Adaptable Tokenizer for Programming Languages

Lect. Drd. Dan Popa
Univ. Bacău, Spiru Haret Nr 8, Bacău 5500, România
mail to: popavdan@yahoo.com cc: vpopa@ub.ro

Abstract: New extensions of the programming languages are frequently created. But implementing such extensions usually becomes an expensive and difficult task. Sometimes compilers and interpreters are generated based on sets of specifications. But the creation of such sets and/or the task of building a CWL (compiler writing language) is still difficult. It will be extremely easy for the language developer if the compiler or at least a part of it will be able to dynamically adapt itself to the needs of the new language without any modifications inside the program. This new concept of adaptable language (i.e. *without metalanguage specifications written by human hand*) is introduced here, based on the research of Dan Popa. Such a software component will be entirely reusable, dramatically reducing the cost of the project to which it belongs. Applications will be from language development to Internet browsing tools building, and more. The goal of the paper is to announce the apearence of an adaptable tokenizer which is closest to the AI technology and is performing adaptive analysis.

1. Introduction

Usually, a CWL (compiler writing language) is used to transform a set of specifications into a part of a compiler. For example, the lexical analysis is usually made by a set of carefully design finite determinist automata. The design specifications can be transformed in a final tokenizer on two ways: by writing the tokenizer by hand or by processing the set of specifications with a tokenizer generator like Lex or Flex. By consequence, the tokenizer should be rebuild every time when a new class of tokens is introduced in the language.

Let's take a look from an other point of view, because the future computer systems probably will be adaptable (or at least more adaptive)and will be able to automatically adapt their behavior to your needs and to your language.

2. The tasks of the tokenizer and the involved problems

The main task of a tokenizer (or lexical analysis tool) is to cut the stream of characters from the input. It is the most visible, but it is not the heavy one. The next one is to *classify* the atoms in groups like: numbers, identifiers, operators etc ... This is highly important. Tipical tokenizers are only able to recognize the classes of atoms which have already been built in the program of the tokenizer itself.

On the opposite side, It will be easy for an adaptable (or adaptive)tokenizer to learn new patterns of the atoms (words, numbers, identifiers, operators) which it had never ever seen before. The problem of recognizing new cathegories of atoms is in fact more difficult that the reader believes. Let's see the main difficulty:

The problem is how to realize that an other complex word is in fact a generalization of a previous one ! Remember, the rules are not known !

Let's take some examples:

a) 41 is a number. If an other number, like 1245 will come from the input stream, how did the tokenizer realize that it belongs to the same group ?

b) 12.34 is a real number. Will be the tokenizer smart enough to tell that 456.8775 belongs to the same group ?

c) If a strange kind of atom is coming from the input followed by a similar and maybe more complex one, how can the tokenizer realize that this new similar atom belongs to the same class ?

In fact, in our opinion, the difficult task of an adaptable tokenizer, which is able to learn by itself, is to *generalize* the informations it has processed in such a great mode that similarly tokens will be accepted as being parts of a previous identified class. During the research we had solved this problem by using a special data structure, which *is not a set of finite automata*. It did *not* used genetic algorithms or neural networks.

Let's see how the adaptable tokenizer is built:

3. Structure of the adaptable tokenizer

The adaptable tokenizer consists of some simple pieces:

- The buffer which stores the string to be processed.
- A classifier routine (GetClassCode) which is able to separate the digits (0,1,2,3,4,5,6,7,8,9,0), the letters of the alphabet ('a','b', 'c' ...'z', 'A', 'B' ...'Z') and the remaining symbols (';' "<", ">" ....everything). The space is considered the tokens separator because to say if "JOIANA" (or "JOEANA") is a pair of tokens (one of them being "ANA") is impossible without an inserted space in the middle of the sequence.

Remark: In the learning phase of the adaptable tokenizing process, the space (" ") between tokens should be necessary. But *after* the system had learned the classes of tokens, the informations extracted from it's data structure may be used to generate a classic, non adaptable and non space dependent tokenizer, if you wish.
- A special data structure (SDS). It acts as a self constructed base of extensions. This is the third piece.
- The mechanism used for generalization which is based mainly on the (SDS) properties. It allows the tokenizer to accept a generalization of a previous token as an element of a known class of tokens. In this case, a new class of token will *not* be created.
- The main loop of the Learning procedure is using the ClassCode Numbers given by GetClassCode routine and stores it in the SDS. (That's it, we are forced to clasify somehow the symbols of thealphabet itself, before the process.) When the current token ends, the data inside the SDS reflects the atom's class.

4. Alghorithm of this adaptable (adaptive) tokenizer

In the first phase the tokenizer is able to learn new kinds of atoms. Every new kind is getting a new number. The following alghorithm is used:

```
REPEAT
        Store the atom's informations in the SDS.
UNTIL TheAtomIsFinished
IF TheCreatedSignature get us something new.
THEN Inc(ClassNumber)
         Display.WriteStr(“A new class of atoms !”)
ELSE RollBack
END
```

The sequence of informations concerning the new atom is stored in the SDS. It may or it may not match an old signature.

When the atom ends, the tokenizer is looking for something new in the signature. If found, then a new kind of atoms have been discovered.

Otherwise the SDS is cleared, by removing the data entered by the previous loop. Let's remark the fact that such a procedure may basically double the time of the analysis, in the *worst case*. This means “when all the atoms belong to the same type”.

5.Running the adaptable tokenizer

The prototype of the tokenizer build during February 2004 in Bacău, Romania had only a 80 characters buffer. In order to respect the history and the language of the author, the output of the computer was recorded as it was. The English speaker is asked to tolerate some romanian words in the listing below and to look for the values of the set of classes, during the sequential processing of the tokens. In fact this is the goal of the adaptable tokenizer, to process texts which had been written in, virtually, *every language*. If a new kind of token is revealed, it's number is added to the set, as you can see below:

```
 DUmmy Precise LEXical analizer - DUPLEX
 The Ultimate Adaptable Lexical Analizer v.0.27.II.2004
 (c) Copyright by  Dan Popa 2004 popavdan@yahoo.com
 A week-end project by a non PH.D. from Bacau,Romania
 mail to: popavdan@yahoo.com or vpopa@ub.ro

Buffer Cleared ! Enter the text preceded by a space, followed by spaces:
> Carmen 1971 a indragit motanul 6.07.2001

 Processing atom #   1
 C a r m e n  O noua clasa de atomi lexicali !
 Setul de clase - Set Of Clases: { 1,}

 Processing atom #   2
 1 9 7 1  O noua clasa de atomi lexicali !
 Setul de clase - Set Of Clases: { 1, 2,}

 Processing atom #   3
 a  O clasa veche de atomi lexicali, nimic nou de facut !
 Setul de clase - Set Of Clases: { 1, 2,}

 Processing atom #   4
 i n d r a g i t  O clasa veche de atomi lexicali, nimic nou de facut !
 Setul de clase - Set Of Clases: { 1, 2,}

 Processing atom #   5
 m o t a n u l  O clasa veche de atomi lexicali, nimic nou de facut !
 Setul de clase - Set Of Clases: { 1, 2,}

 Processing atom #   6
 6 . 0 7 . 2 0 0 1  O noua clasa de atomi lexicali !
 Setul de clase - Set Of Clases: { 1, 2, 6,}
```

6. Conclusion

The first step in the new world of the adaptable language processing is made.This kind of systems are able to adapt themself to your language as you type, correctly identifying the different classes of tokens without knowing anything about them before. The adaptable (adaptive) tokenizer is build since February 2004 and it is proved to be functional.

The complexity of the algorithm is linear, but it is dependent by the implementation of the SDS.

To build such an adaptable tokenizer was an acceptable task, the program, written in Oberon being short (less than 300 lines of code, comments included).

So we don't need to rebuild the tokenizer every time when we are building a new language or every time when we extends an old one. The only thing to do is to run an adaptable tokenizer using a text written in the new language and save the data structure generated by it for the future analisis. Th compiler's tokenizer will use such informations instead the specifications of the language's atoms but will *reject* the new kind of tokens as errors.

Bibliography

Aaby, A. Anthony ***Introduction to Programming Languages*** **, http://cs.wwc.edu/aabyan/221_2/PLBOOK/ ; 1998-2002**

Crenshaw, Jack W. **Let's Build a Compiler (Compiler Building Tutorial 1.8, April 11, 2001) WWW resource**

Denning, J. Peter; Metcalfe, Robert M. ***Beyound Calculation, The Next Fifty Years Of Computing*****, Copernicus, Springer Verlag, New York, 1998;**

Micușa,D; Todoroi D ; Tsapcov V ; Drucioc N; Chelaru M ***Extensibilities' Cube in Object – Oriented Programming: Implementation of the Level-Level-Preprocesor-Compiler*****, The 9th Roumanian Symp. On Computer Science: ROSYCS'93, Iași, Univ Al. I. Cuza, pg. 265-288.**

Pârv Bazil, Vancea Alexandru ***Fundamentele limbajelor de programare*****, Editura Albastră , Cluj Napoca, 1996**

Popa, Dan **- *Rezultate conexe cercetărilor despre interacțiunile dintre compilatoare și interpretoare,* Simpozionul Internațional al**

Tinerilor Cercetători, ASEM, Chișinău 2003, p.387-388;

Popa, Dan ***Building an extensible language*** **– Internal code for the Interpreters with two program counters, Conferința Internațională - Rolul științei și învățământului economic în realizarea reformelor economice din Republica Moldova, ASEM Chișinău 2003, p.573-576.**

Popa, Dan **Programarea Calculatoarelor în Limbajul Oberon, (Note de curs),Universitatea Bacău, 2003-2004**

Șerbănați, Luca Dan ***Limbaje de programare și compilatoare*** **, Editura Academiei Republicii Socialiste România ; 1987**

Tery, P.D. **Compilers an Compilers Generators, an introduction with C++ , P.D. Terry, Rhodes University, 1996**

Vaida, Dragoș ***Algoritmi de compilare ,***
Editura Didactică și Pedagogică, București, 1971 ;

Zenger, Matthias; Odersky Martin
Implementing Extensible Compilers
Swiss Federal Institute of Technology,
INR Ecublens, 1015 Lausanne, Switzerland
(WWW ressource)