Vesta's System Description Language

An Introduction to Vesta's Language for Describing Builds and Expressing Configurations
Vesta SDL Introduction

- What SDL is/isn't
- Syntax
- Data Types
- Many “Hello World”s
- Operators
- Scoping
- More complex examples
What is Vesta SDL?

- A functional programming language
- A way of manipulating files and directories
- A way of running tools in an encapsulated environment and capturing the changes made by those tools
- A method for expressing configurations (sets of specific versions which go together)
Vesta SDL isn't

• Like a Makefile
  – SDL is a programming language with data structures and functions
  – Makefiles are nearly flat lists of commands used to generate result files and dependencies
• A way of expressing dependencies
  – The Vesta evaluator detects dependencies automatically
Syntax: Overview

- Vesta SDL syntax is similar to C/C++
  - Whitespace separates but is otherwise insignificant
  - Statements are terminated with a semicolon ;
  - Blocks of statements enclosed in curly braces { }
  - Strings are enclosed in double quotes, using backslash to escape special characters (\”, \n, \t)
Syntax : Comments

• C style comments:
  - /* comment */ not in the comment

• C++ style comments:
  - // comment goes to end of line

• Special comments (aka “pragmas”):
  - /**nocache**/
  - /**pk**/
  - /**noupdate**/
Syntax : Identifiers

- Identifiers can be made up of any sequence of:
  - Letters
  - Decimal digits
  - Underscores
  - Periods

- But, anything that can be parsed as an integer will be treated as a numeric literal
Syntax : Identifiers

- Some valid identifiers:
  - myVar
  - foo.c
  - __
  - 36.foo
  - 123_456
  - 3.14159
Simple Data Types

- **Boolean.** Literals: `TRUE, FALSE`
- **Integers.** Example literals: `0, 1024, 07531, 0xa0`
- **Text strings.** Example literals:
  - "Simple text."
  - "Text with "quotes"." 
  - "Examples of\n\ntescaped whitespace.\n"
Data Types : Lists

- A list is a linear sequence of values
- Lists can contain any data type (including lists and other complex types)
- List literals are enclosed in angle brackets (<>), with commas separating elements (final comma optional)
- Examples:
  - `<1, "abcdefg", FALSE,>
  - `< < 1, 2, 3 >, <"a", "b", "c"> >
Data Types : Bindings

- A binding is a sequence of name/value pairs (similar to Perl hashes, Python dictionaries)
- Bindings can contain any data type (including lists and bindings)
- Binding literals are enclosed in square brackets ([ ]) with elements separated by commas (final comma optional)
- Example:
  ```
  - [ foo = 1, bar = TRUE, msg = "a string", ]
  ```
More Binding Syntax

- Nested bindings made by specifying a path with names separated by slashes:
  - [ foo/a = 1, bar/b = 2 ]
  - [ foo = [ a = 1 ], bar = [ b = 2 ] ]

- Placing a variable in binding with the same name as the variable:
  - [ foo, bar ]
  - [ foo = foo, bar = bar ]
More Binding Syntax

- A text stored in a variable used as the name:
  - `name = "foo"; [ $name = 1 ]`
  - `[ foo = 1 ]`

- A text expression used as the name:
  - `[ $("foo" + "bar") = TRUE ]`
  - `[ foobar = TRUE ]`
Files and Directories

• Manipulating files and directories is easy, because:
  – A file is just a text value
    • Using a source file becomes a text value in SDL
    • Returning a text value creates a file when shipped
  – A directory is just a binding
    • Using a directory becomes a binding value in SDL
    • Returning a binding value creates a directory when shipped
Data Types : Functions

- Functions are just another data type
- They can be assigned to variables and passed as arguments
- Function values can be created in two ways:
  - Defining a function creates a variable with the name of the function
  - Importing another SDL file, because models are functions
First Model: hello.ves

• Each Vesta SDL model is a function that returns a value. Here's a simple one:

```ves
{ return "Hello World!";
}
```

• If we evaluate and ship this, it will create a text file.
Filenames: `hello_name.ves`

- If a model returns a binding, shipping it creates files and directories for the binding elements

```javascript
{
    return [ msg.txt = "Hello World!" ];
}
```

- Shipping the result of this model will create a file named `"msg.txt"`
Directories: `hello_subdir.ves`

- Result files can be placed in a subdirectory just by adding a binding level

```java
{
    return [ foo/msg.txt = "Hello World!" ];
}
```

- Shipping the result of this model will put the "msg.txt" file in a directory named "foo"
Debugging: \texttt{hello\_print.ves}

- Let's look at two new things: a variable assignment and the \texttt{_print} primitive function:

  ```
  r = [ msg.txt = "Hello World!" ];
  return \_print(r);
  ```

- \texttt{_print}, which is handy for debugging, prints and then returns the value passed to it
Functions: **hello_func.ves**

- Here's an example of defining a function:
  ```
  { 
    hi(msg) 
      { 
        return [ msg.txt = msg ]; 
      };
    return hi("Hello World"); 
  }
  ```

- Note the semicolon after the function body
Imports: **hello_import.ves**

- Importing a model in the same directory:

  ```python
  import
  hi = hello.ves
  {
    return [ msg.txt = hi() ];
  }
  ```

- The import creates a variable named "**hi**" containing a function which is the model "**hello.ves**"
Dot ( . ) : The Special Variable

- Every function (including models) has a special, undeclared, final parameter named “." (also called “dot” or “the environment”)

- You can explicitly pass a value for this parameter

- If you don't explicitly pass a value, the value of dot in the calling context is passed

- This is often used to pass a build environment that includes specific version of tools and functions to run those tools
Dot Example

• **hello_import2.ves:**
  ```
  import
  
  hi = dot_msg.ves;
  {
    . = [ msg = "Hello World!" ];
    return hi();
  }
  ```

• **dot_msg.ves:**
  ```
  {
    return [ msg.txt = ./msg ];
  }
  ```
Files: **hello_files.ves**

- We can get a variable with the contents of a file in the same directory as our model:

```ves
files
    msg.txt;
{
    return [ msg.txt ];
}
```

- This is how source files are used in SDL
Files: `hello_files2.ves`

- We can also get a variable with the contents of a directory as a binding:

```ves
files
dir;
{
    return [ msg.txt = dir/hello.txt ];
}
```
In-line Code: \texttt{hello\_inline.ves} \\

• Now let's have some real fun and build ourselves a little program:

\begin{verbatim}
from /vesta/vestasyso.org/platforms/linux/redhat/i386 import std_env/9;
{
  . = std_env()/env_build();
  code = "#include <stdio.h>\n" +
         "main(){printf("Hello World!\n");}\n";
  return ./C/program("hello", [ foo.c = code ], [],
                      <./libs/c/libc>);
}
\end{verbatim}

• There's a lot in this example, so let's go through it piece by piece
In-line Code: `hello_inline.ves`

- We start by importing another SDL file from another directory:
  ```python
  from /vesta/vestasys.org/platforms/linux/redhat/i386 import std_env/9;
  ```
- This is how configurations are expressed in SDL by referring to specific versions of models in other packages.
- This actually imports:
  ```plaintext
  /vesta/vestasys.org/platforms/linux/redhat/i386/std_env/9/build.ves
  ```
In-line Code: `hello_inline.ves` 

- Next we use `std_env` so set the value of `dot`:
  
  ```ves
  . = std_env()\)/env_build();
  ```

- This calls the `std_env` model as a function.

- It performs a binding lookup (`/`) in the result of `std_env` for the name “`env_build`” and then calls that as a function.

- Finally, it assigns the result of `env_build` to `dot`
In-line Code: `hello_inline.ves`

- After assigning the variable “code” a text value containing a short C program, we call a function to compile it into an executable:
  
  ```
  return ./C/program("hello", [ foo.c = code ], [], <./libs/c/libc>);
  ```

- This does a two-level binding lookup within dot to get a function which builds C programs

- Arguments: target name, code, headers, libraries

- This is one of many functions provided by `std_env`
In-line Code: **hello_inline.ves**

- With the standard C/C++ bridge, libraries include their headers
  - Without `./libs/c/libc`, there would be no `stdio.h`, and compilation of our little program would fail
- The file **foo.c** only exists in the temporary filesystem used during compilation
  - The user never sees a **foo.c** file in any directory
• Let's wrap that up in a little function:

```python
from /vesta/vestasys.org/platforms/linux/redhat/i386 import std_env/9;
{
    . = std_env()/env_build();
    hi(name, msg) {
        code = ("#include <stdio.h>\n" +
                "main(){printf(""+msg+"\n");}\n");
        return ./C/program(name, [ foo.c = code ], [],
                             <./libs/c/libc>);
    };
    return hi("hello", "Hello World!");
}
```
Appending text values

- The plus operator can be used to combine text values:

  ```
  code = ("#include <stdio.h>
  "
  "main(){printf(""+msg+"\n");}
  ");
  ```

- This even works for combining files, or appending/prepending text to files
hello_inline3.ves

• Now let's call multiple times:

```python
from /vesta/vestasys.org/platforms/linux/redhat/i386 import std_env/9;
{
  . = std_env()/env_build();
  hi(name, msg) {
    code = ("#include <stdio.h>
      "main(){printf(""+msg+"\n");}\n");
    return ./C/program(name, [ foo.c = code ], [],
      <./libs/c/libc>);
  };
  r = [];
  foreach [ n = m ] in [ hello = "Hello World!",
      goodbye = "Goodbye World!" ] do
    r += hi(n,m);
  return r;
}
```
hello_inline3.ves

• **foreach** can be used to iterate over bindings and lists:

```ves
r = [];
foreach [ n = m ] in [ hello = "Hello World!",
                     goodbye = "Goodbye World!" ] do
    r += hi(n, m);
```

• Similar to C/C++, SDL has assignment operators that modify an existing variable

  – `+=` can be used to merge into an existing binding variable
What happens when `foo.c` changes between `hello` and `goodbye`?

- It's just like compiling against two different versions of the same source file: Vesta notes the difference in contents and recompiles.
- The two different intermediate `foo.o` files and final executables are stored separately, each recorded with dependencies on the specific contents of `foo.c` that produced them.
hello_inline4.ves

- Here's a better way to loop over a binding:

```python
from /vesta/vestasys.org/platforms/linux/redhat/i386 import std_env/9;
{
    . = std_env()/env_build();
    hi(name, msg) {
        code = (
            "#include <stdio.h>\n" +
            "main(){printf(""+msg+"\n\n");}\n";
        return ./C/program(name, [ foo.c = code ], [],
        <./libs/c/libc>);
    }
    return _map(hi, [ hello = "Hello World!",
         goodbye = "Goodbye World!" ]));
}
```
**_map**

- The **_map** primitive function will call a function once for each element of a list or binding
  - Function must take one argument for lists
  - Function must take two arguments (name and value) for bindings
- **_par_map** is equivalent to **_map**, but performs the different function calls in parallel
- The SDL programmer chooses when to parallelize, but there's no difference in the result
Binding Ops: **bind_plus.ves**

- When used on bindings, `+` is called "binding overlay":

```plaintext
{ 
  b1 = [x=1, y=2];
  b2 = [x=3, z=4];
  return b1+b2;
}
```

- Names in the right-hand operand take precedence, so this returns:

```
[ x=3, y=2, z=4 ]
```
Binding Ops: `bind_app.ves`

- The `_append` primitive function is similar to `+`, but only works when there are no name overlaps:

```plaintext
{ 
    b1 = [x=1, y=2];
    b2 = [z=4];
    return _append(b1, b2);
}
```

- In general, you should use `_append` if you know that there won't be name overlaps.

- A name overlap will cause a run-time error.
Binding Ops: `bind_diff.ves`

• The `–` operator removes names:

```plaintext
{ 
  b1 = [x=1, y=2];
  b2 = [x=3, z=4];
  return b1–b2;
}
```

• Names in the right-hand operand are removed from the left-hand operand, so this returns:

```
[ y=2 ]
```

• Values in the right-hand binding are ignored
Binding Ops: `bind_test.ves`

- The `!` operator tests whether a name exists in a binding and returns a boolean:

```plaintext
{ 
  f(b) {
      return (if b!x then b/x else 0);
  }
  return <f([x=1,y=2]), f([y=3,z=4])>;
}
```

- This returns:

```plaintext
<1, 0>
```
If Expressions

• Note the return expression in that function:
  \[
  \text{return } \ (\text{if } b!x \ \text{then } b/x \ \text{else } 0)\;
  \]

• In SDL, if is a type of \textit{expression} \textbf{not} a type of \textit{statement}

• This is similar the ternary operator in C/C++
  \[
  (\text{test} \ ? \ \text{true} : \ \text{false})
  \]
**Binding Ops : bind_pp.ves**

- Related to `+` is `++`, the “recursive overlay” operator:
  
  ```
  {
    b1 = [foo/x=1, bar/y=2];
    b2 = [foo/u=3, bar/v=4];
    return <b1+b2, b1++b2>;
  }
  ```

- With `+`, names are replaced. With `++`, nested bindings are recursively merged.

- `++` is very useful for making directory structures
Binding Ops: `bind_pp2.ves`

- `++` only recurses when the value on both sides are bindings
- If only one is a binding, the right-hand side value gets used (just like `+`):
  ```
  { 
  b1 = [foo=1, bar/y=2];
  b2 = [foo/u=3, bar=4];
  return b1++b2;
  }
  ```
- In this case, the result is identical to `b2`
Integer Operations

- Integer operations work pretty much as you would expect:
  - Binary operators: +, −, *
  - Unary − negates
  - Primitive functions: \_div, \_mod, \_min, \_max
  - Comparison: <, <=, ==, !=, >=, >
Text Operations

• Text operations are also pretty self-explanatory:
  – Concatenation: +
  – Comparison: ==, != (Note: no relative comparison)
  – Primitive functions: _length, _sub, _find, _findr, _elem
Assignment operators

• Here are all the modify-in-place assignment operators:
  – `+=` : works on bindings, lists, texts, integers
  – `++=` : works on bindings
  – `--=` : works on bindings, integers
  – `*=` : works on integers
Scoping: `scoping1.ves`

- There are no global variables, but functions do capture their definition context:

```java
{x = 1;
 f(y) { return x+y; }
 x = 2;
 return f(3);
}
```

- The function body sees the first value for `x`, so the result is 4, not 5!
Scoping: scoping2.ves

- If a function modifies a variable, that change is local:

```c
{x = 1;
 f(y) { x += y; return x; };
 return <f(2), f(3), x>;
}
```

- \(x\) is unmodified by the function call, so the result is:

\(<3, 4, 1>\)
Scoping: `scoping3.ves`

- A block of statements can be used as an expression, but assignments are local:
  ```ves
  {  
    x = 1; y = 2;
    z = { x += y; return x; };
    return <x, y, z>;
  }
  
  • `x` is unmodified by the block, so the result is:
  `<1, 2, 3>`
Scoping: `scoping4.ves`

- The reason assignments in blocks confuse people is because the rule is different for `foreach` blocks:

  ```ves
  {
  x = 1;
  foreach y in <2, 3, 4> do {
    x += y;
  };
  return x;
  }
  ```

- The result of this is 10
Scoping: scoping5.ves

● Remember that all functions have an implicit final parameter “.” which is usually taken from the calling context, but can be passed explicitly:

```ves
{  
  f() { return ./x+1; };  
  . = [ x = 1 ];  
  return <f(), f([x = 3])>;  
}  

● The result is:  
  <2, 4>
```
Real Examples

• Let's look at some models used to build part of Vesta.
  – These models are come from:
    /vesta/vestasys.org/vesta/config/16
  – We'll look at:
    • src/docs.ves – Create the vgetconfig man page
    • src/lib.ves – The config library
    • src/progs.ves – The vgetconfig program
Excluding comments, here it is:

```c
files
  mtex_files = [ vgetconfig.1.mtex ];
{
  return ./mtex/mtex(mtex_files);
}
```

The `files` clause creates a binding with the file `vgetconfig.1.mtex` stored in `mtex_files`

It returns the result of calling `./mtex/mtex` with `mtex_files` as an argument
/vesta/vestasys.org/vesta/config/16/src/lib.ves

```c
files
    c_files = [ VestaConfig.C ];
    h_files = [ VestaConfig.H ];
{
    ovs = [ Cxx/options/thread_safe = TRUE ];
    return ./Cxx/leaf("libVestaConfig.a",
        c_files, h_files, /*priv_h_files=*/ [], ovs);
}
```

- **The files** caluse creates two bindings:
  - `c_files` containing `VestaConfig.C`
  - `h_files` containing `VestaConfig.H`
The variable **ovs** is set to a nested binding with a compile override:

```
ovs = [ Cxx/options/thread_safe = TRUE ];
```

The result is from the function **./Cxx/leaf** which builds a C++ library from source:

```
return ./Cxx/leaf("libVestaConfig.a",
 c_files, h_files, /*priv_h_files=*/ [], ovs);
```

The arguments to **./Cxx/leaf** are: library name, code, public headers, private headers, overrides
• Too much for one slide, so ...
• The files clause brings in a single source file, but also creates an empty binding in `h_files`:

```c
files
    vgetconfig_c = [ vgetconfig.C ];
    h_files = [ ];
```

• The body of the model starts by setting some build options:

```c
// set build switches
. += [ env_ovs/Cxx/options/thread_safe = TRUE ];
ovs = [];
```
Next, we create an in-line source file with a version identifier:

```c
vgetconfig_c += [ Version.C = "const char *Version = "" +
    (if(!version_string) then ./version_string
    else
    ./generic/replace_text(./generic/replace_text(_model_name(_self ), "/vesta/vestasys.org/vesta/", ""),
    "/src/progs.ves", ""))
    + "\";\n"
];
```

- This uses `./version_string` if set
- If not, it generates a string from the model path
Finally, the source is compiled into a binary:

```c
libs = < ./libs/vesta/config, ./libs/basics/basics_umb >;
return
./Cxx/program("vgetconfig", vgetconfig_c, h_files, libs, ovs);
```

- `./Cxx/program` is similar to `./C/program`

Here we put the libraries in the variable `libs`:

- `./libs/vesta/config` was defined in `src/lib.ves`
- `./libs/basics/basics_umb` is a collection of support libraries
Bonus: **hello_inline5.ves**

- One more “Hello World” using `_run_tool`:

```python
from /vesta/vestasys.org/platforms/linux/redhat/i386 import std_env/8;
{
  . = std_env()/env_build();
  code = ("#include <stdio.h>\n" +
           "int main(){printf("Hello World!\n");" +
           "return 0;}\n");
  exe = ./C/program("hello", [ foo.c = code ], [],
                    <./libs/c/libc>);
  . ++= [ root/.WD = exe ];
  r = _run_tool(./target_platform, <"hello">,
                /*stdin=*/ "",
                /*stdout_treatment=*/ "value");
  return [hello.out = r/stdout];
}
```
Where to go Next

• Examples from this presentation can be found in:
  - /vesta/vestasys.org/examples/sdl_intro
  - See the README file for some suggested exercises

• More documentation on the web:
    • Similar to these slides
    • Detailed examination of code for running lex

• The main reference