Interpreters

José Miguel Rivero

rivero@lsi.upc.edu

Barcelona School of Informatics (FIB)

Technical University of Catalonia (UPC)

Summary

- Introduction
- Efficiency. Alternatives

Summary

- Introduction
- Efficiency. Alternatives
- Just-in-time (JIT) Systems
- Self-interpreters

Summary

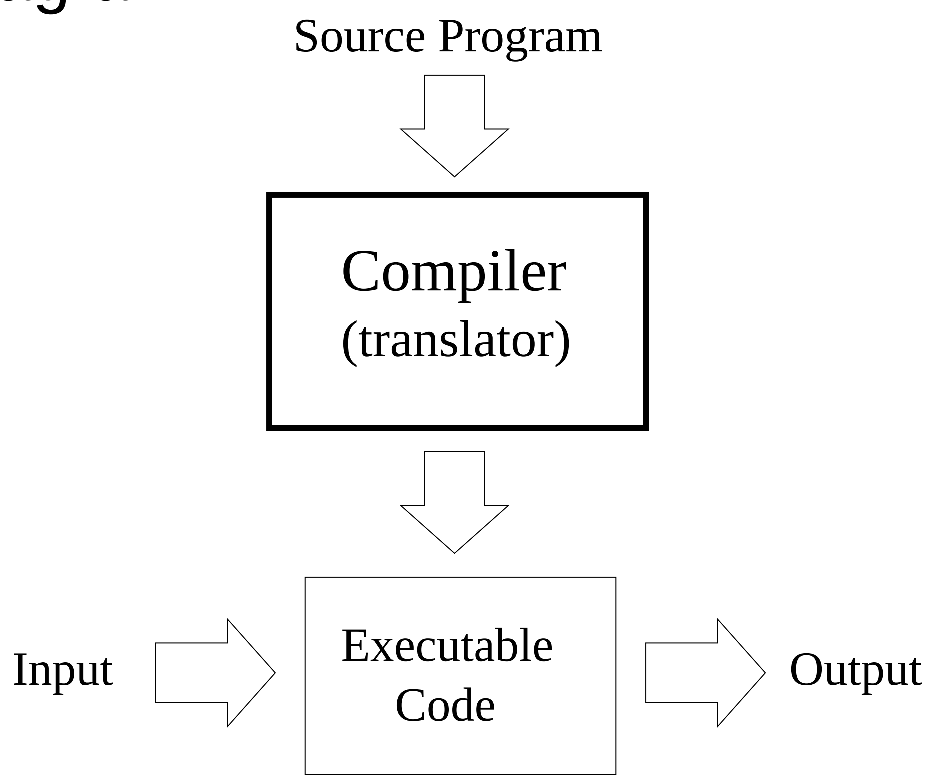
- Introduction
- Efficiency. Alternatives
- Just-in-time (JIT) Systems
- Self-interpreters
- Virtual Machines
- Usual Structure of an Interpreter

Summary

- Introduction
- Efficiency. Alternatives
- Just-in-time (JIT) Systems
- Self-interpreters
- Virtual Machines
- Usual Structure of an Interpreter
- A Very Simple Example
- Exercises

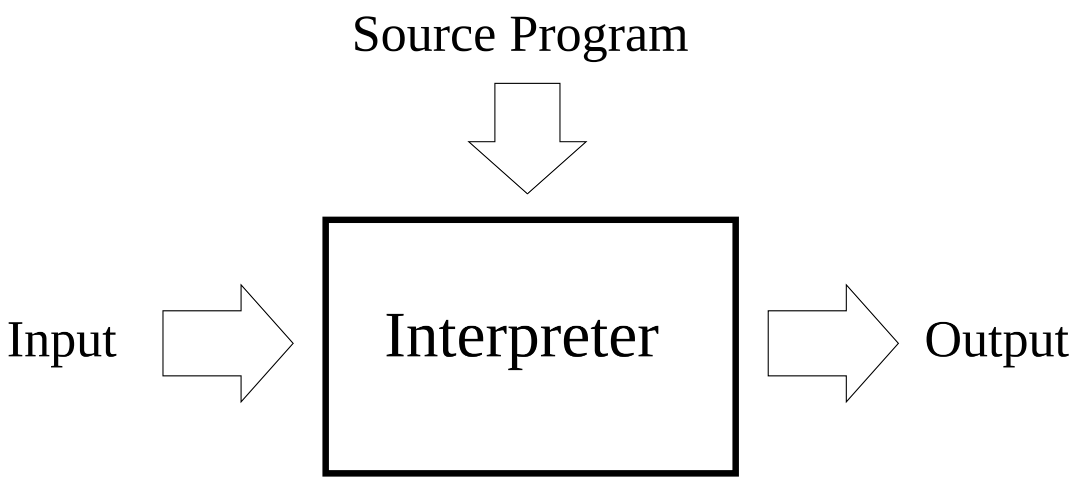
Introduction

- **Interpreter**: program that executes instructions written in some programming language
- Every language can be compiled or interpreted
- Compiler Diagram:



Introduction

- **Interpreter**: program that executes instructions written in some programming language
- Every language can be compiled or interpreted
- Interpreter Diagram:



Efficiency

- The difference in execution time is usually one o more orders of magnitude
- But many times the interpretation time is lower than the compile time plus the execution time of the compiled code

Efficiency

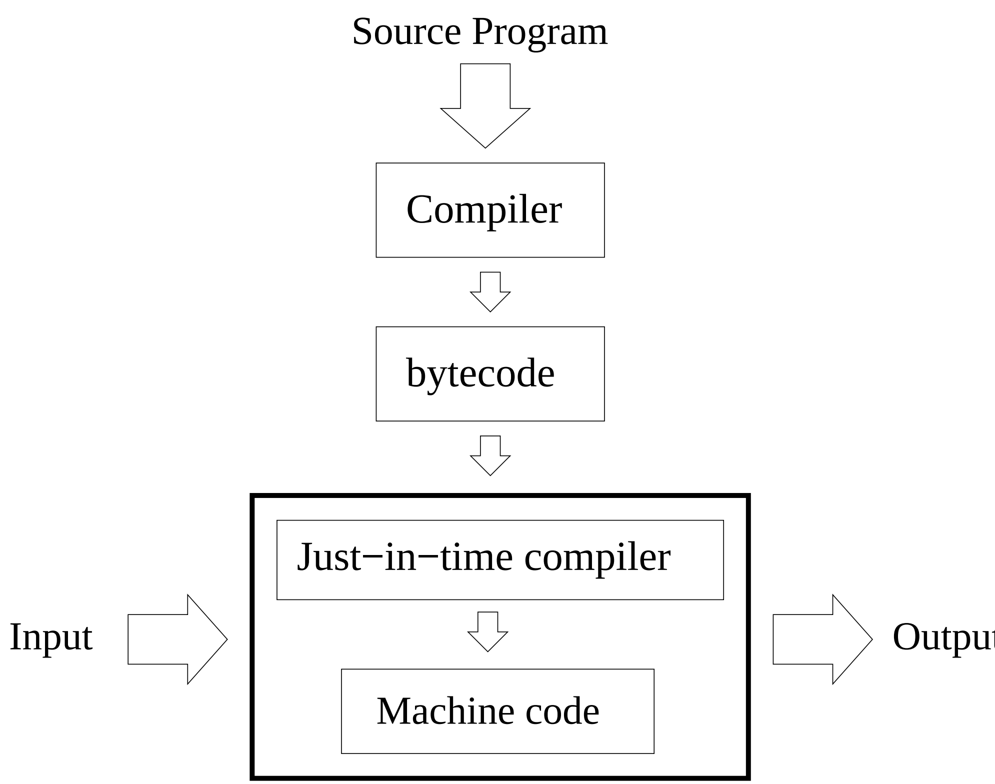
- The difference in execution time is usually one o more orders of magnitude
- But many times the interpretation time is lower than the compile time plus the execution time of the compiled code
- Interpreted languages: Java, Matlab, Mathematica, Lisp, Prolog, Perl, Python, Ruby, script languages (sh, csh, bash, tcl, awk, . . .)

Efficiency

- The difference in execution time is usually one o more orders of magnitude
- But many times the interpretation time is lower than the compile time plus the execution time of the compiled code
- Interpreted languages: Java, Matlab, Mathematica, Lisp, Prolog, Perl, Python, Ruby, script languages (sh, csh, bash, tcl, awk, . . .)
- Alternative: Compilation + Execution
  Examples: Java (bytecode), Perl, Python

Just-in-time (JIT) Systems

- The intermediate representation is compiled into native machine code at runtime
- More efficient but increases memory use
- Diagram:



Just-in-time (JIT) Systems

- The intermediate representation is compiled into native machine code at runtime
- More efficient but increases memory use
- *Adaptive optimization*: the interpreter analyzes the program profile and compiles the most frequently executed parts
- Examples: Java, Python, Smalltalk

Self-interpreters

- Written in the same language they interpret. An initial interpreter must be built in another language (bootstrapping). Ex: Smalltalk, Lisp/Scheme, Ruby, . . .

Self-interpreters

- Written in the same language they interpret. An initial interpreter must be built in another language (bootstrapping). Ex: Smalltalk, Lisp/Scheme, Ruby, ...
- Meta-circulars interpreters
  - Self-interpreters in which the key language constructs are implemented in term of themselves, thus no additional semantics is required for new constructs.

Self-interpreters

- Written in the same language they interpret. An initial interpreter must be built in another language (bootstrapping). Ex: Smalltalk, Lisp/Scheme, Ruby, . . .
- Meta-circulars interpreters
  - Self-interpreters in which the key language constructs are implemented in term of themselves, thus no additional semantics is required for new constructs.
  - Homoiconic Languages
    - Program representation is made in a basic structure of the language (homo [the same], iconic [representation])
    - Assume that a basic structure is the string. We can write strings containing function declarations, and apply these strings to other data structures (Snobol)

Self-interpreters

- Written in the same language they interpret. An initial interpreter must be built in another language (bootstrapping). Ex: Smalltalk, Lisp/Scheme, Ruby, . . .
- Meta-circulars interpreters
  - Self-interpreters in which the key language constructs are implemented in term of themselves, thus no additional semantics is required for new constructs.
  - Homoiconic Languages
    - Program representation is made in a basic structure of the language (homo [the same], iconic [representation])
    - Usually functional/logical languages: Lisp (s-expressions, *eval*), Prolog (terms, *unification*, *resolution*), . . .

Virtual Machines

- Security and portability.
  Independence wrt. platform: hardware and OS
- Some examples: JavaVM (bytecode), P-Machine (Pascal), Smalltalk, Warren Abstract Machine (Prolog), Macromedia Flash Player, Dalvik (Google Android), . . .

Virtual Machines

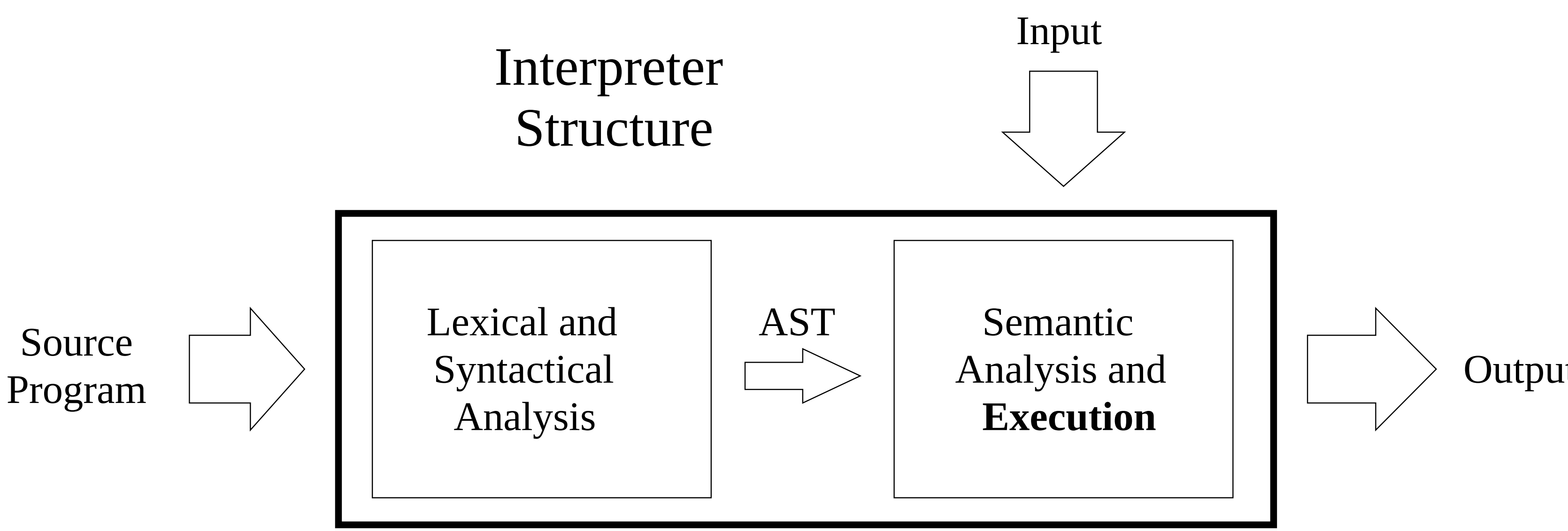
- Security and portability.
  Independence wrt. platform: hardware and OS
- Some examples: JavaVM (bytecode), P-Machine (Pascal), Smalltalk, Warren Abstract Machine (Prolog), Macromedia Flash Player, Dalvik (Google Android), . . .
- Architecture specifications
  - Memory model and management:
    static memory, runtime stack, heap, . . .
  - Instructions set

Virtual Machines

- Security and portability.
  Independence wrt. platform: hardware and OS
- Some examples: JavaVM (bytecode), P-Machine (Pascal), Smalltalk, Warren Abstract Machine (Prolog), Macromedia Flash Player, Dalvik (Google Android), . . .
- Architecture specifications
  - Memory model and management:
    static memory, runtime stack, heap, . . .
  - Instructions set
  - Instruction execution:
    - “program counter”
    - For each instruction: memory transformations produced
    - Control flow instructions

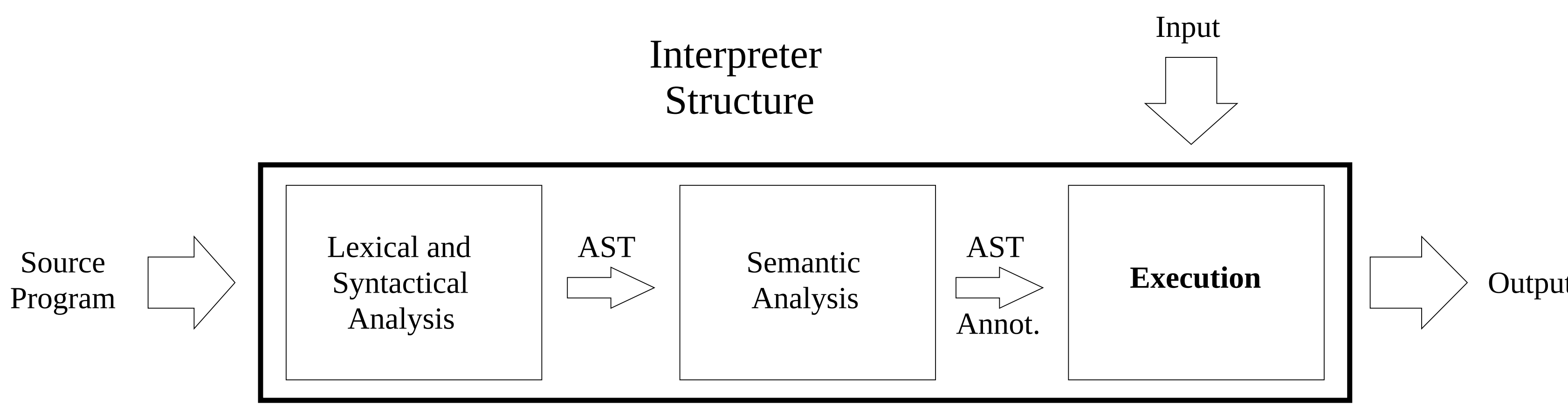
Usual Structure of an Interpreter

- Scanning (lexical analysis)
- Parsing (syntactic analysis) and AST construction
- Semantic analysis and AST execution.
  Diagram:



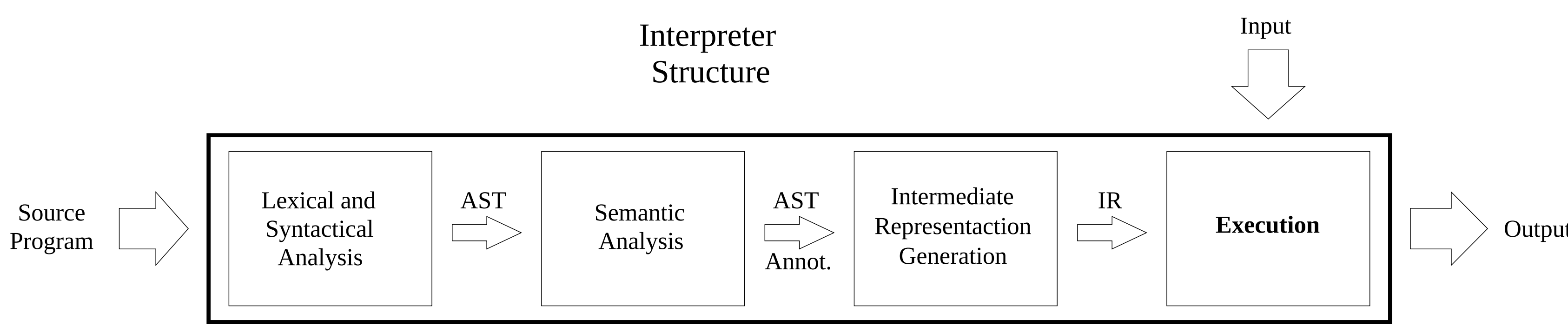
Usual Structure of an Interpreter

- Scanning (lexical analysis)
- Parsing (syntactic analysis) and AST construction
- Semantic analysis
- AST execution. Diagram:



Usual Structure of an Interpreter

- Scanning (lexical analysis)
- Parsing (syntactic analysis) and AST construction
- Semantic Analysis
- Intermediate representation (IR) generation
- IR execution. Diagram:



Simple Example (Grammar)

```
l_instrs
      :  ( instruction ) *
      ;
instruction
      :  IDENT   ASSIG^  expr
      |  WRITE^  expr
      ;
expr
      :  expr_simple   ( PLUS^  expr_simple ) *
      ;
expr_simple
      :  INTCONST
      |  IDENT
      ;
```

Simple Example (Interpreter)

```
map<string, int>  Mem;

int   evaluate( AST * a )   {
      if  ( a == NULL )   return 0;
      if  ( a->kind == "intconst" )   {
            return  atoi( a->text.c_str( ) );
      }  else  if  ( a->kind == "+" )   {
            v1 = evaluate( a->down );
            v2 = evaluate( a->down->right );
            return  v1 + v2;
      }  else  if  ( a->kind == "ident" )   {
            return  Mem[ a->text ];
      }
}
```

Simple Example (Interpreter)

```
map<string, int>  Mem;

void  execute( AST * a )  {
    if ( a == NULL )  return;
    if ( a->kind == ":=" )  {
        Mem[ a->down->text ] = evaluate( a->down->right );
    } else if ( a->kind == "write" )  {
        cout << evaluate( a->down ) << endl;
    }
    execute( a->right );        // executes the rest of siblings of a
}
```

Exercise

Complete the previous interpreter adding `if` and `while` instructions.

```
void  execute( AST * a )  {
    if ( a == NULL )  return;
    if ( a->kind == ":=" )  {
        Mem[ a->down->text ] = evaluate( a->down->right );
    } else if ( a->kind == "write" )  {
        cout << evaluate( a->down ) << endl;
    } else if ( a->kind == "if" )  {
    } else if ( a->kind == "while" )  {
    }
    execute( a->right );          // executes the rest of siblings of a
}
```

Exercise

Complete the previous interpreter adding `if` and `while` instructions.

```
void execute( AST * a ) {
            ⋮
    } else if ( a->kind == "if" ) {
        if ( evaluate( a->down ) != 0 ) {
            execute( a->down->right );
        }
    } else if ( a->kind == "while" ) {
```

Exercise

Complete the previous interpreter adding `if` and `while` instructions.

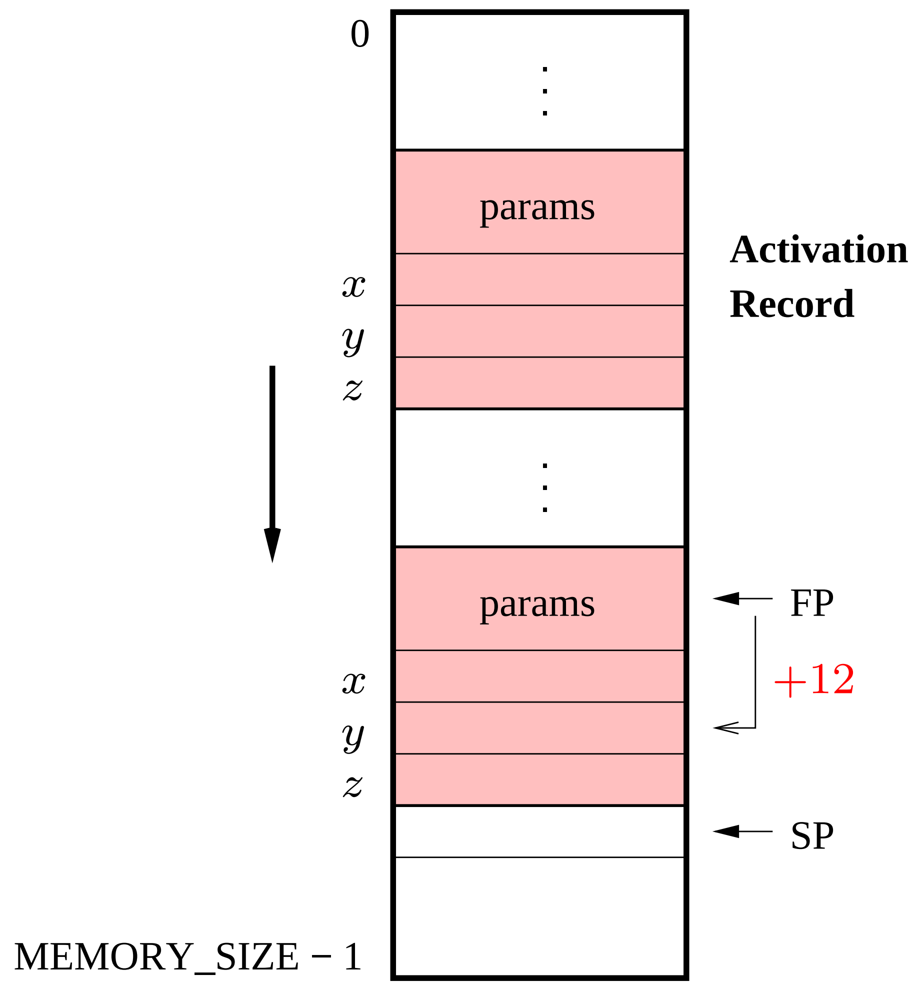
```
void execute( AST * a ) {
             ⋮
    } else if ( a->kind == "if" ) {
        if ( evaluate( a->down ) != 0 ) {
            execute( a->down->right );
        }
    } else if ( a->kind == "while" ) {
        while ( evaluate( a->down ) != 0 ) {
            execute( a->down->right );
        }
    }
             ⋮
```

Simple Example (adding Calls)

[Current] Symbol Table:

Ident	Offset
⋮	⋮
x	$+8$
y	$+12$
z	$+16$

Memory (Runtime Stack):



Simple Example (cont.)

```
int   Mem [ MEMORY_SIZE ];
int   SP, FP;                          // Stack Pointer, Frame Pointer

int   addressMemory( String  id )   {
      InfoSym info = SearchSymbol( CurrentSymbolTable,  id );
      return   FP  +  info->offset;
}
```

Simple Example (cont.)

```
int   Mem [ MEMORY_SIZE ];
int   SP, FP;                        // Stack Pointer, Frame Pointer

int   addressMemory( String  id )   {
      InfoSym info = SearchSymbol( CurrentSymbolTable,  id );
      return   FP  +  info->offset;
}

void   execute( AST * a )   {
                    ⋮
      if  ( a->kind == ":=" )   {
            addr = addressMemory( a->down->text );
            Mem[ addr ] = evaluate( a->down->right );
      }  else  if  ( a->kind == "write" )   {
                    ⋮
```

Simple Example (cont.)

```
int   Mem [ MEMORY_SIZE ];
int   SP, FP;                          // Stack Pointer, Frame Pointer

int   addressMemory( String  id )   {
      InfoSym info = SearchSymbol( CurrentSymbolTable,  id );
      return   FP  +  info−>offset;
}

int   evaluate( AST * a )   {
                  ⋮
      }  else  if  ( a−>kind == "ident" )   {
            addr = addressMemory( a−>text );
            return  Mem[ addr ];
      }
                  ⋮
```

Exercise

Complete the interpreter adding procedure calls.

```
void  execute( AST * a )  {
                 ⋮
    }  else  if  ( a−>kind == ”ident” )  {          // procedure call
        param = a−>down;                             // first parameter
        int SP0 = SP;  {
        while  ( param )  {
            . . .
            param = param−>right;                    // next parameter
        }
        proc = FindASTProcedure( a−>text );
        execute( proc );             // saving and restoring registers . . .
        . . .
    }
```

Exercise

Complete the interpreter adding procedure calls.

```
void  execute( AST * a )  {
                 ⋮
    } else if ( a->kind == "ident" )  {          // procedure call
        param = a->down;                         // first parameter
        int SP0 = SP;  {
        while ( param )  {
            push( evaluate( param ) );           // Mem[SP++] = ...
            param = param->right;                // next parameter
        }
        proc = FindASTProcedure( a->text );
        execute( proc );               // saving and restoring registers ...
        SP = SP0;                      // pop the params
    }
```