Comparison of Parsing Techniques For Formal Languages

Sunanda Mulik, Sheetal Shinde, Smita Kapase

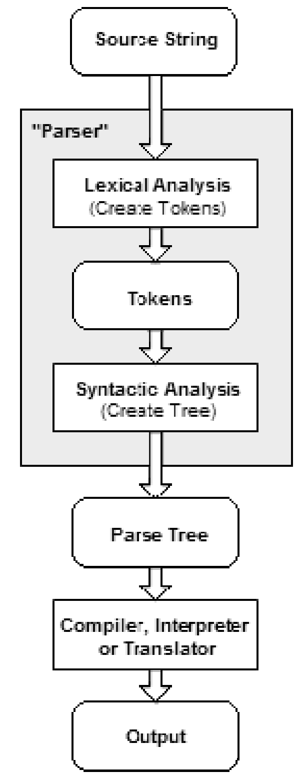
Bharati Vidyapeeth Pune

Abstract: A parser is one of the components in an interpreter or compiler, which checks for correct syntax and builds a data structure (often some kind of parse tree, abstract syntax tree or other hierarchical structure) implicit in the input tokens. Parsers may be programmed by hand or may be (semi-)automatically generated by a parser generating tool. Various techniques are available for parsing formal languages. The objective of this paper is to compare these techniques. The paper is organized in two sections. The first section does discuss about the parsing problem and process. In the second section we study and compare three parsing techniques theoretically .Finally the paper concludes with the suggestion of a new parsing technique.

Keywords: Parsing, classical, fuzzy, NLP technique, generic, machine learning

1. Introduction

1.1 Definition: In computer science and linguistics, parsing, or, more formally, syntactic analysis, is the process of analyzing a text, made of a sequence of tokens to determine its grammatical structure with respect to a given (more or less) formal grammar. Programming languages tend to be specified in terms of a context-free grammar so that fast and efficient parsers can be written for them. Parsers are written by hand or generated by parser generators. To extract information from the programs, they first parse the program code and produce an abstract syntax tree (AST) for further analysis and abstraction.



The process is divided into three phases

The first stage is the token generation, or lexical analysis, by which the input character stream is split into meaningful symbols defined by a grammar of regular expressions. The lexer would contain rules for that.

The next stage is parsing or syntactic analysis, which is checking that the tokens form an allowable expression. This is usually done with reference to a context-free grammar which recursively defines components that can make up an expression and the order in which they must appear. The rules for type validity and proper declaration of identifiers can be formally expressed with attribute grammars.

The final phase is semantic parsing or analysis, which is working out the implications of the expression just validated and taking the appropriate action. In the case of a calculator or interpreter, the action is to evaluate the expression or program; a compiler, on the other hand, would generate some kind of code. Attribute grammars can also be used to define these actions.

1.2 Overview of process

2. Parsing Techniques

2.1 Classical parsing

Classical parsers for formal languages have been known for many years. They conventionally accept a context-free language defined by a context free grammar. For each program, the parser does produce a phrase structure referred to as an abstract syntax tree (AST) .Parsers including error stabilization and AST-constructors can be generated from context-free grammars for parsers (Kastens et al., 2007). But a parser for a new language still requires the development of a complex specification. Moreover, error stabilization often throws away large parts of the source . These parsers are robust but do not care about maximizing accuracy. In this technique context is not taken into account. There are many cases where we want to work with the context of a detected element. But it never cares about the context of structures of interest. Also it does not provide error recovery. If the parsing of a system fails these approaches do not help the user to recover from the error. Mostly we do not even get feedback why and where there were problems. In addition it has a space for ambiguity.

Classical parsing techniques may be applied as long as the program conforms to the syntax of a programming language. This, however, cannot be assumed in general, as the programs to analyze can be incomplete, erroneous, or conform to a dialect or version of the language. Despite error stabilization, classical parsers then lose a lot of information or break down.

2.2 Fuzzy Parsing

Fuzzy Parsing (Koppler, 1997) was designed to efficiently develop parsers by performing the analysis on selected parts of the source instead of the whole input. It is specified by a set of fuzzy context free sub grammars each with their own axioms. Unlike conventional parsing, it does not require strict adherence to a language grammar. It scans for instances of the axioms and then parses according to the grammar. It makes parsing more robust since it ignores source fragments including missing parts, errors and deviations therein – that subsequent analyses abstract from anyway.

Island grammars (Moonen, 2001) generalize on Fuzzy Parsing in which parsing is controlled by two grammar levels, island and sea, where the sea-level is used when no island-level production applies. The island-level corresponds to the sub-grammars of fuzzy parsing. Island grammars have been applied in reverse-engineering, specifically, to bank software (Moonen, 2002). Like conventional parsers in this technique context is not taken into account: It does not care about the context of structures of interest. Also it does not provide error recovery: If the parsing of a system fails these approaches do not help the user to recover from the error. Mostly we do not even get feedback why and where there were problems. Also there is a space for ambiguity.

The following case studies show the limitations of fuzzy parsers

1. **DelphiXPG** utilises a Context/Fuzzy parsing technology for building a cross-reference of identifiers and has following limitations due to fuzzy parsing
   - Binary DFMs are not currently processed by the parser
   - if identifiers are referenced via a CPU register then the parser cannot identify these items
   - The parser will fail to identify an overloaded method if it is not uniquely identifiable though the number of parameters or by the data types of the arguments being passed to the method.
2. **OPARI** is a source-to-source translation tool which automatically adds all necessary calls to the pomp runtime measurement library which allows to collect runtime performance data of Fortran, C, or C++ OpenMP applications and has following limitations due to fuzzy parsing

   - Fortran 77/90:

     1. The !$OMP END DO and !$OMP END PARALLEL DO directives are required (and not optional as described in the OpenMP specification)
     2. The *atomic expression* controlled by a !$OMP ATOMIC directive has to be on a line all by itself.
     3. If the measurement environment does not support the automatic recording of user function entries and exits, the OPARI runtime measurement library has to be initialized by a !$OMP INST INIT directive prior to any other OpenMP directive.

- C/C++:

    1. structured blocks describing the extend of an OpenMP pragma need to be either compound statements {....}, while loops, or simple statements. In addition, for loops are supported after omp for and omp parallel for pragmas. Complex statements like if-then-else or do-while need to be enclosed in a block ({....}).
    2. If the measurement environment does not support the automatic recording of user function entries and exits, the OPARI runtime measurement library has to be initialized by a omp inst init pragma prior to any other OpenMP pragma.

2.3 Using Natural Language Processing(NLP)

The natural language processing techniques can be applied to formal languages. Dependency structure is one way of representing the syntax of natural languages. The same can be applied to formal languages also. The idea was introduced by Nilsson et al., 2009. The general approach is divided into two phases, training and production.

In the training phase, we need to train and adapt the generic parsing approach to a specific programming language. It consist of
1) Generate training data automatically by producing syntax trees and then dependency trees for correct programs
2) Train the generic parser with the training data.
This automated training phase needs to be done for every new programming language we adapt to.

In the production phase, we extract the information from programs which need not be correct and complete. It consist of
3) Parse the new source code into dependency trees.
4) Convert the dependency trees into syntax trees

This technique automatically generates the language specific information extractor using machine learning and training of a generic parsing, instead of explicitly specifying the information extractor using grammar and transformation rules. Also the training data can be generated automatically. It does increase the development efficiency of parsers, since only examples has to be provided. Unlike conventional and fuzzy parsers , they produce exactly one syntactic structure for every input, even if the input does not conform to a grammar and thus eliminates ambiguity.

The summary of various techniques is shown in following table

Compiler Technique ► / Characteristics ▼	Conventional	Fuzzy	NLP
Uses	CFG	CFG	Classifier
Advantages	Robust	More Robust	More Robust
	Error stabilization	Parses only selected parts of source	Accuracy approximately 100%
		Error stabilization	Automatic parser generation
			Needs no language specification
			Parser can be easily adapted for new languages
			Provide error recovery and feedback
			No Ambiguity
			Language independent generic parser
Limitations	Parses whole source-time consuming		Classifier is bound to commit errors even if the input is acceptable according to a grammar.
	Needs complete language specification	Needs language specification	Training phase must be repeated for every new language
	Parser development for new language is complex	Parser development for new language is complex	
	does not care about maximizing accuracy	does not care about maximizing accuracy	
	Does not care about the context of structures of interest	Does not care about the context of structures of interest	
	No error recovery and feedback	No error recovery and feedback	
	Ambiguity	Ambiguity	
	Language specific parser	Language specific parser	

Conclusion: From the table above it can be easily seen that parsers using NLP techniques have major advantages over classical and fuzzy parsing. Due to machine learning and automatic parser generation, it eliminates overhead of generating parser for every new language or version of the same and also does not need language specification. In addition, it provides approximately 100% accuracy. Fuzzy parsers allow for the fuzziness but they need language specification. If the two techniques can be combined together then we will be able to have a language independent generic parser which does provide fuzziness. Here we suggest to develop a new parsing technique based on the combination of above two techniques and our future work will be in that direction.

Acknowledgement: We kindly extend our gratitude towards the researchers and authors whose work help us a lot while writing this paper. Also we are grateful to Dr. Om Prakash Gupta as he has been a constant source of inspiration and motivation for us. Finally we can not forget to thank our family members and colleagues for the support they provided.

References:

[1] http://en.wikipedia.org/wiki/Parsing
[2] Jens Nilsson, Proceedings of the 11th International Conference on Parsing Technologies (IWPT), Paris, October 2009 c Association for Computational Linguistics
[3] delphixpg.com/docs/context**parsing**.htm
[4] http://www.fz-juelich.de/jsc/kojak/opari/
[5] A. Aho, J.ulhman, Principles of compiler construction