Some Methods of Applying Attributes for the Definition of Static Semantics

Ludmila Fedorchenko, Alexander Geida
St. Petersburg Federal Research Center of the Russian Academy of Sciences (SPC RAS)
St. Petersburg, Russia
{lnf, geida}@iias.spb.su

***Abstract*—Traditional theoretical works on language processing systems define the syntax of a formal language as a set of rules of grammar, which a compiler can check, all other language aspects, which we can detect only in runtime. We call them "semantics". Moreover, static semantics – i.e., those language properties which may be checked at the translation (compilation) time, distinguish from dynamic semantics – all other properties of this language. This article describes various approaches to defining the static semantics of an implemented language. A comparison of attribute-based language specification methods is also given. Examples provided. In addition, a new attribute-based approach to the implementation of contextual conditions in a translational context-free grammar in regular form are presented in the article. The attention is paid to the attributed automation model in the framework of automata theory to the decomposition problem and, to the application of attributed automata from the point of view of model developed.**

I. Introduction

Modern language processing systems are aimed not only at parsing the input text in a formal language, but at its transformation into some other form as well, provided the language syntax and semantics are properly specified. A common example is translation of programs in a high-level programming language into their internal machine representation to be executed by a computing device.

Conventional theoretical works in this area define the syntax of a formal language as a set of rules, which a compiler can check, all other language aspects, which we can detect only in runtime being referred to as "semantics". Moreover, static semantics – i.e., those language properties which may be checked at the translation (compilation) time, distinguish from dynamic semantics – all other properties of this language.

In this paper, we assume that grammar rules of a formal language represent its syntax. Some other rules, which determine correct usage of language words, compose its static semantics and all other properties of language expresses their dynamic semantics.

We suppose that a program in a formal language consists of words rather than of individual characters and therefore, terminal symbols of its context-free grammar (CF-grammar) are language words.

In comparison with the natural languages we find out that a lot of the programming languages have a strange feature, – significant set of the legal words have no predefined meaning or the meaning is partly defined and their semantic attributes are fixed only in the program. For example, an identifier in ALGOL 68 can denote any type of constant or variable or can denote a label or an operation, etc. In Basic, a letter followed by a left bracket, may denote an array (either one- or two-dimensional), etc. Therefore, the majority of programming languages have embedded mechanisms to create syntactical (semantical) attributes for all allowable language words.

In modern compilers syntactic checks are usually performed with tables where current attributes of already analyzed words (language constructs) are registered, check goes in the program text in the program text systematically from left to right.

Unlike powerful language such as ALGOL 68, Ada, Java or FORTRAN, in many very simple languages, like those for calculators, industrial or household equipment etc, and each word has a clear predefined meaning. In these cases, there is no static semantics in the language and formal correctness of a program in such simple language may be formally checked with its context-free syntax only. Although the proper work of the program is not guaranteed.

The branch of computer science, which deals with the theory of formal languages, since early 1950-es have developed a variety of abstract formalisms for defining programming languages in three, various directions:

- Substituting another, more powerful grammar for the initial context-free grammar;
- Extending the initial context-free grammar with various attributes and predicates (checks);
- Systematic modifying the initial grammar with a grammar transformer.

A number of attempts failed in the first direction until the van Wiingaarden grammar (VW-grammar) was introduced in order to formally define ALGOL 68 in 1965 [0], [[12]], [[13]], [[14]] with certain context dependent features embedded into the grammar (like "an applied occurrence of an identifier should match its defining occurrence").

Knuth highlighted the second direction in paper [[8]], which introduced inherited and synthesized attributes for each syntactic construct. This idea was enriched with Koster affix grammars [[9]], [[10]], [[11]] and became a theoretical basis for CDL [[11]], [[15]]. In 1973, Griffits proposed a similar method [[18]] independently of Knuth.

The third direction was based on Ledgard's work [[16]] and was applied in practice by Williams [[17]] et al. This approach assumes a table or a similar structure for current words with their attributes, while certain functions related to syntax constructs modify its contents. Terminal symbols of a context-free grammar act as transition functions, which force the parser to transit from one state to another. Attributes for syntax units of higher orders are introduced in a way similar to VW-grammars.

II. Comparison of VW- and affix Grammars

A VW-grammar is a context-free grammar with an infinite set of production rules presented recursively.

Grammar symbols are denoted by alphabetical strings, these are the so-called *protonotions*. The terminal symbols terminate with the word "symbol". The individual symbols are separated by commas.

For the construction of context-free production rules, schemes (so-called *hyper-rules*) are given. In the hyper-rules at both sides such symbols are composed of small-letter words (protonotions) and capital-letter words (*metanotions*). The metanotions are parameters. For each metanotion there are a possibly infinite set of protonotions generating a context-free grammar .

A production rule is generated from a hyper-rule in such a way that all occurrences of each metanotion must be replaced by a corresponding protonotion so that the same metanotion must be replaced by the same protonotion (synchronous replacement).

Using the metanotions we can generate (automatically or by hand) production rules which can fulfil any algorithm in a style similar to a Markovian algorithm. Those symbols in the hyper-rules have a special role from which we can derive the empty string. Such symbols are called predicates. They can express a relation between the parameters denoted by metanotions. If the relation is true, we can derive the empty string. If it is false, the derivation stops and a non-terminal symbol remains in the derived string. Therefore, the derivation is not valid.

A. Example of a VW-grammar

The following example demonstrates how to check that each variable is declared only once in a program (semicolons separate alternatives in meta-production rules.)

Metaproductions:

```
'ALPHA': : A, B;... X; Y; Z.
'LETTER': :  letter 'ALPHA'.
'NAME': :'LETTER'; 'LETTER' 'NAME'.
'DEF': : 'NAME' has 'MODE'.
'TABLE': : 'DEF'; 'TABLE' 'DEF'.
'DEFSETY': : TABLE'; 'EMPTY'.
'EMPTY': : .
```

Hyper rules:

```
Program:   Begin symbol,
           Declare of TABLE', TABLE' restrictions,
TABLE' statement train,
           end symbol.
```

(The " 'TABLE' restrictions" symbol is a predicate, which checks the unique declaration.)

```
'DEFSETY' 'NAME' has 'MODE' restrictions:
           where 'NAME' is not in 'DEFSETY',
'DEFSETY' restrictions;
            where 'DEFSETY' is 'EMPTY'.

Where 'NAME1' is not in 'NAME2' has 'MODE'
'DEFSETY':
         where 'NAME1' differs from 'NAME2', where
'NAME1' is not in 'DEFSETY';
         where 'NAME1' differs from 'NAME2', where
'DEFSETY' is 'EMPTY'.

Where 'EMPTY' is 'EMPTY': 'EMPTY'.

Where 'NAME1' letter 'ALPHA1' differs from 'NAME2'
letter 'ALPHA2':
         where 'NAME 1' differs from 'NAME2';
         'ALPHA1' is not 'ALPHA2'.

Where 'NAME' letter 'ALPHA 1' differs from letter
'ALPHA2': 'EMPTY'.
Where letter 'ALPHA1' differs from 'NAME' letter
'ALPHA2': 'EMPTY'
A is not B: 'EMPTY'.
A is not C: 'EMPTY'.
A is not D: 'EMPTY'
    .....etc.
```

As can be seen from the example, this type of definition is easily legible and comprehensive. On the other hand, we can see that the definition is rather redundant, not mathematically but in practice, since very simple functions are implemented in an artful way, using sophisticated string manipulations. Such a string manipulation is solvable in a computer but it is surely an ineffective solution.

B. Example of an affix-grammar

D.E. Knuth [[8]] proposed a context-free grammar in which every grammatical unit has a set of attributes. An attribute is called *ascendant* if it is derived from attributes of lower-level units and called *descendant* if it originates from a higher-level grammatical unit.

This concept and the concept of VW-grammar were effectively combined in the notion of an *affix grammar* [[9], [10], [11]].

In an affix grammar three types of objects are considered. Non-terminal symbols denote grammatical units, terminal symbols are words of programs and checks are *predicates* over attributes. All types of objects have a definite number of attributes; terminal symbols have no attributes.

Using a set of context-free substitution rules, non-terminal characters are replaced by a string consisting of terminals, non-terminals, and checks. Attributes of non-terminals and checks in both sides of each rule are connected with objects. An attribute is denoted by a symbol or it can have constant value, too.

A non-terminal symbol with attributes with certain values can be replaced with the string in the right-hand side of the respective rule with this non-terminal in the left-hand side). The attributes of the objects must be given so that *attributes, which were denoted by the same symbol must have the same value*. Then, in the new string, we must substitute for the non-terminal symbols repeatedly and the checks must be evaluated. Every check means a recursive predicate over the attributes. If the check is true, the empty string will substitute it. If it is false, it will be substituted for by a non-terminal symbol, which has no further derivation. The derivation is finished when a string consisting of terminal symbols only is produced.

The example below extends the previous one with more detail on the derivation mechanism

```
Program: Begin,
       Declaration+'TABLE',
       Restrictions+'TABLE',
        Statement train+ 'TABLE' ,
        End.
```

Here the "restriction" is a predicate over the domain of the 'TABLE's. It checks that every name is unique in the 'TABLE'.

```
Declaration 'TABLE': Declare+ 'MODE'+ 'TABLE';
               Declare+'MODE'+'SUBTABLE 1',
               Declaration+'SUBTABLE2' ,
               Union+'TABLE' +'SUBTABLE1+'SUBTABLE2'.
```

'Union' checks that the 'TABLE' is the union of the two 'SUBTABLE's

```
Declare+'MODE'+ 'TABLE': Declarer+'MODE',
             Idlist+ 'TABLE'+ 'MODE'.

Idlist+ 'TABLE + 'MODE': Identifier*'NAME',
          Include+'TABLE'+'NAME'+'MODE',
          Semicolon;
          Identifier+'NAME',
           Include+'TABLE'+'NAME'+'MODE',
           Comma,
           Idlist+'TABLE'+ 'MODE'.
```

("Include" checks that 'NAME' having 'MODE' is included in TABLE'.)

Based on affix grammars, Koster created the Compiler Description Language (CDL) [[15]]. A CDL program is a syntactic/semantic definition, which can be translated into a code (subprogram) to parse the program text. In the form of the grammar, there were modifications, which turn the description of languages, and the execution of the parsing shorter.

Text of a CDL program are translated into calls recursive procedures, while terminal symbols and checks are translated into substitutions of user-defined *macros*.

The body of a procedure is a sequence of macro and procedure calls. The calls are given in accordance with the sequence of objects in the substitution rule. Parsing of the text goes on from top to bottom and from left to right.

Attributes are parameters of the procedures and macros. Descendant attributes are input parameters and ascendant ones are output parameters. So left-recursive rules are excluded.

Thus, an affix grammar is a frame for the execution program, where syntax and data united. In affix grammars, the values of attributes are strings generated by a context-free grammar though tables, lists, or some other data structures would be more natural to represent them. In CDL, attributes are data structures, integers and integer arrays. They are used to represent the necessary logical data structures mentioned above.Conceptions and the solution of the mentioned problem with VW- and affix grammar are very similar, but the latter mechanism is more explicit and thus much easier for realization.

The CDL-method appeared to be much less efficient than expected. Its inefficiency is due to two reasons: recursive calls to procedures and a large number of parameters transferred at each call. Majority of attributes are logically related to terminal symbols which themselves are deprived of attributes. Therefore, another model was proposed, which at least partly resolved this issue.

C. State transition methods

A VW-grammar is a *synchronous model* in the sense that all substitutions resulting from applying grammar rules are simultaneously. As a generic mechanism for defining a formal language with context dependences, it assumes that first terminals and attributes of the language model are defined and them the respective language grammar is designed.

An affix grammar can be regarded as both *synchronous, and* generic model too. But this model may be enforced through consideration of the distinction between ascendant and descendant attributes which helps to determine how to build the parsing tree of a program. On the other hand, this additional consideration means *a restriction either for the grammar or for the parsing algorithm*, or for both.

In practice, one should distinguish between implementer's problems and user's problems. The implementer must read and check existing programs, so another model (called diachronic) is more adequate for this activity. The diachronic model means that the program is considered in its development in time. In the program, new words are created first (defined, declared, etc.), later on these words get used. Sometimes they get new attributes, which are valid in a limited scope. In majority of the programming languages, words and attributes must be created before they are applied.

CDL, as an implementer-oriented realization of the affix grammar technique, assumes the diachronic nature of its subject languages. Sometimes this limits its applicability. For example, it is not applicable to languages, where declarations of identifiers can appear everywhere in the program.

Ledgard proposed another approach known as VDM (Vienna Definition Method) was given by H.F. Ledgard, later applied for definition of static semantics of PL/I [[16]]. This method was refined by Williams [[17]] and by Farkas [[19]] in various directions.

In VDM attributes are not included in the context-free grammar but rather are enclosed into tables or similar structures. The table state changes gradually, while advancing through the program text. Each syntactic unit; i.e., each substitution rule in the context-free grammar of the language being considered is connected to *a state transition function*. When recognition of the next syntactic unit is completed, 'the associated state transition function is called and the associated state transition is performed. Upon completion of the overall parsing, it must be checked, whether the table is in a legal final state.

In this automaton model the diachronic nature of the subject language is thoroughly utilized, and parsing goes on from beginning to the end of text

D. Example of the diachronic approach.

In this example there are two variables 'MODE' and 'NAME', and a table with the name 'TABLE'.

```
Program: Begin, Declaration, Statement Train, End.
Declaration: Declare;
                 Declare, Declaration.
Declare: Declarer /'MODE':=MODE/, idlist.

Idlist: Identifier /"NAME" := Identifier,
'TABLE':='TABLE' + ('NAME', 'MODE')/,Semicolon;
Identifier / 'NAME' := Identifier,
'TABLE'  :=  'TABLE'  +  (NAME'  +  'MODE')/,  Comma,
Idlist.
     ……etc.
```

Checking, whether every declaration is unique, is done with the help of function 'UNIQUE' that defined by a program, similarly to functions in the substitution rules.

This method is realized through a state table and a table handling function. In contrast to the former approaches, in this method state transition functions are connected to terminal symbols only. Then the question arises, how to implement transitions related to higher-level syntactic units (e.g. blocks, etc.). The solution is very simple: each such higher-level unit has its first and last terminal symbols and the necessary actions relate to these ones. Therefore, the state transition function is defined in a separate table where three items are specified: the terminal symbol the lexical unit, and the state transition action. In the example below the prefix "D" indicates a defining occurrence of an identifier.

E. Another Example

```
Program: Begin, Declaration, Statement Train, End.
Declaration: Declare; Declare, Declaration.
Declare: Declarer, Idlist.
Idlist:  D.  Identifier,  Semicolon;  D.  Identifier,
Comma, Idlist.
etc.
```

An advantage of this method is that lexical rules, syntax, and static semantics are separated, and static semantics has no impact on the parsing method implementation.

III. The Attribute Technique for CFR grammar

Knuth introduced the notion of attribute [[8]] as a means to describe the semantics of languages generated by classical context-free grammars of Chomsky [0]. His attribute approach assumes construction of a parsing tree for source language statements in the given context-free grammar and then calculation of attribute values in each vertex of the tree according to rules derived from grammar rules used in parsing. This approach was implemented in a series of toolsets for compiler building, such as YACC [[7]], Eli [[5]], or CUP [[18]].

In 1970-ies, regularized versions of context-free grammars started to be used for defining the context-free language syntax, such as regular context-free (CFR)[3], regular Backus-Naur form (RBNF), and extended Backus-Naur form (EBNF) grammars, as well as their graphic representation – syntactic graph-schemes [[2], [3]]. Each non-terminal is matched to one component in such a diagram, which represents a rule of the respective CFR-grammar. The right-hand part of such rule is a generalized regular expression over the symbols of the united grammar alphabet [[3]], [[4]] [20] consisting of the alphabets of terminals, non-terminals, semantics names, and predicate names.

Techniques which employ CF-grammars with attributes (like affix grammars) usually transform the source grammar into an equivalent unambiguous grammar. In case of a CFR-grammar, the names of semantics and predicates are used in the grammar rules along with terminal symbols, and in syntactic charts they are placed on arcs which connect vertices marked with terminals and non-terminals.

The technique of building syntax state tables for a syntax-controlled language processor implemented in the SynGT tool [[3]] uses syntactic graph-schemes and translational CFR-grammars to define translation of source language statements into the target language.

Fig. 1 presents a syntactical graph-scheme for the non-terminal "statement" in the C language.

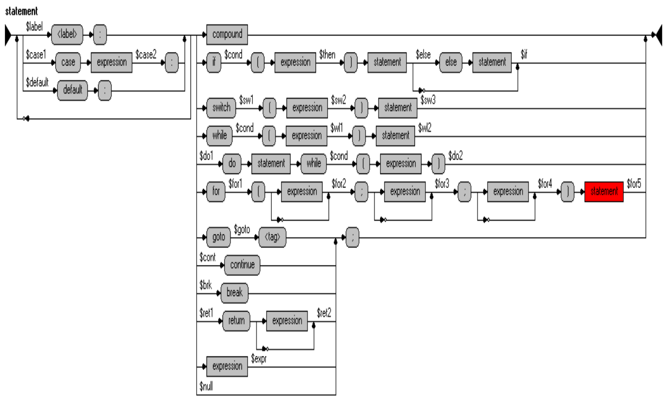

Fig. 1. Syntactical graph-scheme for the non-terminal statement in C

A. The Technique

By definition [0] *Translation* from the language L_1 into the language L_2 is defined as a relation $\tau \subseteq L_1 \times L_2$.

According to requirements for language descriptions formulated in [[12]], a translation specification should ensure automated synthesis of a translator from specifications, analysis of the input language properties, and visualization of the description of input language syntax and semantics.

Attractiveness of CF-grammars for language analysis and their simplicity (grammar rules have the form A→α where α is a string of symbols from the united alphabet of the grammar) for creating efficient parsers stimulated creation of a variety of classes and subclasses of grammars with various language constraints. The strongest language constraints (LL(k)), LR(k), LALR, SLR) allow to build parsers with linear complexity w.r.t. to the length of the source program.

Developed in SynGT system method of translation specification allows to define context-dependent languages. This is achieved due to predicates introduced to limit the choice among alternatives by context dependencies when parsing source statements. The context state is modified by semantic procedures introduced directly into the grammar rules or graph-schemes and processed along with terminal symbols. Grammar pre-processing translates occurrences of predicate and semantic names into invocations of respective predicated functions and semantic procedures, thus ensuring a computational support for the parser. These functions and procedures are parameterized implicitly through a common computational stack (or stacks) and attribute values considered as their shared global variables.

In [[8]] the semantics of a source statement are defined from its context-free syntax structure. In case of semantics parameterization, our technique uses attributes as classical Knuth attributes. Semantics of an input statement is defined by its context-dependent syntax structure, which means that parsing analysis and semantics computations are inseparable and interleave in time.

When attributes parameterize predicates, non-terminals parameterization with affixes is used, same as in CDL3 [[15]]. Thus, a CFR-grammar with attributes becomes a two-level one [[12]]. VW-grammars [[14]] are other representatives of this class of two-level grammars, along with Koster affix grammars.

As a rule, formal languages used in practice are often context-dependent. Two-level grammars are powerful enough to describe regular sets and to specify the syntax of respective languages within a unified formalism as well as to express context dependencies (i.e., static semantics).

As SynGT uses top-down syntax parsing, its major constraint on the class of grammars it can process is absence of left-recursive rules. I need equivalent transformations of grammar rules. An algorithm of grammar transformation, which eliminates any recursions in CFR-grammars, including the left one, is described in detail in [[2]], [[3]], [[20]], [21].

Introducing attributes on this parsing mechanism assumes a means of passing context-dependent data among procedures, which analyze various language constructs in the process of parsing the source language statements. The process develops along several parallel routes in the syntactic graph-scheme which represents the CFR-grammar of the language. Certain constraints to be observed with introduced attributes are imposed on the grammar in order to preserve the determinism of the parsing process.

An analogy may be drawn between this mechanism and analysis with the recursive descendant method in terms of a set of finite automata [[2]] cooperating in a common operational environment. Each rule of the CFR-grammar, which defines a non-terminal A, is mapped into a procedure A. The right-hand side of the rule is mapped into the body of the procedure A, which specifies checking conditions (context-free or context-dependent) regulating selection of the respective branch in parsing. A context-free condition consists in checking that the current input symbol coincides with the respective terminal symbol in the right-hand side of the rule for A. If this check succeeds, then a move forward in the input text is scheduled along with transition to the next position in the right-hand side of this rule. If a non-terminal B occurs in the current position of the rule for A then an invocation of a procedure which is an image of the rule for B is performed.

Context-dependent conditions are implemented by respective predicate functions; their invocations are denoted in grammar rules by names of semantic procedures of by respective labels on the graph-scheme arcs. Invocations of semantic procedures should be specified explicitly in the rules right-hand sides.

Implementation of back-tracking with the recursive descendant method is relatively complicated in a sequential environment. Back-tracking is used to cancel semantic actions because of detected violations of some context-free or context-dependent condition in the current branch of parsing. Therefore, the approach adopted in SynGT is equivalent to mapping several rules into one procedure, which ensures simultaneous mapping of several rules into one procedure. This leads to simultaneous building of several parallel variants of inference and rejecting inappropriate variants as the parsing progresses.

Developing further the analogy with procedures, one can match each rule with a set of input and output formal parameters and local variables. Elements of this set are formal attributes. In order to pass data to procedures, which correspond to grammar rules or implement semantics and predicates, actual attributes in form of actual parameters may be used.

Therefore, at the grammar level, attributes describe only the ordering, which the context data is passed in among elements of a particular translation algorithm. Their main purpose is to deliver data to semantic procedures and predicates through parameters of respective procedure calls.

Possible variants of analysis continuation being developed in parallel and being formed by lists of stacks, constraints imposed on CFR-grammars allow for ambiguous grammars, provided local semantic unambiguity is preserved taking into account the determinism of analysis. On the other hand, semantic actions along with context modifications and translation are performed in parallel with parsing of the source language statements. Sometimes a semantic action depends on data available only at completion of parsing the whole language statement. In this case, this action may be postponed till the phase of looking at the parallel stacks and semantic procedures invocations resulting from additional analysis. At this phase, the parser states are reviewed in the reverse order, and the postponed semantic procedures are invoked and more exact identifying of the syntax structure of the input chain through predicates.

The context of postponed actions should be restored to perform them. In other words, storing/restoring of context data should be ensured in such exceptional cases.

B. Informal Definition of Attributed Translation

Every occurrence of a non-terminal in the left-hand side of a rule is bound with a set of formal attributes, and each occurrence of a non-terminal, semantic procedure, or predicate in the right-hand side of rules a set of actual parameters is bound. Formal attributes may be inherited, synthesized, or local. Actual parameters may be inherited or synthesized.

The proposed definition of attributed translation is close the Early algorithm [5] used for analysis of arbitrary CF-grammars. The recursive descendant method and transformation of grammar rules into procedures with parameters is similar to the Koster's affix approach. The major differences from these methods are: a) taking into account the context conditions specified with predicates; b) executing semantic actions; c) specifying grammar rules with generalized regular expressions; and d) using attributes to handle data.

Constraints on the attributed specification of compilation are derived from the algorithm of the language processor work and from the definition of attributed translation; they are primarily reduced to absence of undefined values.

These requirements follow from general constraints of the class of CFR-grammars used in SynGT to obtain deterministic language processors of linear complexity.

Attributes, both formal and actual, are divided into direct pass attributes and postponed attributes; some local attributes may be defined as two-way attributes, which are computed at the direct pass and may be used in case of postponed computation (when the stacks are reviewed in the reverse order).

When the translation algorithm executes, the processor behaves as a deterministic finite automaton (DFA) on particular parts of the input text. In certain cases, the work of such DFA suspends and control is passed to some other DFA for parsing some sub-construct. Upon completion of processing the sub-construct, control is returned to the suspended DFA. Each DFA corresponds to one or several grammar rules. Values of formal attributes related to non-terminals in the left-hand sides of the rules, determine the local context of the DFA. Attribute values are considered to be computed when a DFA reaches its terminal state or the source text becomes exhausted. At this moment these attribute values should be stored in order to be restored later in case of postponed checks. The values are stored along with the numbers of states which control is returned to (the so called return states).

Control flow among DFAs at the postponed pass is consistent and occurs when a return state is accepted. At this moment, a new local context is formed from the values of postponed attributes and the values of respective two-way attributes are restored from the respective records.

Conclusion

Most of the programming languages have the feature that a set of words has no predetermined meaning. The meaning of these words is rather determined by the actual program. The rules regarding the consistent use of these words is called static semantics in this paper.

All 4 considered methods of syntax parsing are based on the idea that the necessary attributes of program elements are collected, registered and checked. There is a great difference between the methods, how correspondence between these elementss and attributes is established, represented, and handled. Anyway, each method assumes recursive functions to be used, which results in generating or accepting enumerable recursive sets.

Similarly, it was demonstrated that in each of the 4 methods three types of attributes are used (this fact was not mentioned explicitly in the literature): the literal, the integer and the pointer-type. The literal-type attribute denotes presence or absence of an attribute, the integer-type attribute has an integer value. The pointer-type attribute is the name of another word with attributes; in practice, it is usually implemented through pointers).

We can classify the definitional methods into two groups. In the first group, the methods are more suitable to create programs, we can call them user-oriented methods. These are the generative synchronous models. The other class of methods is more suitable for checking existing programs, which can be denoted as implementer-oriented methods.

For the first time the attribute technique was applied explicitly to regular expressions in context-free grammar rules when building a language processor, taking the following translation algorithm features into account: 1) attributes are computed in the process of building the inference tree, taking into account the context determined by these attributes; 2) attributes are used for context-dependent resolution of semantic ambiguity of translation. The obtained results further develop the technique of syntax-directed data processing with the SynGT tool, improving the development environment.. A prototype of the attributed system is planned to be developed on the .NET platform and derived from the SynGT code developed at SPCRAS.

References

[1] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] L. Fedorchenko, "On Regularization of Context-Free Grammars". *Izvestiya vuzov. Priborostroyeniye.* vol.49(11), 2006. pp. 50–54. (In Russian).

[3] L. Fedorchenko, *Regularization of Context-Free Grammars*. Saarbrucken: LAP LAMBERT Academic Publishing. 2011.

[4] S. Baranov, C. Lavarenne, "Open C Compiler in Forth", *in Proc EuroForth'95 conf*, 27–29 Oct. Schloss Dagstuhl. 1995.

[5] Robert W. Gray, Vincent P. Heuring, Steven P. Levi Anthony M. Sloane, William M. Waite "Eli: a complete, flexible compiler construction system" . Communications of ACM, Vol. 35, Num. 2, pp. 121–130, 1992.

[6] Jay Early. "An Efficient Context-Free Parsing Algorithm". Communications of the ACM, Vol. 13, Num. 2, pp. 94–102. 1970.

[7] S.C. Johnson Yacc yet another compiler compiler. Computer Science *Technical Report 32.* AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[8] D.E. Knuth "Semantics of context-free languages". Mathematical Systems Theory 2:2., pp. 127–145. 1968

[9] C.H.A. Koster *On the Construction of ALGOL-Procedures for Generating, Analyzing and Translating Sentences in Natural Languages.* Report MR72, Mathematisch Centrum, Amsterdam, 1965.

[10] C.H.A. Koster "Affix Grammars". *In: Proceedings of IFIP Conference on ALGOL 68 Implementation.* Munich, pp. 95–109. 1970.

[11] C.H.A. Koster "Affix Grammars for Programming Languages" *In: Attribute Grammars, Applications and Systems. International Summer School SAGA*. Prague. 1991.

[12] A. van Wijngaarden *A. Orthogonal Design and Description of a Formal Language.* Report MR76, Mathematisch Centrum, Amsterdam, 1965.

[13] B. J. Mailloux, J. E. L. Peck, C. H. A. Koster "Report on the algorithmic language ALGOL 68" Numerische Mathematik 14, pp. 79–218 1969) Springer-Verlag Berlin.

[14] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, C.H. Lindsey, M. Sintzoff, L.G.L.T. Meertens, R.G. Fisker "Revised report on the algorithmic language Algol 68"// ACTA Informatica 5 pp.1–236, 1974).

[15] C.H.A. Koster CDL3 manual Web:\\http://www.cs.ru.nl/~kees/vbcourse/ivbstuff/cdl3.pdf .

[16] H.F. Ledgard "Production system or can we do better than BNF?" CACM, pp.94–102. 1974/2

[17] V.H. Williams " Static semantics features of Algol 60, and BASIC". The Computer Journal. Vol. 21. No. 3. pp. 234–242.

[18] M. Griffiths "Relationship between definition and implementation of language". //Advanced courses on software engineering. Lecture Notes in Economics and Math Syst. Springer-Verlag 1973.

[19] Erno Farcas "Comparison of Some Methods for The Definition of Static Semantics", Hungarian Academy of Sciences / *Computational Linguistics and Computer Languages Vol. XII.* Hungary.

[20] Ludmila Fedorchenko and Sergey Baranov "Equivalent Transformations and Regularization in Context–Free Grammars". Bulgarian Academy of Sciences. *Cybernetics and Information Technologies CIT).* Vol. 14, No 4, pp.29–44. Sofia 2013/

[21] Lukichev A.S. Use of attributes in SYNTAX technology //Bulletin of St.Petersburg. university. Series 1. Issue 2. SPb. 2005. Pp. 64-73