4-1-1993

Dynamic semantic specification by two-level grammars for a block structured language with subroutine parameters.

Mehdi Badii

SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT
Number 60, April 1993

Dynamic Semantic Specification by Two-Level Grammars for a Block Structured Language with Subroutine Parameters

Mehdi Badii

Department of Computer Science
Pace University
Pleasantville, NY 10570

Fatemeh Abdollahzadeh

Department of Computer Science
Central Connecticut State University
New Britain, CT 06056

PACE UNIVERSITY

This manuscript is to appear in
a forthcoming issue of
ACM SIG PLAN NOTICES

Mehdi Badii is an Associate Professor of Computer Science at Pace University. He earned his doctorate in computer science in England at the Loughborough University of Technology. He is an active researcher in the areas of the detection of parallelism, the formal description of programming languages, and compiler design. Several papers have resulted from his work in these areas.

Fatemeh Abdollahzadeh is an Associate Professor of Computer Science at the Central Connecticut State University. Holding a doctorate in computer science from the Loughborough University of Technology, she investigates and contributes papers in the area of grammatical means for detecting parallelism as well as in the area of compiler design.

Dynamic Semantic Specification by Two-Level Grammars for a Block Structured Language with Subroutine Parameters

M. Badii
Department Computer Science
Pace University

F. Abdollahzadeh
Department Computer Science
Central Connecticut State University

Abstract: This paper presents the application of a formal description language --Van Wijngaarden Two-Level grammar -- to define the dynamic semantics of a block structured language consisting of several types of parameter passing mechanisms. We will show, in a fairly understandable, concise and clear fashion, that the metalinguistic formalism of Two-Level grammar is a fine descriptive device for this purpose.

1. INTRODUCTION

Although in functional programming languages the use of higher order functions are customary, passing subroutines as parameters in imperative languages has not been received in the past several years. In this paper we shall introduce a precise and English-like language formalism to define the dynamic semantics of a blocked structure language including parametrized-subroutine. It is our objective to use this feature and make a bridge between imperative and functional programming languages.

We introduce a new formalism for the control stack. This work is different from Velazques [1] in that all the operations and the details of the run time stack are precisely and formally defined. Our approach for formalizing the dynamic semantics of subroutine parameter is similar to the idea used by Birtwistle and Loose [2], where a formal subroutine parameter points to its corresponding actual parameter. The formalism is a more understandable English-like language.

In this system the stack is a number of substrings, each one representing an activation record. An activation record is also partitioned to a number of substrings to represent a variable location , a parameter location, a location for a subroutine name, a location for the returned value in case of a function, a record number to make the activation record unique among the others, or a number to represent the static link of the record. In this formalism there is no need for a dynamic link or a return address -- the recursive nature of Two-Level Grammar is powerful enough to resolve these two phenomena. We believe a simple underlying formalism makes a language more precise and understandable. The formalism is also essential for program correctness and verification.

This paper is divided into five sections. Section 2 describes and defines the Two-Level grammar. Section 3 presents the abstract program, and section 4 covers our approach in handling the control stack and subroutine call. Section 5 contains the conclusion.

2. Two-Level Grammar (TLG)

The concept of Two-Level grammar (van Wijngaarden grammar, W-grammar) became widely known as a powerful formalism in the revised report on Algol 68 [3], where a single Two-Level grammar was used to define all aspects of the syntax, including the context dependent conditions (static semantics) of the language. It is shown that TLG is equivalent to computable function [6] and therefore it is as powerful as Turing machines, Markov algorithms and recursive functions. This makes the application of TLG for program verifications and correctness possible. The class of TLGs have efficiently been parsed by an LL(1)-based algorithm [4] using the concurrent programming language Occam, which makes this class more practical. The formal system TLG is used as practical means to define axiomatic semantics [5], and as a programming language [7,8,9].

This paper presents a TLG to define the complete dynamic semantics of a Pascal-like language, incorporating integer variables, procedures, functions, and several parameter-passing mechanisms known as call-by-value, call-by-reference, call-by-procedure, and call by-function. The statements of the language are assignment statements, subroutine calls and conditional statements. The static semantics and other features such as structured type definitions and other statements are formally defined in [10] by TLG.

A TLG consists of two finite sets of rules, the metaproductions (the production rules of a first context free grammar), and the hyperrules (the model productions). By replacing the strings derived from metaproductions in Hyper rules, a set of production rules of a second context free grammar is obtained. This set is capable of describing the syntax and the semantics of a language. Formally, similar to [11, 12], a TLG is defined as a 4-tuple:

$$W = (X,\ T,\ (M,\ Q),\ (H,\ R,\ S)),\ \text{where}$$

X is a finite alphabet called the **orthovocabulary,**
T is a finite alphabet of terminals,
Q is a finite set of context-free productions called **metaproductions,** and M is the set of their nonterminals called **metanotions.** The pair (M, Q) is called **meta-level,**
H is a set of **hypernotions** and is a subset of $((M * I) \cup X)^+$, where I is a set of nonnegative integers, R is a set of **hyperrules** which is a subset of $H * (M \cup X)^*$. The triple (H, R, S) is called the **hyper-level,** $S \in H$, is the start hypernotion.

Alternatives in a metaproduction are separated by a semicolon. Different nonterminals in the same alternative are separated by spaces. A double colon is used to separate the left and right sides of a metaproduction. A semicolon separates two alternatives in the same hyperrule and different hypernotions in the same alternative are separated by a comma. A colon is used to separate the left and right sides of a hyperrule.

Tables I and II show the metaproductions and hyperrules defined for this language. In these Tables:

X is the set of lower case letters, +, -, *, /, +, <>, <, <=, >, >=, and 1. These characters contribute phrases such as "and", "has block 1", or "exit" in Table II.
T is a singleton set consisted of empty string ϵ.
M is the set of nonterminals of productions of Table I
Q is the set of all productions in Table I.
H is the set of all phrases in Table II, separated by a colon, a comma, and a semicolon.
R is the set of model productions in Table II.
S is the phrase "BLOCKS and STMTSETY has block 1 STMTSETY1 exit execute main block " on the left side of H01, in Table II.
The symbol @n, where n is an integer, means that the corresponding hypernotion appears on the left side of the hyperrule numbered Hn.

In making the second context free productions, all occurrences of the same metanotion in a hyperrule must be replaced by the same sentence derived from the metanotion. A valid program is defined to have a parse tree derived from the second context free grammar. The parse tree of this grammar consists of the **semantic** subtrees with the leaves as empty string ϵ to impose the dynamic semantics of the program.

Table I Metaproductions for the TLG Definition of the Language

M01	ALPHA	:: A; B; C;...;Z; a; b; c;...z.
M02	AMOUNT	:: undefined; NUMERAL.
M03	ARG	:: EXPRESSION.
M04	ARGS	:: ARG ARGSETY.
M05	ARGSETY	:: ARGS; EMPTY.
M06	ARITOPERATOR	:: +; -; *; /.
M07	BLOCK	:: the ONES th PARSETY VARSETY SUBROUTINSETY nested in ONSETY block.
M08	BLOCKS	:: BLOCK BLOCKSETY.
M09	BLOCKSETY	:: BLOCKS; EMPTY.
M10	BOOL	:: false; true.
M11	CALL	:: call NAME with ARGSETY endcall
M12	CHAR	:: ALPHA; ARITOPERATOR ; RELOPERATOR.
M13	CHARS	:: CHAR CHARSETY.
M14	CHARSETY	:: CHARS; EMPTY.
M15	EMPTY	::.
M16	EXPRESSION	:: OPERAND EXPTAILETY
M17	EXPTAILETY	:: ARITOPERATOR OPERAND; EMPTY.
M18	FUNCLOC	:: funcloc NAME with AMOUNT endfuncloc.
M19	FUNCLOCETY	:: FUNCLOC; EMPTY.
M20	FUNPROC	:: function; procedure.

Table I (Continued)

```
M21 LOC              :: FUNCLOC; PARLOC; VARLOC; SUBLOC .
M22 LOCS             :: LOC LOCSETY.
M23 LOCSETY          :: LOCS; EMPTY.
M24 NAME             :: letter ALPHA.
M25 NUMERAL          :: 0; ONES.
M26 NUMERALETY       :: NUMERAL; EMPTY.
M27 NAMETY           :: NAME; EMPTY.
M28 ONES             :: 1 ONESETY.
M29 ONESETY          :: ONES; EMPTY.
M30 OPERAND          :: NUMERAL; NAME; CALL.
M31 PAR              :: VALUEPAR; REFPAR; SUBPAR.
M32 PARLOC           :: REFPARLOC; SUBPARLOC.
M33 PARS             :: PAR PARSETY.
M34 PARSETY          :: PARS; EMPTY.
M35 RECORD           :: record ONES th LOCSETY link to ONSETY recordend.
M36 RECORDS          :: RECORD RECORDSETY.
M37 RECORDSETY       :: RECORDS; EMPTY.
M38 REFPAR           :: refpar NAME endrefpar.
M39 REFPARLOC        :: ref NAME points to NAME in ONES record endref.
M40 RELOPERATOR      :: =; <>; <; <=; >; >=.
M41 STACK            :: RECORDSETY.
M42 STMT             :: NAMETY has ONES block; exit; CALL;
                        assign NAME by EXPRESSION endassign ;
                        if OPERAND RELOPERATOR OPERAND then STMT else
                        STMT endif;
                        begin STMTSETY end;
                        function NAME return EXPRESSION endfunction.
M43 STMTS            :: STMT STMTSETY.
M44 STMTSETY         :: STMTS; EMPTY.
M45 SUBLOC           :: FUNPROC NAME has block ONES endsubroutine.
M46 SUBPAR           :: subpar NAME PARSETY endsub.
M47 SUBPARLOC        :: sub NAME points to NAME in ONES record endsub.
M48 SUBROUTINE       :: FUNPROC NAME has block ONES endsubroutine.
M49 SUBROUTINS       :: SUBROUTIN SUBROUTINSETY.
M50 SUBROUTINSETY    :: SUBROUTINS; EMPTY.
M51 VALUEPAR         :: valuepar NAME endvaluepar.
M52 VAR              :: var NAME endvar.
M53 VARLOC           :: var NAME with AMOUNT endvar.
M54 VARS             :: VAR VARSETY.
M55 VARSETY          :: VARS; EMPTY.
```

Table II Hyperrules for the TLG Definition of the Language

H01 BLOCKS and STMTSETY has block 1 STMTSETY$_1$ exit execute main block:
where STACK is record 1 th link to recordend,
BLOCKS and STMTSETY allocate from 1 th block STACK becomes
STACK$_1$, @2
BLOCKS and STMTSETY execute statements STMTSETY$_1$ STACK$_1$ becomes
STACK$_2$. @16

H02 BLOCKSETY the ONES th PARSETY VARSETY SUBROUTINSETY nested in
ONESETY block BLOCKSETY$_1$ and STMTSETY allocate from ONES th block
ARGSETY STACK becomes STACK$_1$:
BLOCKSETY the ONES th PARSETY VARSETY SUBROUTINSETY in ONESETY
block BLOCKSETY$_1$ and STMTSETY allocate PARSETY and ARGSETY make
parameter locations STACK becomes STACK$_2$, @3
VARSETY make variable locations STACK$_2$ becomes RECORDSETY record
ONES$_1$ th LOCSETY link to ONESETY recordend , @26
where STACK$_1$ is RECORDSETY record ONES$_1$ th LOCSETY SUBROUTINSETY
link to ONESETY recordend.

H03 BLOCKS and STMTSETY allocate PARSETY and ARGSETY make parameter
locations STACK becomes STACK$_1$:
BLOCKS and STMTSETY allocate PARSETY and ARGSETY make single par
location STACK becomes STACK$_1$; @4
where PARSETY ARGSETY is EMPTY,
where STACK$_1$ is STACK.

H04 BLOCKS and STMTSETY allocate PAR PARSETY and ARG ARGSETY make
single par location STACK becomes STACK$_1$:
BLOCKS and STMTSETY allocate PAR and ARG location for formal
parameter STACK becomes STACK$_2$, @5
BLOCKS allocate PARSETY and ARGSETY make parameter locations
STACK$_2$
becomes STACK$_1$. @3

H05 BLOCKS and STMTSETY allocate PAR and ARG location for formal
parameter STACK becomes STACK$_1$:
where PAR is valuepar NAME endvaluepar,
BLOCKS and STMTSETY evaluate ARG to NUMERAL STACK becomes
RECORDSETY record ONES th LOCSETY link to ONESETY recordend, @6
where STACK$_1$ is RECORDSETY record ONES th LOCSETY var NAME with
NUMERAL endvar link to ONESETY recordend;
where PAR is refpar NAME endrefpar,
obtain ARG in ONES record of STACK, @25
where STACK is RECORDSETY record ONES th LOCSETY ref NAME points
to ARG in ONES record endref link to ONESETY recordend;
where PAR is subpar NAME PARSETY endsub,
obtain ARG in ONES record of STACK, @25
where STACK is RECORDSETY record ONES th LOCSETY sub NAME points
to NAME in ONES record endsub link to ONESETY recordend.

Table II (Continued)

H06 BLOCKS and STMTSETY evaluate OPERAND EXPTAILETY to NUMERAL STACK becomes STACK$_1$:
BLOCKS and STMTSETY evaluate binary expression OPERAND EXPTAILETY to NUMERAL STACK becomes STACK$_1$; @7
where EXPTAILETY is EMPTY,
BLOCKS and STMTSETY evaluate operand OPERAND to NUMERAL STACK becomes STACK$_1$. @8

H07 BLOCKS and STMTSETY evaluate binary expression OPERAND1 ARITOPERATOR OPERAND$_2$ to NUMERAL STACK becomes STACK$_1$:
BLOCKS and STMTSETY evaluate operand OPERAND$_1$ to NUMERAL$_1$ STACK becomes STACK$_2$, @8
BLOCKS and STMTSETY evaluate operand OPERAND$_2$ to NUMERAL$_2$ STACK$_2$ becomes STACK$_1$, @8
NUMERAL$_1$ ARITOPERATOR NUMERAL$_2$ equal to NUMERAL. @28, 29, 30, 33

H08 BLOCKS and STMTSETY evaluate operand OPERAND to NUMERAL STACK becomes STACK$_1$:
where OPERAND is NUMERAL;
find value of name OPERAND from STACK to be NUMERAL, @9
where STACK$_1$ is STACK;
BLOCKS and STMTSETY call the subroutine OPERAND to NUMERAL STACK becomes STACK$_1$. @12

H09 find value of name NAME from STACK to be NUMERAL :
from NAME find NAME$_1$ in ONES record of STACK, @10
where STACK is RECORDSETY record ONES th LOCSETY var NAME$_1$ with NUMERAL endvar LOCSETY$_1$ link to ONESTY recordend RECORDSETY$_1$.

H10 from NAME find NAME$_1$ in ONES record of RECORDSETY record ONES$_1$ th LOCSETY link to ONESETY recordend :
where LOCSETY contains var NAME with AMOUNT endvar,
where ONES is ONES$_1$,
where NAME$_1$ is NAME;
where LOCSETY contains ref NAME points to NAME$_2$ in ONES$_2$ record endref,
cut the RECORDSETY record ONES$_1$ th LOCSETY link to ONESETY recordend at ONES$_2$ record to STACK, @11
from NAME$_1$ find NAME$_2$ in ONES record of STACK; @10
cut the RECORDSETY record ONES$_1$ th LOCSETY link to ONESETY recordend at ONES$_2$ record to STACK, @11
from NAME find NAME$_1$ in ONES record of STACK. @10

H11 cut the RECORDSETY record ONES th LOCSETY link to ONESETY recordend RECORDSETY$_1$ at ONES record to RECORDSETY record ONES th LOCSETY link to ONESETY recordend : EMPTY.

Table II (Continued)

H12 BLOCKS and STMTSETY call subroutine call NAME with ARGSETY endcall to NUMERALETY STACK becomes $STACK_1$:
search STACK for NAME to be FUNPROC NAME1 has block ONES endsubroutine in $ONES_1$ record, @13
according to FUNPROC NAME1 make FUNCLOCETY, @14
where STACK is RECORDSETY record $ONES_2$ th LOCSETY link to ONESETY recordend,
where STMTSETY is $STMTSETY_1$ NAME1 has ONES block $STMTSETY_2$ exit $STMTSETY_3$,
BLOCKS and STMTSETY allocate from ONES th block RECORDSETY record $ONES_2$ th LOCSETY link to ONESETY recordend record 1 $ONES_2$ th FUNCLOCETY link to $ONES_1$ recordend becomes $STACK_2$, @2
BLOCKS and STMTSETY execute statements $STMTSETY_2$ $STACK_2$ becomes $RECORDSETY_1$ RECORD, @16
NAME could be function to return NUMERALETY in RECORD, @15
where STACK is RECORDSETY.

H13 search RECORDSETY record ONES th LOCSETY link to ONESETY recordend for NAME to be FUNPROC NAME1 has block $ONES_1$ endsubroutine in $ONES_2$ record :
where LOCSETY contains sub NAME points to NAME2 in ONES3 record endsub,
cut the RECORDSETY record ONES th LOCSETY link to ONESETY recordend at ONES3 record to STACK, @11
search STACK for NAME2 to be FUNPROC NAME1 has block $ONES_1$ endsubroutine in $ONES_2$ record; @13
where LOCSETY contains FUNPROC NAME has block $ONES_1$ endsubroutine,
where NAME1 is NAME,
where $ONES_2$ is ONES;
cut the RECORDSETY record ONES th LOCSETY link to ONESETY recordend at ONESETY record to STACK, @11
search STACK for NAME to be FUNPROC NAME1 has block $ONES_1$ endsubroutine in $ONES_2$ record. @13

H14 according to FUNPROC NAME make FUNCLOCETY :
where FUNPROC is function,
where FUNCLOCETY is funcloc NAME with undefined endfuncloc;
where FUNCLOCETY is EMPTY.

H15 NAME could be function to return NUMERALETY in RECORD :
where RECORD contains funcloc NAME with NUMERALETY;
where NUMERALETY is EMPTY.

H16 BLOCKS and STMTSETY execute statements STMTSETY1 STACK becomes $STACK_1$:
BLOCKS and STMTSETY execute all statement STMTSETY1 STACK becomes $STACK_1$; @17
where $STMTSETY_1$ is EMPTY,
where $STACK_1$ is STACK.

Table II (Continued)

H17 BLOCKS and STMTSETY execute all statements STMT $STMTSETY_1$ STACK becomes $STACK_1$:
BLOCKS and STMTSETY execute single statement STMT STACK becomes $STACK_2$, @18, 19, 21, 23, 24
BLOCKS and STMTSETY execute statements $STMTSETY_1$ $STACK_2$ becomes $STACK_1$; @16

H18 BLOCKS and STMTSETY execute single statement function NAME return EXPRESSION endfunction STACK becomes $STACK_1$:
BLOCKS and STMTSETY evaluate EXPRESSION to NUMERAL STACK becomes RECORDSETY record $ONES_1$ th funcloc NAME with AMOUNT endfunction LOCSETY link to ONESETY recordend , @6
where $STACK_1$ is RECORDSETY record $ONES_1$ th funcloc NAME with NUMERAL endfunction LOCSETY link to ONESETY recordend.

H19 BLOCKS and STMTSETY execute single statement assign NAME by EXPRESSION endassign STACK becomes $STACK_1$:
BLOCKS and STMTSETY evaluate EXPRESSION to NUMERAL STACK becomes $STACK_2$, @6
acquire NAME to be $NAME_1$ in ONES record of $STACK_2$, @20
where $STACK_2$ is RECORDSETY record $ONES_1$ th LOCSETY var $NAME_1$ with AMOUNT endvar $LOCSETY_1$ link to ONESETY recordend RECORDSETY,
where $STACK_1$ is RECORDSETY record $ONES_1$ th LOCSETY var $NAME_1$ with NUMERAL endvar $LOCSETY_1$ link to ONESETY recordend RECORDSETY.

H20 acquire NAME to be $NAME_1$ in ONES record of RECORDSETY record $ONES_1$ th LOCSETY link to ONESETY recordend :
where LOCSETY contains ref NAME points to $NAME_2$ in $ONES_2$ record endref,
cut the RECORDSETY record ONES1 th LOCSETY link to ONESETY recordend at $ONES_2$ record to STACK, @11
acquire $NAME_2$ to be NAME1 in ONES record of STACK; @20
where LOCSETY contains var NAME,
where $NAME_1$ is NAME,
where ONES is $ONES_1$;
cut the RECORDSETY record $ONES_1$ th LOCSETY link to ONESETY recordend at ONESETY record to STACK, @11
acquire NAME to be $NAME_1$ in ONES record of STACK; @20

H21 BLOCKS and STMTSETY execute single statement if $OPERAND_1$ RELOPERATOR $OPERAND_2$ then $STMT_1$ else $STMT_2$ endif STACK becomes $STACK_1$:
BLOCKS and STMTSETY evaluate operand $OPERAND_1$ to $NUMERAL_1$ STACK becomes $STACK_2$, @8
BLOCKS and STMTSETY evaluate operand $OPERAND_2$ to $NUMERAL_2$ $STACK_2$ becomes $STACK_3$, @8
$NUMERAL_1$ RELOPERATOR $NUMERAL_2$ equal to BOOL, @35-40
BLOCKS and STMTSETY based on BOOL execute $STMT_1$ else $STMT_2$ $STACK_3$ becomes STACK. @22

Table II (Continued)

H22 BLOCKS and STMTSETY based on BOOL execute $STMT_1$ else $STMT_2$ STACK becomes $STACK_1$:
where BOOL is true,
BLOCKS and STMTSETY execute single statement $STMT_1$ STACK becomes $STACK_1$; @18, 19, 21, 23, 24
BLOCKS and STMTSETY execute single statement $STMT_2$ STACK becomes $STACK_1$. @18, 19, 21, 23, 24

H23 BLOCKS and STMTSETY execute single statement begin STMTSETY end STACK becomes $STACK_1$:
BLOCKS and STMTSETY execute statements $STMTSETY_1$ STACK becomes $STACK_1$. @16

H24 BLOCKS and STMTSETY execute single statement call NAME with ARGSETY endcall STACK becomes $STACK_1$:
BLOCKS and STMTSETY call subroutine call NAME with ARGSETY endcall to NUMERALETY STACK becomes $STACK_1$. @12

H25 obtain NAME in ONES record of RECORDSETY record $ONES_1$ th LOCSETY link to ONESETY recordend :
where LOCSETY contains NAME,
where ONES is $ONES_1$;
cut the RECORDSETY record $ONES_1$ th LOCSETY link to ONESETY recordend at ONESETY record to STACK, @11
obtain NAME in ONES record of STACK. @25

H26 VARSETY make variable locations STACK becomes $STACK_1$:
VARSETY single variable location STACK becomes $STACK_1$; @27
where VARSETY is EMPTY,
where $STACK_1$ is STACK.

H27 var NAME endvar VARSETY single variable location RECORDSETY record $ONES_1$ th LOCSETY link to ONESETY recordend becomes RECORDSETY record $ONES_1$ th LOCSETY var NAME with undefined endvar link to ONESETY recordend :
VARSETY make variable locations RECORDSETY record $ONES_1$ th LOCSETY var NAME with undefined endvar link to ONESETY recordend becomes STACK. @26

H28 $NUMERAL_1$ + $NUMERAL_2$ equal to NUMERAL :
where $NUMERAL_1$ is 0,
where NUMERAL is $NUMERAL_2$;
where $NUMERAL_2$ is 0,
where NUMERAL is $NUMERAL_1$;
where NUMERAL is $NUMERAL_1$ $NUMERAL_2$.

H29 $NUMERAL_1$ - $NUMERAL_2$ equal to NUMERAL :
NUMERAL + $NUMERAL_2$ equal to $NUMERAL_1$. @28

Table II (Continued)

H30 $NUMERAL_1$ * $NUMERAL_2$ equal to NUMERAL :
where $NUMERAL_1$ is 0,
where NUMERAL is 0;
where $NUMERAL_2$ is 0,
where NUMERAL is 0;
multiply $NUMERAL_1$ by $NUMERAL_2$ to be NUMERAL. @31

H31 multiply 1 ONESETY by ONES to be ONES $ONESETY_1$:
multiply ONESETY by ONES to be $ONESETY_1$; @31
where ONESETY ONESETY1 is EMPTY,
where $ONESETY_1$ is ONESETY.

H32 $NUMERAL_1$ / $NUMERAL_2$ equal to NUMERAL :
$NUMERAL_1$ < $NUMERAL_2$ equal to true, @35
where NUMERAL is 0;
$NUMERAL_1$ div $NUMERAL_2$ to be NUMERAL. @33

H33 $ONES_1$ div $ONES_2$ to be ONESETY :
$ONES_1$ < $ONES_2$ equal true,
where ONESETY is EMPTY;
where $ONES_1$ is $ONES_2$,
where ONESETY is 1;
divide $ONES_1$ by $ONES_2$ to have ONESETY. @34

H34 divide $ONES_1$ by $ONES_2$ to have 1 ONESETY :
$ONES_1$ - $ONES_2$ equal to $ONES_3$, @29
$ONES_3$ div $ONES_2$ to be ONESETY. @33

H35 $NUMERAL_1$ < $NUMERAL_2$ equal to BOOL :
$NUMERAL_2$ - $NUMERAL_1$ equal to ONES, @29
where BOOL is true;
where BOOL is false.

H36 $NUMERAL_1$ <= $NUMERAL_2$ equal to BOOL :
$NUMERAL_2$ - $NUMERAL_1$ equal to 0, @29
where BOOL is true;
$NUMERAL_2$ < $NUMERAL_1$ equal to BOOL. @35

H37 $NUMERAL_1$ = $NUMERAL_2$ equal to BOOL :
where $NUMERAL_1$ is $NUMERAL_2$,
where BOOL is true;
where BOOL is false.

Table II (Continued)

H38 $NUMERAL_1$ <> $NUMERAL_2$ equal to BOOL :
where $NUMERAL_1$ is $NUMERAL_2$,
where BOOL is false;
where BOOL is true.

Table II (Continued)

H39 $NUMERAL_1$ > $NUMERAL_2$ equal to BOOL :
    $NUMERAL_2$ < $NUMERAL_1$ equal to BOOL. @35

H40 $NUMERAL_1$ >= $NUMERAL_2$ equal to BOOL :
    $NUMERAL_2$ <= $NUMERAL_1$ equal to BOOL. @36

H41 where CHARSETY is CHARSETY : EMPTY.

H42 where $CHARSETY_1$ CHARS $CHARSETY_2$ contains CHARS : EMPTY.

Let us look at the semantic subtree of a program that contains an operation "5 -5". Note that for simplicity we use non-negative integers in the language and represent them as unary numbers in our grammar. In order to show that "5 - 5 = 0" is a valid statement, following substitutions in the corresponding hyperrules are made to obtain the second context free productions.

	Productions	Substitutions
1)	11111 - 11111 equal to 0 : 0 + 11111 equal 11111.	drive 11111, 11111, and 0 from M25 and substitute them for $NUMERAL_1$, $NUMERAL_2$, and NUMERAL in H29 respectively.
2)	0 + 11111 equal to 11111 : where 0 is 0, where 11111 is 11111; where 11111 is 0, where 11111 is 0; where 11111 is 0 11111.	drive 0, 11111, and 11111 from M25 and substitute them for $NUMERAL_1$, $NUMERAL_2$, and NUMERAL in H28 respectively.
3)	where 0 is 0:.	drive 0 from M14 and substitute in H41.
4)	where 11111 is 11111.	drive 11111 from M14 and substitute in H41.

3. The Abstract Program

The following sample program serves as a source example to illustrate the Stack and Procedure invocation in the formalism described in Table I and II.

```
program T;
var
  R: integer;

function F(n: integer):integer;
begin
  if n = 1 then F := 1
  else F := n * F(n - 1)
end;

function S(function f(a:integer):integer; n:integer):integer;
begin
  if n = 0 then S := 0
  else S := f(n) + S(f, n - 1)
end;

begin
  R := S(F, 3)
end.
```

This program evaluates 1! + 2! + 3!.

The abstract program consists of two substrings derived from M08 (which represents the scope of the identifiers) and M42 (which represents the abstract statements). These two strings are generated when the static semantics of the program are inspected [10]. For the program, the first substring generated from BLOCKS is:

```
the 1 block
  var letter R endvar
  function letter F has block 11 endsubroutine
  function letter S has block 111 endsubroutine
nested in block
the 11 block
  valuepar letter n endvaluepar
nested in 1 block
the 111 block
  subpar letter f valuepar letter a endvaluepar endsub
  valuepar letter n endvaluepar
nested in 1 block
```

and the second substring generated from STMTS is:

```
letter F has block 11
if letter n = 1 then
  function letter F return 1 endfunction
else
  function letter F return letter n *
    call letter F with letter n - 1 endcall
  endfunction
endif
exit
letter S has 111 block
if letter n = 0 then
```

```
    function letter S return 0 endfunction
  else
    function letter S return call letter f with letter n endcall +
      call letter S with letter n - 1 endcall
    endfunction
  endif
  exit
  has block 1
  assign letter R by call letter S with letter F with 111 endcall
  endassign
  exit
```

The compound statement forming the body of each subroutine, corresponds to the following abstract statement:

$$\text{NAMETY has ONES block } STMT_1 \; STMT_2 ... STMT_n \text{ exit}$$

Since this string is a part of the abstract statements, the unique string "NAMETY has ONES block" and immediate "exit" determine the beginning and the end of the corresponding compound statement. NAMETY represents the name of the subroutine and is the an empty string for the main program. The metanotion ONES show the block number.

4. Formalizing the Stack and Procedure Invocation.

In order to define the dynamic semantics of a program, it is required to have a broad understanding of the "stack" STACK in M41. The Metanotion STACK can drive a sequence of records RECORD in M35 as:

$$RECORD_1 \; RECORD_2 ... RECORD_n$$

A RECORD holds local information required for either a subroutine call or the program. It is in the form of :

record ONES th LOCSETY link to ONESTY recordend.

where ONES and ONESETY define the "record number" and the "static link" respectively. Each "record location" LOC of LOCSETY introduces a location for a variable, a parameter, or a nested subroutine name in the program.
In the case of a function there is a LOC for its returned value.

Starting from H01, the state of the stack before the execution of the first statement of the program is the following string that must be replaced for $STACK_1$ in this rule :

```
record 1 th
  function letter F has block 11 endsubroutine
  function letter S has block 111 endsubroutine
  var letter R with undefined endvar
link to recordend
```

The above string shows that the first activation record has the name of the function F and its nested block number 2 (i.e. 11), the name of function S and its nested block number 3 (i.e. 111), and a variable R with its initial value as undefined.

When in the main program the function S(F, 3) is activated, in order to obtain the correct semantic tree, the metanotion $STACK_2$ in H12 must be replaced by the string:

```
record 1 th
  function letter F has block 11 endsubroutine
  function letter S has block 111 endsubroutine
  var letter R with undefined endvar
link to recordend.
record 11 th
  funcloc letter S with undefined endfuncloc
  sub letter f points to letter F in 1 record endsub
  var letter n with 111 endvar
link to 1 recordend
```

Here the second activation record is added to the stack. In this record, there is a substring representing a location for function S to return its value; a substring for formal function parameter f that must be replaced by actual parameter F (when f is activated); and a substring for the value parameter n which receives the value 3 (i.e. 111). The substring "link to 1 recordend" describes that the second record is linked to the first one. Notice that there is no difference between value parameters and local variables of a subroutine.

When the function f(n) is activated in the body of procedure S, the metanotion $STACK_2$ in H12 must be replaced by the string:

```
record 1 th
  function letter F has block 11 endsubroutine
  function letter S has block 111 endsubroutine
  var letter R with undefined endvar
link to recordend.
record 11 th
  funcloc letter S with undefined endfuncloc
  sub letter f points to letter F in 1 record endsub
  var letter n with 111 endvar
link to 1 recordend
record 111 th
  funcloc letter F with undefined endfuncloc
  var letter n with 111 endvar
link to 1 recordend
```

The first hypernotion on the right side of H12 in Table II refers to H13. This hyperrule describes how to find the actual parameter F, its record number (i.e. 1 for ONES), and its block number (i.e. 11 for $ONES_1$). The second hypernotion on the right side of H12 uses H14 and makes a substring, representing a location for function F to return its value. The fifth hypernotion on the right side of H12 which appears on the left side of H02 makes a substring for the value parameter n which receives the value 3 (i.e. 111). The substring "link to 1 recordend" describes that the third record is also linked to the first one.

According to H21 and H18, when the conditional statement in function F is executed, the string "funcloc letter F with undefined endfuncloc" in this record is changed to "funcloc letter F with 111111 endfuncloc" to describe that the returned value F is 6 (i.e. 111111).

If this process is continued and the complete semantic tree is derived then $STACK_2$ in H01 has been replaced by the string:

```
record 1 th
  function letter F has block 11 endsubroutine
  function letter S has block 111 endsubroutine
  var letter R with 111111111 endvar
link to recordend.
```

This shows that the value of the variable R is 9.

5. CONCLUSION

The Two-Level Grammar system is based on the familiar notion of context free grammar. Therefore, the mechanism of the system can be easily and quickly understood. Since a hypernotion is written as a pseudo-English sentence, a semantic action can be understood from the content of its corresponding Hypernotion.

In this paper a Two-Level Grammar is defined to describe the dynamic semantics of a blocked structure programming language containing call by value, reference, procedure, and function parameter, with assignment, conditional, and subroutine call statements. The semantics are defined through the process of driving a semantic parse tree for a given program. The precise formal definition of each statement in the program corresponds to a subtree of this tree. The formal stack in this system is a string of characters derived from rule M41. Different strings derived from this rule reflect the changes in the stack and appear in the interior nodes of the semantic tree. Upon the formal description of the last statement in the program, we have a complete semantic tree with leaves as empty strings. The formalism does not give explicit indication of the errors in a program. Any error stops the process of constructing the semantic tree, leaving no error message. However, it is possible to add error indications as the last alternative of most of the Hyper rules.

References

[1] Velazques Ituribide, Formalization of Control Stack, Sigplan Notices, 24, March 1989.

[2] Birtwistle G. and Loose K., A Model for Procedure Passed as Parameters, Sigplan Notices, 23, Feburary 1988.

[3] Wijngaarden Van A. et al., Revised report on the algorithmic language ALGOL 68, Acta Inform. 5 (1975) 1-236.

[4] Fisher A. J., Practical LL(1)-Based Parsing of Van Wijngaarden Grammars, Acta Informatica 21 (1985) 559-584.

[5] Barrett R. et al., Two-Level grammar as an impletable Metalanguage for Axiomatic Semantics, Comput. Lang. Vol. 11, No. 3/4, pp. 173-191, 1986.

[6] Kupka I, van Wijngaarden grammars as a special information processing model, Mathematical Foundations of Computer Science, edited P. Dembinski, pp 387-401, LNCS 88, Springer-Verlag 1980.

[7] Wijngaarden Van A., Languageless programming. The Relationship Between Numerical Computation and Programming Languages, edited J. K. Reid, pp 361-371, North-Holand Publishing Co., 1982.

[8] Edupuganty B. and Bryant B., Two-Level Grammar as a Functional Programming Language, The Computer Journal, Vol. 32, No 1, pp. 36-44, 1989.

[9] Bryant B. et al., Two-Level Grammar as a Programming Language for Data Flow and Piplined Algorithms, Proceedings IEEE Computer Society 1986 International Conference on Computer Languages, Miami, Florida, 27-30 October 1986, pp 136-143.

[10] BADII M. "A Study of Two Formal Description Languages and their Applications to PASCAL" Ph. D. Thesis, Loughborough University of Technology, U. K. (1981).

[11] Graaf De J. and Ollongarn A., On Two-Level Grammars, International Journal of Computer Mathematics, 15(1984), pp. 269-288.

[12] Maluszynski, J., Twoards a Programming Language Based on the Notion of Two-Level Grammar, Theoretical Computer Science, 28(1984), pp. 13-43.

Hey snake... I appreciate the offer,
but I like <u>real</u> apples better
Let's make MEN the Geeks.