



DGL Grammars
Are Universal

Peter M. Maurer

VCAPP Laboratory
Report SE-4

DARL

CONTEXT FREE GRAMMARS WITH VARIABLES ARE UNIVERSALLY POWERFUL*

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

This report presents a proof that the enhanced grammars presented in VCAPP Technical Report number SE-3 are universally powerful. The proof shows that an arbitrary Turing Machine can be implemented using the "variable" construct introduced in SE-3.

* This research was supported by the University of South Florida Center for Microelectronics Design and Test.

CONTEXT FREE GRAMMARS WITH VARIABLES ARE UNIVERSALLY POWERFUL*

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

1 Introduction

An extension to context free grammars was presented in [1]. This extension allows a finite number of variables to be declared. At some point in the leftmost derivation of a string, a variable can be assigned a value that contains terminals and non-terminals. The value of the variable can then be inserted into the output at any convenient place to the right of the assignment. When the value of a variable contains non-terminals, these non-terminals are replaced by one of the alternatives of the referenced productions.

The purpose of this paper is to show that context free grammars enhanced with variables are universal in the sense that when such a grammar is interpreted using the technique presented in [1], that variables can be used to simulate the action of an arbitrary Turing Machine.

2 The Theorem and Proof

Theorem: *Context Free Grammars with variables can be used to simulate an arbitrary Turing Machine, when interpreted as follows. Beginning with the start symbol, select an alternative and pass it to the interpreter. The interpreter scans the alternative from the left and outputs all terminal symbols in the order encountered. When a non-terminal is found, the interpreter chooses an alternative from the referenced production and calls itself recursively to interpret it. When an assignment of the form $\%{a.b}$ is encountered, the output derived from "a" is assigned to "b". A nonterminal of the form $\%b$ or $\%{b}$ causes the value of "b" to be passed to the interpreter. A the pair $\%\%$ causes a single $\%$ to be assigned to a variable, or output, and does not cause the following symbol to be passed to the interpreter. Non-terminals constructed by storing $\%$ symbols in a variable are treated as any other non-terminal.*

Proof: Let M be an arbitrary Turing Machine with one read-only input tape, one write-only output tape, and one work tape. Let I be the set of input symbols, T be the set

* This research was supported by the University of South Florida Center for Microelectronics Design and Test.

of work-tape symbols, O be the set of output symbols, and S be the set of states. Let f be the state transition function that maps elements of $I \times T \times S$ to elements of the form (t, o, ia, ta) , where t is the new tape symbol, o is the new output symbol or to represent no output, ia is the input tape action, and ta is the work-tape action. It is assumed that the input tape action will be either "Move-Right" or "Don't-Move", while the work-tape action includes these plus "Move-Left."

The output tape is simulated by the normal output of the interpretation process. The elements of O must be encoded as alphanumeric strings. Let n be the size of I then the elements of I are represented as variables named $i1, i2, \dots, in$. Let m be the size of T . The elements of T are encoded as variables named $t1, t2, \dots, tm$. Similarly, let k be the size of S . The elements of S are represented as variables named $s1, s2, \dots, sk$. The current state of the Turing Machine is represented by a variable named *state*. The input tape is represented by a variable named *itape*, and the work-tape is represented by two variables, *wtape_right* and *wtape_left* which represent the right and left halves of the tape respectively. For every element in the domain of the transition function f , there is one production named *ixtysz* where *ix* is the input-tape symbol, *ty* is the work-tape symbol, and *sz* is the state symbol of the domain element. The right-hand side of each of these productions will produce the output symbol, (if any,) the new tape symbol, and the tape action. Precisely how this is done will be explained below. For simplicity, it is assumed that the Turing Machine is deterministic, so each production of the form *ixtysz* has a single alternative. (Nondeterministic machines can be represented by allowing *ixtysz* to have several alternatives, but the interpreter is not guaranteed to produce an appropriate set of transitions for each input.)

The Turing Machine will be simulated using the following sequence of steps.

1. Split the three tape variables (*itape*, *wtape_left*, and *wtape_right*) into two pieces each. For each variable, two new variables will be assigned values. The value of the first variable is the symbol closest to the read head, while the value of the second is the string of all symbols on the tape except the one assigned to the first variable. For example if the input tape contains the string *aaabbb*, the first variable will be assigned the value "a" and the second will be assigned the value "aabbb." (The tape is assumed to be read from left to right.)
2. Process the current input symbol. This process breaks the set of productions named *ixtyjz* into subsets based on the value of x . The subset of productions for which x matches the current input symbol is passed to the next step.
3. Process the current work-tape symbol. The subset of productions passed from step 2 is further broken down by the value of y , where the production name is *ixtysz*. Out of this set, those whose value of y corresponds to the value of the current work-tape symbol is passed to the next step.
4. Process the current state symbol. This process selects the one production that corresponds to the current state from the set of productions passed from step 3. This production is interpreted, which causes tape symbols to be written, tape actions to take place, and state transitions to occur.
5. If the machine has halted, stop, otherwise go back to step 1.

Let "main" be the start symbol of the grammar. The overall flow is represented by the following production.

```
main: % {split_input} % {split_right} % {split_left} % {current_input}
      % {current_work} % {state} % {next_action};
```

The *split_input* production works as follows. First every symbol on the input tape is represented by a non-terminal $\% \{ix\}$. For example, if the input tape contains "abc" and a, b, and c are represented as the variables $i1$, $i2$, and $i3$ respectively, then the variable *itape* will contain the value " $\% \{i1\} \% \{i2\} \% \{i3\}$ ". The *split_input* production is defined as follows.

```
split-input: % {init_input} % {itape.BODY};
```

The "init_input" production initializes each of the variables, $i1$ through in , to the value " $\% \{ijb.HEAD\} \% \{ijc.current_input\} \% \{redefine_input\}$ ". None of the symbols in the initial value produce output. The initial value is interpreted when the non-terminal $\% \{itape.BODY\}$ is encountered by the interpreter. This causes the initial value to be interpreted for the leftmost symbol on the tape, which in turn causes the $\% \{redefine_input\}$ symbol to be interpreted. The "redefine_input" production causes the value of each of the variables $i1$ through in to be changed to $\% \% \% \% \% \% \{il\}$ through $\% \% \% \% \% \% \{in\}$ respectively. Thus only the leftmost symbol on the tape will cause assignments to be made to *HEAD* and *current_input*. All symbols after the first will cause strings of the form $\% \% \% \{ix\}$ to be assigned to *BODY*. After the symbol $\% \{itape.BODY\}$ is interpreted, the contents of *HEAD* and *BODY* will be assigned to the variables *itape_head* and *itape_body* respectively, so the complete definition of the production *split_input* is as follows.

```
split_input: % {init_input} % {itape.BODY}
             % {HEAD.itape_head} % {BODY.itape_body};
```

The value eventually assigned to *itape_head* will be of the form " $\% \% \{ix\}$ ", while the value eventually assigned to *itape_body* will be of the form " $\% \% \{ix\} \dots \% \{iz\}$ ". The value assigned to *current_input* will be $\% \{ixd\}$ where x is the number of the current input symbol. The processing of the left and right halves of the work tape is identical to the processing of the input tape. The variable *current_work* will contain the value $\% \{tyd\}$, where ty is the leftmost symbol in the variable *wtape_left*. The four variables containing work-tape symbols are *wtape_right_head*, *wtape_right_body*, *wtape_left_head*, and *wtape_left_body*. By concatenating these variables together with the new tape symbol, any tape action can be simulated.

The selection of the production $ixtysj$ proceeds as follows. After splitting, the variable *current_input* will contain " $\% \{ixd\}$ ", and the variable *current_work* will contain the value " $\% \{tyd\}$ ". For each combination of state and tape symbol there is one auxiliary variable $tysz$ defined. The production ixd has a single alternative that assigns the value of " $\% \{ixtysz\}$ " to the variable $tysz$. One such assignment is done for each auxiliary variable.

This is the first level of selection described above. Now the production *tyd* has a single alternative that assigns the current value of the variable *tysz* to the variable *sz*. This is the second level of selection. The third level of selection is accomplished by interpreting the variable "state." This variable contains the value $\% \{sz\}$ where *sz* is the current state. Interpreting the the symbol $\% \{state\}$ causes the symbol $\% \{sz\}$ to be interpreted, which in turn causes the symbol $\% \{ixtysz\}$ to be interpreted. The right-hand side of *ixtysz* causes the appropriate tape actions to be executed, the output to be generated, if any, the state to change, and the machine to halt, if necessary. Tape actions are executed by concatenating the appropriate variables and values and assigning the result to the appropriate tape variables. The state is changed by making an assignment to *state*. The machine is halted by assigning a null value to the variable *next_action*. (See the *main* production above.) The value of *next_action* is initialized to " $\% \{main\}$ " which causes the machine to continue if no assignment is made to *next_action*.

This concludes the proof of the theorem.

3 Conclusion

This report shows that context free grammars augmented with variables are universal. This fact is referred to in [1] but due to the tutorial nature of that report, and the intended audience, it seemed unwise to include this material in that report.

4 References

1. P. M. Maurer, "Reference Manual for a Data Generation Language Based on Probabilistic Context Free Grammars," Department of Computer Science and Engineering, University of South Florida, VCAPP Technical Report SE-3, 1995.

APPENDIX

The following is a DGL grammar that implements a Turing Machine for recognizing strings of the form $a^n b^n c^n$.

```

/* Input tape symbols.
a=%{i1}
b=%{i2}
c=%{i3}
blank=%{i4} */

/* Work tape symbols
a=%{t1}
blank=%{t2} */

/* States
%{s1}=start,
%{s2}=accept,
%{s3}=reject,
%{s4}=copy the a's to the work tape,
%{s5}=rewind work tape to check b's,
%{s6}=check b's on input against a's on work tape.
%{s7}=rewind work tape to check for c's.
%{s8}=check c's on input against a's on work tape. */

repeat: 1;
nosave: seed;

/* Each invocation of main represents one state-change of the */
/* Turing machine */
main: %{split_input}%{split_right}%{split_left}
      %{iwork}%{wwork}%{state}%{next_action};

/* this variable contains %{main} if the machine execution is to continue */
next_action: variable %{main};

/* current state */
state: variable %{s1};

/* Current input symbol indicator */
iwork: variable;

/* Current tape symbol indicator */
wwork: variable;

/* work tape symbol definitions */
t1: variable;
t2: variable;

/* input symbol definitions */
i1: variable;
i2: variable;
i3: variable;
i4: variable;

/* state variables */
s1: variable;
s4: variable;

```

```

s5: variable;
s6: variable;
s7: variable;
s8: variable;
/* The following two messages are for debugging */
s3: "Machine starts in REJECT state and halts\n";
s2: "Machine starts in ACCEPT state and halts\n";

/* Input tape */
itape: variable %{i1}%{i1}%{i1}%{i2}%{i2}%{i2}%{i3}%{i3}%{i3};
itape_head: variable;
itape_body: variable;

/* Work tape Right half */
wtape_right: variable;
wtape_right_head: variable;
wtape_right_body: variable;

/* Work tape Left half */
wtape_left: variable;
wtape_left_head: variable;
wtape_left_body: variable;

/* tape processors */
split_right: %{init_work}%{t2b.head}%{t2c.wwork}%{wtape_right.body}
             %{head.wtape_right_head}%{body.wtape_right_body};
split_left:  %{init_work}%{t2b.head}%{t2c.wwork}%{wtape_left.body}
             %{head.wtape_left_head}%{body.wtape_left_body};
split_input: %{init_input}%{i4b.head}%{i4c.iwork}%{itape.body}
             %{head.itape_head}%{body.itape_body};

/* work variables */
head: variable;
body: variable;

/* tape action 1: write an "a" and move right */
wtape1: %{wtape_right_body.wtape_right}%{wtape1a.wtape_left};
wtape1a: %{wtape_right_head}%{t1}%{wtape_left_body};

/* tape action 2: rewrite the current symbol and move left */
wtape2: %{wtape_left_body.wtape_left}%{wtape2a.wtape_right};
wtape2a: %{wtape_left_head}%{wtape_right_head}%{wtape_right_body};

/* tape action 3: rewrite the current symbol and move right */
wtape3: %{wtape_right_body.wtape_right}%{wtape3a.wtape_left};
wtape3a: %{wtape_right_head}%{wtape_left_head}%{wtape_left_body};

/* tape action 4: don't rewrite or move head */
wtape4: "";

/* input tape action: move head */
input1: %{itape_body.itape};

/* input tape action: don't move head */
input2: "";

/* output tape actions */
output1 : "";
output2 : YES\n;
output3 : NO\n;

```

```

/* First-level selection variables */
slt1: variable;
slt2: variable;
s4t1: variable;
s4t2: variable;
s5t1: variable;
s5t2: variable;
s6t1: variable;
s6t2: variable;
s7t1: variable;
s7t2: variable;
s8t1: variable;
s8t2: variable;

/* this is the master-list of actions */
slt1i1: %{input1}%{wtape1}%{output1}%{s4a.state};
slt1i2: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
slt1i3: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
slt1i4: %{input2}%{wtape4}%{output2}%{s2a.state}%{.next_action};
slt2i1: %{input1}%{wtape1}%{output1}%{s4a.state};
slt2i2: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
slt2i3: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
slt2i4: %{input2}%{wtape4}%{output2}%{s2a.state}%{.next_action};
s4t1i1: %{input1}%{wtape1}%{output1}%{s4a.state};
s4t1i2: %{input2}%{wtape2}%{output1}%{s5a.state};
s4t1i3: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s4t1i4: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s4t2i1: %{input1}%{wtape1}%{output1}%{s4a.state};
s4t2i2: %{input2}%{wtape2}%{output1}%{s5a.state};
s4t2i3: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s4t2i4: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s5t1i1: %{input2}%{wtape2}%{output1}%{s5a.state};
s5t1i2: %{input2}%{wtape2}%{output1}%{s5a.state};
s5t1i3: %{input2}%{wtape2}%{output1}%{s5a.state};
s5t1i4: %{input2}%{wtape2}%{output1}%{s5a.state};
s5t2i1: %{input2}%{wtape3}%{output1}%{s6a.state};
s5t2i2: %{input2}%{wtape3}%{output1}%{s6a.state};
s5t2i3: %{input2}%{wtape3}%{output1}%{s6a.state};
s5t2i4: %{input2}%{wtape3}%{output1}%{s6a.state};
s6t1i1: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s6t1i2: %{input1}%{wtape3}%{output1}%{s6a.state};
s6t1i3: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s6t1i4: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s6t2i1: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s6t2i2: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s6t2i3: %{input2}%{wtape2}%{output1}%{s7a.state};
s6t2i4: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s7t1i1: %{input2}%{wtape2}%{output1}%{s7a.state};
s7t1i2: %{input2}%{wtape2}%{output1}%{s7a.state};
s7t1i3: %{input2}%{wtape2}%{output1}%{s7a.state};
s7t1i4: %{input2}%{wtape2}%{output1}%{s7a.state};
s7t2i1: %{input2}%{wtape3}%{output1}%{s8a.state};
s7t2i2: %{input2}%{wtape3}%{output1}%{s8a.state};
s7t2i3: %{input2}%{wtape3}%{output1}%{s8a.state};
s7t2i4: %{input2}%{wtape3}%{output1}%{s8a.state};
s8t1i1: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s8t1i2: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};
s8t1i3: %{input1}%{wtape3}%{output1}%{s8a.state};
s8t1i4: %{input2}%{wtape4}%{output3}%{s3a.state}%{.next_action};

```

```

s8t2i1: % {input2} % {wtape4} % {output3} % {s3a.state} % { .next_action };
s8t2i2: % {input2} % {wtape4} % {output3} % {s3a.state} % { .next_action };
s8t2i3: % {input2} % {wtape4} % {output3} % {s3a.state} % { .next_action };
s8t2i4: % {input2} % {wtape4} % {output2} % {s2a.state} % { .next_action };

/* auxilliary productions for handling tape symbols */
t1a: %% {t1b.head} %% {t1c.wwork} %% {redefine_work};
t1b: %%% %%% %%% {t1};
t1c: %% {t1d};
t1d: % {s1t1.s1} % {s4t1.s4} % {s5t1.s5} % {s6t1.s6} % {s7t1.s7} % {s8t1.s8};
t1e: %%% %%% %%% %%% %%% %%% {t1};

t2a: %% {t2b.head} %% {t2c.wwork} %% {redefine_work};
t2b: %%% %%% %%% {t2};
t2c: %% {t2d};
t2d: % {s1t2.s1} % {s4t2.s4} % {s5t2.s5} % {s6t2.s6} % {s7t2.s7} % {s8t2.s8};
t2e: %%% %%% %%% %%% %%% %%% {t2};

redefine_work: % {t1e.t1} % {t2e.t2};
init_work: % {t1a.t1} % {t2a.t2};

/* auxilliary productions for handling input symbols */
i1a: %% {i1b.head} %% {i1c.iwork} %% {redefine_input};
i1b: %%% %%% %%% {i1};
i1c: %% {i1d};
i1d: % {s1t2i1a.s1t2} % {s1t1i1a.s1t1} % {s4t2i1a.s4t2} % {s4t1i1a.s4t1}
    % {s5t2i1a.s5t2} % {s5t1i1a.s5t1} % {s6t2i1a.s6t2} % {s6t1i1a.s6t1}
    % {s7t2i1a.s7t2} % {s7t1i1a.s7t1} % {s8t2i1a.s8t2} % {s8t1i1a.s8t1};
i1e: %%% %%% %%% %%% %%% %%% {i1};

i2a: %% {i2b.head} %% {i2c.iwork} %% {redefine_input};
i2b: %%% %%% %%% {i2};
i2c: %% {i2d};
i2d: % {s1t2i2a.s1t2} % {s1t1i2a.s1t1} % {s4t2i2a.s4t2} % {s4t1i2a.s4t1}
    % {s5t2i2a.s5t2} % {s5t1i2a.s5t1} % {s6t2i2a.s6t2} % {s6t1i2a.s6t1}
    % {s7t2i2a.s7t2} % {s7t1i2a.s7t1} % {s8t2i2a.s8t2} % {s8t1i2a.s8t1};
i2e: %%% %%% %%% %%% %%% %%% {i2};

i3a: %% {i3b.head} %% {i3c.iwork} %% {redefine_input};
i3b: %%% %%% %%% {i3};
i3c: %% {i3d};
i3d: % {s1t2i3a.s1t2} % {s1t1i3a.s1t1} % {s4t2i3a.s4t2} % {s4t1i3a.s4t1}
    % {s5t2i3a.s5t2} % {s5t1i3a.s5t1} % {s6t2i3a.s6t2} % {s6t1i3a.s6t1}
    % {s7t2i3a.s7t2} % {s7t1i3a.s7t1} % {s8t2i3a.s8t2} % {s8t1i3a.s8t1};
i3e: %%% %%% %%% %%% %%% %%% {i3};

i4a: %% {i4b.head} %% {i4c.iwork} %% {redefine_input};
i4b: %%% %%% %%% {i4};
i4c: %% {i4d};
i4d: % {s1t2i4a.s1t2} % {s1t1i4a.s1t1} % {s4t2i4a.s4t2} % {s4t1i4a.s4t1}
    % {s5t2i4a.s5t2} % {s5t1i4a.s5t1} % {s6t2i4a.s6t2} % {s6t1i4a.s6t1}
    % {s7t2i4a.s7t2} % {s7t1i4a.s7t1} % {s8t2i4a.s8t2} % {s8t1i4a.s8t1};
i4e: %%% %%% %%% %%% %%% %%% {i4};

redefine_input: % {i1e.i1} % {i2e.i2} % {i3e.i3} % {i4e.i4};
init_input: % {i1a.i1} % {i2a.i2} % {i3a.i3} % {i4a.i4};

/* these feed the state variable */
s1a: %% {s1};

```

```

s2a: %%{s2};
s3a: %%{s3};
s4a: %%{s4};
s5a: %%{s5};
s6a: %%{s6};
s7a: %%{s7};
s8a: %%{s8};

/* feeders for slt2, slt1, s2t2, s2t1, ... */
slt1l1a: %%%{slt1l1};
slt1l2a: %%%{slt1l2};
slt1l3a: %%%{slt1l3};
slt1l4a: %%%{slt1l4};
slt2i1a: %%%{slt2i1};
slt2i2a: %%%{slt2i2};
slt2i3a: %%%{slt2i3};
slt2i4a: %%%{slt2i4};
s4t1l1a: %%%{s4t1l1};
s4t1l2a: %%%{s4t1l2};
s4t1l3a: %%%{s4t1l3};
s4t1l4a: %%%{s4t1l4};
s4t2i1a: %%%{s4t2i1};
s4t2i2a: %%%{s4t2i2};
s4t2i3a: %%%{s4t2i3};
s4t2i4a: %%%{s4t2i4};
s5t1l1a: %%%{s5t1l1};
s5t1l2a: %%%{s5t1l2};
s5t1l3a: %%%{s5t1l3};
s5t1l4a: %%%{s5t1l4};
s5t2i1a: %%%{s5t2i1};
s5t2i2a: %%%{s5t2i2};
s5t2i3a: %%%{s5t2i3};
s5t2i4a: %%%{s5t2i4};
s6t1l1a: %%%{s6t1l1};
s6t1l2a: %%%{s6t1l2};
s6t1l3a: %%%{s6t1l3};
s6t1l4a: %%%{s6t1l4};
s6t2i1a: %%%{s6t2i1};
s6t2i2a: %%%{s6t2i2};
s6t2i3a: %%%{s6t2i3};
s6t2i4a: %%%{s6t2i4};
s7t1l1a: %%%{s7t1l1};
s7t1l2a: %%%{s7t1l2};
s7t1l3a: %%%{s7t1l3};
s7t1l4a: %%%{s7t1l4};
s7t2i1a: %%%{s7t2i1};
s7t2i2a: %%%{s7t2i2};
s7t2i3a: %%%{s7t2i3};
s7t2i4a: %%%{s7t2i4};
s8t1l1a: %%%{s8t1l1};
s8t1l2a: %%%{s8t1l2};
s8t1l3a: %%%{s8t1l3};
s8t1l4a: %%%{s8t1l4};
s8t2i1a: %%%{s8t2i1};
s8t2i2a: %%%{s8t2i2};
s8t2i3a: %%%{s8t2i3};
s8t2i4a: %%%{s8t2i4};

```