' , as well as names and arguments of adaptive functions A and B, may be either constants or variables not used elsewhere.

An inspection mechanism searches the set of adaptive productions of the current instance of the adaptive automaton for any actual production whose components match

the production pattern's corresponding constants, and fills the variables with the current values of the corresponding production's elements.

If no such a production is found in the current instance of the adaptive automaton, the values assumed by variables associated to unknown elements remain undefined, and these variables will not be written any more.

Elimination-type elementary adaptive actions also reference adaptive production forms, and their initial operation is similar to that of inspection-type ones: the production pattern must be fully defined at the instant the elimination-type action is executed, otherwise it will be ignored, and matching productions will be eliminated from the set of adaptive productions.

Insertion-type elementary adaptive actions operate in the opposite way: the production pattern must be fully defined at the instant insertion-type actions take place, otherwise it will have no effect, and a production matching the production pattern will be included in the set of adaptive productions

The proposed notation provides abbreviations to denote, by means of universal quantifiers, sets of similar adaptive actions whose elements differ only in one of its basic component values. Set notation is used properly to implement these abbreviations:
{ list of actions A local variable e set }
where
list of actions refers to any set of elementary adaptive actions referencing a local variable, which assume successively the values of the elements in the set, so building the desired actions to be performed.

Note that each instance of the list of actions builds a corresponding set of actions by successively replacing, by consistent substitution, all occurrences of the local variable by each of the elements in the chosen set.

Local variables have their scope limited to such abbreviations of sets. The function must not declare them as variables or generators, but can use them locally when defining other sets, provided the scopes of those definitions are disjoint.

5. Operation

Alternatively to what has been stated before, the operation of an adaptive automaton may be described in terms of the transitions defined by its set of adaptive productions.

First, the automaton must be set to an initial situation

(Z , e , a)

meaning that the stack is empty, the automaton is at its initial state and the whole input string is ready to be processed.

Being the adaptive automaton in its current situation

(γ p , e , s a)

set P is searched for some production that matches that situation.

In the case only one such a production exists, the transition it represents will be deterministically executed.

If two or more productions match the current situation, a non-deterministic transition will take place by executing in parallel all corresponding transitions.

If no productions consuming some input atom match the current situation, a similar trial will be made for matching empty transitions.

If neither kind of productions is found that allow a transition to be executed, then the input string a will be rejected, otherwise another transition will be executed, and so forth, until a final situation is reached:

(Z , e , e)

where the automaton exhibits an empty stack, the current state is one of its final states, and its input string has been exhausted.

The language defined by an adaptive automaton is the set of all input strings that lead it to some final situation.

In order to complete the definition of adaptive automata, the way adaptive productions are executed must be described.

Let

(γ p , e , s a) : A ,→(γ p' , e' , s' a) , B

be a general form of some matching production to be applied to current situation of an adaptive automaton.

If A is present, it will be executed first. As a consequence, the current production may eventually be excluded from the set of productions defining the adaptive automaton.

In such a case the application of that production will be aborted, and a next production to be applied to the current situation will be searched for.

If a change is specified to the contents of the stack, an eventually explicit p will be first popped from it, and then p'_, if explicitly present, will be pushed onto the stack.

If a change is specified to the contents of the input string, it will be treated as a stack: an eventually explicit s will be popped, by pointing the input cursor to the symbol just to its right, and an eventually explicit s' will be pushed onto the resulting input stack, to the left of the remaining input string, at the current cursor position.

Next, the automaton leaves state e and goes to state e', and if some adaptive action B is present, it will finally be executed.

6. Example

A simple example is given below for illustrating the use of the above notation for specifying a context-sensitive language by means of an adaptive automaton.

The sentences of the language chosen for this example are simple sequences of five elements

- a left curl bracket ({)
- a list of identifiers, separated by commas
- a colon (:)
- a list of assignment statements, separated by semicolons
- a right curl bracket (})

Each identifier is a string of letters and digits, initiated by a letter. Like FORTRAN identifiers, those started by I, J, K, L, M or N are to be used as integer variables.

Each assignment statement is a sequence of three elements

- a left side, consisting of a single identifier previously declared in the list of identifiers

- an assignment symbol (=)
- a right side, consisting of a simple expression combining integer constants and declared identifiers of the same type as the one found in the left side

A full formal definition of the syntax for the above language may be described by means of an adaptive automaton as follows.

The adaptive automaton M will be initially stated as a set of three submachines L, S and A:

- a lexical analyzer L, responsible by the extraction of basic elements of the language, such as delimeters, reserved words, identifiers, punctuation symbols and integer constants
- a non-recursive submachine S, for defining the syntactic structure of a whole sentence
- a self-recursive submachine A, for defining the syntax of simple arithmetic expressions, used in assignment statements

Let ASCII, LETTER and DIGIT represent the sets of ASCII symbols, roman uppercase letters and decimal digits, respectively.

For s e ASCII, let Ds represent a unique token associated to the character s denotes, and by which submachines S and A, implementing the syntax analyzer, will recognize it.

Submachine→is automatically called each time the syntax analyzer needs an atom, and plays the role of a lexical analyzer.

L is given by the following set of productions (L1 is its initial state, and L6, L7, L8, L9 are final states that return to the calling submachine, respectively, id , id-int , id and id-real as the next token to be extracted from the input string):

- Ignoring blanks:→stays in state L1, consuming all blank characters found in the input string: (γ , L1 , " " a) :→(γ , L1 , a)
- Extracting special characters: in state L1, submachine→extracts the next input character, s, and converts it to the corresponding token Ds, returning to the calling submachine after inserting Ds at the start of the remaining input string:

{ (γ p , L1 , s a) :→(γ , p , Ds a)
A s e ASCII - (DIGIT u LETTER u { " " }) }

- Extracting integer numerals: When a digit is extracted in state L1,→consumes all digits following it until a non-digit is found, which is not consumed. A token num is then returned to the calling submachine as the next token to be read in

{ (γ , L1 , s a) :→(γ , L2 , a) A s e DIGIT }
{ (γ , L2 , s a) :→(γ , L2 , a) A s e DIGIT }
{ (γ , L2 , s a) :→(γ , L3 , s a) A s e ASCII - DIGIT }
(γ p , L3 , a) :→(γ , p , num a)

- Extracting identifiers: First, when a letter initiating an identifier of integer variables (I, J, K, L, M or N) is found at state L1, the remaining sequence of letters and decimal digits is extracted at state L4, otherwise at state L5. L6 and L8 are final states, reached whenever a new identifier is declared, and both return token id to the syntax analyzer. L7 and L9 correspond to final states, reached only when previously found integer or real identifiers are found, respectively. Adaptive action B is activated whenever a symbol is found that had never occurred before in the same position, and it

acts by eliminating the current production and inserting a new similar one with a different destination state, specifically created for recording the new path. In order to preserve the syntactical meaning of the automaton, from the newly added state an insertion is done of a set of productions equivalent to those emerging from the former destination state. From states L4 and L5 transitions emerge which activate adequate calls to adaptive action D, changing the token id returned by submachine→to id-int or id-real, respectively

```
 { ( γ , L1 , s a ) : B( L1 , s , L4 , L6 , L7 )→( γ , L4 , a )
A s e { I, J, K, L, M, N } }
 ( γ , L4 , a ) :→( γ , L6 , a ) , D( L4 , L6 , L7 )
 ( γ p , L6 , a ) :→( γ , p , id a )
 ( γ p , L7 , a ) :→( γ , p , id-int a )
 { ( γ , L1 , s a ) : B( L1 , s , L5 , L8 , L9 )→( γ , L5 , a )
A s e LETTER - { I, J, K, L, M, N } }
 ( γ , L5 , a ) :→( γ , L8 , a ) , D( L5 , L8 , L9 )
 ( γ p , L8 , a ) :→( γ , p , id a )
 ( γ p , L9 , a ) :→( γ , p , id-real a )
```

Adaptive functions B and D are declared as:

```
B( x , s , y , z , t ) = { j* :
 + [ ( γ , x , s a ) :→( γ , j , a ) ]
 { + [ ( γ , j , q a ) : B( j , q , y , z , t )→( γ , y , a ) ]
A q e DIGIT u LETTER }
 { + [ ( γ , j , q a ) :→( γ , z , q a ) , D( j , z , t ) ]
A q e ASCII - ( DIGIT u LETTER ) }
 - [ ( γ , x , s a ) : B( x , s , y , z , t )→( γ , y , a ) ] }

D( j , z , t ) = { :
 { - [ ( γ , j , q a ) :→( γ , z , q a ) , D( j , z , t ) ]
A q e ASCII - ( DIGIT u LETTER ) }
 { + [ ( γ , j , q a ) :→( γ , t , q a ) , D( j , z , t ) ]
A q e ASCII - ( DIGIT u LETTER ) }          }
```

Much simpler than L, submachine S is the initial submachine of the starting automaton and is given by the following productions, where adaptive action C inhibits the lexical analyzer to incorporate new identifiers as typed variables after declaration segment's end, while A configures submachine A to reject inconsistent variable types. The way this submachine works has no tricks. S8 is the final state of the adaptive automaton, identified by a trap in the last production:

```
( γ , S1 , D{ a ) :→( γ , S2 , a )
( γ , S2 , D: a ) :→( γ , S4 , a )
( γ , S2 , id a ) :→( γ , S3 , a )
( γ , S3 , D, a ) :→( γ , S2 , a )
( γ , S3 , D: a ) :→( γ , S4 , a ) , C ()
( γ , S4 , D} a ) :→( γ , S8 , a )
```

(γ , S4 , id-int a) :→(γ , S5 , a) , A(id-int)
(γ , S4 , id-real a) :→(γ , S5 , a) , A(id-real)
(γ , S5 , D= a) :→(γ , S6 , a)
(γ , S6 , expr a) :→(γ , S7 , a)
(γ , S7 , D; a) :→(γ , S4 , a)
(γ , S7 , D} a) :→(γ , S8 , a)
(γ , S8 , a) :→(γ , S8 , a)

Adaptive functions A and C are defined below:

A(s) = { :
- [(γ , A1 , id-int a) :→(γ , A2 , a)]
- [(γ , A1 , id-real a) :→(γ , A2 , a)]
+ [(γ , A1 , s a) :→(γ , A2 , a)] }

C () = { :
- [(γ , L4 , a) :→(γ , L6 , a) , D(L4 , L6 , L7)]
- [(γ , L5 , a) :→(γ , L8 , a) , D(L5 , L8 , L9)]
+ [(γ , L4 , a) :→(γ , L6 , a)]
+ [(γ , L5 , a) :→(γ , L8 , a)] }

Finally, define a simple arithmetic expression in the classical way, by means of self-recursive submachine A, through the followinγ set of productions.
Although A seems to be context-free, a production from A1 to A2, consuming id-int or id-real, is not shown in the set below, but is added and removed dinamically by adaptive function A, executed during the operation of S

(γ , A1 , num a) :→(γ , A2 , a)
(γ , A1 , D(a) :→(γ , A3 , a)
(γ , A2 , D+ a) :→(γ , A1 , a)
(γ , A2 , D- a) :→(γ , A1 , a)
(γ , A2 , D* a) :→(γ , A1 , a)
(γ , A2 , D/ a) :→(γ , A1 , a)
(γ , A3 , expr a) :→(γ , A4 , a)
(γ , A4 , D) a) :→(γ , A2 , a)
{ (γ , A2 , q a) :→(γ , A5 , q a)
A q m { D+ , D- , D* , D/ } }
(γ p , A5 , a) :→(γ , p , expr a)

7. Applications

One of the most important applications of adaptive automata is, obviously, in activities related to the implementation of computer languages, by providing an efficient

way to syntactical handling of context-dependent features such as: name handling, reserved word extraction, nested scoping, object typing, type checking, parametric macro definition and expansion, and other syntactical extension mechanisms.

All these applications may be implemented without any help of semantic actions. Instead, they may be done entirely by properly using adaptive transitions where needed.

In addition, some additional activities, usually implemented by other means, such as static semantics in a conventional language processor, may also be included in the syntactical handling provided by adaptive automata or by transducers based on adaptive automata: canonical code generation, syntax error handling, some kinds of code optimizations based on pattern-matching or pattern-substitution mechanisms, and even the replacement of storage-handling procedures by syntactical tools, implemented by adaptive transitions within an adaptive automaton.

Naturally, when implementing non-conventional compilers, such as those intended to be used in parallel or distributed environments, one may explore the powerful features of adaptive automata to implement devices that feature pattern-recognition mechanisms, and then use them to achieve some goals, such as identifying segments of the code that might be executed in parallel, recognizing mutual dependence among parts of the code, locating segments of the code suitable to be improved by using adequate machine-dependent instructions, and so on.

Many other fields of interest may use adaptive automata instead of conventional mechanisms to solve specific problems: digital communications (protocols, error handling), software engineering (formal specifications, man-machine interfaces, automation of design and implementation activities and tools), systems software tools (generators of systems programs from formal specifications), meta-programming, artificial intelligence (context-sensitive recognizing mechanisms, natural language processing, pattern recognition, learning devices, expert systems), computer-aided instruction, etc.

Because of its easy-to-learn general structure from which traditional recognition devices may be directly derived by imposing trivial restrictions to the former general model, and because of the existence of intuitive methods to obtain a good implementation from the formal specification it represents, adaptive automata are excellent to rapidly introducing formal language concepts and implementation devices to people not previously exposed to a full theoretical course in formal languages and automata.

8. Final remarks

Adaptive automata are somewhat general theoretical devices that allow one to exploit their special features to represent the behavior of the language being formally specified without loosing performance for simple languages.

That is particularly true because structured pushdown automata may be implemented from adaptive automata simply by using no adaptive transitions in its formulation, and finite state automata may be specified as a particular case of structured pushdown automata in which there is only one non-recursive submachine that does never use the adaptive automaton's stack.

So, once the design of an adaptive automaton is completed, it may be properly implemented in the cheapest way by choosing the simpler model that has power enough to fulfill the needs of the language being implemented.

It can be easily shown that one may use the proposed formal model for building extremely efficient context-free recognizers which operate deterministically in time O(n), as informally sketched below.

If the automaton is built from a set of submachines such that each submachine is deterministic and correspond to an essential non-terminal of a context-free grammar (the root of the grammar plus a set of independent self-recursive non-terminals), the stack will be modified strictly at the occurrence of self-embedded constructions.

Whenever such a syntactic form is identified, a stack symbol is pushed onto the stack at the time a submachine called, to be later popped out at the time a return is done to its caller.

All the rest of the automaton's operation is done through internal transitions, which consume one atom each.

As the total cost of recognizing a string with length n will depend linearly on the number of internal transitions, and on the number of submachine call-return transition pairs, which is also linearly limited by the length n of the input string, the resulting cost function will also be proportional to n.

This is of course an excellent result, and, as the operation of adaptive automata may be regarded as a sequence of operations of structured pushdown automata, one can also expect similar performance from adaptive automata, provided that there be an upper limit to the time taken by adaptive transitions to operate.

Although it does not apply in general, the last hypothesis does hold in several useful and somewhat complex applications, making it attractive to employ adaptive automata in many interesting situations.

9. References

[1] Barnes, B. H.
"A programmer's view of automata"
ACM Computing Surveys, vol 4, n. 2, Feb. 1972.

[2] Burshteyn, B.
"On the Modification of the Formal Grammar at Parse Time"
ACM SIGPLAN Notices, vol. 25, No. 5, May 1990, pp. 45-53.

[3] Burshteyn, B.
"Generation and Recognition of Formal Languages by Modifiable Grammars"
ACM SIGPLAN Notices, vol. 25, No. 12, Dec. 1990, pp. 45-53.

[4] Cabasino, S., Paolucci, P.S., Todesco, G.M.
"Dynamic Parsers and Evolving Grammars",
ACM SIGPLAN Notices, vol. 27, No. 11, Nov. 1992, pp. 39-48.

[5] Christiansen, H.
"A Survey of Adaptable Grammars"
ACM SIGPLAN Notices, vol. 25, No. 11, Nov. 1990, pp. 35-44.

[6] Conway, M. E.
"Design of a separable transition diagram compiler"
Communications of the ACM,vol. 6, n. 7, Jul. 1963, pp 396-408.

[7] Lomet, D. B.
"A formalization of transition diagram systems"
Journal of the ACM, vol 20, n. 2, Feb. 1973, pp. 235-257.