DaMPL
What is DaMPL?

• “Data Manipulation Programming Language”
• High-level abstraction language
• Features tools to read, process and write data
• Translator generates efficient C code
Quick-start guide
/* No need to previous declare them */
/* Types inferred and bind at first usage */

```
int = 0;  /* i inferred as integer */
str = "Hi!"; /* str inferred as text */
num = 1.2;  /* num inferred as real */
test = true;  /* teste inferred as boolean */
```
Assignments

```java
a = 1; b = 2*a;
print(b);  /* Outputs 2 */
c = d = b+1;
print(c); print(d);  /* Both output 3 */
/* However, you can't change a variable type */
a = "DaMPL";  /* Illegal */
```
Strings

s1 = "Hi ";
s2 = "Professor ";
s3 = "Edwards";

/* The + operator concats strings */
print(s1 + s2 + s3 + "!");
/* Output: Hi Professor Edwards! */
/* Cast functions int(), str(), float() */
message = "Your grade is ";
grade = 0;

print(message + grade); /* Illegal operation */

print(message + str(grade)); /* Much better */
Functions

/* Function declaration in DaMPL */
fun foo(a,b) {
    return a+b;
}

print(foo(1,3)); /* prints 4 */
print(foo("abc","def")); /* prints abcdef */

/* Notice how it works for multiple types */
v = 4;
arr = [1,2,3,v,v+1]; /* Array init */
arr[] = 10; /* Appends 10 to arr */
arr[0] = -100; /* Sets pos 0 to -100 */
print(arr); /* Prints [-100,2,3,4,5,10] */
print(arr[1:4]); /* Prints [2,3,4] */
arr[1:5] = [200];
print(arr); /* Prints [-100,200,10] */
Arrays

/* Arrays can be multidimensional */
new = [["Good","morning"],["Good","night"];
/* @ precedes insertions */
@new[0][1] = "shiny";
print(new);
/* [["Good","shiny","morning"],["Good","night"]]
/* Types still need to be respected */
new[0][1] = 1; /* Illegal */
new[0] = "abc"; /* Illegal */
Tuples

/* tuples hold structured data */
tuple Student{name:text,age:integer,grade:real}
/* If you don’t declare a type, text is default*/
/* So, student could also be defined as: */
tuple Student{name,age:integer,grade:real}

t=Student; /* tuple instantiation */
t$name = “Michael”; t$age = 20; t$grade = 99.5;
print(t$name); /* Prints Michael */
tuple Student{name:text,age:integer,grade:real}

/* Tuples can be also accessed by attr index */
/* However, the operation will be always string*/
t=Student; /* tuple instantiation */
t$(0) = "Michelle"; t$(1) = "20"; t$(2) = "99.5";
/* Types violations are null-valued */
a=1; t$(a)="not an valid age";
print(t$age); /* Prints 0 */
tuple Student{name: text, age: integer, grade: real}

/* Tables works as 1D-only arrays */
relation = Student[]; /* table instantiation */
t = Student;
t$name = "Michael"; t$age = 20; t$grade = 99.5;
relation[] = t; /* Same array operations */
/* You can also append as array of string */
relation[] = ["Bob", "25", "95.0"]; /* Attribute extraction */
print(relation$age); /* Prints [20,25] */
Control Structures

if(condition) { ... }
if(condition) { ... } else { ... }

while(condition) { ... }

/* For statements loop over arrays or tables */
a = [10,20,30,40];
for i in a {
    print(str(a) + " ");
}

/* Outputs 10 20 30 40 */
The compiler
translator
The translator

.mpl file → Translator → .c code → C Compiler → Program

.mpl includes

DaMPL libs in C
Inside the translator

- Input files
- Scanner / Parser
- AST
- Semantic Checker
- Semantic Tree
- Code Generator

C Code
DaMPL code

```plaintext
fun foo(p1,p2) {
    return p1+p2;
}
a=1;
b=1.2;
c=foo(a,b);
d="Hi ";
e="again";
f=foo(d,e);
```

C code

```c
int dampl_a; float dampl_b; float dampl_c;
String dampl_d; String dampl_e; String dampl_f;

float dampl_foo__int_float
    (int dampl_p1,float dampl_p2) {
        return dampl_p1+dampl_p2;
    }

String dampl_foo__str_str
    (String dampl_p1,String dampl_p2) {
        return dampl_str_concat(
            dampl_p1,dampl_p2);
    }

int main() {
    dampl_a=1; dampl_b=1.2;
    dampl_c=dampl_foo__int_float(dampl_a,dampl_b);

    dampl_d="Hi "; dampl_e="again";
    dampl_f=dampl_foo__str_str(dampl_d,dampl_e);

    return 0;
}
```
The Parsing Stack

<table>
<thead>
<tr>
<th>DaMPL code</th>
<th>Translate and check process</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun ping(a) {</code></td>
<td>First, build a function map with known functions including parameter count.</td>
</tr>
<tr>
<td><code>if(a&gt;0) {</code></td>
<td><code>-&gt; [“ping”,1] [“pong”,1]</code></td>
</tr>
<tr>
<td><code>print(“Ping... “);</code></td>
<td><strong>Init stack with “<em>global</em>”</strong></td>
</tr>
<tr>
<td><code>pong(a);</code></td>
<td><strong>Then, start reading statements</strong></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>-&gt; ping(int) -&gt; put “dampl_ping__int” on stack</code></td>
</tr>
<tr>
<td><code>fun pong(a) {</code></td>
<td><strong>Start interpreting dampl_ping__int:</strong></td>
</tr>
<tr>
<td><code>print(“pong!\n”);</code></td>
<td><code>-&gt; if statement -&gt; bool condition -&gt; OK!</code></td>
</tr>
<tr>
<td><code>ping(a-1);</code></td>
<td><code>-&gt; print(str) -&gt; use builtin “dampl_print__str”</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>-&gt; pong(int) -&gt; put “dampl_pong__int” on stack</code></td>
</tr>
<tr>
<td><code>ping(3);</code></td>
<td><strong>Start interpreting dampl_pong__int:</strong></td>
</tr>
<tr>
<td></td>
<td><code>-&gt; print(str) -&gt; use builtin “dampl_print__str”</code></td>
</tr>
<tr>
<td></td>
<td><code>-&gt; ping(int) -&gt; “dampl_ping__int” already on stack</code></td>
</tr>
<tr>
<td></td>
<td>\ <code>-&gt; ignore</code></td>
</tr>
<tr>
<td></td>
<td><code>-&gt; end of dampl_pong__int -&gt; pop “dampl_pong__int”</code></td>
</tr>
<tr>
<td></td>
<td><code>-&gt; end of dampl_ping__int -&gt; pop “dampl_ping__int”</code></td>
</tr>
</tbody>
</table>