NA'VI/PL Compiler User Manual

Paul Gatterdam, Robert Bieber

April 20, 2010

Contents

Chapter 1

Setup

1.1 Compilation

To compile the NA'VI/PL compiler, first navigate to the proper directory within the computer system using the terminal by using the command "cd directory" to navigate through each level of the computer's file system to get to the directory where the compiler was downloaded and stored. When that directory is reached, use the command "make" to compile the compiler. The GCC C compiler and a BASH compatible shell are required to compile and run the NA'VI/PL compiler.

1.2 Execution

To run the compiler, enter the command "./compiler inputfile". The compiler will then run the input file through the scanner first, then the parser, and then will finally put the compiled program through the virtual machine and run the program. If any errors are encountered at any step of the process, the errors will be displayed and the compiler will quit.

Once the NA'VI/PL compiler is running, appropriate error messages will be displayed after each part of the compiler is run if necessary. The error messages are detailed enough that the error should be easily understood and fixable in the code that was run through the compiler. If no errors are encountered, the code will run to completion and display the final assembly language program that is then interpreted by the virtual machine. Each step of the virtual machine and the contents of the program counter, lexicographical level, offset, and the contents of the stack will also be shown alongside each step in the execution of the assembly code program that was generated

Chapter 2

The NA'VI/PL Language

2.1 Program Structure

The structure of a NA'VI/PL program is as follows; each section of the program will be explained in detail. In this listing, the text contained within brackets is descriptive, not actual code. The bracketed parts are also optional. Your program might, for instance, contain only constant declarations and a statement: the other parts are not required.

```
/* Comments can be mixed throughout the program */
[constant declarations];
[variable declarations];
[procedure declarations];
[statement].
```

2.2 Comments

You can include comments anywhere within your NA'VI/PL program. A comment is any block of text that begins with the two-character sequence "/*" and ends with the sequence "*/". The compiler will ignore any text in between these sequences, so you can include whatever text you need to describe the function of your program within them.

Example:

```
a := 2; /* This stores the value 2 in the variable a */
```

2.3 Constant Declarations

A constant is a symbol that represents a number in your program. A constant must be named, and you may use any valid identifier for this purpose. A valid identifier is a series of characters beginning with a letter and containing only letters and numbers that is not a reserved word. Examples of valid identifiers are "pi", "Count1", and "Three8Nine". Examples of **invalid** identifiers are "0times", "The_END", and "const". Remember that identifiers can not contain symbols or begin with a number.

Once you have decided on a name for your constant, you can assign it a value. This value will never change once it has been assigned, so it is important to use constants only where the same value will be used repeatedly. For instance, if you wished to approximate the area of a circle, you may wish to define the constant "pi" with the value 3, to avoid mistakenly typing a different value for pi at some point in your program. Once you have decided on a name and value for a constant, you can declare a constant with the syntax "const [name] = [value];", or "const [name1] = [value1], [name2] = [value2], ...;" to declare multiple constants.

Example:

```
const x = 5; /* Declares a single constant */
const x = 5, y = 3; /* Declares two constants */
```

2.4 Variable Declarations

A variable is a place to store a value that can change throughout the program's execution. A variable must be named by a valid identifier in the same manner as a constant. Once a variable is declared, it will initially have the value 0. During the program's execution variables can be changed to other values using statements, discussed in section 2.6. The syntax to declare a variable is the same as the syntax to declare a constant, except that the "const" keyword has been replaced by "int", and the "= [value]" parts are omitted, because a value can not be specified in a variable declaration.

Example:

```
int x; /* Declares a single variable */
int x, y; /* Declares multiple variables */
```

2.5 Procedure Declarations

After declaring variables, you can define subprocedures. A subprocedure is a section of code that can be executed multiple times within your program. Executing a subprocedure is referred to as "calling" it, and will be discussed in more detail in section 2.6. The syntax to declare a procedure is as follows.

```
procedure <identifier>;
[constant declarations];
[variable declarations];
[procedure declarations];
[statement];
```

<identifier> can be any valid identifier used to name the procedure. Constants and variables can be declared in a procedure in the same way that they were declared before. A constant or variable declared within a procedure is said to be "local" to that procedure, and it can only be accessed by that procedure or other procedures contained within it. Local procedures and variables will be discussed in section 2.6.

Example:

```
/* Defines a procedure that doubles a variable named y */
procedure double;
const x = 2;
        y := y * x;
```

2.6 Statements

A statement is a command that instructs the program to take an action, and statements make up the functional parts of programs and procedures. The main program and procedures both allow you to specify a single statement to execute. The following sections will list the types of statements you can use.

2.6.1 Assignment

Assigning a new value to a variable is a common task. The syntax to assign to a variable is "<identifier> := <expression>". <identifier> can be the name of any variable that has been previously declared. It can be either a local variable of the current procedure, or a local variable of any procedure that the current procedure is a subprocedure of. If the same name is used at multiple levels, then the variable accessed will be the one closest to the current procedure: first the compiler will look for variables declared locally, then in the parent procedure, then in its parent procedure, etc.

<expression> can be any valid expresion. Expressions will be discussed in more detail in section 2.7. When an assignment statement is executed, the value of <expression> is stored in the variable named by <identifier>

Example:

```
/* Doubles the variable x */
x := 2 * x
```

2.6.2 Procedure Calls

To execute a procedure, use the syntax "syaw <procedure>", where <procedure> is the name of a procedure. At any time, you can call any subprocedure of the current procedure, or any procedure that has the same parent procedure as the current procedure.

Example:

```
/* Calls the function Z */
syaw Z
```

2.6.3 Input

To receive input from the user on the console, use the syntax "mi <identifier>", where <identifier> is the name of any variable that can be accessed from the current procedure (rules for variable access are the same for input as they are for assignment). When this statement is executed, the program will wait for the user to enter a number at the terminal, and it will be stored in the variable named.

Example:

```
/* Gets input into the variable x */
mi x
```

2.6.4 Output

To print output on the console, use the syntax "wrrpa <identifier>", where <identifier> is the name of any variable that can be accessed from the current procedure. When this statement is executed, the program will print the value of the variable to the console.

Example:

```
/* Prints the variable x */
wrrpa x
```

2.6.5 Conditional Execution

Sometimes, you may wish to execute some statement only if some condition is true. For instance, you may want to decrease the value of a variable, but only if it is currently greater than zero. To accomplish this, you can use the syntax "txo <condition> tsakrr <statement>", where <condition> is any valid conditional expression (conditionals will be discussed in more detail in section 2.8) and <statement> is any valid statement.

Example:

```
/* Decrease x if it is greater than 0 */
txo x > 0 tsakrr x := x − 1
```

You may also specify a statement to execute if the condition is not met, with the syntax "txo <condition> tsakrr <statement> txokefyaw <statement>". This syntax is the same as before, except that the second statement will be executed if the condition is not met.

Example:

```
/* Decrease x if it is greater than 0 */
/* If x is less than 0 set it to 5 */
txo x > 0 tsakrr
        x := x − 1
txokefyaw
        x := 5
```

2.6.6 Loops

A loop allows you to repeat the same statement multiple times, as long as some condition is true. You can use the syntax "tengkrr <condition> si <statement>" to do this, where <condition> is any valid conditional expression and <statement> is any valid statement. The statement will be repeated over and over again until the condition becomes false. If the condition is false initially, then the statement will not be executed at all.

Example:

```
/* Loop until x = 0 */
tengkrr x != 0 si
        x := x − 1
```

2.6.7 Multiple Statements

Usually, you will want to execute multiple statements at a time, rather than just one. The syntax to do this is "snga'i [statements] fpe'", where [statements] is any number of statements such that any statement other than the first is preceded by a semicolon. Note that the keywords "snga'i" and "fpe'" must be entered with the ASCII apostrophe character, not the Unicode right-single-quote character.

Example:

```
/* Display every number from 0 to 4 */
snga'i
  x := 0;
  tengkrr x < 5 si
    snga'i
      wrrpa x;
      x := x + 1;
    fpe';
fpe'
```

2.7 Expressions

When an expression is required, you may use any combination of variables, constants, numbers, parenthesis, and the operators +, -, *, and / to form a mathematical expression. For instance, the expression "x * 5 + y" would evaluate to the value of x multiplied by five and added to the value of y. The * and / operators are always evaluated before the + and - operators, and any parts of an expression in parenthesis are evaluated together. This means that, for instance, "(5 + 2) * 3" will evaluate to 21, while "5 + 2 * 3" will evaluate to 11. The operators that you can use are as follows.

Expression With Operator	Result
x + y	Evaluates to the value of x added to the value of y
x - y	Evaluates to the value of y subtracted from the value of x
x * y	Evaluates to the value of x multiplied by the value of y
x / y	Evaluates to the whole number part of the value of x divided by the value of y

Example:

```
/* Sets x equal to three times the sum of it and y */
x := 3 * (x + y)
```

2.8 Conditionals

Conditionals are expressions that are either true or false. These may be statements of equality or inequality. The syntax for a conditional is either "<expression> <operator> <expression>" or "odd <expression>", where <expression> can be any valid expression, and <operator> can be any valid conditional operators. If the "odd" keyword is used, the conditional will evaluate to true if the expression evaluates to an odd number. The operators you can use are as follows.

Condition with Operator	Evaluates True If
x < y	The value of x is less than the value of y
x <= y	The value of x is less than or equal to the value of y
x > y	The value of x is greater than the value of y
x >= y	The value of x is greater than or equal to the value of y
x = y	The value of x is equal to the value of y
x != y	The value of x is not equal to the value of y

Example:

```
/* Display every number from 0 to 4 */
snga'i
  x := 0;
  tengkrr x < 5 si
    snga'i
      wrrpa x;
      x := x + 1;
    fpe';
fpe'
```

Chapter 3

Sample Programs

3.1 Recursive Factorial Program

This program will compute the factorial of a number entered at the command line using recursive procedure calls.

```
int f, n;
procedure fact;
     int ans1;
     snga'i
          ans1:=n;
          n:= n-1;
          txo n = 0 tsakrr f := 1 txokefyaw f := 2;
          txo n > 0 tsakrr syaw fact;
          f:=f*ans1;
     fpe';
snga'i
     mi n;
     syaw fact;
     wrrpa f;
          mi;
          wrrpa;
fpe'.
```

In the above example, the first line “int f, n;” declares two new variables, f and n. The second line “procedure fact;” declares a new procedure named fact. The third line “int ans1;” declares a new variable within the local scope of the procedure fact named ans1.

The fourth line “snga'i” indicates that a statement section is about to follow. The fifth line “ans1 := n;” gives the value stored in n to the variable ans1. The sixth line “n := n-1;” subtracts one from the value of n and stores the result back in the variable n.

The seventh line "txo n = 0 tsakrr f := 1 txokefyaw f := 2;" is an txo statement that checks the condition that n = 0 and if that condition evaluates to true, sets the value of f to 1, otherwise sets the value of f to 2. The eighth line "txo n > 0 tsakrr syaw fact;" is another txo statement that checks to see if the value of n is greater than zero and if it is, calls the procedure fact (this is an example of very basic recursion). The ninth line "f := f*ans1;" multiplies the value stored in the variable f by the value in the variable ans1 and stores the result in f. The tenth line "fpe';" signifies the end of the statement segment of code in the procedure fact.

The eleventh line "snga'i" indicates the start of a segment of multiple statements. The twelfth line "mi n;" takes input from the user and stores the input value in the variable n. The thirteenth line "syaw fact;" calls and executes the procedure fact. The fourteenth line "wrrpa f;" outputs the value stored in the variable f to the screen. The fifteenth line "mi;" takes user input and stores it on the top of the stack. The sixteenth line "wrrpa;" outputs the value on the top of the stack to the screen. The seventeenth line "fpe'." signifies the end of the segment of statement code in the main program block and the end of the program.

3.2 Nested Procedure Program

This program demonstrates the use of nested procedures.

```
const k = 3;
int x,y,z,v,w;
/* This is a comment */
procedure a;
  int x,y,u,v;
  procedure b;
    int y,z,v;
    procedure c;
      int y,z;
      snga'i
        z:=1;
        x:=y+z+w
      fpe';
    snga'i
      y:=x+u+w;
      syaw c
    fpe';
  snga'i
    z:=2;
    u:=z+w;
    syaw b
  fpe';
snga'i
```

```
  x:=1; y:=2; z:=3; v:=4; w:=5;
  x:=v+w;
  wrrpa z;
  syaw a;
fpe'.
```

The above program is designed to show the nesting of procedures. The first line, "const k = 3;" defines a constant named k that is given the value 3 and cannot be changed at any point in the program, but can be read in the exact same manner as a variable. The second line, "/* This is a comment */", is a comment and is completely ignored by the compiler.

The third line, "int x, y, z, v, w;" is a declaration of many variables. The fourth line, "procedure a;" is the declaration of the procedure named a and the code following it as the function. The fifth line, "int x, y, u, v;", declares four variables within the local scope of the procedure a (note that if x, y, or v is called by procedure a or any of the nested procedures within a, these three variables will override the variables declared in main first if they are within or within the parent of the function being currently executed).

The sixth line, "procedure b;" simply declares the procedure b. The seventh line, "int y, z, v;" declares these three variables within the local scope of procedure b (again note that y, z, and v will override any variables with the same name above their level if possible).

The eighth line, "procedure c;" declares the procedure named c. The ninth line, "int y, z;" declares two variables with a local scope to the procedure c. The tenth line, "snga'i" declares the start of a segment of statement code. The eleventh line, "z := 1;" assigns the value 1 to the variable z (which is the variable declared within procedure c). The twelfth line, "x := y+z+w", sums the variables y (in procedure c), z (in procedure c), and w (in the main section of code) and assigns them to x (in procedure a). The thirteenth line, "fpe';" signifies the end of the statement segment of procedure c.

The fourteenth line, "snga'i" begins the statement segment of code that is part of procedure b. The fifteenth line, "y := x+u+w;", sums the values of x (in procedure a), u (in procedure a), and w (in the main section of code) and stores the value in y (in procedure b). The sixteenth line, "syaw c", calls and executes the procedure c. The seventeenth line, "fpe';" signifies the end of the statement segment of procedure b.

The eighteenth line, "snga'i" begins the statement segment of procedure a. The nineteenth line, "z := 2;" assigns 2 to the value of z (in the main section of code). The twentieth line, "u := z+w;", sums the values of z (in the main section of code) and w (also in the main section of code) and stores the result in the variable u (in procedure a). The twenty-first line of code, "syaw b", calls and executes the procedure b. The twenty-second line of code "fpe';", signifies the end of the statement segment of procedure a.

The twenty-third line of code, "snga'i", begins the statement segment of the main code. The twenty-fourth line of code, "x := 1; y := 2; z := 3; v := 4; w := 5;" assigns the given values to their respective variables (at this point, all variables used are strictly in the scope of the main code, or in a sense, the entire program). The twenty-fifth line of code, "x := v+w;", sums the variables v and w and stores them in the variable x. The twenty-sixth line of code, "wrrpa z;" displays the value of the variable z to the screen. The twenty-seventh line of code, "syaw a;" calls and executes the procedure a. The twenty-eighth and final line of code, "fpe'." signifies the end of the statement segment in the main section of code and the end of the program.