CMPE 152: Compiler Design

October 5 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017

Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Reminders

- A VARIABLE node in the parse tree contains a pointer to the variable name's symbol table entry.
  - Set in the front end by method **`VariableParser::parse()`**
  - The method that takes two parameters.

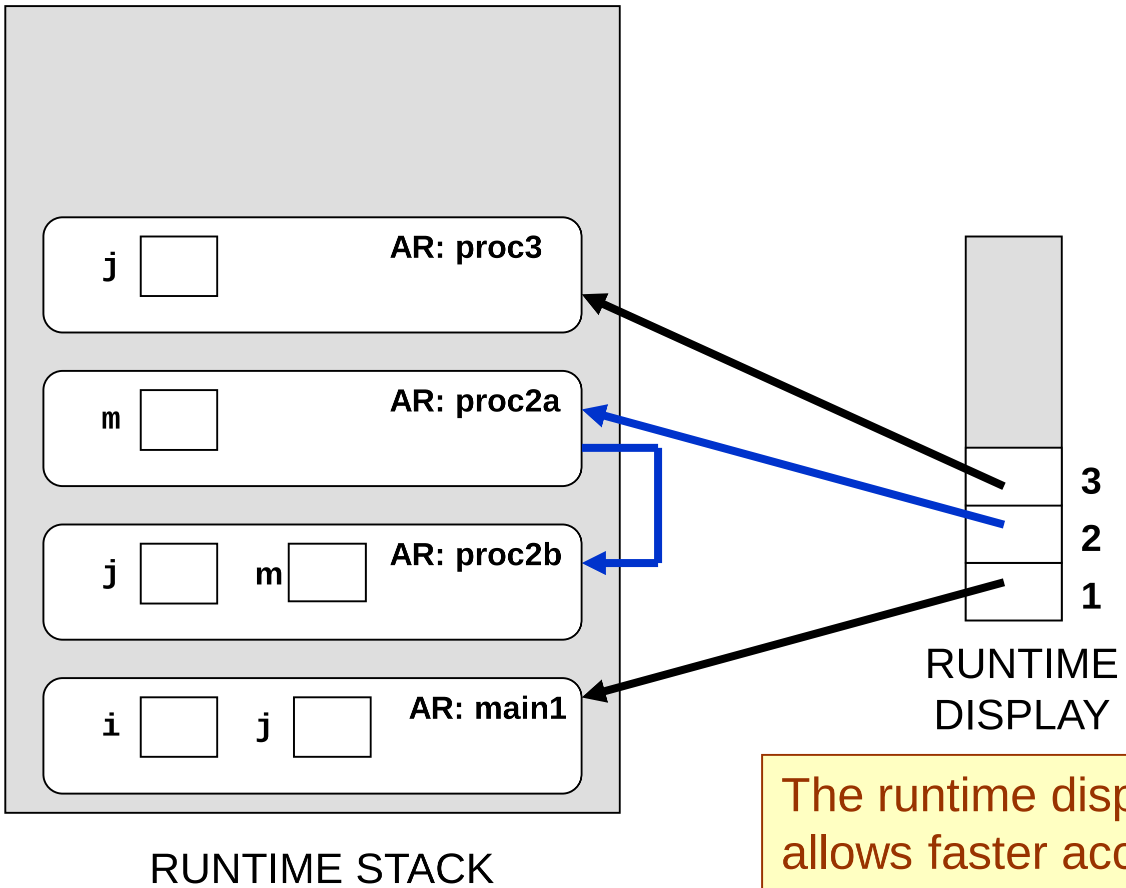
- A symbol table entry contains a pointer to its parent symbol table.
  - The symbol table that contains the entry.

Reminders, *cont'd*

- Each symbol table has a nesting level field.
- Therefore, at run time, for a given VARIABLE node, the executor can determine the nesting level of the variable.

Runtime Access to Nonlocal Variables

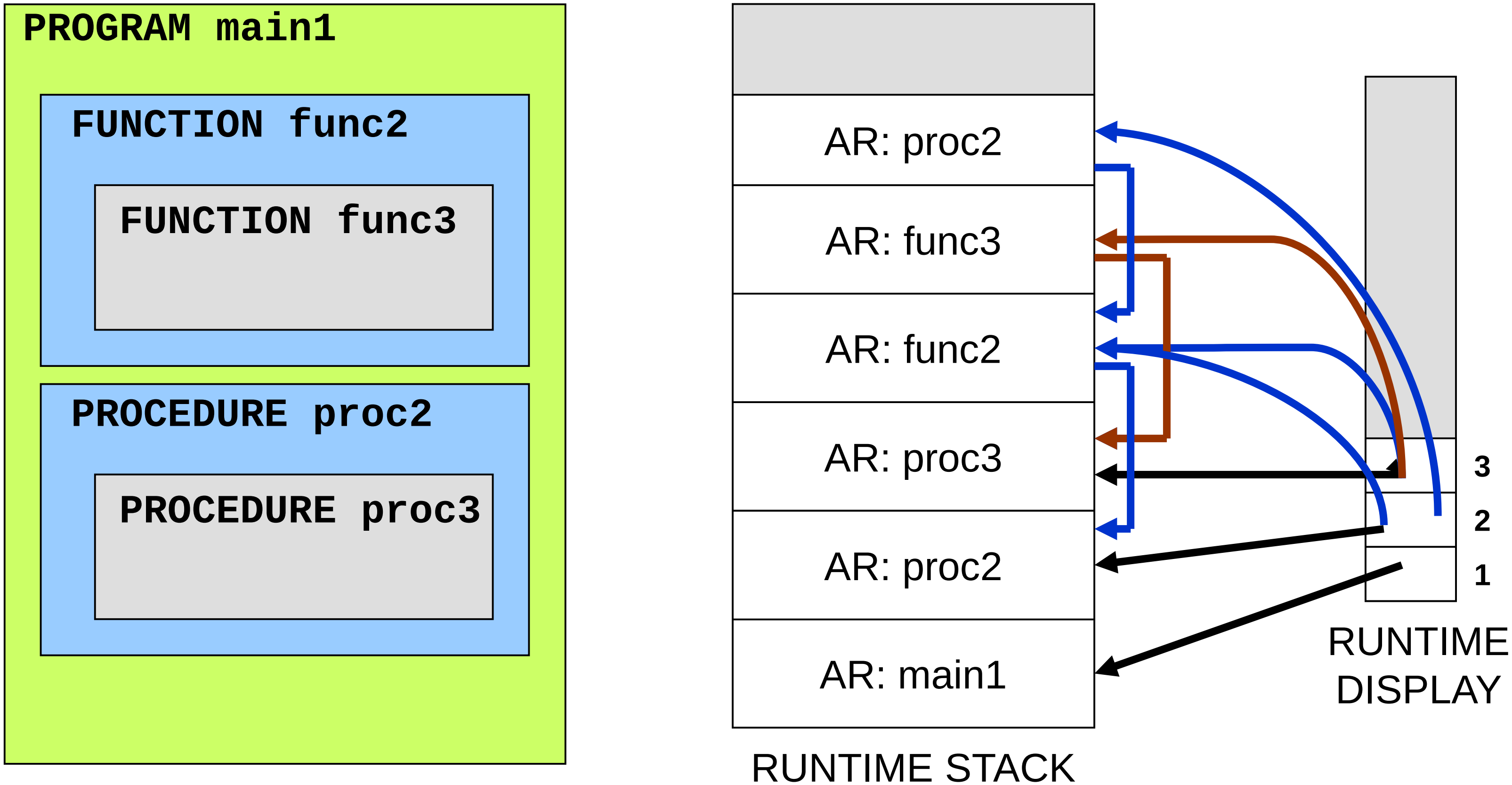
```
PROGRAM main1;
VAR i, j : integer;

  PROCEDURE proc2a;
    VAR m : integer;

    PROCEDURE proc3;
        VAR j : integer
        BEGIN
          j := i + m;
        END;

    BEGIN {proc2a}
      i := 11;
      m := j;
      proc3;
    END;

  PROCEDURE proc2b;
    VAR j, m : integer;
    BEGIN
      j := 14;
      m := 5;
      proc2a;
    END;

  BEGIN {main1}
    i := 33;
    j := 55;
    proc2b;
  END.
```



main1 → proc2b → proc2a → proc3

The runtime display allows faster access to **nonlocal** values.

Recursive Calls

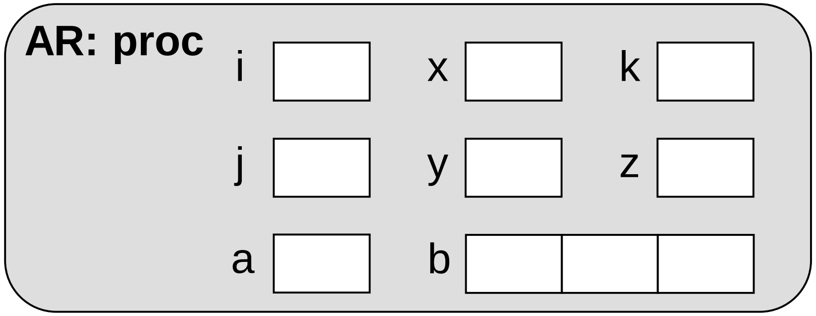


main1 ➔ proc2 ➔ proc3 ➔ func2 ➔ func3 ➔ proc2

Allocating an Activation Record

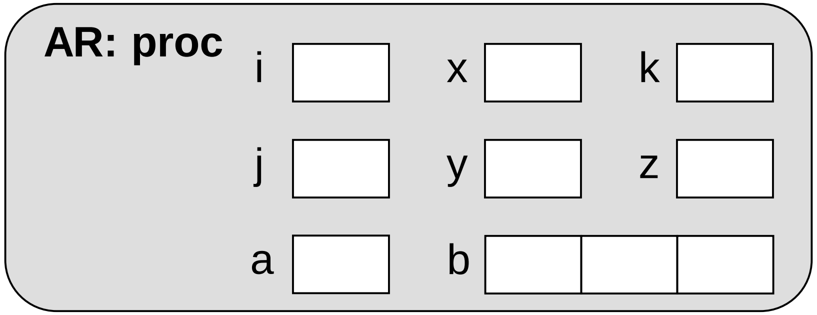
- The activation record for a routine (procedure, function, or the main program) needs one or more "data cells" to store the value of each of the routine's <u>local variables</u> and <u>formal parameters</u>.

```
TYPE arr = ARRAY[1..3] OF integer;
...
PROCEDURE proc(i, j : integer;
               VAR x, y : real;
               VAR a : arr;
               b : arr);
VAR
    k : integer;
    z : real;
```



Allocating an Activation Record

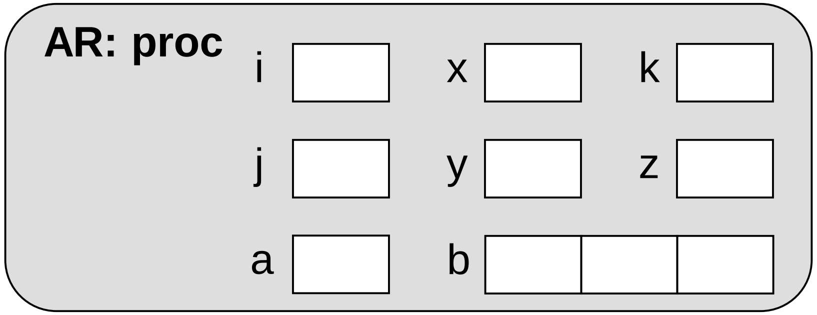
```
TYPE arr = ARRAY[1..3] OF integer;
...
PROCEDURE proc(i, j : integer;
               VAR x, y : real;
               VAR a : arr;
               b : arr);
VAR
    k : integer;
    z : real;
```



- Obtain the names and types of the local variables and formal parameters from the <u>routine's symbol table</u>.

Allocating an Activation Record

```
TYPE arr = ARRAY[1..3] OF integer;
...
PROCEDURE proc(i, j : integer;
               VAR x, y : real;
               VAR a : arr;
               b : arr);
VAR
    k : integer;
    z : real;
```

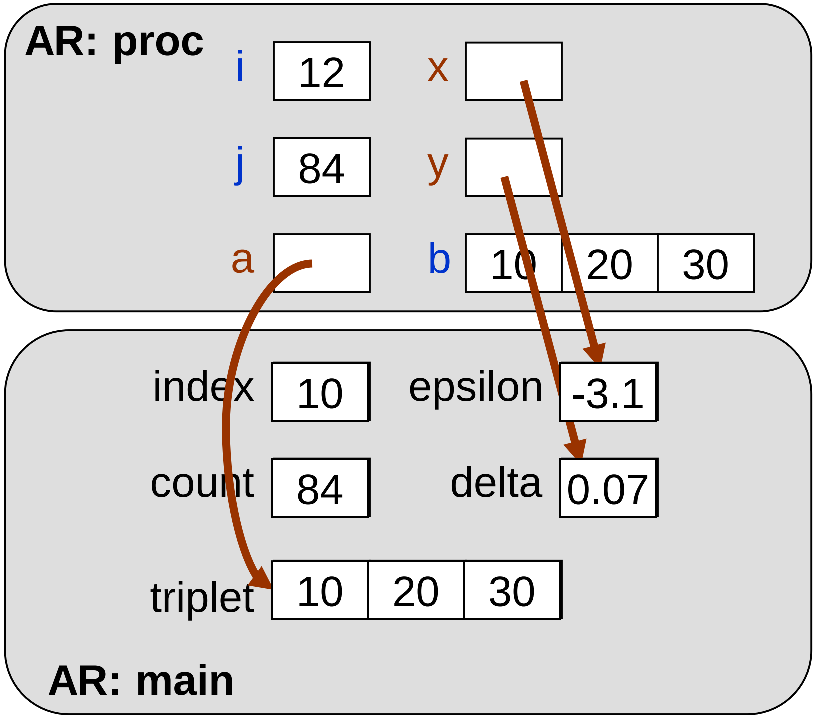


- Whenever we call a procedure or function:
  - <u>Create</u> an activation record.
  - <u>Push</u> the activation record onto the runtime stack.
  - <u>Allocate</u> the memory map of data cells based on the symbol table.

No more storing runtime values in the symbol table!

Passing Parameters During a Call

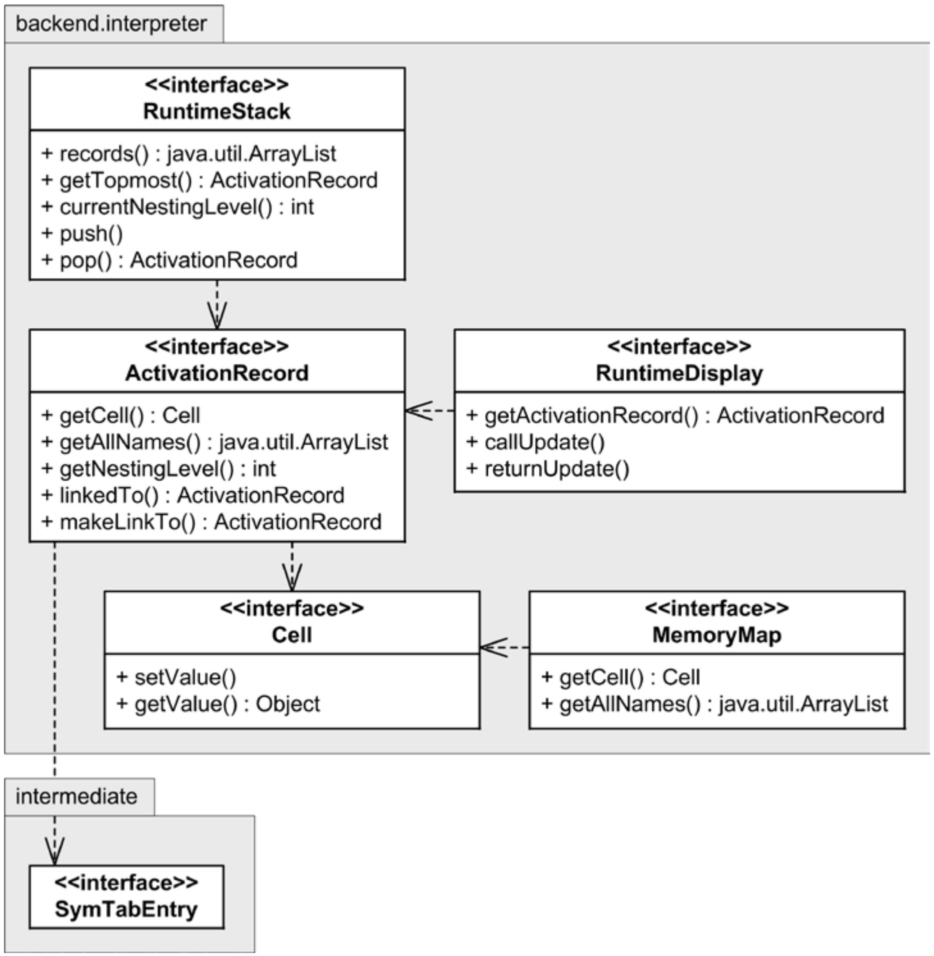
```
PROGRAM main;
TYPE
  arr = ARRAY[1..3] OF integer;
VAR
  index, count : integer;
  epsilon, delta : real;
  triplet : arr;

PROCEDURE proc(i, j : integer;
               VAR x, y : real;
               VAR a : arr;
               b : arr);
  ...
BEGIN {main}
  ...
  proc(index + 2, count, epsilon,
       delta, triplet, triplet);
END.
```



RUNTIME STACK

- Value parameters: A copy of the value is passed.
- VAR parameters: A reference to the actual parameter is passed.

Memory Management Interfaces



- Implementations:
  - Runtime stack:
    `vector<ActivationRecord *>`
  - Runtime display:
    `vector<ActivationRecord *>`
  - Memory map:
    `map<string, Cell *>`
- Class `MemoryFactory` creates:
  - runtime stack
  - runtime display
  - memory map
  - cell

Class `RuntimeStackImpl`

- In namespace `backend::interpreter::memoryImpl`
- Implemented by `vector<ActivationRecord *>`
- Methods
  - `push(ActivationRecord *ar)`
    Push an activation record onto the runtime stack.
  - `ActivationRecord pop()`
    Pop off the top activation record.
  - `ActivationRecord *get_topmost (int nesting_level)`
    Return the topmost activation record at a given nesting level. (Uses the runtime display!)

Class `RuntimeDisplayImpl`

- In package `backend::interpreter::memoryImpl`
- Implemented by `vector<ActivationRecord *>`

- Methods
  - `ActivationRecord *get_activation_record(int nesting_level)`
    Get the activation record at a given nesting level.
  - `call_update(int nesting_level, ActivationRecord *ar)`
    Update the display for a call to a routine at a given nesting level.
  - `return_update(int nesting_level)`
    Update the display for a return from a routine at a given nesting level.

Class **MemoryMapImpl**

- In package **backend.interpreter.memoryimpl**
- Implemented by **map<string, Cell *>**

- Methods
  - **MemoryMapImpl(SymTab *symtab)**
    Allocate the memory cells based on the names and types of the local variables and formal parameters in the symbol table.

  - **Cell get_cell(string name)**
    Return the memory cell with the given name.

  - **vector<string> get_all_names()**
    Return the list of all the names in this memory map.

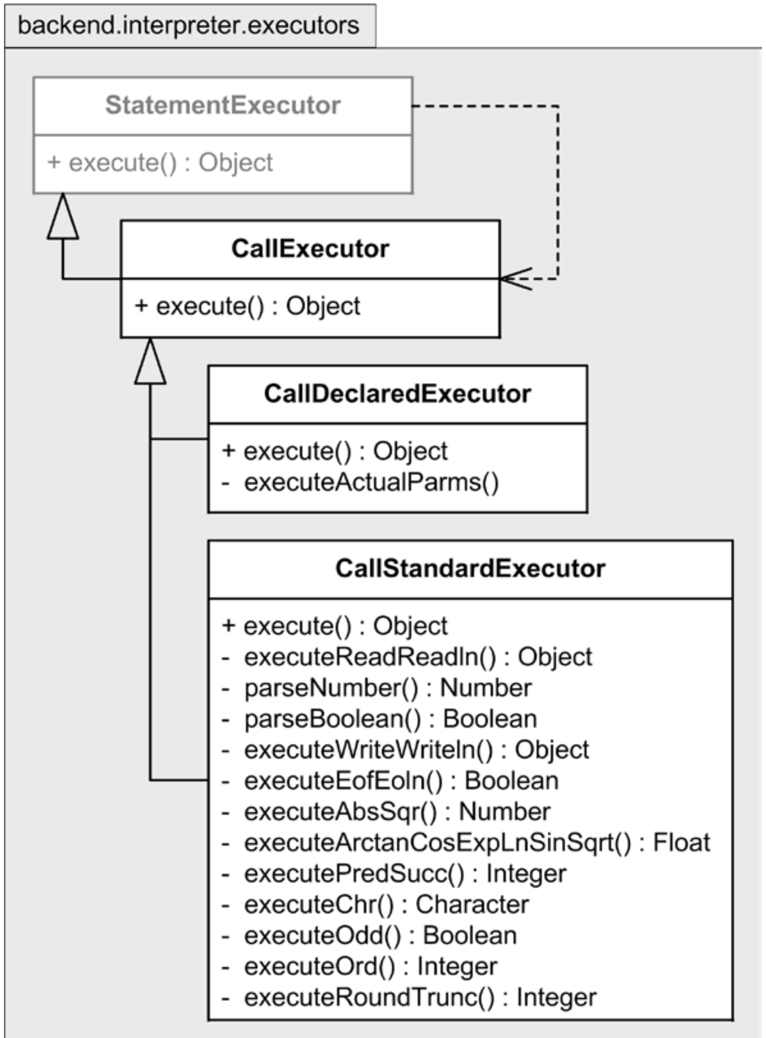
Class `ActivationRecordImpl`

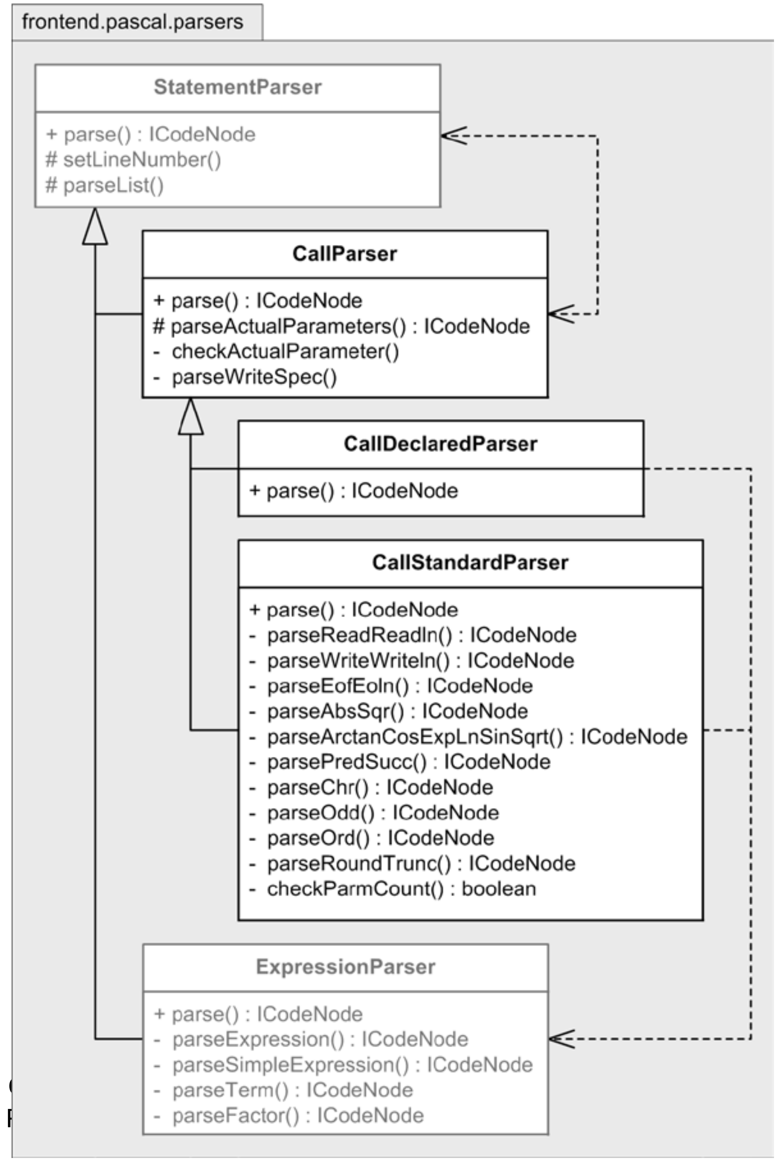
- In namespace `backend::interpreter::memoryimpl`

- Fields
  - `int nesting_level`
  - `MemoryMap *memory_map`
    Values of the local variables and formal parameters
  - `ActivationRecord *link`
    Link to the previous topmost activation record with the same nesting level in the runtime stack

- Method `Cell *get_cell(string name)`
  - Return a reference to a memory cell in the memory map that is keyed by the name of a local variable or formal parameter.

Executing Procedure and Function Calls





Class `CallDeclaredExecutor`

- Method `execute()`
  - Create a new activation record based on the called routine's symbol table.
  - Execute the actual parameter expressions.
  - Initialize the memory map of the new activation record.
    - The symbol table entry of the name of the called routine points to the routine's symbol table.
    - Copy values of actual parameters passed by value.
    - Set pointers to actual parameters passed by reference.

Class `CallDeclaredExecutor` *cont'd*

- Method **`execute()`** *cont'd*
  - Push the new activation record onto the runtime stack.
  - Access the root of the called routine's parse tree.
    - The symbol table entry of the name of the called routine points to the routine's parse tree.
  - Execute the routine.
  - Pop the activation record off the runtime stack.

Class `CallDeclaredExecutor`

- Function **`execute_actual_parms()`**
  - Obtain the formal parameter cell in the new activation record:

```
Cell *formal_cell = new_ar->get_cell(formal_id->get_name());
```

Class `CallDeclaredExecutor`

- Method `execute_actual_parms()` *cont'd*
  - Value parameter:

```
CellValue *cell_value =
                    expression_executor.execute(actual_node);

assignment_executor.assign_value(actual_node, formal_id,
                                 formal_cell, formal_typespec,
                                 cell_value, value_typespec);
```

Set a <u>copy of the value</u> of the actual parameter into the memory cell for the formal parameter.

Class `CallDeclaredExecutor`

- Method `execute_actual_parms()` *cont'd*

  - VAR (reference) parameter:

```
Cell *actual_cell =
    expression_executor.execute_variable(actual_node);

formal_cell->set_value(new CellValue(actual_cell));
```

Set a <u>reference to the actual parameter</u>
into the memory cell for the formal parameter.

  - Method `ExpressionExecutor::executeVariable()` executes the parse tree for an actual parameter and returns the <u>reference</u> to the value.

struct CellValue

- What each memory cell can hold.
  - Declared in **wci::backend::interpreter::Cell.h**

```
struct CellValue
{
    DataValue *value;
    Cell *cell;
    Cell **cell_array;
    MemoryMap *memory_map;

    ...
};
```

Class `Cell`

```
class Cell
{
public:
    /**
     * Destructor.
     */
    virtual ~Cell() {}

    /**
     * Defined by an implementation subclass.
     * @return the value in the cell.
     */
    virtual CellValue *get_value() const = 0;

    /**
     * Set a new value into the cell.
     * Defined by an implementation subclass.
     * @param new_value the new value.
     */
    virtual void set_value(CellValue *new_value) = 0;
};
```

Runtime Error Checking

- Range error
  - Assign a value to a variable with a subrange type.
  - Verify the value is <u>within range</u>
    - Not less than the minimum value and
      not greater than the maximum value.
  - Method **`StatementExecutor::check_range()`**
- Division by zero error
  - <u>Before</u> executing a division operation,
    check that the divisor's value is not zero.

Pascal Interpreter

- Now we can execute entire Pascal programs!
  - Demo

Assignment #4: Complex Type

- Add a built-in complex data type to Pascal.
  - Add the type to the global symbol table.
  - Implement as a record type with real fields **re** and **im**.

- Declare complex numbers:

```
VAR
    x, y, z : complex;
```

- Assign values to them:

```
BEGIN
    z.re := 3.14;
    z.im := -8.2;
    ...
```

Assignment #4, *cont'd*

- Do complex arithmetic:

```
z := x + y;
```

- The backend executor does all the work of evaluating complex expressions. Use the following rules:
  - $(a+bi)+(c+di)=(a+c)+(b+d)i$
  - $(a+bi)-(c+di)=(a-c)+(b-d)i$
  - $(a+bi)(c+di)=(ac-bd)+(ad+bc)i$
  - $\frac{a+bi}{c+di}=\frac{(ac+bd)+(bc-ad)i}{c^2+d^2}$

Assignment #4, *cont'd*

- Start with the C++ code from Chapter 12.
- Examine
  `wci::intermediate::symtabimpl::Predefined`
  to see how the built-in types like **`integer`** and **`real`** are defined.
- Examine
  `wci::frontend::pascal::parsers::RecordTypeParser`
  to see what information is entered into the symbol table for a **`record`** type.