Compiler construction – a pedagogical approach

João José Neto, César Bravo Pariente, Fabrizio Leonardi
Escola Politécnica da Universidade de São Paulo, Brazil.
Departamento de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, travessa 3, número 158
05508-900 - São Paulo - SP - Brazil
E-mails: jjneto@pcs.usp.br , lupus@usp.br , fabrizio@cci.fei.br

Abstract

The aim of the present paper is to revisit some important topics regarding pedagogical issues on the teaching of concepts and techniques of programming languages and compiler construction. Essentially, a unified formal model is used in the proposed approach to explore the exposition of the students to a set of lessons with growing complexity, with a heavy practical component, all sharing the same common formal model. Experiments start with the definition of regular languages, aimed to introduce concepts and give the students familiarity with formal languages and their description through a metalanguage suitable for direct mapping into finite-state automata. Context-free languages are then considered as a simple extension of regular languages, both in grammar and acceptor aspects. The proposed formulation allow considering context-dependent languages as natural extensions of context-free ones, by adding some arrangements, while covering issues related to scopes, types, dynamic syntax, static semantics and language extensibility.

Keywords: Compiler construction, Context-free languages, Structured pushdown automata, Adaptive automata

1. Introduction

For many years we have been teaching computer languages and their compilers for Electrical and Computing Engineering students, and many experiences have been made while searching for some pedagogical approach that turns easier the task of transferring to the students information, technical issues and, especially, scientific foundations for the techniques.

At the time they have attended our classes, those students have already been exposed to related topics in other disciplines, and they have expressed preference in technical but non-heavily-theoretical disciplines, making the task of teaching pure computer-science disciplines a very hard task.

Because of their unquestionable importance in the formation of good professionals in this area, conceptual subjects like computer language issues, formal languages, automata theory, compiler construction and related topics do demand special care to be taught, in order to maintain motivation, despite how arid their theoretical aspects may appear to an Engineering student.

We decided to reduce this problem by addressing these subjects in a lighter form, without losing accuracy and without omitting any relevant aspect of the needed scientific foundations.

2. Conceptual framework

Our approach has been to adopt a single unified notation for all classes of languages, in order to minimize the need of learning a different notation for each kind of language [1,2].

Since Computer and Electrical Engineering students are familiar with the state-transition models they derive digital circuits from, our first choice when specifying our notation would be also trying to state our language descriptions in terms of states and transitions.

Next, we searched for a notation that be the closer possible to familiar notations that are frequently used for representing sequential circuits.

It would also be convenient that descriptions of simpler languages be simpler than those of more complex ones. We searched for conceptual differences between each class of languages and the next more complex one, in order to determine the correspondence between the presence of those features in the language and the corresponding notational needs.

From that investigation, we were able to select a notation which satisfies our requisites by showing a hierarchical structure that allows us to represent languages of any kind of complexity by strictly using the corresponding features of the notation that are really needed for their representation.

In the remainder of this section we justify and comment the more significant aspects of the features included in the notation.

Regular languages

- regular languages need no more than a finite-state machine for their descriptions.
- non-determinism may be easily eliminated from any finite-state machine by applying well-known classical algorithms.
- standard finite-state machines may accept any regular language in linear time.

Context-free languages

- non-regular context-free languages need pushdown automata to accept them. We may build such a device by using the model called structured pushdown automata, with one state machine for each essential syntactical construct in the language, and a pushdown store for coordinating their operation.
- it is possible to minimize the use of the pushdown store by properly choosing and designing the set of state-machines, so that auxiliary memory be used strictly when starting or finishing each embedded construct in the input text
- non-determinism may be eliminated in many languages by proper manipulation of the language description and of the corresponding state machines
- with this scenario, for deterministic languages, each state machine operates as a finite-state machine, except when handling embedded constructs, when an additional access to the pushdown store is needed at the start, and another one at the end of each embedded construct present in the input string. So, we may identify several substrings in the input text, each handled in linear time by some state-machine. We identify also several changes of control among the state machines, each demanding an extra access to the pushdown store. As we know, in any finite string the maximum possible number of embedded constructs is proportional to its length, so the time needed for handling embedded constructs is also a linear function of its length. We conclude that such a scheme lead to an accepting device that operate in linear time for deterministic languages.

- the demands for space is also linear for deterministic languages: what is needed is a static space for the transition tables, which, in the worst case, request space proportional to the total number of states, and a dynamic space for holding control information in the pushdown store, which is proportional to the length of the input string. So, for deterministic languages, space demands are also linear.
- for intrinsically non deterministic context-free languages, although there is no way to find any deterministic acceptor for them, we can manage to build structured pushdown automata that operate deterministically for some chosen subset of the language. For these devices, response time remains linear while input strings belong to that language subset. For sentences that do not belong to the subset, its operation will remain on deterministic. In this case, deterministic behavior is obtained for sentences in the chosen language subset at the cost of growing the size of the automaton. The wider the set of sentences we need the automaton to accept deterministically, the bigger will be the resulting automaton.

Other languages

- for languages that are neither regular nor context-free, the device we are looking for must have features that allow it to handle context-dependencies. By observing context-sensitive languages, we can verify that it differs from context-free ones in a fundamental point: whenever any context-sensitive construct is present somewhere in the input text, the whole text is reinterpreted accordingly. For an accepting device, this fact means that such an occurrence would have, as a response, an adequate change in the behavior of the acceptor. This kind of reasoning led us to the model named *adaptive automaton*, which is no more than a structured pushdown automaton, in which transitions associated to the discovery of some context-dependency in the source text drive the execution of attached *adaptive actions*, responsible for changing the automaton's behavior, which is accomplished by means of adequate editing operations on the set of transitions of the automaton.

Comments on the proposed notation

- from all the former information, we are now ready to state our notation. It will have three elements: the first one is a set of state transitions, needed to represent finite-state automata; the second is a pushdown store, needed to perform control actions in structured pushdown automata; the third element are the adaptive actions, which are attached to transitions in order to allow them to edit the automaton. Any transition in the adaptive automaton will always have the first element, but the others are optional.
- being strictly syntactical devices, adaptive automata allow describing rather complex languages without using anything else such as semantic routines or similar elements. For instance, a high-level syntactically extensible language may be completely described formally by means of adaptive automata, including the lexical and extension mechanisms, which are traditionally described separately, and its full static-semantics, including symbol tables, structuring scopes, type-checking and many others, which use to be handled by extra-syntactic elements, like semantic routines.
- once determined the notation to be used for describing languages, we should search for methods for automating the generation of processors for these languages. In this way we have developed a software tool that accepts as input a language description, drawn in a notation similar to that we had just chosen for the adaptive automata, that simulates the adaptive device it describes.

The proposed notation [3]

Any automaton will be represented as a collection of productions of the form:

$$(t, s, a, B) \rightarrow (t', s', a', B')$$

where each quadruple denote, in this order, the top of a stack, a state, an input symbol and an adaptive action. The left one denotes the configuration of the automaton before the transition takes place, and the right quadruple represents the configuration after the execution of the transition.

B represents an adaptive action to be performed before applying the production, and B' is another adaptive action, to be executed after the application of the production.

Adaptive actions are calls to adaptive functions, which are declared as lists of elementary adaptive actions of inspection, deletion and addition, applied to the current set of productions in the automaton. Elementary adaptive actions allow to edit the set of productions, in order to impose modifications to the behavior of the automaton, and has the following aspect:

$$\otimes[(t, s, a, B) \rightarrow (t', s', a', B')]$$

where ⊗ assumes one of the values ?, – or +, for inspections, deletions and additions, respectively.

Inspections allow asking for productions with a given shape in the current set of productions. Deletions allow eliminating productions with a given shape from the current set of productions. Additions allow adding new productions to the current of set productions.

Regular languages do not use terms t, B, t', a', B'. Context-free languages do not use B, a', B'. Both regular and context-free languages do never use adaptive actions. In this paper, we are going to use these simplified notation only, since no context-dependent features are needed here.

So, regular transitions will be denoted as $(s, a) \rightarrow s'$ and context-free transitions in which the stack is used will take the form $(t, s, a) \rightarrow (t', s')$.

3 Teaching issues [4]

In this section we report an experience made by applying the proposed notation in teaching practical aspects of compiler construction in an undergraduate computer engineering course, to students with basic knowledge of finite-state machines and context-free grammars.

Two monthly two-hour classes have been divided in two halves, the first of which was used to review and to introduce concepts and techniques, and the last one to perform graded practical exercises in class, preparing the student to implement the corresponding program-exercise in a computer, as a homework.

As a whole, the project consisted in the following steps:

The first stage was intended to exercise the ability of the students in formally defining regular and context-free languages. Starting from very simple languages, this step ended after the student has been able to define all important sublanguages of a typical programming language.

With the formal definition of all components of a language already available, the next step has been to integrate them into a single definition of a simple but complete programming language, defined by the student with the aid of textbooks on programming language design.

After that, but without being exposed to compiling techniques, students were asked to convert their formal grammars into automata, starting from regular constructs, which resulted in programs implementing all major parts of a lexical analyzer for the language, and then the rest of the language, resulting in parts of the syntax acceptor of the language.

Then, usual recognition techniques were exercised, through exercises in which the former grammars have been mapped into top-down and bottom-up parsers. The main results from this activity has been the increase in the capacity of the student to accurately decide when and why to use each technique.

The next topic was a deeper study of the particular technique described in the following section, in which a method is used that allows easy building of a recognition device for a language directly from a given grammatical formal definition of its syntax. In this topic, students have the opportunity of comparing the adopted technique with the others, especially in terms of building effort needed, and of execution performance. As a homework, the language defined by the students are manually converted into the corresponding recognition device. If time permits, the resulting acceptor is implemented in a computer, giving the students a good feeling of the process.

For shorter courses, the next step will be adding semantic routines to the recognizer, in order to generate code, and the course will be finished.

For normal courses, there is time enough to add a project of a generator of syntax acceptors, so introducing the concepts and practical issues of metacompiling.

Being already trained in manually building syntax recognizers, students are asked to build another acceptor, for the language of Wirth Notation texts. It will accept any grammar in Wirth Notation, so it would be tested with the formal definition of the language designed by the students as input.

Next, semantics will be added manually to the acceptor, in order to automate the activity already done manually by the student when building from the Wirth grammar the acceptor for the language. The resulting acceptor must be the equivalent to the one previously done manually.

The last step will be putting the acceptor to work and manually decorating it with semantic routines for code generation. This will be, indeed, a minor problem, since all this task will be reduced to migrating into our acceptor the already designed semantic routines, which we have implemented and tested in a previous step.

If time permits, a further test would be performed by using our tool with other grammars, in order to generate acceptors for other languages.

4 The method proposed [4]

The following text describes a step-by-step conversion from BNF-like context-free grammars into equivalent deterministic automata. For convenience, we will use the Wirth Notation as an intermediate metalanguage, in order to turn it easier the mapping of the grammar into an equivalent automaton.

A very simple method is then used that maps this Wirth grammar into a structured pushdown automaton which accepts texts in Wirth notation.

Semantic routines are then manually added to an optimized version of the resulting automaton, in order to force it to perform the same procedures we used to manually construct structured pushdown automata from Wirth grammars.

Denoting context-free grammars in Wirth notation

BNF-like notations are very used to state programming language syntax. The first step in the construction of our automaton from such context-free grammars is to convert them into Wirth notation (for other metalanguages the procedure is very similar). Apply the following steps:

- Associate a different set to each nonterminal in the original grammar, and collect in each set all rules defining the corresponding nonterminal.
- Eliminate from each set all left- and right-recursive rules defining the corresponding nonterminal, retaining self-embedded expressions:
- Let X be a nonterminal, and let all terms different from X in the following expressions represent arbitrarily long strings of terminals and nonterminals. In the set defining X in terms of expressions of the forms Xa_i, b_jX, c_kXd_k, e_m, Xf_nX, we will interpret, for convenience, expressions of the form Xf_nX as being of the form Xa_i, and c_kXd_k as being of the form e_m.
- It is easy to show that the corresponding Wirth expression is X = {b} e {a}, where b, e and a abbreviate $b_1|b_2|...|b_r$, $e_1|e_2|...|e_s$, and $a_1|a_2|...|a_t$, respectively, with r, s and t representing the number of terms of that form in the group. So, rewrite the set defining X in the form. X = {b} e {a}.
- Eliminate from the grammar all non-essential nonterminals. Essential nonterminals are the root of the grammar and a minimal set of independent self-embedded nonterminals in the grammar. In case of cyclic dependencies among nonterminals, choose from the cycle the nonterminal that be most directly derivable from the root . All other non-essential nonterminals may be eliminated by substituting successively all their occurrences in the grammar by the Wirth expression defining them, and then repeating the elimination steps until no more non-essential nonterminals remain.

Preparing a Wirth grammar to be mapped into a deterministic automaton

Once obtained the set of expressions defining all essential nonterminals, they must be manipulated in order to guarantee that they will lead, as far as possible, to an automaton with deterministic transitions.

We may eliminate, from the expressions defining each nonterminal, all constructs that cause nondeterministic transitions to appear in the resulting automaton:

- Explicit empty-string symbols appearing in the expressions will generate empty transitions, so we would eliminate them: convert expressions of the form a(ε | b)c into a(c | bc)
- Explicit nondeterminisms caused by the occurrence of common prefixes among alternative expressions may be eliminated by left factorization of the longest possible common prefix for the largest set of alternatives, and then, successively factoring eventually remaining common–prefixes in parentheses: Being a ~~be~~ the longest common prefix in $ab_1|ab_2|...|ab_n$, we rewrite this expression as a $(b_1| b_2|...| b_n)$.
- Hidden nondeterminisms due to the presence of nonterminals as prefixes in one or more of the expressions in a group of alternatives may be handled by replacing the nonterminal by the Wirth expression that defines it. Assuming that X→a is the definition of X, expressions starting by nonterminal X, like Xb, will be replaced by (a)b.
- Hidden nondeterminisms caused by optional cycles like {a} may be first turned explicit by rewriting it as a{a} |ε, and then eliminating the explicit empty string from the expression.
- After every transformation, the resulting expression must be checked for remaining nondeterminisms that have to be eliminated. For these ones, select in the expression the most local scope in which that nondeterminism is active, then eliminate all factorations previously made in that scope of the expression, before reapplying the former rules until no more nondeterminisms remain in the expression.

Mapping a prepared grammar into an automaton

Use one different initial state for each submachine, associated to one corresponding non-terminal in the grammar. Make the initial state of the starting submachine the initial state of the whole automaton.

All states corresponding to the left ends of those expressions are assigned the same state.

Each occurrence of a terminal in the expression corresponds to an internal transition between two states in the submachine corresponding to the nonterminal being developed.

Each occurrence of a nonterminal in the expression corresponds to a transition that calls the submachine associated to that nonterminal by pushing the next state into the stack and transferring control to the initial state of the called submachine.

States corresponding to the right ends of all expressions representing the syntactical options for a nonterminal are considered final states of the corresponding submachine.

Final states include an empty transition that returns control to the calling submachine. That is performed by simply popping the state contained in the top of the stack into the current state, so returning control to the calling submachine at the target state of the calling transition.

Groups of expressions in parentheses or brackets will have two distinguished states: one associated to its lefthand extreme and other, to its righthand extreme. The left state must be same state associated to the left extremes of all its expressions, and the states corresponding to all right ends of the expressions must converge into the single state associated to the right extreme of the group. Denoting an optional syntax, bracketed expressions ask for an extra empty transition from the states associated to their lefthand extreme to the one corresponding to their righthand extreme.

Groups of expressions in braces also ask for this empty transitions, but in this case the left extreme of each expression is associated to the righthand state of the group, and all states associated to their right ends will also converge into the state associated to the right extreme of the group, closing the loop.

Adding semantic actions

Semantic routines are needed to perform several functions in any language processor. In our case, adding them to a recognizer will allow it to drive them in order to generate code. For the Wirth notation, semantic routines will produce as output the transition function of the automaton that implements a recognition device for the language defined by the given Wirth grammar.

The semantics of the generation related to the structured pushdown automata from Wirth expressions may be resumed as follows:

For each Wirth expression, defining some nonterminal N :

- Start with an empty stack.
- Initialize state counters: CS= N0 (current state), NS= N2 (next state to be assigned).
- Assign N0 to the initial state of the automaton, and N1 to its final state.
- Push the pair (N0, N1) onto the stack, in order to memorize this assignment for future use.
- Scan the Wirth expression defining N from left to right. For each element in this expression, execute the corresponding action A, B, … , H:

A. **Terminal**: Generate a new transition from CS to NS, consuming **Terminal**.
Assign NS to CS. Increment NS

B. **Nonterminal**: Generate a new transition from CS to NS, calling submachine **Nonterminal**.
Assign NS to CS. Increment NS (e a pilha?)

C. | : Inspect the ordered pair (L, R) at the top of the stack.
Generate an empty transition from CS to R. Assign L to CS.

D. (: Push the pair (CS,NS) onto the stack.
Increment NS.

E. [: Push the pair (CS,NS) onto the stack.
Generate an empty transition from CS to NS. Increment NS.

F. { : Push the pair (NS,NS) onto the stack.
Generate an empty transition from CS to NS. Assign NS to CS. Increment NS.

G.),], } : Pop the pair (L,R) from the stack.
Generate an empty transition from CS to R. Assign R to CS.

H. . : Pop the ordered pair (L, R) from the stack.
Generate an empty transition from CS to R.

5 An example project

In order to apply the ideas commented above, we chose a small complete project through which many concepts and practical issues may be explored.

Being rather trivial, lexical analysis issues are not covered. Instead, we study syntactical and semantic aspects of the construction of a meta-recognizer by sketching its building it in detail.

Lexical analysis

Lexical analysis are easily addressed as a direct application of finite automata to the recognition of word categories in a language. For use in compilers, lexical analyzers are expected to perform as finite-state transducers, that extract input symbols from the source program, and generates a token for each lexical item found.

So, a lexical analyzer will have a finite-state automaton for extracting, from the source code, strings representing whole lexical items (identifiers, numbers, reserved words, punctuation, operators, etc).

Once a lexical item has been recognized, lexical analyzers classify them in their appropriate categories and generate as output a token, consisting of an ordered pair: (*class*, *value*) where *class* represents the categories to which belongs the extracted item (for use in syntactical analysis), and *value* represents complementary information needed for semantic analysis and code generation.

Lexical analyzer construction may be reduced to the design of a finite-state automaton that, whenever called, extracts from its input stream a maximum-length string that follow any of the formation rules for valid lexical items.

At each final state, when accepting another lexical item, the automaton generates as output the corresponding token, consisting of the information on the class to which belongs the extracted item, and the corresponding string of input symbols.

Syntactical recognition

The way we chose for accepting syntax in this experiment has been the standard acceptance of an input string by a structured pushdown automaton.

Being the acceptor formed by a set of finite-state automata, one of these so-called submachines is used as a starting submachine, whose initial state is the initial state of the whole automaton.

Starting from its initial state, the automaton performs successive transitions, based on the current symbol of the input string and the current state of the automaton.

As we have seen before, there will be two kinds of transitions: finite-state transitions and stack-dependent ones. The finite-state transitions operate just like in the case of finite-state automata. There are two situations where stack-dependent transitions will be used, both performing empty transitions between submachines (calls and returns) only.

For each transition, if the set of transitions in the automaton includes a single transition that is compatible with its current configuration, that transition will be deterministically executed, usually changing the current state and consuming an input symbol.

The input string is accepted by the automaton if and only if a final state in the automaton is reached, with the stack empty after the input stream is fully consumed by successive applications of valid transitions. Although it is not mandatory to impose that our stack to be empty in final configuration requirements, any other choice would be unnatural as a final configuration in our case.

In deterministic automata, all valid transitions will be unique. In non-deterministic automata, there may be several compatible transitions, then the configuration of the automaton will evolve non-deterministically, by simultaneously performing all compatible transitions in parallel.

In this case, we define that the input sequence of symbols is accepted by the automaton if any of the resulting paths leads to some final state, with the stack empty, when the input string is emptied. In any other casem the input string will be rejected.

Syntax description

The start point for this project is a grammar describing the syntax of the Wirth notation, stated in the same Wirth notation (consider **Terminal** and **Nonterminal** as terminals).

Wirth = **Rule** { **Rule** } .
Rule = **Nonterminal** "=" **Expression** "." .
Expression = **Term** { "|" **Term** } .
Term = **Factor** { **Factor** } .
Factor = **Nonterminal** | **Terminal** | "e" | "(" **Expression** ")" | "[" **Expression** "]" | "{" **Expression** "}".

We can simplify the former grammar rules by eliminating the non-essential nonterminals **Rule**, **Term** and **Factor**, by applying the following steps to the expressions defining **Wirth** and **Expression**:

Substitute the occurrences of **Rule** by its defining expression; then, substitute in the resulting expression the occurrences of **Term** by its defining expression; substitute all occurrences of **Factor** by its defining expression

The following grammar results, stated in terms of terminals and essential nonterminals only:

Wirth = Nonterminal “=” Expression “.” { Nonterminal “=” Expression “.” } .
Expression =
Nonterminal | Terminal | “e” | “(” Expression “)” | “[” Expression “]” | “{” Expression “}”
{ Nonterminal | Terminal | “e” | “(” Expression “)” | “[” Expression “]” | “{” Expression “}” }
{ “|” (Nonterminal | Terminal | “e” | “(” Expression “)” | “[” Expression “]” | “{” Expression “}”
{ Nonterminal | Terminal | “e” | “(” Expression “)” | “[” Expression “]” | “{” Expression “}” }) }.

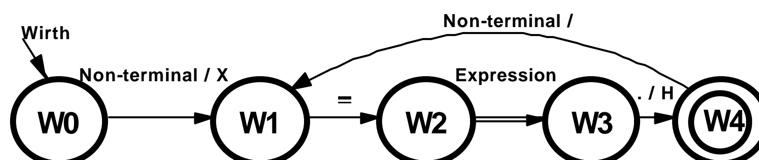
Construction of a recognizer for the Wirth Notation

>From the expressions above, the following structured pushdown automaton may be easily constructed manually by first applying our method, then eliminating empty transitions and equivalent states, through classical well-known algorithms.

Submachine **Wirth** (transition-table format):

	Nonterminal	Expression	=	.
0 (Initial State)	1/X			
1			2/–	
2		3/–		
3				4/ H
4 (Final State)	1/X			

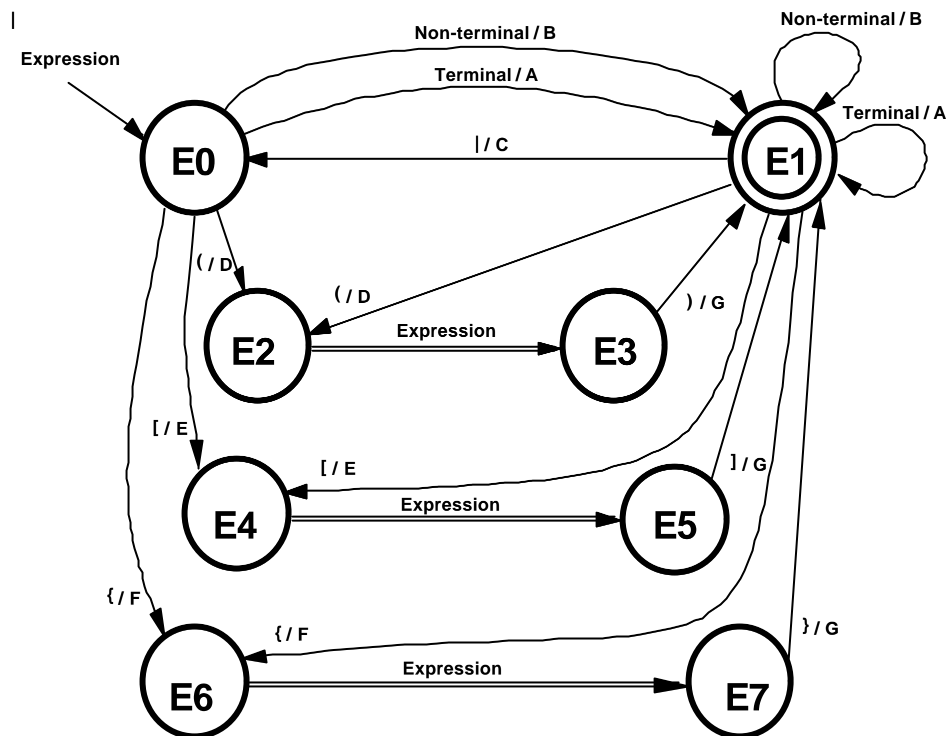
Submachine **Wirth** (transition-diagram format):



Submachine **Expression** (transition-table format):

	(	)	[	]	{	}	|	Terminal	Nonterminal	Expression
0 (Initial state)	2/D		4/E		6/F			1/A	1/B	
1 (Final state)	2/D		4/E		6/F		0/C	1/A	1/B	
2										3/–
3		1/G								
4										5/–
5				1/G						
6										7/–
7						1/G				

Submachine **Expression** (transition-diagram format):



A set of semantic actions are then manually attached to the transitions of the structured pushdown automaton, so converting it into a compiler of Wirth grammars into structured pushdown automata that accept the languages described by the input grammars.

In the tables above, we called X the routine that initializes the variables and the stack before each nonterminal definition. Routines A, B, ... , H refer to the homonymous routines sketched in the preceding section. All these semantic routines are executed whenever the corresponding transition is activated during syntax recognition.

By feeding this automaton with the Wirth expressions we just derived for nonterminals **Wirth** and **Expression**, the semantic routines are called in the correct order so that it will generate an automaton with submachines **Wirth** and **Expression**. These submachines, although being equivalent to ours, are not the same as the ones shown above, because they present many empty transitions and equivalent states. By removing such undesired transitions, it will result exactly the automaton we have presented above.

Pedagogically, this project is twofold: while serving as a base for the study of the construction of compilers, by addressing issues on the building of syntax acceptors and semantic routines, this project allows introducing concepts and practical aspects concerning the automatic generation of parsers for context-free languages, in a very natural form, thus abbreviating the exposition of students to those topics, otherwise usually uncovered in most courses.

6 Future Work

Although adaptive formal devices are still in their beginning as language description tools, they have already shown to be effective not only as pedagogical aid in teaching language-related subjects but also as excellent implementation models.

Many applications have been devised for adaptive devices, including language recognition

(adaptive automata), reactive systems description (adaptive statecharts), language generation (adaptive grammars), stochastic generative devices (adaptive Markov chains), and artificial intelligence applications, especially those related to machine learning.

Some of the main aspects of the study of those formal devices have been already addressed, but there are many others that remains to be explored.

In the particular case discussed in this article, no adaptive features have been used. We explored the hierarchical structure of the formal model, which allowed us to use a unified notation for both finite-state and pushdown automata, avoiding unnecessary and time-consuming teaching of more than one single notation.

As we mentioned before, a software tool is available that allows one to enter a specification of an adaptive automata (with any desired simplifications), from which the system automatically builds a simulator for the device being specified.

Such a tool is very useful for developing new languages, testing existing automata, and automatically building language processors from formal specifications. The example included in this paper shows in a very simplified form how such a metasystem works.

An integrated laboratory may be designed to explore this tool as a common framework for teaching many important topics in computer science, like: Introduction to computer science, formal languages and automata, introduction to compiler writing, programming languages, formal specifications of languages, systems programming, the design of sequential devices, and many others.

7 Conclusions

The approach we have used in the experience related to this paper has been completely successful in a number of aspects:

- the grammatical notation based on Wirth's notation is adequate for our purpose since it shows to be so close to the corresponding automaton and that a very simple mapping algorithm allows us to draw acceptors for the language through very simple manipulation of a given grammar.
- transducers may be obtained by extending such acceptors with an adequate output function. By choosing this function conveniently, one may map these acceptors into parsers, or produce an object language, or both.
- the unified notation based on adaptive automata is easy to learn and well-suited for the formal definition of acceptors for languages of any Chomsky type
- adaptive features are all fully processed in a strictly syntactical way, with no need for externally-coded routines
- by exploring adaptive features, extensible languages may be both defined and processed from a single formal definition, always by means of syntactic methods only, so providing a clear and easy-to-understand formal definition of those languages
- by exploring intuitive extensions of already-known simple models, this approach allows quick teaching of subjects related to quite complex topics, in a very simple manner, so motivating the students to go further into the corresponding theoretical topics
- by means of the use of this approach and accompanying each lecture with practical assignments regarding the next class, understanding of the subject has been significantly increased, and teaching speed grew by some 50%, with a simultaneous relative increase in the average grade

8 References

[1] Hopcroft, J.E. and Ullman, J.D. - Introduction to automata theory, languages and computation - Addison-Wesley, 1979.

[2] Aho, A.V. and Ullman, J.D. - The theory of parsing, translation and compiling - vol 1 - Prentice Hall, 1972.

[3] Neto, J.J. - Adaptive automata for context-dependent languages - SIGPLAN Notices, vol 29, no. 9, september, 1994.

[4] Neto, J.J. - Introdução à compilação - Editora LTC, Rio de Janeiro, 1987. (in Portuguese)