PROLOG AND AUTOMATIC LANGUAGE IMPLEMENTATION SYSTEMS

Marjan MERNIK, Matej ČREPINŠEK
Institute of Computer Science, Faculty of Electrical Engineering and Computer Science, University of Maribor,
Smetanova 17, 2000 Maribor, Slovenia, tel. (+386 2) 220 7455,
E-mail: {marjan.mernik, matej.crepinsek}@uni-mb.si

SUMMARY

In the paper the use of Prolog, their benefits and drawbacks in automatic language implementation systems are presented. Various formal methods such as attribute grammars, operational semantics and denotational semantics are briefly described and implemented in Prolog. The advantages of Prolog basically stem from the use of unification and nondeterminism, and the price paid for the advantages are slower execution times. Various Prolog implementation of formal semantics method show that Prolog is viable tool for programming language development, design and prototyping.

Keywords: *programming language design and prototyping, logic programming, attribute grammars, operational semantics, denotational semantics, Prolog*

1. INDRODUCTION

Syntax and semantics of programming languages are usually described in natural languages. These descriptions are understandable and accessible to a wide variety of users. On the other hand, natural descriptions have many disadvantages such as: lack of clarity, ambiguities, and various interpretations. Therefore, a formal method is necessary which will describe syntax and semantics in a precise and unambigous manner. BNF is widely accepted for formal syntax description. Unfortunatelly in the area of semantics, the subject is more complicated because semantics is much more difficult to describe as syntax. Also, in semantics no standard method exist such as BNF. There are many approaches such as: axiomatic semantics, operational semantics, denotational semantics, and attribute grammars. In [34] was stated that formal methods must play a much more central role in language design. In such engineering approach to language design first formal specification is written. The next stage is to use formal specification to derive a prototype implementation. This implementation must be completed quickly, with little or no attention being paid to efficiency. But must allow to gain useful experience of writing and running programs in new language. This experience might well suggest improvements to the language design, requiring modifications to the specifications and prototype. In the last stage the efficient compiler must be implemented.

The effectiveness of Prolog as a language for rapid prototyping compilers and for developing scanner generators, parser generators and code generators has already been shown [4, 33]. However, these works covered in formal manner only lexical and syntax part of language definition. In this paper the idea is extended to formal semantic definition. Prototype implementations for various formal methods such as attribute grammars, operational semantics and denotational semantics are presented and implemented in Prolog.

The main goal of the paper is to show how different formal methods for programming language description can be implemented in Prolog achieving rapid language implementation. The organization of the paper is as follows. In section 2 attribute grammars and their Prolog implementation is briefly described. Relating logic programming and operational semantics is presented in section 3, followed by possible implementation of denotational semantics using logic programming in section 4. Related work is described in section 5. Finally, the conclusion is given in section 6.

2. ATTRIBUTE GRAMMARS

An attribute grammar [1, 6, 15, 21] is a context-free grammar $G = (N, T, S, P)$ augmented with attributes A and semantic rules R. An attributed tree for a program is a derivation tree where each node n, labelled by X, is attached with attribute instances that correspond to the attributes of X. Attribute evaluation is a process that computes values of attribute instances within an attributed tree according to the semantic rules R. The meaning of a program consists of the values of the synthesized attribute instances associated with the root node of the attributed tree. Semantic rules set up dependencies between attributes, which are foundation for several subclasses of attribute grammars (S-attributed, L-attributed, absolutely noncircular attribute grammars). An attribute grammar is S-attributed if it has only synthesized attributes. An attribute grammar is L-attributed if in each production $p \in P$ following condition holds: attribute occurrence $X_i.a$ can only depend on attribute occurrence $X_j.b$ and $j < i$. Therefore, an attribute grammar is L-attributed if the value of attribute occurrences associated to some grammars symbol X is computed from values of attribute occurrences associated to grammar symbols left from X. In absolutely noncircular attribute grammar attributes can depend on any attributes associated to the grammar symbols in this production, providing

non-circularity of augmented dependency graphs. For S-attributed and L-attributed grammars the values of attribute occurrences can be computed in one-pass, which is most commonly interleaved with syntax analysis, while in absolutely noncircular attribute grammars attribute values can be computed in several passes.

Semantics is with attribute grammars given in descriptive rather then algorithmic notation and therefore have many common features with logic programming. In the works [5, 13, 27] were shown that logic programming paradigm can additionally improve semantic expressiveness of attribute grammars. Relating attributes with logical variables, it is possibly to delay certain attribute evaluations. With this feature we can evaluate attributes of certain class of attribute grammars in one pass, while ordinary attribute grammars require multiple passes. An example of such attribute grammar is shown in Fig. 1 for simple calculation language DESK [27]. Attribute grammar for DESK language is absolutely noncircular, but not L-attributed, due to a right-to-left arc in local dependency graph for the first production (Fig. 2). The attribute evaluation can not be totally done during parsing. Name analysis and code generation for expression part have to be postponed until the constant definition part of the program has been processed. However, with logical programming paradigm concepts – unification and logic variable, the evaluation process can be done in one pass. Logic programming paradigm and attribute grammar formalism both share a common structural representation: the proof tree in logic programming and the attributed tree in attribute grammars. The relationship is based on the correspondence *nonterminal=predicate*. This correspondence has been adopted in a number of syntactic tools build on top of logic programming. The most well known logic grammar formalism is the definite clause grammars (DCGs) [28]. When considering the arguments as attribute values and the embedded Prolog code as semantic rules, DCGs match very closely with L-attributed grammars. On the basis of logic programming a new class of attribute grammars is established: logical one-pass attribute grammars, which is proper superset of L-attribute grammars. Relating attributes with logical variables, it is possibly to delay certain attribute evaluations. The idea is to allow the value of an attribute instance to be undefined as long as it is not needed in making control decisions in the attribute evaluation process. Attribute grammar in the Fig. 1 is logical-one-pass attribute grammar and therefore can be implemented in Prolog (Fig. 3).

productions	Semantic functions
PROG → **print** EXP CONSTPART	PROG.*code* = EXP.*code* + "PRINT 0" +"HALT 0" EXP.envi = CONSTPART.envs
$EXP_0 \rightarrow EXP_1$ + FACTOR	EXP_0.*code* = EXP_1.*code* + "`ADD` " + FACTOR.*value* EXP_1.envi = EXP_0.envi FACTOR.envi = EXP_0.envi
EXP → FACTOR	EXP.*code* = "`LOAD` " + FACTOR.*value* FACTOR.envi =EXP.envi
FACTOR → CONSTNAME	FACTOR.*value* = getvalue(CONSTNAME.*name*, FACTOR.*envi*)
FACTOR → NUMBER	FACTOR.*value* = NUMBER.*value*
CONSTNAME → ID	CONSTNAME.*name* = ID.*name*
CONSTPART→ ε	CONSPART.*envs* = ()
CONSPART→ **where** CONSTDEFLIST	CONSPART.*envs* = CONSTDEFLIST.*envs*
$CONSTDEFLIST_0 \rightarrow CONSTDEFLIST_1$, CONSTDEF	$CONSTDEFLIST_0$.*envs* = $CONSTDEFLIST_1$.*envs* + (CONSTDEF.*name*,CONSTDEF.*value*)
CONSTDEFLIST → CONSTDEF	CONSTDEFLIST.*envs* = (CONSTDEF.*name*, CONSTDEF.*value*)
CONSTDEF → CONSTNAME = NUMBER	CONSTDEF.*name* = CONSTNAME.*name* CONSTDEF.*value* = NUMBER.*value*

Fig. 1 Attribute grammar for DESK language

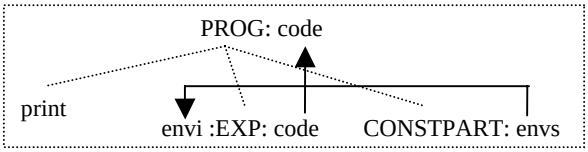

Fig. 2 Local dependency graph for the first production

```
program(Code) --> [print],
  exp(Code1, Env),
  constpart(Env),
  {append(Code1, [print, 0, halt, 0],
          Code)}.

exp(Code1,Env) --> factor(Env, Value),
    exp1(Code2, Env),
 {append([load, Value], Code2, Code1)}.

exp1(Code1, Env) -->
  [+], factor(Env, Value),
  exp1(Code2, Env),
  {append([add, Value], Code2, Code1)}.
exp1([], _) --> {null(epsilon)}.

factor(Env, Value) --> constname(Name),
    {getvalue(Name, Env, Value)}.
factor(_,X) --> [X],{integer(X)}.

constname(Name)--> [Name],{atom(Name)}.

constpart(Env) -->   [where],
    constdeflist(Env).
constpart([], _, _).   % epsilon

constdeflist([Name, Value|Env]) -->
  constdef(Name, Value),
  constrest(Env).

constrest(Env) --> [','],
                   constdeflist(Env).
constrest([]) --> {null(epsilon)}.

constdef(Name, X) --> constname(Name),
    [=],[X],{integer(X)}.

getvalue(Name, [Name,Value|_], Value).
getvalue(Name,[_, _|Env], Value) :-
  getvalue(Name, Env, Value).

null(epsilon).
```

Fig. 3 Prolog implementation of DESK attribute grammar

In this manner rapid language implementation is obtained and language can be tested by writing simple programs such as:

```
?- program(P, [print,2,+,3,+,5], []).
P = [load, 2, add, 3, add , 5, print,
0, halt, 0]

?- program(P, [print, x, +, y,
     where, x, =, 7, ',', y, =, 3],[]).
P = [load, 7, add, 3, print, 0, halt,0]

?- program(P, [print, 9, +, y,
   where, x, =, 7, ',' , y, =, 3], []).
P = [load,9, add, 3, print, 0, halt, 0]
```

One of the benefits of formal methods is also the possibility of automatically generating compiler or interpreter. Attribute grammars are very suitable for this task and many compiler-compiler systems exist such as FNC/2 [14] and LISA [22]. Some among them PANDA [8] and PROFIT [26] implements logical attribute grammars, which from attribute grammars automatically produce Prolog code.

3. OPERATIONAL SEMANTICS

In an operational semantics we are concerned with how to execute programs [11, 25]. We are interested with the relationship between initial and final state of execution. The notation $\langle S, s\rangle \rightarrow s'$ represents such relationship. The statement S will be executed in state s and terminated in final state s'. The main mathematical tool used in operational semantics is induction, where induction rules have following general form:

$$\frac{\langle S_1, p_1\rangle \rightarrow q_1' \ldots \langle S_n, p_n\rangle \rightarrow q_n'}{\langle S, p\rangle \rightarrow q'} \textbf{\textit{if}}\ \textit{condition},$$

where S is constructed from immediate constituents $S_1, \ldots, S_n$. A rule has a number of premises (written above solid lines) and one conclusion (written below the solid line). A rule may also have a number of conditions (written to the right of the solid line) that have to be fulfilled whenever the rule is applied. Rules with an empty set of premises are called axioms and inference rules that characterize semantic behavior of the language constructs. The logical framework of operational semantics is based on unification and nondeterminism and therefore can be related to logic programming. Rule in general form can be translated to the following Prolog statement

```
(s, P, Q') :- condition , !,
              (s1, P1, Q1')
                  ...
              (sn, Pn, Qn').
```

We have been implemented a simple operational semantics directed language implementation systems which made above transformation. In Fig. 4 an input (operational semantics for simple calculator language [29]) to our generator is presented, while in Fig. 5 an output (Prolog interpreter) is shown.

```
Bnf:
 P ::= on SE.
 SE ::= E total off | E total SE.
 E ::= E1 '+' E2 | E1 '-' E2 |
       if '(' E1 ',' E2 ',' E3 ')' |
       Number | lastanswer
```

```
Relations:
  =>A : Numeral -> Exp -> Numeral
  =>S : Numeral -> Seq_Exp -> Numeral*
  =>P : Program -> Numeral*

Semantics:
 Axioms:
  L |- lastanswer  =>A  L
  L |- Number      =>A  Number


 Rules:
L|-E1=>A V1, L|-E2=>A V2, sum(V1,V2,V)
            L|-E1 '+' E2  =>A V

L|-E1=>A V1, L|-E2=>A V2, dif(V1,V2,V)
            L|-E1 '-' E2  =>A V

positive(E1), L|- E2  =>A  V
  L|- if '(' E1 ',' E2 ',' E3 ')'=>A V

          L|- E3  =>A  V
  L|- if '(' E1 ',' E2 ',' E3 ')'=>A V

          L|- E  =>A  V
       L|- E total off =>S [V]

     L|- E  =>A  V, V |- SE =>S  VS
       L|- E total SE =>S [V|VS]

          0|- SE  =>S  S
            on SE =>P S
```

Fig. 4 Input to operational semantics directed generator

```
a([lastanswer], L, L).
a([Number], L, Number).
a([E1, '+', E2], L, V) :-
  a(E1,L,V1), a(E2,L,V2), sum(V1,V2,V).
a([E1, '-', E2], L, V) :-
  a(E1,L,V1), a(E2,L,V2),dif(V1,V2,V).
a([if,'(',E1,',',E2,',',E3,')'],L,V) :-
  positive(E1),  a(E2, L, V).
a([if,'(',E1,',',E2,',',E3,')'],L,V) :-
  a(E3, L, V).

s([E, total, off], L, [V]) :-
  a(E, L, V).
s([E, total, SE], L, [V|VS]) :-
  a(E, L, V), s(SE, V, VS).

p([on, SE], S) :- s(SE, 0, S).
```

Fig. 5 Automatically generated Prolog interpreter

Again, rapid language implementation is obtained and language can be tested by writing simple programs such as:

```
?- p([on,  [[11], total,
    [[lastanswer], total, off]]], K).
K = [11, 11]

?- p([on,[[10], total,
     [[if, [lastanswer], [10], [2]],
       total, off]]], K).
K = [10, 2]
```

4. DENOTATIONAL SEMANTICS

In denotational semantics [29], abstract syntactic constructs of the defined language are denoted by mathematical objects. The denotation is usually a semantic function which models the meaning of the constructs. Semantic functions map the language constructs into various semantic domains. The main part of a denotational definition of a language consists of a set of semantic equations which define the semantic functions. These are typically expressed in terms of λ-calculus. In spite of very simple syntax λ-calculus is strong enough to describe all mechanically computable functions and can be viewed as a very simple programming language. Many languages (imperative, functional, process functional [16, 18, 19]) have been design and prototyped with denotational semantics, proving that denotational semantics is an excellent tool for programming language design.

In order to be denotational semantics executable first task was construction of lambda machine interpreter in Prolog (Fig. 6). Following λ-expressions have been taking into account:

- Lambda abstractions ($\lambda x.E$): a anonymous function is defined with formal parameter x and body E. This is converted into Prolog data structure `lambda(X, E)`.
- Lambda application ($f\ x$): the function f is applied to argument x. This is represented in Prolog as `apply(F, X)`.
- Let expressions ($let\ x=E_1\ in\ E_2$): let expression is only syntactic sugar for $(\lambda x.E_2)E_1$. In Prolog this is represented as `let(X, E1, E2)`.
- Conditional expression ($if\ E_1\ then\ E_2\ else\ E_3$): expression E_1 is evaluated to boolean value; if it is true then the value of conditional expression is E_2; if it is false then the returned value is E_3. The Prolog representation is `cond(E1,E2, E3)`.
- Fix-point combinator ($fix\ F$): where F is a functional of the form $\lambda f.E$ and f is a function appearing in the body of E. The application of the fix-point combinator simulates a recursive application of function. It is represented in Prolog as `fix(F, E)`.
- Primitive functions: there are also a number of standard arithmetic and relational functions with usual semantic interpretation and are defined straightforwardly in Prolog.

```
lambda_machine(neg(E), X) :-
    lambda_machine(E, V1),
    neg(V1, X).
lambda_machine(eq(E1, E2), X) :-
    lambda_machine(E1, V1),
    lambda_machine(E2, V2),
    eq(V1, V2, X).
lambda_machine(add(E1, E2),X) :-
    lambda_machine(E1, V1),
    lambda_machine(E2, V2),
    X is V1 + V2.
lambda_machine(sub(E1, E2), X) :-
```

```
    lambda_machine(E1, V1),
    lambda_machine(E2, V2),
    X is V1 - V2.
lambda_machine(mul(E1, E2), X) :-
    lambda_machine(E1, V1),
    lambda_machine(E2, V2),
    X is V1 * V2.
lambda_machine(div(E1, E2), X) :-
    lambda_machine(E1, V1),
    lambda_machine(E2, V2),
    X is V1 / V2.
/* applicative evaluation order */
lambda_machine(
 apply(apply(E1,E2),E3), X) :-
   lambda_machine(apply(E1, E2), X1),
   lambda_machine(apply(X1, E3), X).
lambda_machine(
 apply(lambda(X1, E1), E2), X) :-
   lambda_machine(E2, X1),
   lambda_machine(E1, X).
lambda_machine(
 apply(fix(X, E1), E2), Y) :-
   fresh_copy(fix(X, E1), fix(XX, EE)),
   X = fix(XX, EE),
   lambda_machine(apply(E1, E2), Y).
lambda_machine(let(X, E1, E2), Y) :-
   lambda_machine(
      apply(lambda(X,E2), E1), Y).
lambda_machine(cond(E1, E2, _), X) :-
   lambda_machine(E1, X1),
   X1 == true, !,
   lambda_machine(E2, X).
lambda_machine(cond(_, _, E3), X) :-
   lambda_machine(E3, X).
/* reduction is not possible */
lambda_machine(X, X).

/* fresh copy */
fresh_copy(X, Y) :-
  fresh_copy1(X, Y, [], _).
fresh_copy1(X, Y, S, S) :-
  var(X), exist(X, S, Y).
fresh_copy1(X, Y, S, [X, Y|S]) :-
  var(X).
fresh_copy1(X, X, S, S) :- atom(X).
fresh_copy1(X, X, S, S) :- integer(X).
fresh_copy1([], [], _, _).
fresh_copy1([X|Xs],[Y|Ys], S1, S2) :-
  fresh_copy1(X, Y, S1, S3),
  fresh_copy1(Xs, Ys, S3, S2).
fresh_copy1(X, Y, S, S1) :-
  X =.. T,
  fresh_copy1(T, Z, S, S1),
  Y =.. Z.

exist(X,[Z, Y|_], Y) :- X == Z.
exist(X, [_, _|Rest], Y) :-
  exist(X, Rest, Y).

eq(V1, V2, true) :- V1 == V2, !.
eq(_, _, false).

neg(true, false).
neg(false, true).
```

Fig. 6 Lambda machine interpreter in Prolog

With such lambda machine interpreter we are ready to interpret any lambda expression, such as:

```
% (λx.x+1) 6
?-lambda_machine(
    apply(lambda(X, add(X, 1)), 6), R).
X = 6,
R = 7

% (λx.λy.x + y) 5 1
?-lambda_machine(
    apply(apply(lambda(X,
     lambda(Y, add(X, Y))), 5), 1), R).
X = 5,
Y = 1,
R = 6

% factorial function
% fix F = (λf.λx.if x = 0 then 1
%                else x * f(x - 1)
?- lambda_machine(
    apply(fix(F, lambda(X,
     cond(eq(X, 0), 1,mul(X,
      apply(F, sub(X, 1)))))), 3), R).
X = 3,
R = 6
```

In transformation from denotational semantics to Prolog each semantic equation of the form

$$\boldsymbol{f}\ [\![\ldots q \ldots r \ldots]\!] = \lambda arg.\ \ldots \boldsymbol{f'}\ [\![q]\!]\ \ldots \boldsymbol{f''}\ [\![r]\!]\ \ldots$$

is written as following Prolog statement:

```
f(Phrase, Arg, Lambda_exp) :-
   ...
   f'(Subphrase_q, Arg, Lambda_exp_q),
   ...
   f''(Subphrase_r, Arg, Lambda_exp_r),
   ...
   .
```

Finally, lambda expression is then interpreted on lambda machine to produce the meaning of phrase. In Fig. 7 the Prolog implementation of the following denotational semantics for arithmetic expressions of While language is presented [25]:

Abstract syntax
 n ∈ Num
 x ∈ Var
 a ∈ Aexp

 a ::= n | x | a1 + a2 | a1 * a2 | a1 - a2

Semantic domains
 Integer = {... -3, -2, -1, 0, 1, 2, 3 ...}
 Truth-Value = {true, false}
 State = Var → Integer

Semantic valuation functions
 $\boldsymbol{N}$: Num → Integer
 $\boldsymbol{A}$: Aexp → State → Integer
 $\boldsymbol{A}\ [\![n]\!] = \lambda s.\ \boldsymbol{N}\ [\![n]\!]$
 $\boldsymbol{A}\ [\![x]\!] = \lambda s.\ s\ x$
 $\boldsymbol{A}\ [\![a1 + a2]\!] = \lambda s.\ \boldsymbol{A}\ [\![a1]\!]\ s + \boldsymbol{A}\ [\![a2]\!]\ s$
 $\boldsymbol{A}\ [\![a1 * a2]\!] = \lambda s.\ \boldsymbol{A}\ [\![a1]\!]\ s * \boldsymbol{A}\ [\![a2]\!]\ s$

$\boldsymbol{A}$ ⟦a1 - a2⟧ = λs. $\boldsymbol{A}$ ⟦a1⟧ s - $\boldsymbol{A}$ ⟦a2⟧ s

```
/*  A : Aexp -> State -> Integer  */
a(X,State,apply(lambda(_,X),State)) :-
  integer(X).
a(X,State,apply(lambda(_,Val),State):-
  atom(X),
  lookup(X, State, Val).
a(+(A1,A2),State,add(A1_den,A2_den)) :-
  a(A1, State, A1_den),
  a(A2, State, A2_den).
a(*(A1,A2),State,mull(A1_den,A2_den)):-
  a(A1, State, A1_den),
  a(A2, State, A2_den).
a(-(A1,A2),State,sub(A1_den,A2_den)) :-
  a(A1, State, A1_den),
  a(A2, State, A2_den).

lookup(X,[X,Y|_],Y).
lookup(X,[_,_|Rest],Y) :-
  lookup(X, Rest, Y).

evaluate(Program, State, Integer) :-
  a(Program, State, Lambda_code),
  lambda_machine(Lambda_code,Integer).
```

Fig. 7 Prolog interpreter of While language

Again, rapid language implementation is obtained and language can be tested by writing simple programs such as:

```
?- a(+(7, 9),[],R).
R = add(apply(lambda(_288,7),[]),
             apply(lambda(_318,9),[]))

?- evaluate(+(7, 9), [], R).
R = 16

?- evaluate(+(7, i), [I, 1], R).
R = 8
```

5. RELATED WORK

In the engineering approach to language design [34] from formal specifications a prototype language implementation is derived, which allow language designer to gain useful experience with the language. This is a basis for further improvements to the language design, requiring modifications to the specifications and prototype. This approach, which is similar to approach described in [17], is also very suitable for domain-specific languages [7, 23, 30] - languages for solving problems in a particular domain, since domain-specific languages change more frequently [24]. In [9, 10] a constraint logic programming-based framework for specification, efficient implementation, and automatic verification of domain specific languages have been presented. Their framework is based on using Horn logic, and eventually constraints, to specify denotational semantics of domain specific languages. More efficient implementations of domain-specific langauge can be automatically derived via partial evaluation. Additionally, the executable specification can be used for automatic or semi-automatic verification of programs written in the domain-specific language. This work is further extended in [31, 32] where logical framework for automatically generating domain-specific language infrastructure is described. Domain-specific language infrastructure (interpreter, compiler, debugger, profiler, etc) can be rapidly develop using logical framework. Latter is a particular example of the approach described in [12].

In [20] a Prolog-based approach to the development of language processors such as: preprocessors, frontends, evaluators, tools for software modification and analysis have been presented. Their tool *Laptob* is an experimental framework for language design, language processing and program transformation using Prolog.

Using the same logical framework various formal methods for programming language description can be integrated. In [2] the Minotaur system has been described. Minotaur is a generic interactive environment based on the integration of the Centaur system [3] and the FNC-2 system [14]. It is shown how attribute grammars techniques can be adequate for evaluation of a quite large subclass of natural semantics, which are special kind of operational semantics. Further possible integration of different formal methods for programming language description based on the same logical framework is left to future work.

6. CONCLUSION

In the paper we show that Prolog is viable tool for programming language development, design and prototyping. Various formal methods such as attribute grammars, operational semantics and denotational semantics were implemented in Prolog. Semantics is with attribute grammars given in descriptive rather than algorithmic notation and therefore have many common features with logic programming. Logic programming paradigm can additionally improve semantic expressiveness of attribute grammars. Relating attributes with logical variables, it is possibly to delay certain attribute evaluations. With this feature we can evaluate attributes of certain class of attribute grammars in one pass, while ordinary attribute grammars require multiple passes. In addition to attribute grammars, the logic programming paradigm can be related with other semantic formalisms. In the paper also transformation from operational semantics and denotational semantics to Prolog programs are shown. Operational semantics use axioms and inference rules that characterize semantic behavior of the language constructs. Operational semantics can be easily implemented in Prolog, because logical framework of operational semantics is based on unification and nondeterminism. In denotational semantics we used semantic valuation functions which maps syntactic constructs into mathematical objects such as numbers, cartesian products,

functions, etc. Semantic valuation functions are written in lambda notation and can be executed with the lambda machine. Thus, a lambda machine interpreter is constructed in Prolog to allow execution of semantic functions.

REFERENCES

[1] H. Alblas, B. Melichar (Eds). Attribute Grammars, Applicatons and Systems, LNCS, Vol. 545, 1991.

[2] I. Attalli, D. Parigot. Integrating Natural Semantics and Attribute Grammars: The Minotaur System. TR No. 2339, INRIA, 1994.

[3] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Khan, B. Lang, V. Pascual. CENTAUR: The System. In Proceedings of SIGSOFT'88, ACM Sigplan Notices, Vol. 24, No. 2, pp. 14 – 24, 1989.

[4] J. Cohen, T. J. Hickey. Parsing and compiling Using Prolog. ACM Transactions on Programming Languages and Systems, Vol. 9, No. 2, pp. 125 – 163, 1987.

[5] P. Deransart, M. Maluszynski. A Grammatical view of Logic Programming. Proceedings of the International Workshop on Programming Languages Implementation and Logic Programming, PLILP'88, pp. 219 – 251, 1988.

[6] P. Deransart, M. Jourdan (Eds). Attribute Grammars and their Applicatons, LNCS Vol. 461, 1990.

[7] A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography. ACM Sigplan Notices, Vol. 35, No. 6, pp. 26 – 36, 2000.

[8] A. Feng, Y. Sugiyama, M. Fuji, K. Torii. Generating Practical Prolog Programs from Attribute Grammars. In Proceedings IEEE COMPSAC'87, pp. 605 – 612, 1987.

[9] G. Gupta. Horn Logic Denotations and Their Applications. The Logic Programming Paradigm: A 25 years Perspective. Springer, LNAI, pp. 127 – 160, 1999.

[10] G. Gupta, E. Pontelli. Specification, Implementation, and Verification of Domain-Specific Languages: A Logic Programming-Based Approach. LNCS, Vol. 2407, pp. 211 – 239, 2002.

[11] J. Hannan. Operational Semantics – Directed Compilers and Machine Architectures. ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4, pp. 1215 – 1247, 1994.

[12] J. Heering, P. Klint. Semantics of Programming Languages: A Tool-Oriented Approach. ACM Sigplan Notices, Vol. 35, No. 3, pp. 39 – 48, 2000.

[13] P. Henriques. A semantic evaluator generating system in Prolog. Proceedings of the International Workshop on Programming Languages Implementation and Logic Programming, PLILP'88, pp. 201 – 218, 1988.

[14] M. Jourdan, D. Parigot. Internals and externals of the FNC-2 attribute grammar system. In Attribute Grammars, Applications and Systems, LNCS, Vol. 545, pp. 485 – 506, 1991.

[15] D. E. Knuth. Semantics of context-free languages. Math. Syst. Theory, Vol. 2, No. 2, pp. 127 – 145, 1968.

[16] J. Kollar. Process Functional Programming. Proc. ISM'99, Rožnov pod Radhoštem, Czech Republic, April 27-29, pp. 41 – 48, 1999.

[17] J. Kollar, V. Novitzka. From requirements specification to design specification. Journal of Information, Control and Management Systems, Vol. 1, No. 2, pp. 55 – 64, 2003.

[18] J. Kollar. The Conception and Application of PFL: A Process Functional Programming Language. Problemy programmirovanija, No. 1, pp. 5 – 23, 2004.

[19] J. Kollar, V. Novitzka. Semantical Equivalence of Process Functional and Imperative Programs. Acta Polytechnica Hungarica, Vol. 1, No. 2, pp. 113 – 124, 2004.

[20] R. Laemmel, G. Riedewald. Prological Language Processing. First Workshop on Language Descriptions, Tools and Applications, LDTA'01, pp. 117 – 141, 2001.

[21] M. Mernik, D. Parigot (Eds). Attribute Grammars and Their Applications. Informatica, Vol. 24, No, 3, Special issue, 2000.

[22] M. Mernik, M. Lenič, E. Avdičaušević, V. Žumer. LISA: An Interactive Environment for Programming Language Development. 11th International Conference on Compiler Construction, CC'02, LNCS, vol. 2304, pp. 1 – 4, 2002.

[23] M. Mernik, J. Heering, T. Sloane. When and How to Develop Domain-Specific Languages. CWI Technical Report SEN-E0309, 2003.

[24] M. Mernik, V. Žumer. Incremental Programming Language Development. Computer Languages, Systems and Structures, No. 31, pp. 1 – 16, 2005.

[25] H. R. Nielson, F. Nielson. Semantics with applications. John Wiley & Sons, 1992.

[26] J. Paakki. PROFIT: A System integrating logic programming and attribute grammars. Proceedings of the International Workshop on Programming Languages Implementation and Logic Programming, PLILP'91, pp. 243 – 254, 1991.

[27] J. Paakki. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, Vol. 27, No. 2, pp. 196 – 255, 1995.

[28] F. Pereira, D. Warren. Definite Clause Grammars for Language Analysis – A Survey

of the Formalism and a Comparison with Augmented Transition Networks. Artificial Intelligence, Vol. 13, No. 3, pp. 231 – 278, 1980.

[29] D. A. Schmidt. Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, 1986.

[30] D. Spinellis. Notable design patterns for domain-specific languages. The Journal of Systems and Software, No. 56, pp. 91 – 99, 2001.

[31] Q. Wang, G. Gupta. Rapidly Prototyping Implementation Infrastructure of Domain-Specific Languages: A Semantic-based Approach. ACM Symposium on Applied Computing, SAC'05, To appear, 2005.

[32] Q. Wang, G. Gupta. Towards Provably Correct Code Generation via Horn Logical Continuation Semantics. In Proceedings of the International Symposium on Practical Aspects of Declarative Languages, PADL'05, To appear, 2005.

[33] D. Warren. Logic Programming and compiler writing. Software Practice and Experience, Vol. 10, No. 2, pp. 97 – 125, 1980.

[34] D. A. Watt. Programming Language Syntax and Semantics. Prentice Hall, 1991.

BIOGRAPHY

Marjan Mernik received his M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently an associate professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He was a visiting professor in the Department of Computer and Information Sciences at the University of Alabama at Birmingham in 2004. His research interests include principles, paradigms, design and implementation of programming languages, compilers, formal methods for programming language description and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Matej Črepinšek received the BSc degree in computer science at the University of Maribor, Slovenia in 1999. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science. His research for PhD degree is concerned with grammatical inference. His research interest in computer science include also grammar-based systems, programming languages and evolutionary algorithms.