Ingénierie des Systèmes d'Information
Vol. 28, No. 6, December, 2023, pp. 1637-1642
Journal homepage: http://iieta.org/journals/isi

An Innovative Approach to Syntax-Free Interpretation in Functional Programming Languages

Omar Alaqeeli

Department of Computer Science, Saudi Electronic University, Riyadh 11673, Saudi Arabia

Corresponding Author Email: o.alaqeeli@seu.edu.sa

https://doi.org/10.18280/isi.280621

Received: 4 September 2023
Revised: 28 November 2023
Accepted: 5 December 2023
Available online: 23 December 2023

Keywords:
functional programming, lexical analysis, syntax analysis, parse tree, compiler, interpreter, lucid

ABSTRACT

In the realm of programming languages, interpreters fundamentally rely on syntax analysis (parsing) for establishing a correct evaluation hierarchy. Traditional parsing methods, however, present limitations in terms of optimization. This study introduces an innovative approach that circumvents syntax analysis in the interpretation of functional programming languages. The proposed method employs a novel subroutine, transforming program expressions into a series of atomic expressions, herein referred to as the "molecular program." Each atomic expression within this molecular program constitutes an element of the program's lexicon, assigned a unique identifier that supplants its role in the original expression. The evaluation process adopts a recursive methodology, where the evaluation of a single variable invariably leads to the sequential evaluation of related variables. For the purposes of clarity and demonstration, this approach is exemplified using Lucid, a notable functional programming language. It is posited that this syntax-free interpretation method can be universally applied to any functional programming language that operates on the principles of expressions, functions, or formulas. The efficacy of this method is validated through rigorous testing, suggesting an enhancement in the efficiency of programming language interpretation.

1. INTRODUCTION

1.1 Overview of lexical and syntax analysis in interpreters

In the architecture of programming language interpreters, two fundamental components are predominantly recognized: the Lexical Analyzer and the Syntax Analyzer. The Lexical Analyzer, also known as the Semantics Analyzer in certain studies [1-5], primarily serves to scan the source code, thereby generating a 'dictionary'. This dictionary maps each token to its corresponding conceptual identity, such as integers, identifiers, operators, and reserved words, effectively categorizing patterns within the source program [6]. This phase, constituting the initial stage of program interpretation and compilation, adheres to a set of predefined rules.

Subsequently, the Syntax Analyzer, often referred to as the 'parser' in literature, employs either Context-Free Grammar or, in specific instances, Synchronous Context-Free Grammar rules [7-10]. This process is designed to evaluate program statements accurately, utilizing the dictionary formulated by the Lexical Analyzer. For instance, an expression like '$x = 1$' undergoes syntactic analysis as *<id><operator><int>*, while a compound expression '$x + y - z$' is dissected into *<id><operator><id><operator><id>*. This marks the secondary phase of program interpretation.

1.2 Limitations of conventional parsing approaches

The implementation of the Lexical Analyzer is generally straightforward and can be conducted either manually or through automated means [11-15]. However, the deployment of the Syntax Analyzer presents more complex challenges. Particularly in scenarios involving significant similarities between program statements [16], or in the domain of image processing [17-19], the intricacies of Syntax Analysis are amplified. To address these complexities or to enhance the efficiency of Syntactical Analysis, various strategies have been explored. These include the implementation of combinators [20, 21] and the optimization of existing combinators [22-24]. In the context of functional programming, the design of such combinators, conceptualized as functions that operate on other functions, introduces additional layers of complexity. These complexities often manifest in increased runtime and memory consumption, which necessitate further resolution [25]. An alternative approach involves the utilization of external tools [26], such as Yacc, which perform functions akin to those of the Syntax Analyzer [27].

Within the scope of this study, the Lucid programming language, a functional language [28], is employed to exemplify the proposed methodology of bypassing the Syntax Analysis phase. This approach is anticipated to augment the efficiency of program interpretation, concurrently reducing the load on system memory and compilation time.

Lucid's syntax encompasses operators like '*fby*' and '*sby*', which regulate the flow and output of a value stream. For instance, in the expression '$i = 1$ *fby* $i+1$', 'i' represents the

sequential stream 1, 2, 3, and so on. Additionally, operators like '*first*' and '*init*' retrieve the initial value in a stream, while '*next*', '*right*', and '*succ*' access subsequent values. This is in conjunction with standard operators, such as arithmetic and logical operators, including '+', '-', '*AND*', '*OR*', and others.

Consider, for example, the Triangular number formula 'T_n' outputs the stream of values, and is represented in Lucid as $n = 0$ *fby* $n + 1$, and $x = n * (n + 1)/2$:

$$\sum_{x=1}^{n} x = 1+2+3+\ldots+n = \frac{n(n+1)}{2}$$

where, n denotes the stream commencing with 0, followed by (*fby*) 1, 2, 3, etc., while x computes the Triangular number. The forthcoming sections will elaborate on the computation of 'T_n' using the proposed methodology.

The structure of the remaining paper is outlined as follows: the method for transforming a program into a molecular program is detailed in the 'proposed approach' section. Subsequently, the 'evaluation approach' section elucidates the methodology of evaluation within this framework. Finally, the paper concludes with a discussion on limitations and potential avenues for future research.

2. PROPOSED APPROACH

2.1 Dismantling procedure

From a conventional perspective, the functional program presented previously in Lucid will pass through a Lexical Analyser that is followed by a Syntax Analyser. In our method, we dismantle each functional expression by first scanning the entire string of characters, and then transforming the expression components into atomic expressions. These expressions will be associated with unique identifiers that replace their existence in the original expression. For example, if $x = 3 + (2 + 1)$, then $2 + 1$ will be deducted from x, and its occurrence in x will be replaced by an identifier, say v_0, thus $x = 3 + v_0$. The main purpose of using unique identifiers is to retrieve the corresponding atomic expression in the event of evaluation. In other words, we transform the complete program into a "molecular" program.

Definition 2.1. An atomic expression is an expression that includes only one operator in its syntax. Such that if α is a Lucid atomic expression, then α has the following Context-Free Grammar:

|⟨*operator*⟩⟨*operand*⟩
|⟨*operand*⟩⟨*operator*⟩⟨*operand*⟩
|*if* ⟨*operand*⟩ *then* ⟨*operand*⟩ *else* ⟨*operand*⟩ *fi*

2.2 Transformation into molecular program

The dismantling is not arbitrary. It respects the precedence of operations and parentheses. With respect to Lucid syntax, the first components to be deducted are numbers. Operators that use only one operand, such as *first* and *next*, are deducted next, and then operators that use two operands, such as ∗, /, + and −. The final deduction is performed on operators that output a stream of values, such as *fby* and *sby*. For instance, if $x = 4$+*next i* then 4 is deducted first, and replaced by an identifier, followed by *next* and then +.

In our aforementioned Lucid program of the Triangular number T_n, the dismantling will proceed as follows: In the expression $n = 0$ *fby* n+1, the integer number 0 is deducted first and replaced by an identifier, say v_0. v_0 is also associated with 0 somewhere in the memory (in Java, for example, using HashMap is very efficient), thus ⟨v_0, *int* 0⟩, meaning that the value of v_0 = *int* 0 (the addition of *int* as an operator is necessary for v_0 to be considered an atomic expression).

At this iteration, the original expression $n = 0$ *fby* n+1 becomes $n = v_0$ *fby* n+1. This is followed by the deduction of the integer number 1 and replaced by v_1. Thus, $n = v_0$ *fby* $n + v_1$ and ⟨v_1, *int* 1⟩ is the new association.

The next segment to be deducted is where the operator is $n+v_1$. So $v_2 = n+v_1$ and ⟨v_2, $n + v_1$⟩, therefore, $n = v_0$ *fby* v_2. In the next iteration, n is already an atomic expression, therefore no further deduction is necessary, but n is associated with its atomic expression as ⟨n, v_0 *fby* v_2⟩.

The dismantling routine processes each expression enclosed between parentheses, if any, separately. Precisely, it processes expressions enclosed between the innermost parentheses and proceeds from there. So if $x = 1+(2*(3+4))$, then the innermost parentheses, (3+4), is deducted first, then (2∗(3+4)) and finally 1+(2∗(3+4)).

In the second line of the Lucid program, $x = n*(n+1)/2$, the dismantling starts with the value between the parentheses (n + 1). So 1 is deducted, followed by the addition +. The deduction continues with the second integer number, 2, and then processes the multiplication before the division, and thus operator precedence is preserved. The overall deduction of $x = n * (n + 1)/2$ is as Table 1.

Table 1. The overall deduction of x = n ∗ (n + 1)/2

(2.1)	$x = n * (n + 1)/2$	
(2.2)	$x = n * (n + v_3)/2$	$\longrightarrow \langle v_3, \textit{int}\ 1\rangle$
(2.3)	$x = n * v_4/2$	$\longrightarrow \langle v_4, n + v_3\rangle$
(2.4)	$x = n * v_4/v_5$	$\longrightarrow \langle v_5, \textit{int}\ 2\rangle$
(2.5)	$x = v_6/v_5$	$\longrightarrow \langle v_6, n * v_4\rangle$
(2.6)	$x = v_6/v_5$	$\longrightarrow \langle x, v_6/v_5\rangle$

The list of associations resulted from dismantling can be viewed as a "dictionary." Instead of tokens and descriptions, we will have variables and atomic expressions.

Theorem 2.2. *For every functional program λ, there is a molecular program λ′ that is equivalent to λ and consists of only atomic expressions.*

Proof. Let δ be the set of equations of the functional program λ, and δ′ be the set of equations of the molecular program λ′. The two sets are equivalent in the following sense: any solution to the first set can be expanded (by adding values for the v_n) into a solution for the second; and from any solution to the second we can extract (by discarding the values of the v_n) a solution to the first set.

In particular, the least fixed points (domain-theoretic minimal solutions) correspond in this way. So from the point of view of the Lucid denotational semantics, they have the same meaning.

2.3 Dismantle algorithm

Algorithm1: Transformation into molecular program
Input: Functional Program λ.

Onput: Functional Program λ'.
$i \leftarrow 0$;
$n \leftarrow 0$;
for *tokens in* λ **do**
if $token_i \in$ {N,W,Z,R} **then**
$\langle v_n, type + \text{“ ”} + token_i \rangle$;
$token_i \leftrightarrow v_n$;
$n \leftarrow n + 1$;
for *tokens in* λ **do**
if $token \in$ {*first, next,…NOT*} ***then***
$\langle v_n, token_i + \text{“ ”} + token_{i+1} \rangle$;
$token_i \leftrightarrow v_n$;
eliminate $token_{i+1}$ in λ;
$n \leftarrow n + 1$;
for *tokens in* λ **do**
if $token_i \in$ {*fby, sby, … , +, −, … , AND, OR, >, <, …* }
then $\langle v_n, token_{i-1} + \text{“ ”} + token_i + \text{“ ”} + token_{i+1} \rangle$;
$token_{i-1} \leftrightarrow v_n$;
eliminate $token_i$ & $token_{i+1}$ in λ;
$n \leftarrow n + 1$;
for *tokens in* λ **do**
if $token_i \in$ {*If*} **then**
$\langle v_n, token_i + \text{“ ”} + token_{i+1} + \text{“ ”} + token_{i+2} + \text{“ ”} + token_{i+3} + \text{“ ”} + token_{i+4} + \text{“ ”} + token_{i+5} + \text{“ ”} + token_{i+6} \rangle$;
$token_i \leftrightarrow v_n$;
eliminate $token_{i+1}$, $token_{i+2}$, $token_{i+3}$, $token_{i+4}$, $token_{i+5}$ &
$token_{i+6}$ in λ;
$n \leftarrow n + 1$;

3. EVALUATION APPROACH

3.1 Recursive evaluation logic

The evaluation of the molecular program is a simple recursion. A demand for an evaluation of one variable is a demand for an evaluation of another.

The transformation of T_n Lucid program results in the following molecular program T_n:

$$\langle v_0, int\ 0 \rangle\ \langle v_1, int\ 1 \rangle$$
$$\langle v_2, n + v_1 \rangle$$
$$\langle n, v_0\ fby\ v_2 \rangle$$
$$\langle v_3, int\ 1 \rangle$$
$$\langle v_4, n + v_3 \rangle$$
$$\langle v_5, int\ 2 \rangle$$
$$\langle v_6, n * v_4 \rangle$$
$$\langle x, v_6/v_5 \rangle$$

The computation of T_n starts with the demand for the evaluation of n. Since $n = v_0\ fby\ v_2$, this generates the demand for evaluating v_0 and v_2 (in the actual execution, *fby* evaluates v_0 in the first iteration and v_2 in the following iterations). $v_0 = int\ 0$ so the output of evaluating n in the first iteration is 0.

The evaluation of x follows, so the demand for evaluating x where $x = v_6/v_5$ will generate the demand for evaluating both v_6 and v_5. $v_6 = n * v_4$ generates the demand for evaluating v_4 that is $n + v_3$ and v_3 is *int* 1 thus $v_4 = 0 + 1$ therefore $v_6 = 0 * (0 + 1)$. v_5 on the other hand is *int* 2. Consequently, in the first iteration we will have $x = 0 * (0 + 1)/2 = 0$. Thus, $T_n = 0$.

In the second iteration, the second operand in $n = v_0\ fby\ v_2$ is evaluated. $v_2 = n + v_1$ generates the demand for evaluating v_1 that is 1 and since n in the first iteration is 0 then $n = 1$ in the second iteration. Then x is evaluated which is the demand to evaluate v_6 and v_5. When evaluating v_6, we will evaluate v_4 which is a demand for the evaluation of v_3 that is 1, so $v_4 = 1 + 1$ and therefore v_6 is $1 * 2$. The evaluation of v_5 is 2. Thus, $x = 1$. In the second iteration $T_n = 1$ and so on. Figure 1 shows the hierarchy of the evaluation process.

Contrary to the conventional methodology, the evaluation of an expression is simply a sequence of demands that is initiated by a demand of an evaluation of one variable. This reduces the complexity of interpretation process hence increase the efficiency of program interpretation.

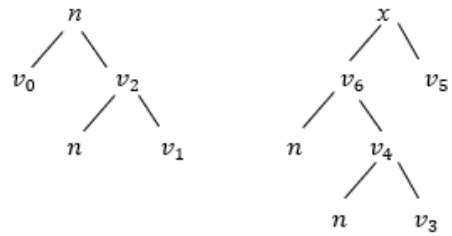

Figure 1. Evaluation hierarchy

3.2 Evaluation algorithm

Algorithm 2: Evaluation process

Input: A variable v_n & dictionary $\langle v_n, Atomic\ Expression_n \rangle$.
Output: Numeric Value.
Function $Eval(v_n, \langle v_n, AE_n \rangle)$ **is**
AE_n is <operator><operand$_1$>
If $v_n = operand_1$ **then**
else if AE_n *is* $\langle operand_1 \rangle \langle operator \rangle \langle operand_2 \rangle$ **then** $v_n = Eval(operand_1, \langle operand_1, AE_1 \rangle) \langle operator_n \rangle$
$Eval(operand_2, \langle operand_2, AE_2 \rangle)$
else if AE_n *is if* $\langle operand_1 \rangle$ *then* $\langle operand_2 \rangle$ *else* $\langle operand_3 \rangle$ *fi* **then**
if $Eval(operand_1, \langle operand_1, AE_1 \rangle) = True$ **then**
$v_n = Eval(operand_2, \langle operand_2, AE_2 \rangle)$
else
$v_n = Eval(operand_3, \langle operand_3, AE_3 \rangle)$

4. IMPLEMENTATION

For the purpose of demonstration, a related simple software called Luminous, written in Java, has been implemented by the author using the same approach. It can be found at: https://github.com/Omar-Alaqeeli/Luminous.

Briefly, Luminous has the following commands:

var v: used to declare a variable *v*, and assign a value to it.

val v: used to evaluate *v* and prints its value according to its dimensions.

defs: used to list all variables stored in the dictionary along with their expressions.

defined v: returns *true* if *v* can be evaluated and returns *false* otherwise.

constant v: returns *true* if there is no temporal operators (except for *first*) involve in an expression that is linked directly or indirectly to *v*. Otherwise return *false*.

dims v: returns dimensions that *v* depends on; *t* for the *time* dimension, *s* for the *space* dimension, *t & s* for both and none for nothing.

We use 1 to represent *true* in expression and 0 to represent *false*.

Luminous software package includes ten classes. Class *amend.java* is called to detect operators, such as +, -, *, / and parentheses, (,) and amends space between each operand and operator when reconstructing equations. This is because when the interpreter scan program statements, it needs to decide the beginning and the end of each string. In other words, distinguishing between operands and operators.

The class *analyzer.java* is used to scan program statements and pass them to *dismantler.java*.

The class *dismantler.java* is used to dismantle program statements into atomic expressions and construct a dictionary. This class uses java *Maps* that required two parameters (in our case two strings); identifiers and their associated atomic expressions.

The class *evaluator.java* retrieves atomic expressions from the dictionary and evaluates them in a hierarchal order. The evaluator considers all types of data, such as *int*, *float* (that is denoted as *flo* in Luminous), *first*, *init*, …etc. It also handles comparison, such as ==,!=, AND, OR, NOT.

The *amender*, the *analyzer*, the *dismantler* and the *evaluator* are the core of Luminous interpreter. There are other classes used for different purposes but they are necessary for Luminous to operate coherently.

The class *constQuery.java* is used to inquire if an operand is a constant or a variable. The class *evalQuery.java* is used to inquire if an operand can be evaluated or not. The class *list.java* is used to print all variables along with their associated expressions in the previously constructed dictionary.

The class *spaceDim.java* and *timeDim.java* are used to inquire about the dimensions that variables depend on. If *s* is the output, then the variable depends only on *space* dimension. If *t* is the output, then the variable depends only on the *time* dimension. If the output is both *s* and *t*, then the variable depends on both, *space* and *time* dimensions. When inquiring about the dimensions of a variable, the interpreter traces all elements related to that variable in the dictionary.

Assuming all the aforementioned files have been downloaded and located on the same directory, using system's terminal, all files can be compiled using the following command (Figure 2):

```
javac *.java
```

Figure 2. Compiling All Files from the Same Directory Using Terminal Commands

To compile one file, we use (Figure 3):

```
javac file_name.java
```

Figure 3. Instructions for Compiling a Single File

To run the interpreter after compiling all its necessary files, we type (Figure 4):

```
java luminous
```

Figure 4. Running the Interpreter Post-Compilation

When Luminous is running, the right shift operator >> appears and user can start typing commands.

Figure 5-11 are screenshots of Luminous which is operated through system's terminal to interpret Lucid programs.

```
>> var a=47;
>> var b=35;
>> var c=a+b;
>> val c
82
```

Figure 5. *var* and *val* in use

Notice that we can't add a and b directly without first defining a variable that is equal to their sum, c, thus the evaluation is inquired for c. Also, when defining variables, each sentence has to be concluded with a semicolon, otherwise error will be thrown.

```
>> var i=1;
>> var j=i fby j+1;
>> val j
1
2
3
4
..
..
```

Figure 6. When variable under evaluation has only one dimension, the stream of outputs is printed vertically (*time* dimension) or horizontally (*space* dimension)

The stream is continued infinitely (due to the nature of *Lucid* programming language) and in this case the terminal has to be interrupted manually.

```
>> var P=100 sby (P fby init P+1);
>> val P
100 100 100 100 100 ...
100 101 101 101 101 ...
100 101 101 101 101 ...
100 101 101 101 101 ...
100 101 101 101 101 ...
... ... ... ...
... ...
```

Figure 7. When evaluating a variable with two dimensions, the stream of values is printed in a form of a matrix.

For demonstration purposes, only five columns are printed (in reality, the stream is infinite). Since outputs are infinite therefore the terminal has to be interrupted.

```
>> var j=i fby i+1;
>> defs

Variable : Expression
V0 : int 1
V1 : i + V0
j :  i fby V1
```

Figure 8. *defs* lists all values in the dictionary; variables and their atomic expressions

```
>> defined j
false
```

Figure 9. *j* can't be defined because *i* is not yet defined. Hence, *j* can't be evaluated

```
>> constant j
false
```

Figure 10. *j* is not a constant since its expression includes the operator *fby* that constantly recalculates *j*

```
>> dims j
t
```

Figure 11. *j* relies on *time* dimension

5. DISCUSSION

Using the transformation approach we presented has indeed shortened the path of program interpretation. Instead of using lexical and syntax analysis, we transform the program into a molecular program that describes the identity of each variable as well as how it will be evaluated. The evaluation is a simple recursive process and no further intervention is needed when processing data, such as what was suggested in literature [29]. Dismantling a program is much simpler than syntactically analysing it. For instance, when evaluating $x = 7 \div (1 + 2 * 3)$, a conventional syntax analyser will scan 1+2*3 and decide the priority of $*$ before + following certain rules. Using our approach, we simply search for $*$ and / within the same close and then search for + and − following common knowledge of the operator's precedence. Although there are limitations, the dismantle approach seems to produce more positive results than anticipated.

5.1 Error handling benefits

Normally, error handling is an important part of the Syntax Analyser when evaluating, since lexical analysis is merely a tokenizing process regardless of the correctness of the program statement. When evaluating, statements are verified for syntax errors according to a syntax tree. Using our approach, we capture errors during transformation. When an operator is detected, its operands are also detected, and following the atomic expression Context-Free Grammar, an error is thrown, if any, before transformation. Since evaluation is a sequence of other variable evaluation then error are captures earlier.

5.2 Limitations and open questions

Some functional programming languages, like Haskell, use indentation as part of their syntax [30]. The dismantling method doesn't handle indentations, although this could be a future topic of research, namely, whether this method could be extended. One way to handle indentation is for every indent we rank program statement and based on that rank we decide whether it belong its precedent or not. Also, although the evaluation process is a simple recursion, in each iteration the dictionary is scanned for the demanded variable. This may be a burden and may negatively affect the efficiency of the program execution, although this is not yet clear.

The method as we use it is limited to functional programming languages; however, we conjecture that since it is applicable on Lucid, then it could be used in another functional language that doesn't use indentation. It is not clear whether it would make code interpretation slower or faster, but it would definitely make the construction of an interpreter much simpler. It is also unclear if this method can be extended to non-functional programming languages, so this is another research question.

6. CONCLUSION

The method as we use it is limited to functional programming languages; however, we conjecture that since it is applicable on Lucid, then it could be used in another functional language that doesn't use indentation. It is not clear whether it would make code interpretation slower or faster, but it would definitely make the construction of an interpreter much simpler. It is also unclear if this method can be extended to non-functional programming languages, so this is another research question.

REFERENCES

[1] Thielecke, H. (2014). On the semantics of parsing actions. Science of Computer Programming, 84: 52-76. https://doi.org/10.1016/j.scico.2013.04.010

[2] Girardot, J.J., Rollin, F. (1987). The syntax of APL, an old approach revisited. ACM SIGAPL APL Quote Quad, 17(4): 441-449. https://doi.org/10.1145/384282.28371

[3] Tayal, M.A., Raghuwanshi, M.M., Malik, L. (2014). Syntax parsing: Implementation using grammar-rules for English language. In 2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies, Nagpur, India, pp. 376-381. https://doi.org/10.1109/ICESC.2014.71

[4] Mitchell, R.J., Mitchell, R.J. (1991). The Analyser. Modula-2 Applied, Palgrave, London, pp. 232-250. https://doi.org/10.1007/978-1-349-12439-8_19

[5] Wirth, N. (1971). The design of a PASCAL compiler. Software: Practice and Experience, 1(4): 309-333. https://doi.org/10.1002/spe.4380010403

[6] Su, Y., Yan, S.Y. (2011). Principles of Compilers. Springer. https://doi.org/10.1007/978-3-642-20835-5

[7] Crescenzi, P., Gildea, D., Marino, A., Rossi, G., Satta, G. (2015). Synchronous context-free grammars and optimal linear parsing strategies. Journal of Computer and System Sciences, 81(7): 1333-1356. https://doi.org/10.1016/j.jcss.2015.04.003

[8] Huang, L., Zhang, H., Gildea, D., Knight, K. (2009). Binarization of synchronous context-free grammars. Computational Linguistics, 35(4): 559-595. https://doi.org/10.1162/coli.2009.35.4.35406

[9] Dyer, C., Lopez, A., Ganitkevitch, J., et al. (2010). cdec: A decoder, alignment, and learning framework for finite-

state and context-free translation models. In Proceedings of the ACL 2010 System Demonstrations, Uppsala, Sweden, Uppsala, Sweden, pp. 7-12.
[10] Gildea, D., Satta, G. (2016). Synchronous context-free grammars and optimal parsing strategies. Computational Linguistics, 42(2): 207-243. https://doi.org/10.1162/COLI_a_00246
[11] Grune, D., Van Reeuwijk, K., Bal, H.E., Jacobs, C.J., Langendoen, K. (2012). Modern compiler design. Springer Science & Business Media. https://doi.org/10.1007/978-1-4614-4699-6
[12] Sanju, V. (2016). An exploration on lexical analysis. In 2016 International Conference on Electrical, Electronics, and Optimization Techniques, Chennai, India, pp. 253-258. https://doi.org/10.1109/ICEEOT.2016.7755127
[13] Waite, W.M. (1986). The cost of lexical analysis. Software: Practice and Experience, 16(5): 473–488. https://doi.org/doi.org/10.1007/BFb0026419
[14] Sugimura, R., Akasaka, K., Kubo, Y., Matsumoto, Y. (1989). Logic based lexical analyser LAX. In Logic Programming'88: Proceedings of the 7th Conference Tokyo, Japan, pp. 188-216. https://doi.org/10.1007/3-540-51564-X_64
[15] Hill, J.M. (1992). Parallel lexical analysis and parsing on the AMT distributed array processor. Parallel Computing, 18(6): 699-714. https://doi.org/10.1016/0167-8191(92)90008-U
[16] Ferreira, R., Lins, R.D., Simske, S.J., Freitas, F., Riss, M. (2016). Assessing sentence similarity through lexical, syntactic and semantic analysis. Computer Speech & Language, 39: 1-28. https://doi.org/10.1016/j.csl.2016.01.003
[17] Zhang, S., Wang, J., Gong, Y., Zhang, S., Zhang, X., Lan, X. (2014). Image parsing by loopy dynamic programming. Neurocomputing, 145: 240-249. https://doi.org/10.1016/j.neucom.2014.05.037
[18] Kovásznay, L.S., Joseph, H.M. (1955). Image processing. Proceedings of the IRE, 43(5): 560-570. https://doi.org/10.1109/JRPROC.1955.278100
[19] Abràmoff, M.D., Garvin, M.K., Sonka, M. (2010). Retinal imaging and image analysis. IEEE Reviews in Biomedical Engineering, 3: 169-208. https://doi.org/10.1109/RBME.2010.2084567
[20] Danielsson, N.A. (2010). Total parser combinators. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, Baltimore Maryland, USA, pp. 285-296. https://doi.org/10.1145/1863543.1863585
[21] Izmaylova, A., Afroozeh, A., Storm, T. V. D. (2016). Practical, general parser combinators. In Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, St. Petersburg, FL, USA, pp. 1-12. https://doi.org/10.1145/2847538.2847539
[22] Kurš, J., Vraný, J., Ghafari, M., Lungu, M., Nierstrasz, O. (2018). Efficient parsing with parser combinators. Science of Computer Programming, 161: 57-88. https://doi.org/10.1016/j.scico.2017.12.001
[23] Kurš, J., Vraný, J., Ghafari, M., Lungu, M., Nierstrasz, O. (2016). Optimizing parser combinators. In Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies, Prague, Czech Republic, pp. 1-13. https://doi.org/10.1145/2991041.2991042
[24] Béguet, E., Jonnalagedda, M. (2014). Accelerating parser combinators with macros. In Proceedings of the Fifth Annual Scala Workshop, Uppsala, Sweden, pp. 7-17. https://doi.org/10.1145/2637647.2637653
[25] Hutton, G., Meijer, E. (1998). Monadic parsing in Haskell. Journal of Functional Programming, 8(4): 437-444. https://doi.org/10.1017/S0956796898003050
[26] Atkey, R. (2012). The semantics of parsing with semantic actions. In 2012 27th Annual IEEE Symposium on Logic in Computer Science, Dubrovnik, Croatia, pp. 75-84. https://doi.org/10.1109/LICS.2012.19
[27] Johnson, S.C. (1975). Yacc: Yet another compiler-compiler (Vol. 32). Murray Hill, NJ: Bell Laboratories.
[28] Wadge, W.W., Ashcroft, E.A. (1985). Lucid, the Dataflow Programming Language. London: Academic Press, 303.
[29] Goodman, D., Khan, B., Luján, M., Watson, I. (2013). Improved dataflow executions with user assisted scheduling. In 2013 Data-Flow Execution Models for Extreme Scale Computing, Edinburgh, UK, pp. 14-21. https://doi.org/10.1109/DFM.2013.10
[30] Adams, M.D., Ağacan, Ö.S. (2014). Indentation-sensitive parsing for Parsec. ACM Sigplan Notices, 49(12): 121-132. https://doi.org/10.1145/2633357.2633369