Proceedings of MASPLAS'04
Mid-Atlantic Student Workshop on Programming Languages and Systems
Seton Hall University, April 3, 2004

Retrieving Critical Information By Code Analysis

A. Louise Perkins[1], Sumanth Yenduri[2], Jim Corbin[3], Avichal Mehra[4], S. S. Iyengar[5]

1, 2, 5 – Department of Computer Science, Louisiana State University
3, 4 – Distributed Marine Environment Forecast System, Mississippi State University

ABSTRACT

We present an analyzer tool that accepts as input a string of programming language statements and outputs possible meanings for those program statements. The Graphical User Interface allows us to see which meanings were proposed for different program statements. The analyzer is packaged as a software tool, which we refer to as the Semantic Extraction tool (SET). SET, through a GUI utilizes a multi-box parsing approach, in combination with a background debugger, infers the meaning of code. The current tool is in its infancy but this proof-of-concept is offered to demonstrate the feasibility of our approach. Our demonstration goal is to identify an arithmetic assignment statement and to derive critical information that might be useful for different kinds of code analysis. We describe the basic underlying principles and the implementation of our tool in this paper.

Keywords: Code analysis, characteristic forms, semantics

1.MOTIVATION

There have been different kinds of program analysis tools developed so far. Most of them are widely acknowledged to be incomplete [4,5]. Particularly the semantic modules need more attention than the syntactic modules. Code analysis is vital for different aspects such as maintenance, testing, error analysis etc. The information extracted by SET could be used for the same. Monolithic codes blur the referencing environment. Vital statistics about the code helps a user understand the code more efficiently. SET was designed as part of the DMEFS project funded by the US Navy and developed at the Mississippi State University. Dr. Aravind Joshi at Pennsylvania State University has pursued work on structured production rules for complex parsers that go beyond the string-based LR-type production rules. Our tool, constructed independently from that work is based on an equivalent theoretical basis. We use characteristic forms analogous to their structured productions [2].

2. INTRODUCTION

The existing software tools that support code development address management & maintenance, time related state inquiry of a program instance, metrics to quantify and some kinds of error analysis [6,9]. Attribute Grammars [1,10] partially address this need. Otherwise the semantic element tends to exist as hand-written routines that insert some semantic information. Actions such as inserting default conditions, completing identifier descriptions and definitions, and converting syntactic structures into semantic structures. Presently no semantic tools are available to assist in comprehending the functionality of the code [7,8].

Knuth [1] developed an Attribute Grammar (AG) in 1968 to couple both the syntax and the semantics of a program. An Attribute Grammar consists of a context independent grammar extended by a set of rules that specify the semantics of the text being analyzed. It associates a function with each node in the parse tree of a given program to bind the semantic content of the node with its syntax. Semantic functions are the rules that specify the computation of the attribute values. The grammar symbols of the productions are used to evaluate these functions. They also have Predicate functions that describe the syntax and semantics of the programming language to some extent. These functions are Boolean expressions over the attribute set of production rules. Sets of semantic functions together with a set of predicate functions are associated with each grammar rule. Every grammar symbol has a set of attributes that consist of synthesized attributes and inherited attributes. Synthesized attributes pass information up a parse tree while inherited attributes move down the tree. For each parse tree node, synthesized attributes depend on the values of their child nodes as well as their inherited attributes.

Formalizing grammars specify domains, on which different dependencies like syntactic, semantic etc can be specified. These kinds of domains are called "domain of locality"[2]. A variety of features of such formalism follow these domains of locality. Dr. Joshi's primary work is based on Tree Adjoint Grammars or TAGs and Lexicalized TAGs because of their linguistic nature and formal relevance. TAGs generate trees rather than strings. Lexicalized grammars methodically associate each elementary structure with a lexical anchor. In each structure a lexical item is identified. It is not adding feature structures and unification equations to the rules of the formalism. The resultant structures specify extended domains of locality over which constraints can be stated. SET employs a bottom up approach unlike them. This allows us to get around the problem of ambiguity to some extent. Even if they backtrack to correct their mistakes it still has the cost overhead to do so.

Our tool SET extracts vital information from different programming language constructs and stores them. This information can be viewed and analyzed for various purposes such as testing, error analysis, measuring degree of concurrency in the program etc. We have successfully derived the information for an assignment statement for the languages C and FORTRAN but can easily extend the tool to operate on any language. In this paper we explain the underlying principles of our tool. For this we need to extract critical information. We discuss the working of the logical analyzer used by SET and put forward a few examples. Our final demonstration goal though is to unambiguously identify if an arithmetic assignment statement is an average, or a discrete finite difference approximation of a derivative.

3. SEMANTIC ANALYSIS

To implement our tool, SET, we proceed as follows. For each *a priori* defined semantic entity that we seek to identify, we require a characteristic form. Both forms provide structured or non-linear derivation rules. We try to parse a code segment into all of the existing characteristic

forms, and when one or more matches occur, we assume that the semantic meaning associated with that characteristic form is a possible meaning for the associated code segment. If more than one characteristic form works, we report all possible meanings. If the user is kind enough to respond, via the GUI, which one was correct, we update our statistics allowing us to build a dynamic context for how often a multiple selection of those forms leads to a specific meaning. Statistics can be kept over a program, over a project, or over the universe of programs, as needed.

Rule based Artificial Intelligence could be used to achieve a comparable result. However, when we view this as a vertical multi-box compiler with semantic content; use lead to an equivalent theory based on structured diagram.

Example 1: Average

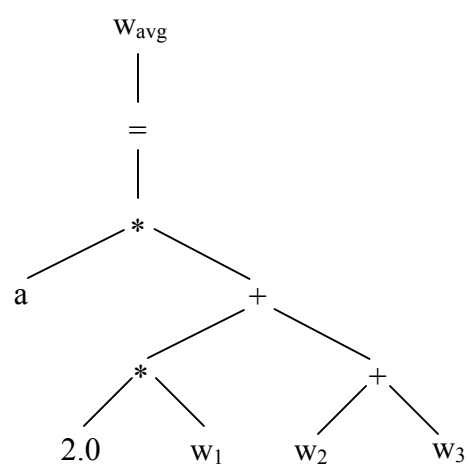
We first define the required characteristic forms and then we use these characteristic forms to see if the coded statement can be associated with a possible meaning. For example a programmer wishes to compute the following average

$$w_{avg} = \tfrac{1}{2}\, w_1 + \tfrac{1}{4}\, w_2 + \tfrac{1}{4}\, w_3$$

which has been coded as:

$$a = \tfrac{1}{4}$$
$$w_{avg} = a * (2.0 * w_1 + w_2 + w_3)$$

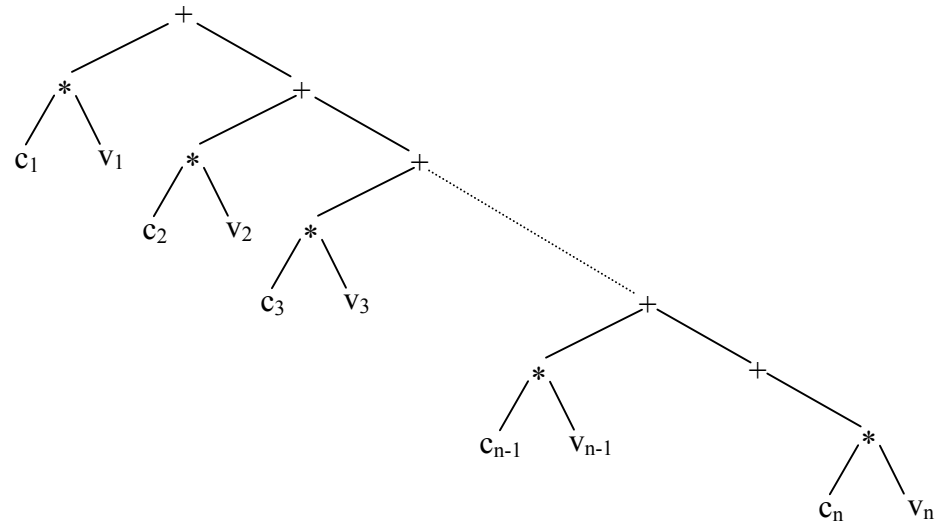
A compiler might generate the following tree to help it translate the function into another language.



We introduce the characteristic average form

$$avg = c_1v_1 + c_2v_2 + \ldots\ldots\ldots + c_nv_n$$

which has been associated with its own characteristic evaluation tree, (the syntactical component).

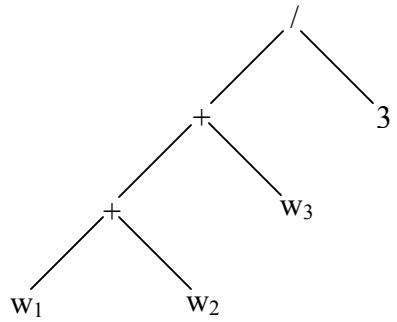


Subject to the constraint that $\sum c_i \equiv 1$. (the semantic component) Note that this is distinct from attribute grammars so our correlation occurs with a parse tree, not a node. If we can encode the program segment given in first tree into the form of the second tree and satisfy all constraints on the characteristic form, then we say that this line of code may be interpreted as an average (in some way) and SET will report an average as one possible meaning.

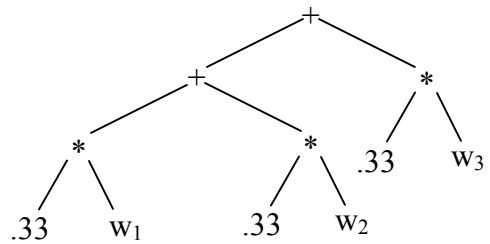
To accomplish this task there are 2 major parts of the SET tool, the form translator and the constraint evaluator. We can think of the form translator to normalize the expression. At this step we change all divides to multiples and all subtractions as to additions. For example,

$$avg = (w_1 + w_2 + w_3)/3$$

Has an expression tree as



When normalized this becomes



As we can always normalize expressions with operators (+,-,÷,*) like this our normal form is ubiquitous. Then when any of our constraints are met we derive semantic meaning. In our example, the constraint for average is easy to pick out. All coefficients were written to the left of

the multiplication symbol, so that we can easily retrieve them. However, in general, the coefficients may be variable names and not constants. We therefore maintain a symbol table for the code being analyzed, and store values into it whenever we have one explicitly assigned. Table 1, we expand our example code to show the corresponding code part of the symbol table. We concurrently execute a debugger in the background with stop points at the code segment interest. The debugger is queried for current values (which may vary for each line through the code, but if it satisfies a constraint the first line through we pre emptorarily conclude the constraint was satisfied).

Weight = 1.93
$w_1 = 5$
$w_2 = 2$
$w_3 = \lambda$
avg = weight * ((w_1+ w_2+ w_3))
Code segment 1

Variable Name	Variable Value
w_1	5
w_2	2
w_3	λ
weight	.33

Table 1

For code segment 1, our characteristic form is normalized as

avg = (weight * w_1+ weight*w_2+ weight*w_3)

And our concurrent semantic constraint is that 3 * weight ≈ 1.0. (Allowing for machine precision tolerances)
We next consider the average encoded as follows,

subroutine avg(w, w_1, w_2)
avg =w * w_1 + w* w_2
Code Segment 2

where we may not have local (may be separately compiled) access to "w" as it. We again rely on the background debugger, place a stop at the expression in question, and query the dynamic debugger for the values needed to decide if the constraints are satisfied.

Example 2: First Order Derivatives

To identify first derivatives, we couple another characteristic form together with the associated numerical consistency constraints. We define an average value spatial difference operator with undetermined coefficients as

Ex 1:

$$\overline{\Delta}_x(U_m^n) = (\frac{1}{v}\overline{\Delta}^{[+]} + \frac{v-1}{v}\overline{\Delta}_x^{[-]})$$

Ex 2:

$$\overline{\Delta}_x^{[+]}(U_m^n) = \frac{\overline{U}_{m+}^n - U_m^n}{\Delta x} \quad \text{and} \quad \overline{\Delta}_x^{[-]}(U_m^n) = \frac{U_m^n - \overline{U}_{m-}^n}{\Delta x}$$

Ex 3:

$$\overline{U}_{m+}^n = \sum_{j=0}^{N_+} \Psi_j^{[+]} U_{m+j}^n \text{ and } \overline{U}_{m-}^n = \sum_{j=0}^{N-} \Psi_j^{[-]} U_{m-j}^n$$

Here $\overline{U}_{m+}^n$ and $\overline{U}_{m-}^n$ are average values of *U(x,t)* located at the right and left of point *m*. For additional information on the use of the generalized form see Corbin and Perkins, 1997 [3]. Here the $\psi_j^{[\pm]}$, are the coefficients to be determined. The factor *v* weights the approximation to the left or right side of point m. Our general form is thus

Ex 4:

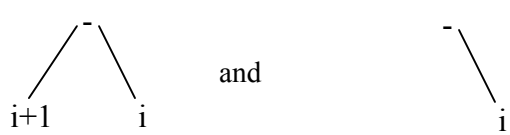
$$[\frac{1}{v}\Delta^+(u_i) + \frac{v-1}{v}\Delta^-(u_i)]/\Delta x$$

This form allows for non-symmetric Upwind FDM's, (higher order derivatives may classified based on the number of factors in the denominator). The offsets are typically expressed, relative to an index, as

$$(u(i+1) - u(i))/\Delta x$$

Although FORTRAN 90/95 departs from this style, it is easy to add a different syntax to account for such variations.

For our second example, we first build 2 separate trees, T^+ and T^-,



subject to the constraints given in Ex 2 and Ex3. We have extended our characteristic forms for higher order derivatives too.

4. LOGICAL ANALYZER USED BY SET

The logical analyzer used by SET reads both FORTRAN and C programs as input, and stores variable names, together with any values assigned, into a symbol table, which is realized in a DBMS. Initially it deletes all the comments, other white space, and handles line continuations ie., it cleans up older FORTRAN codes. It locates assignment statements by matching sequences of tokens (which are generated by flex) into predefined patterns. It keeps track of the current column numbers, interprets string and character escapes, converts integers and floating point numbers and actions specified in the rules section that are applicable during the lexical analysis phase. Finally, it generates an SQL script, which consists of creating a table and all the necessary insert statements. The table is generated with a unique name (user can specify his own table names) with all the fields. This SQL script when run in an SQL server or any database package would produce a database table with all the table entries.

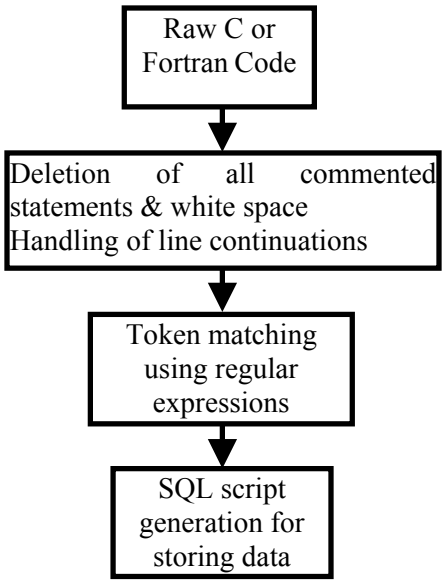


Fig1. Working of Logical Analyzer used by SET

Down below is an example SQL script generated by SET for an input C program. It contains the necessary statements to create the table and store the required information as well. Each of the values in the insert statement represent the file name, line number, variable to which the value is assigned, data type and the value.

INPUT C Code Segment

```
.
.
.
.
if(speed==0)
   use_pot();
else
   sumanth();

a = i++;
b= (j*k+l)/2;
/*
a=b;
*/
        int i=0;
        int j,k;

        i = 1;
        i = i+2;

        j = k+2+i&j/3+k;

        k = (int) j+3;

        *j = ++j;
```

```
        i = i++;
        i++;

        k = -j;
        a =c;

port_b = POSITIONS[  stepper_state ];

a="sumanth;;yenduri";
p='yenduri';
q='yend duri';
"aa=kjfdskjf";
.
.
.
```

OUTPUT SQL Script

```
# creating table Logan_19700926_0937
create table Logan_19700926_0937
(
  FileName varchar (100),
  LineNumber int,
  VariableName varchar (100),
  DataType int,
  AssignedValue varchar (255)
);
# creating table entries for aa.c
insert into Logan_19700926_0937 values ('aa.c', 6,'a', -1, ' i++');
insert into Logan_19700926_0937 values ('aa.c', 26,'a', -1, 'c');
insert into Logan_19700926_0937 values ('aa.c', 30,'a', -1, '"sumanth;;yenduri"');
insert into Logan_19700926_0937 values ('aa.c', 7,'b', -1, ' (j*k+l)/2');
insert into Logan_19700926_0937 values ('aa.c', 11,'i', -1, '0');
insert into Logan_19700926_0937 values ('aa.c', 14,'i', -1, ' 1');
insert into Logan_19700926_0937 values ('aa.c', 15,'i', -1, ' i+2');
insert into Logan_19700926_0937 values ('aa.c', 22,'i', -1, ' i++');
insert into Logan_19700926_0937 values ('aa.c', 17,'j', -1, ' k+2+i&j/3+k');
insert into Logan_19700926_0937 values ('aa.c', 19,'k', -1, ' (int) j+3');
insert into Logan_19700926_0937 values ('aa.c', 25,'k', -1, ' -j');
insert into Logan_19700926_0937 values ('aa.c', 21,'*j', -1, ' ++j');
insert into Logan_19700926_0937 values ('aa.c', 28,'port_b', -1, ' POSITIONS[
stepper_state ]');
insert into Logan_19700926_0937 values ('aa.c', 31,'p', -1, ''yenduri'');
insert into Logan_19700926_0937 values ('aa.c', 32,'q', -1, ''yend duri'');
```

5. CONCLUSIONS

We have demonstrated how to derive possible semantic meaning from PL statements. We use parsing ideas to attempt to conform a PL statement to a character expression. If this conformation is possible, the semantic meaning is one sense of the expression. We also discussed our presentation of the tool via a meta code GUI that reduces the value of textual output found in many existing analysis tools. This on-demand meta-code interface is user friendly. We have demonstrated this capability by construction of a tool that implements the characteristic forms. We have demonstrated that tool for both FORTRAN and C programs. Semantic meaning with context sensitive characteristic forms (that can vary for each context) provide a probabilistic semantic meaning for ambiguous code segments that allows infer semantic meaning with reasonable confidence even in the face of semantic ambiguity.

6. ACKNOWLEDGEMENTS

We built SET upon the foundations of the work of Dr. A. Louise Perkins and Joseph Dixon. It was designed as part of the Distributed Marine Environment Forecast System (DMEFS) project funded by the United States Navy and developed under the guidance of the CAST research arm.

7. REFERENCES

[1] Knuth, D.E., Semantics of Context-free Languages, *Mathematical Systems Theory, 2*(2), 1968, 127-145.
[2] Aravind K. Joshi, Patrick Paroubek & Yves Schabes, XTAG- A graphical workbench for developing tree-adjoining grammars, *Proceedings of the Third Applied Natural Language Processing (ANLP) Conference*, Trento, Italy, April, 1992.
[3] P.A Orlin, A.L. Perkins, and G. Hilburn, "A Frequency Accurate Spatial Derivative Finite Difference Approximation", Numerical Methods in Partial Differnential Equations, 13: 1-12, 1997.
[4] Jan Heering, Paul Klint, Semantics of Programming Languages: A Tool-Oriented Approach, *ACM SIGPLAN Notices, Volume 35*(3), March 2000, 39-48
[5] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools* (Boston, MA: Addison-Wesley Pub Co, 1986)
[6] C. Gunter, *Semantics of programming languages: structures and techniques* (Cambridge, MA: MIT Press, 1992)
[7] H.R. Nielson and F. Nielson, *Semantics with applications: A formal introduction* (Wiley: John Wiley & Sons, 1992)
[8] G. Winskel, *The formal semantics of programming languages: an introduction* (Cambridge, MA: MIT Press1993)
[9] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation (Boston, MA: Addison Wesley Publishing Co, 1979)
[10] J. Penjam, Synthesis of a Semantic Processor from an Attribute Grammar, *Prog. and Computer Software, 9*(1), January 1983, 29-39.