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

Variables

```
/* 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

```
a = 1; b = 2*a;

print(b); /* Outputs 2 */

c = d = b+1;

print(c); print(d); /* Both output 3 */


/* However, you cant 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! */
```

Casting

```
/* 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 */
```

Arrays

```
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 */
```

Tuples

```
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 */
```

Table

```
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] */
```

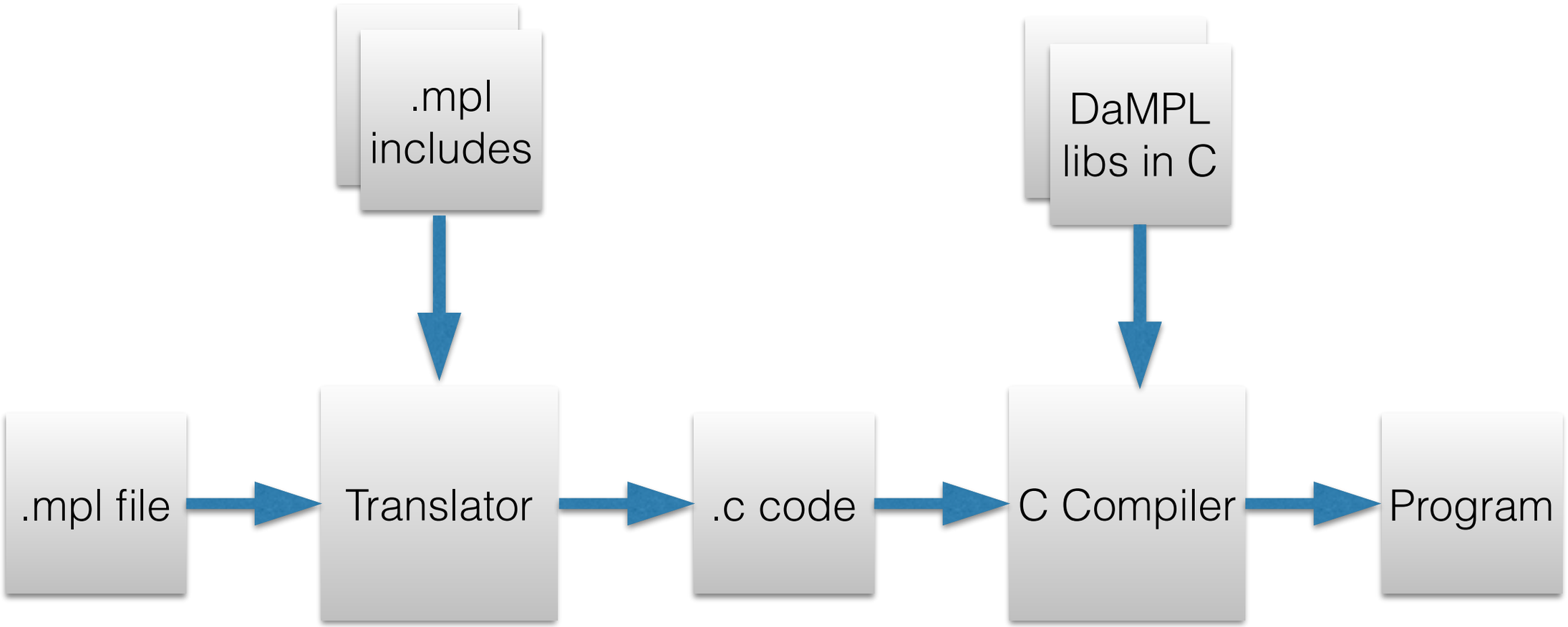
Control Structures

```
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 */
```

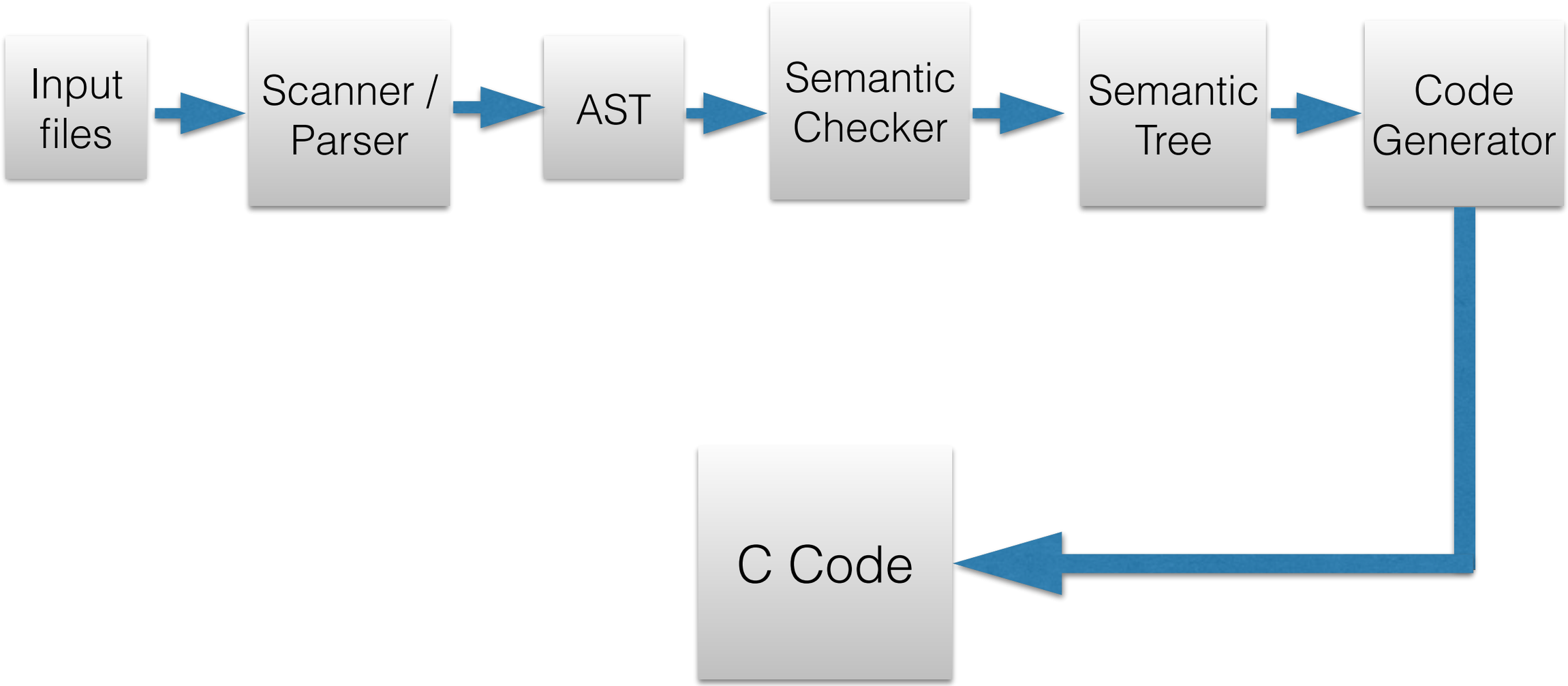
The ~~compiler~~

translator

The translator



Inside the translator



The Translator

DaMPL code

```
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

```
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

DaMPL code

```
fun ping(a) {
    if(a>0) {
        print(“Ping... ”);
        pong(a);
    }
}

fun pong(a) {
    print(“pong!\n”);
    ping(a-1);
}

ping(3);
```

Translate and check process

```
First, build a function map with known functions
including parameter count.
-> [“ping”,1] [“pong”,1]

Init stack with “_global_"
Then, start reading statements
-> ping(int) -> put “dampl_ping__int” on stack

Start interpreting dampl_ping__int:
-> if statement -> bool condition -> OK!
-> print(str) -> use builtin “dampl_print__str”
-> pong(int) -> put “dampl_pong__int” on stack

Start interpreting dampl_pong__int:
-> print(str) -> use builtin “dampl_print__str”
-> ping(int) -> “dampl_ping__int” already on stack
    \ -> ignore

-> end of dampl_pong__int -> pop “dampl_pong__int”
-> end of dampl_ping__int -> pop “dampl_ping__int"
```