COMPILER DESIGN

Intermediate representations

Introduction to code generation

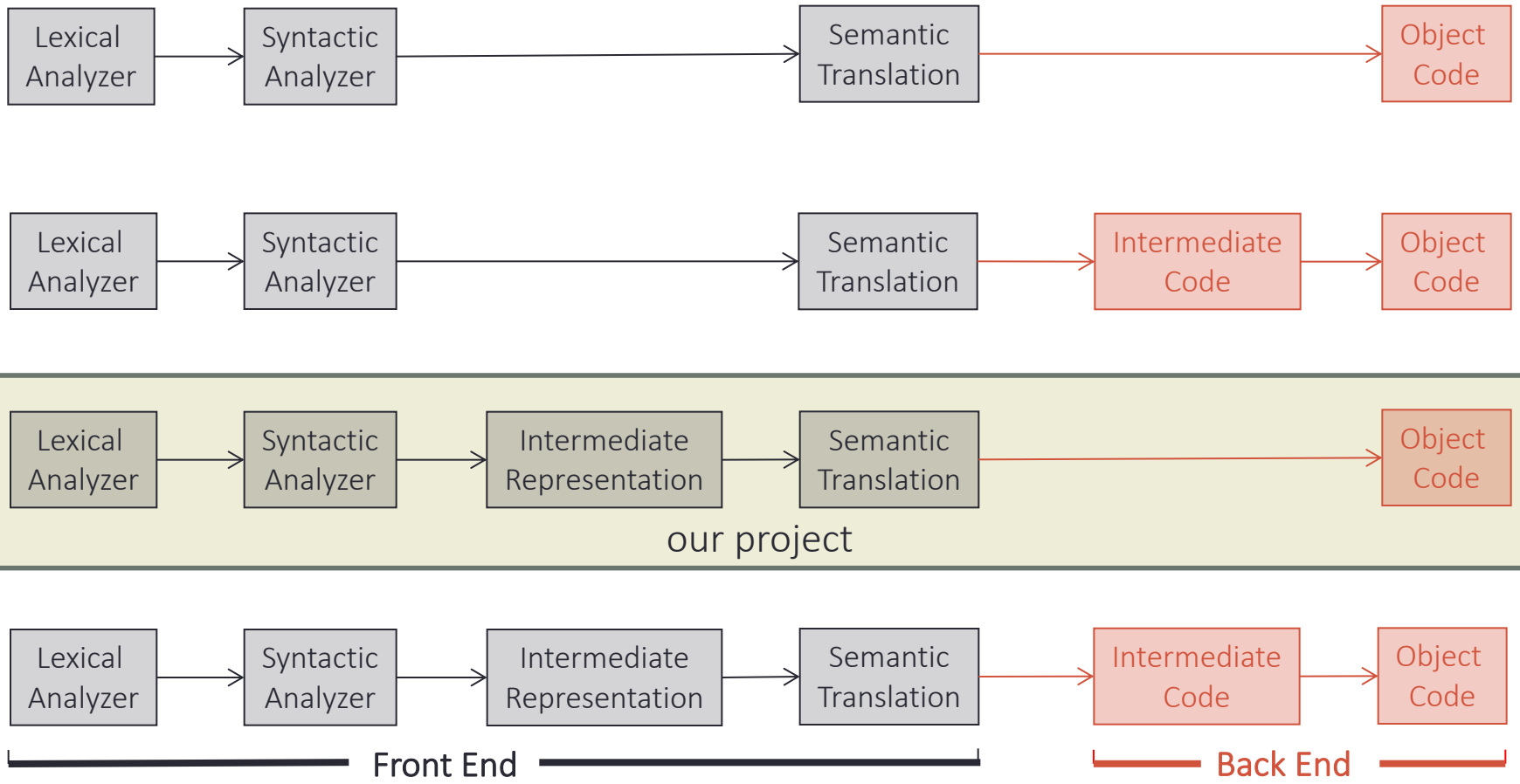
Introduction to code generation

- Front end:
  - Lexical Analysis
  - Syntactic Analysis
  - Intermediate Code/Representation Generation
  - Intermediate Code/Representation Optimization
  - Semantic Analysis
- Back end:
  - Object Code Generation
  - Object Code Optimization
- The front end is **machine-independent**, i.e. the decisions made in its processing do not depend on the target machine on which the translated program will be executed.
- A well-designed front end can be reused to build compilers for different target machines.
- The back end is **machine-dependent**, i.e. these steps are related to the nature of the assembly or machine language of the target architecture.

Introduction to code generation

- After syntactic analysis, we have a number of options to choose from:
  - generate object code directly from the parse.
  - generate intermediate code, and then generate object code from it.
  - generate an intermediate abstract representation, and then generate code directly from it.
  - generate an intermediate abstract representation, generate intermediate code, and then the object code.
- All these options have one thing in common: they are all based on syntactic information gathered/aggregated/processed/verified in the syntactic-semantic analysis.

Possible paths toward object code generation



Intermediate representations and intermediate code

- **Intermediate representations** synthetize the syntactic information gathered during the parse, generally in the form of a tree or directed graph.
- Intermediate representations enable high-level code optimization.
- In out project, we use an Abstract Syntax Tree (AST) as an intermediate representation.

- **Intermediate code** is a low-level coded (text) representation of the program, directly translatable to object code.
- Intermediate code enables low-level, architecture-dependent optimizations.
- We don't have intermediate code in our project.

Intermediate representations

Abstract syntax tree

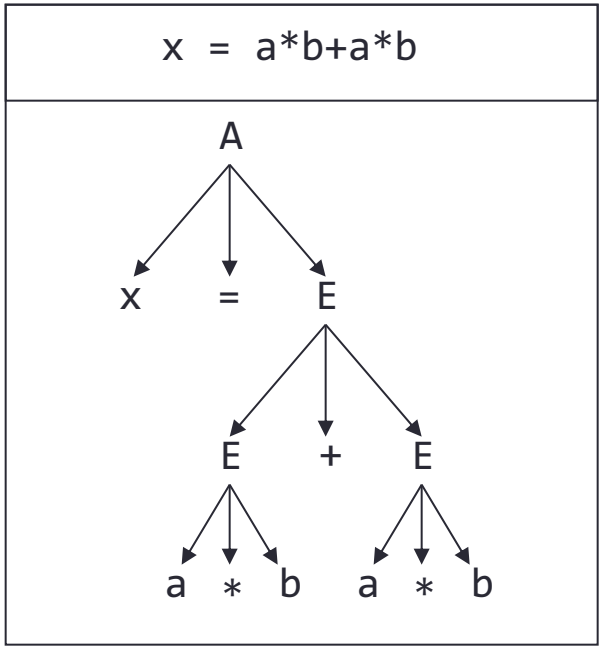
- Each node represents the application of a rule in the grammar.
- A subtree is created only after the complete parsing of a right hand side.
- References to subtrees are sent up and grafted as upper subtrees are completed.

- **Parse trees** (concrete syntax trees) emphasize the grammatical structure of the program, based on the exact concrete syntax of the grammar.
- **Abstract syntax trees** emphasize the actual computations to be performed. They do not refer to the actual non-terminals defined in the grammar, nor to tokens that play no role in defining the translated program, hence their name.

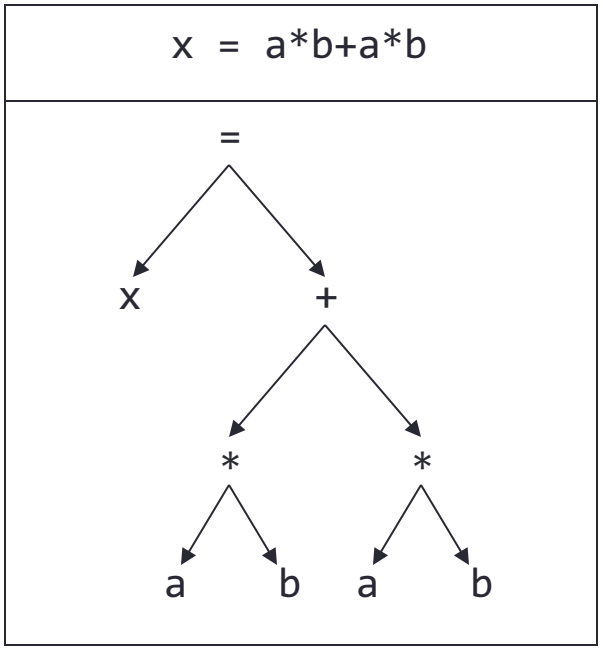
- When implementing an LL parser, there is generally a lot of differences between the parse tree and the corresponding abstract syntax tree.

Parse tree vs. abstract syntax tree

Parse Tree



Abstract Syntax Tree

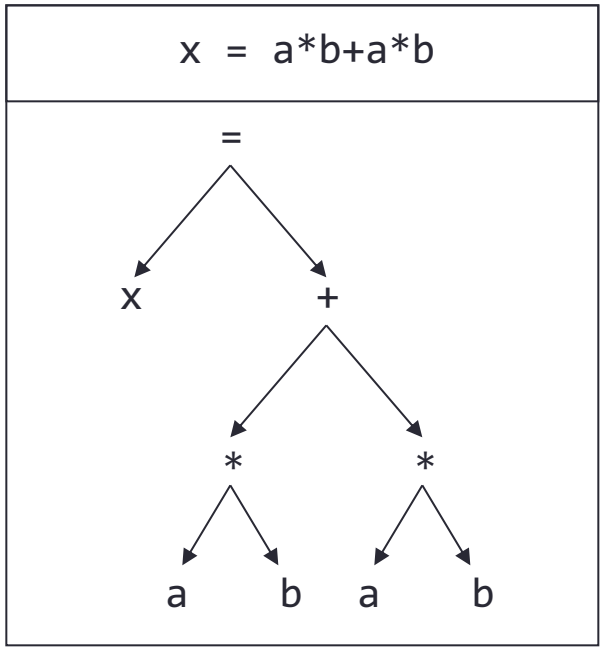


Directed acyclic graph

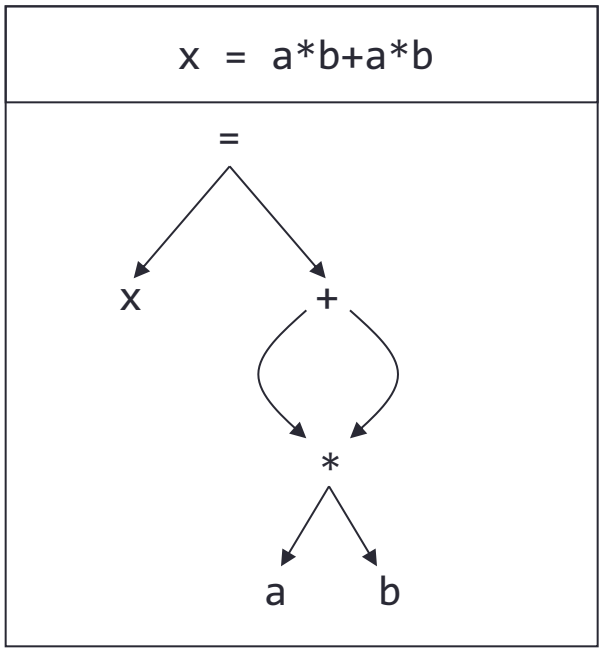
- **Directed acyclic graphs** (DAG) are a relative of syntax trees: they are used to show the syntactic structure of valid programs in the form of a graph with the following restrictions:
  - Edges between nodes can only be **unidirectional**.
  - There cannot be **cycles** in a DAG, i.e. no path can ever lead twice to the same node.
  - Presence of a **root node**.
- For example, in a DAG representation, the nodes for repeated variables and expressions are merged into a single node.
- DAGs are more complicated to build and use than syntax trees, but easily allows the implementation of a variety of **optimization** techniques by avoiding redundant operations.

Abstract syntax tree vs. directed acyclic graph

Abstract Syntax Tree



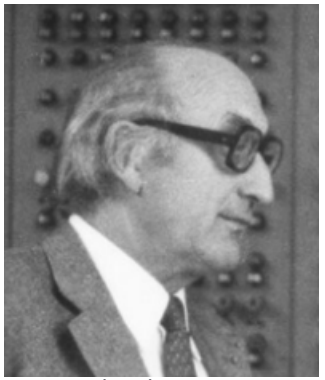
Directed Acyclic Graph



Postfix notation

- **Reverse Polish notation** (RPN), also known as **Polish postfix notation** or simply **postfix notation.**
- "Polish" refers to the nationality of logician Jan Łukasiewicz, who invented it in 1924.
- Mathematical notation in which operators follow their operands, in contrast to Polish notation, in which operators precede their operands.
- It does not need any parentheses as long as each operator has a fixed number of operands.
- The postfix notation was used by **Friedrich L. Bauer** and **Edsger W. Dijkstra** while working on the design of a compiler for the Algol language in the 1960s to reduce computer memory access and utilize a stack to evaluate expressions.

Edsger Dijkstra



Friedrich L. Bauer

Postfix notation

- Easy to generate from a bottom-up parse.
- Can be generated from an abstract syntax tree using post-order traversal.
- Some programming language constructs can be translated in post-fix notation.
- Edsger Dijstra invented the **shunting-yard algorithm** to translate them from infix notation to post-fix notation.
- This algorithm was later expanded into **operator precedence parsing**, which is essentially LR shift-reduce parsing without the notion of state.
- Much less powerful than LR, e.g. cannot deal with grammars that have more than one consecutive non-terminals nor epsilon in a right-hand-side.

```
a+b                                  ⇒ ab+
a+b*c                                ⇒ abc*+
if A then B else C                   ⇒ ABC?
if A then if B then C else D else E  ⇒ ABCD?E?
x=a*b+a*b                            ⇒ xab*ab*+=
```

Postfix notation

- Example of processing post-fix notation:
  - Expression in post-fix notation in black.
  - Current token in black bold.
  - Content of stack in red.

```
15 7 1 1 + - ÷ 3 × 2 1 1 + + - =
15 7 1 1 + - ÷ 3 × 2 1 1 + + - =
15 7 1 1 + - ÷ 3 × 2 1 1 + + - =
15 7 1 1 + - ÷ 3 × 2 1 1 + + - =
15 7 1 1 + - ÷ 3 × 2 1 1 + + - =
15 7   2 - ÷ 3 × 2 1 1 + + - =
15       5 ÷ 3 × 2 1 1 + + - =
           3 3 × 2 1 1 + + - =
           3 3 × 2 1 1 + + - =
               9 2 1 1 + + - =
               9 2 1 1 + + - =
               9 2 1 1 + + - =
               9 2 1 1 + + - =
               9 2   2 + + - =
               9       4 - - =
                       5 - =
                       5
```

- Stack-based evaluation of expressions expressed in post-fix notation:

```
for each token in the postfix expression:
  if token is an operator:
    operand_2 ← pop from the stack
    operand_1 ← pop from the stack
    result ← evaluate token with operand_1 and operand_2
    push result back onto the stack
  else if token is an operand:
    push token onto the stack
result ← pop from the stack
```

Three-address code

- Three-address codes (TAC or 3AC) is an intermediate language that maps directly to “assembly pseudo-code”, i.e. architecture-dependent assembly code.
- It breaks the program into short statements requiring no more than three variables (hence its name) and no more than one operator.

- As it is an intermediate (abstract) language, its “addresses” represent symbolic addresses (i.e. variables), as opposed to either registers or memory addresses that would be used by the target machine code.

- These characteristics allows 3AC to:
  - be more abstract than assembly language, enabling optimizations at the higher abstract level.
  - have very high resemblance to assembly language, enabling very easy translation to assembly language.

Three-address code

source	3AC
`x = a+b*c`	`t := b*c` `x := a+t`

`x = (-b+sqrt(b^2-4*a*c))/(2*a)`	`t1 := b * b`
	`t2 := 4 * a`
	`t3 := t2 * c`
	`t4 := t1 - t3`
	`t5 := sqrt(t4)`
	`t6 := 0 - b`
	`t7 := t5 + t6`
	`t8 := 2 * a`
	`t9 := t7 / t8`
	`x := t9`

`for (i = 0; i < 10; ++i) {`
`  b[i] = i*i;`
`}`
`...`
	`t1 := 0`
	`L1: if t1 >= 10 goto L2`
	`t2 := t1 * t1`
	`t3 := t1 * 4`
	`t4 := b + t3`
	`*t4 := t2`
	`t1 := t1 + 1`
	`goto L1`
	`L2: ...`

Three-address code

- The temporary variables are generated at compile time and may be added to the symbol table.
- In the generated code, the variables will refer to actual memory cells. Their address (or alias) may also be stored in the symbol table.

3AC	ASM
t := b*c	L 3,b M 3,c ST 3,t
x := a+t	L 3,a A 3,t ST 3,x

- 3AC can also be represented as **quadruples**, which are even more related to assembly languages.

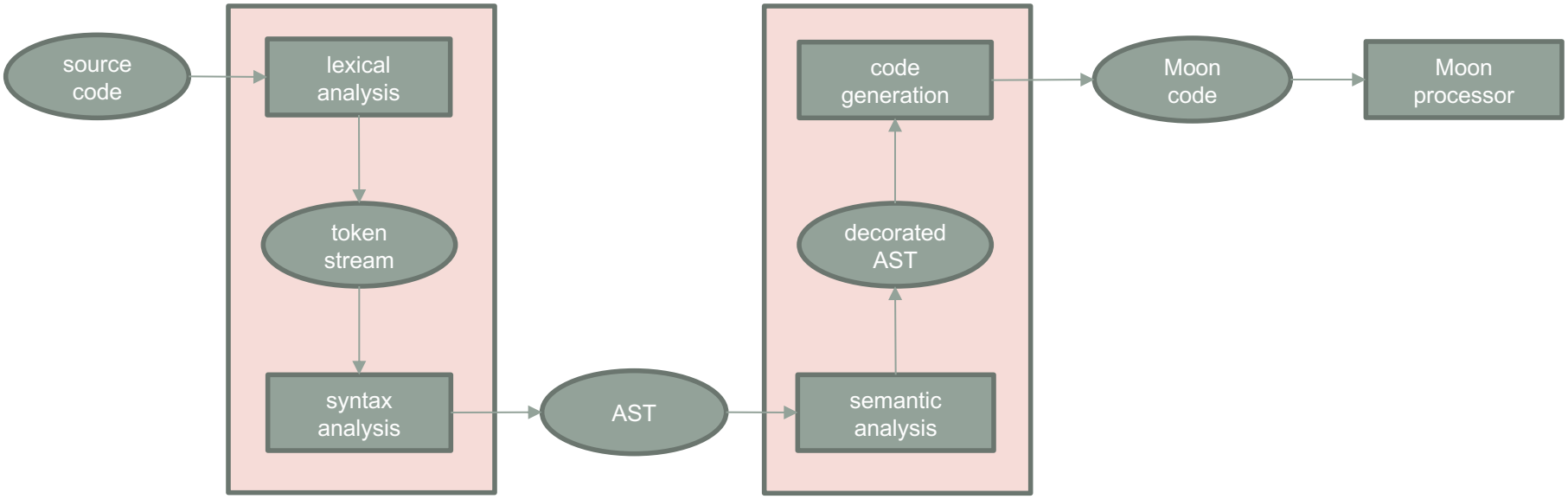
3AC	Quadruples
t := b*c	MULT t,b,c
x := a+t	ADD x,a,t

Intermediate languages

- In this case, we generate code in a language for which we already have a compiler or interpreter.
- Such languages are generally low-level and dedicated to the compiler construction task.
- It provides the compiler writer with a “virtual machine”.
- Various compilers can be built using the same virtual machine.
- The virtual machine compiler can be compiled on different machines to provide a translator to various architectures.
- Many contemporary languages, such as Java, Perl, PHP, Python and Ruby use a similar execution architecture.
- For the project, we have the Moon processor, which provides a virtual assembly language and a compiler/interpreter for that language.

Project architectural overview

Project architectural overview

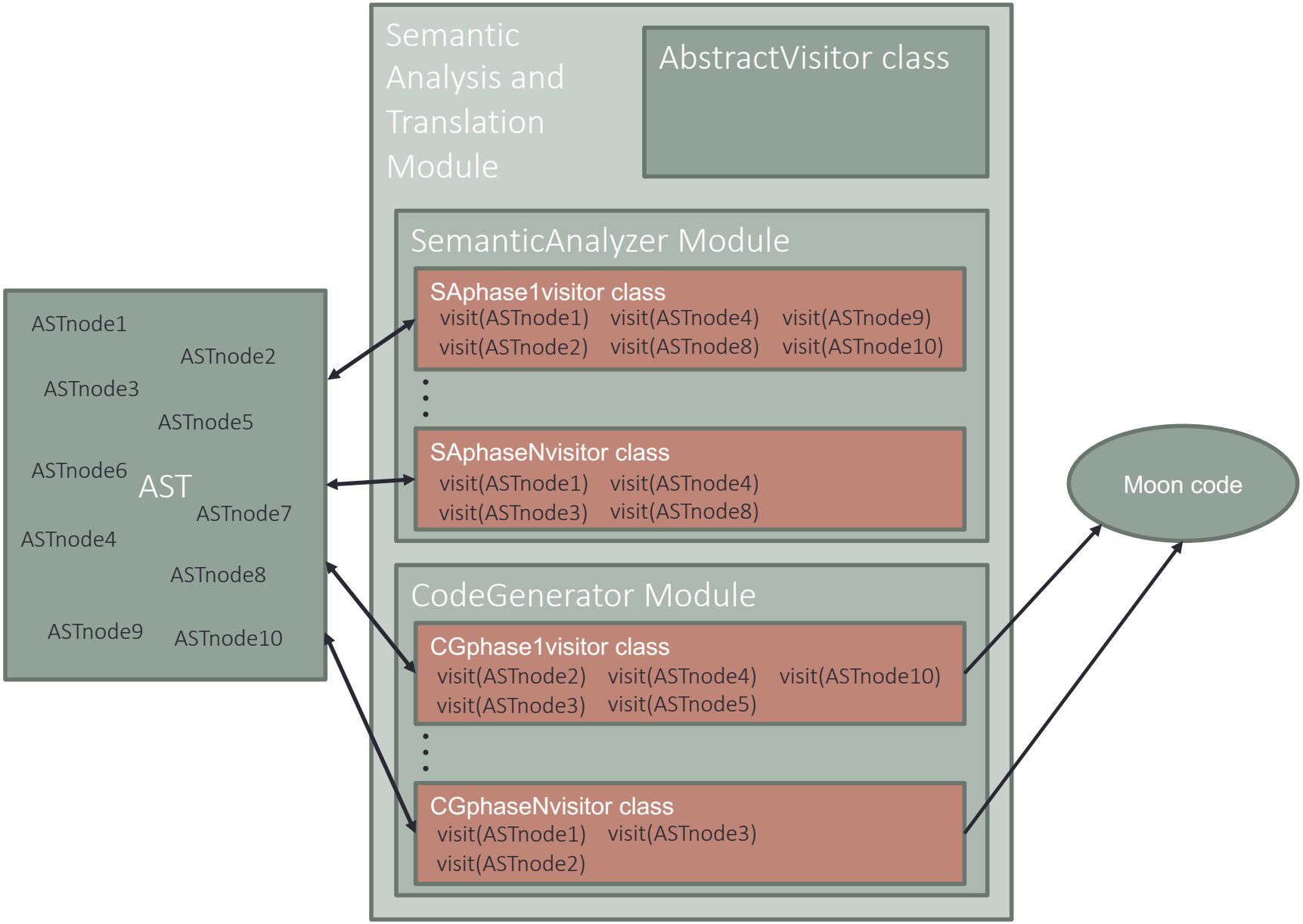


- Your compiler generates Moon code.
- The Moon processor (virtual machine) is used to execute your output program.
- The moon processor is written as a C program.
- Your compiler is thus retargetable by recompilation of the Moon compiler on your target processor.

Project architectural overview

- Semantic verification/translation is done using Semantic Actions.
- As the AST is traversed, semantic actions applicable to the currently traversed node type are triggered.
- Many semantic actions need to be used in order to do the entire semantic verification/translation.
- As some semantic actions need to be applied prior to others, you need to organize them in phases.
- A separate Semantic Actions module should be created that consists of a library of functions that the compiler will call as three AST is traversed. Ideally, there should be one such module for each phase, and each phase is externally triggered in a particular sequence.
- For example, semantic verification actions and semantic translation actions can be separated from each other.
- This can easily be done using the Visitor pattern.

Project architectural overview



Semantic actions and code generation

Semantic actions

- Semantics is about giving a meaning to the compiled program.
- Semantic actions have two counterparts:
  - Semantic checking actions: check if the compiled program can have a meaning, e.g identifiers are declared and properly used, operators and functions have the right parameter types and number of parameters upon calling.
  - Semantic translation actions: translate declarations, expressions, statements and functions to target code.
- Semantic translation is conditional to semantic checking, i.e. if a program is invalid, the generated code would be invalid.
- There is no such thing as code generation errors detection/reporting.

Semantic actions

- There are semantic actions associated with:
  - Declarations:
    - variable declarations
    - type declarations
    - function declarations
  - Control structures:
    - conditional statements
    - loop statements
    - function calls
  - Assignments and expressions:
    - assignment operations
    - arithmetic and logical expressions
- This applies to strongly typed procedural programming.
- Other programming language paradigms often have to be analyzed/translated differently.

Processing declarations

Processing declarations

- Semantic checking
  - In processing declarations, the only semantic checking there is to do is to ensure that every identifier (e.g. variable, type, class, function, etc.) is declared once and only once in the same scope.
  - This restriction is tested using the symbol table mechanism.
  - Symbol table entries are generated as declarations are encountered.
  - A symbol table is created every time a scope is entered.
  - Afterwards, every time an identifier is encountered, a check is made in the symbol table to ensure that:
    - It has been properly defined in the scope where the identifier is encountered.
    - It is properly used according to its type as stored in the symbol table.

Processing declarations

- Semantic translation
  - Code generation in type declarations comes in the form of calculation of the total memory **size** to be allocated for the objects defined.
  - Every variable defined will eventually have to be stored in the computer's memory.
  - Memory allocation must be done according to the size of the variables defined, the data encoding used, and the word length of the computer, which depends on the target machine.
  - For each variable identifier declared you either:
    - Generate a **unique label** that will be used to refer to that variable in the Moon code (tags-based approach).
    - Use a stack-based memory allocation scheme, where all references to variables have to be done using **offset** calculation assuming that all variables are stored in a function call stack frame.
  - If recursion is allowed, stack-based memory allocation is necessary.
  - Examples that follow are using a unique labels non-recursive scheme.
  - Stack-based operation will be explained in more details in the next lecture.

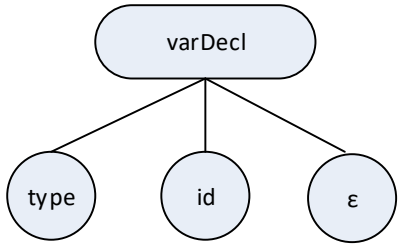
Processing variable declarations

- Semantic checking
  - Verify that an entry has been entered in the symbol table corresponding to the current scope.

- Semantic translation
  - The label corresponding to the variable, or its offset versus the current scope's starting point, is stored in the symbol table entry of the variable. In the case of arrays, the elements' offsets (size of the elements, rows, etc) are often stored in the symbol table record, though it can be calculated from the array's type and dimension list.
  - Such information can also be put in the AST nodes, though it may clutter the AST node structure.
  - If a label was assigned to the variable, then memory space is reserved for the variable according to the size of the type of the variable and linked to a unique label (or offset) in the Moon code.
  - If a stack-operated mode is used, it is just assumed that the variable's value is stored at address stackframe+offset of all the variables preceding it.

Processing variable declarations (basic types)

- Variable declaration (basic type)



- sizeof(int) = 32 bits (4 bytes)
- sizeof(float) = needs to be encoded (non-native)

- Implementation as a visitor method (non-array):

```
public void visit(VarDeclNode node){
  // First, propagate accepting the same visitor to all the children
  // This effectively achieves Depth-First AST Traversal
  for (Node child : node.getChildren() )
    child.accept(this);
  // Then, do the processing of this nodes' visitor
  if (node.getChildren().get(0).getData() == "int")
    moonDataCode += "        % space for variable " + node.getChildren().get(1).getData() + "\n";
    moonDataCode += String.format("%-7s" ,node.getChildren().get(1).getData()) + " res 4\n";
}
```

```
int a;
int b;
int c;
```

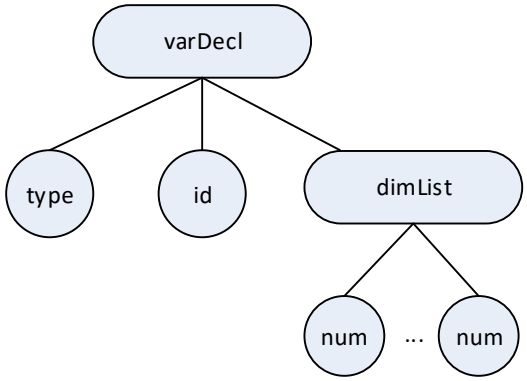
```
        % space for variable a
a       res 4
        % space for variable b
b       res 4
        % space for variable c
c       res 4
```

Processing variables declarations (composite data structures)

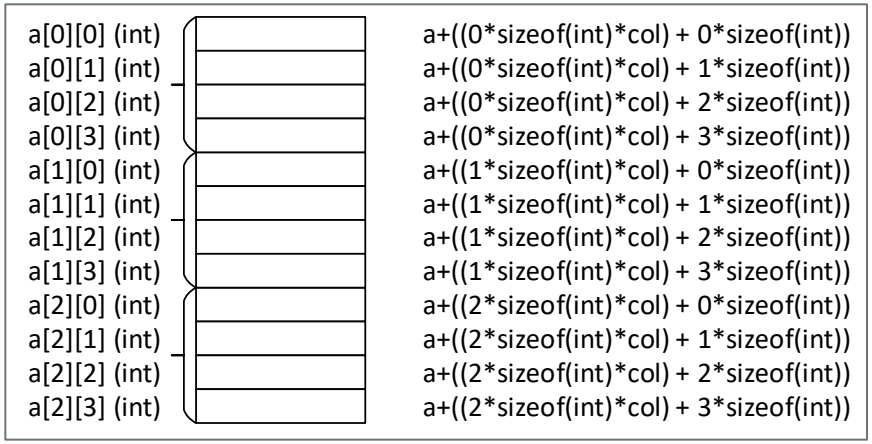
- All variables need to be declared so that they are of fixed size. This is why array dimensions need to be constants.
- This restriction comes from the fact that the memory allocated to the array has to be set at compile time, and be fixed throughout the execution of the program.
- When processing an array declaration, a sufficient amount of memory is allocated to the variable depending on the size of the elements and the cardinality of the array.
- Similarly for objects, memory is assigned for all the data members available to objects of their class.
- Addressing is stored in the symbol table. This is either a Moon label used to refer to the starting address of a data element, or an offset computed from the size of all the data elements coming before it in the current scope. The size of the entire data structure also needs to be stored in the symbol table. This facilitates code generation of array indexing or object member access during code generation, as well as the offset calculation of other variables coming after this one in the current scope.
- Variables of dynamic size are generally implemented using pointers, dynamic memory allocation functions and an execution stack or heap, which requires the implementation of a runtime system to execute the programs. For simplicity, we don't have them in the project.

Processing variable declarations (arrays)

- Variable declaration (arrays)



- sizeof(array) = sizeof(type)
  * num_1
  * num_2
  ...
  * num_n



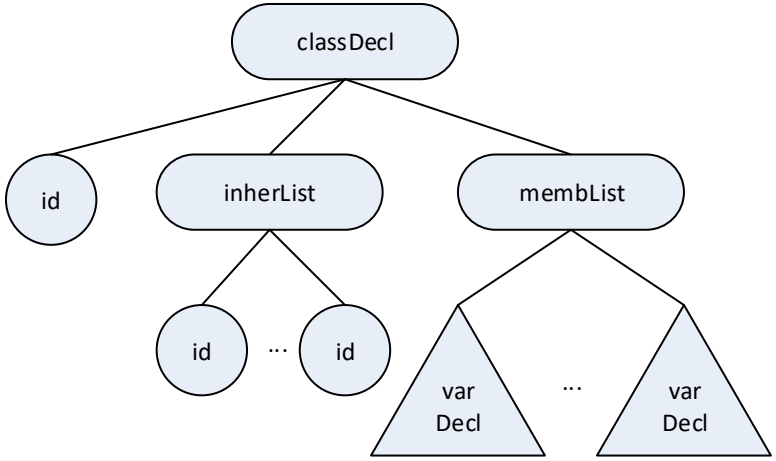

```
int a[3][4];
int b[2];
int c[2][2][2];
```

```
        % space for variable a
a       res 48
        % space for variable b
b       res 8
        % space for variable c
c       res 32
```

Processing variable declarations (objects)

- Variable declaration (objects)

- sizeof(object) = for each inherited class
  size += classSize
  for each member variable
  size += memberSize

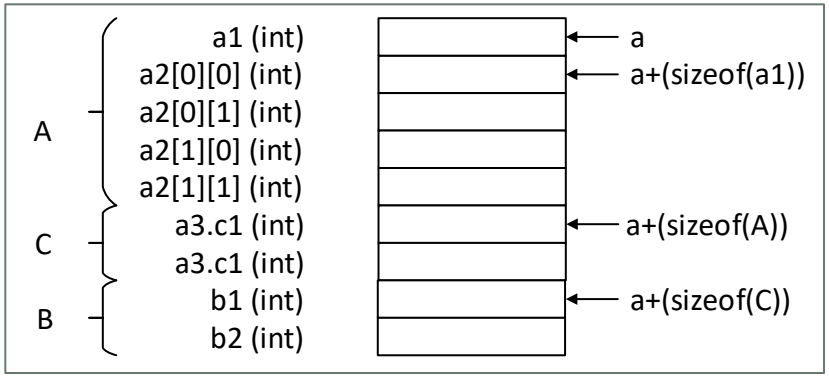



```
class A : B {
  int a1;
  int a2[2][2];
  C a3;
}

class B {
  int b1;
  int b2;
}

class C {
  int c1;
  int c2;
}

program {
  A a;
}
```




```
        % space for variable a
a       res 36
```

Processing expressions/assignment statements

- Semantic records contain the **type** and **location** for variables (i.e. labels in the Moon code, or offsets) or the **type** and **value** for constant factors.
- Semantic records are created at the leaves of the tree when factors are recognized and their information is retrieved, e.g. from the symbol table record for variables, or from the token contained in the AST for literal values.
- Some of the information in these semantic records needs to be migrated up for processing of:
  - Semantic checking, e.g. check if operands' types are valid for the operator
  - Semantic translation, generate code, or reserve memory/calculate offsets.

- For example, when processing expressions, the type of all sub-expressions should be determined and stored in a semantic record made accessible in the root node of the subtree, so that it can be accessed by their parent node.
- Similarly, temporary variables need to be created/allocated to store each temporary result or a subtree. Either a label or is created or an offset is calculated for them and also made available to the parent node.

Processing expressions/assignment statements

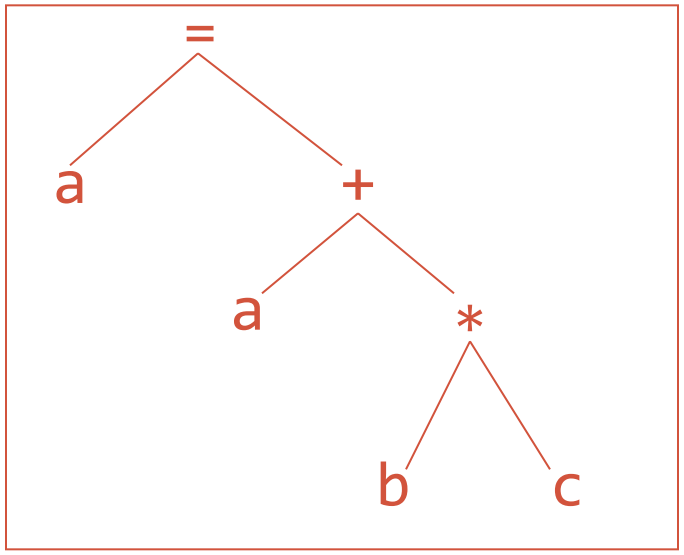
- Each time an operator node is traversed, its corresponding semantic checking and translation is done and its subresult is stored in a temporary memory location for which you have to allocate some memory and generated a label or computed its offset.
- An entry can then be inserted in the symbol table for each intermediate result generated. If so, it can then be used for further reference when doing semantic verification and translation while continuing AST tree traversal.

```
program{
  int a;
  int b;
  int c;
  a = 1;
  put(a);
  b = 2;
  put(b);
  c = 3;
  put(c);
  a = a + b * c;
  put(a + 6);
} // result = 13
```

```
=================================================
| table: global           scope size: 0         |
=================================================
| func      | program   | void                    |
|    ===========================================
|    | table: program          scope size: 40   |
|    ===========================================
|    | var       | a         | int     | 4 | 0  |
|    | var       | b         | int     | 4 | 4  |
|    | var       | c         | int     | 4 | 8  |
|    | litval    | t1        | int     | 4 | 12 |
|    | litval    | t2        | int     | 4 | 16 |
|    | litval    | t3        | int     | 4 | 20 |
|    | tempvar   | t4        | int     | 4 | 24 |
|    | tempvar   | t5        | int     | 4 | 28 |
|    | litval    | t6        | int     | 4 | 32 |
|    | tempvar   | t7        | int     | 4 | 36 |
|    ===========================================
=================================================
```

Processing expressions/assignment statements

- The code is generated sequentially as the tree is traversed:



Subtree	Moon code

t1 = b*c	lw r1,b(r0)
	lw r2,c(r0)
	mul r3,r1,r2
	sw t1(r0), r3

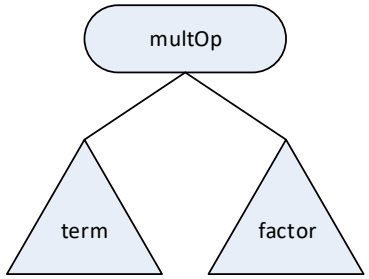
t2 = a+t1	lw r1,a(r0)
	lw r2,t1(r0)
	add r3,r1,r2
	sw t2(r0), r3

x = t2	lw r1,t2(r0) sw a(r0),r1

Processing expressions/assignment statements

- Processing a multiplication node

```
public void visit(MultOpNode node){
  // First, propagate accepting the same visitor to all the children
  // This effectively achieves Depth-First AST Traversal
  for (Node child : node.getChildren() )
    child.accept(this);
  // Then, do the processing of this nodes' visitor
  // create a local variable and allocate a register to this subcomputation
  node.localRegister      = this.registerPool.pop();
  node.leftChildRegister  = this.registerPool.pop();
  node.rightChildRegister = this.registerPool.pop();
  node.moonVarName        = this.getNewTempVarName();
  // generate code
  moonExecCode += "        lw " + node.leftChildRegister + "," + node.getChildren().get(0).moonVarName + "(r0)\n";
  moonExecCode += "        lw " + node.rightChildRegister + "," + node.getChildren().get(1).moonVarName + "(r0)\n";
  moonExecCode += "        mul " + node.localRegister + "," + node.leftChildRegister + "," + node.rightChildRegister + "\n";
  moonDataCode += String.format("%-7s",node.moonVarName) + " dw 0\n";
  moonExecCode += "        sw " + node.moonVarName + "(r0)," + node.localRegister + "\n";
  // deallocate the registers for the two children, and the current node
  this.registerPool.push(node.leftChildRegister);
  this.registerPool.push(node.rightChildRegister);
  this.registerPool.push(node.localRegister);
}
```

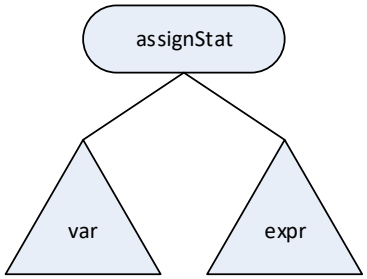


- Expects the Moon labels to be stored in the nodes.
  - Here, for variables, Moon labels are the same as variable names, not always the case.
  - In cases where the factor subtree is a tree, its result needs to be stored as a temporary value. Here, temporary results are stored in labeled memory cells. If using a stack mode of operation, the temporary memory location's offset needs to be used to get to the stored value before it can be loaded.

Processing expressions/assignment statements

- Processing an assignment statement node

```
public void visit(AssignStatNode node){
  // First, propagate accepting the same visitor to all the children
  // This effectively achieves Depth-First AST Traversal
  for (Node child : node.getChildren() )
    child.accept(this);
  // Then, do the processing of this nodes' visitor
  // allocate local register
  node.localRegister = this.registerPool.pop();
  //generate code
  moonExecCode += "        lw " + node.localRegister + "," + node.getChildren().get(1).moonVarName + "(r0)\n";
  moonExecCode += "        sw " + node.getChildren().get(0).moonVarName + "(r0)," + node.localRegister + "\n";
  //deallocate local register
  this.registerPool.push(node.localRegister);
}
```



Processing expressions/assignment statements

- A solution needs to be implemented for:
  - **Register allocation** : many operations use registers. You need a facility to allocate registers for operations and deallocate them after they have been used.
  - **Moon tag name generation**: If you want to use Moon tags for your temporary variables, you can implement that as a method of the visitor that implements the phase that uses name tag generation.

```
public class CodeGenerationVisitor extends Visitor {

    public Stack<String> registerPool = new Stack<String>();
    public Integer       tempVarNum   = 0;
    public String        moonExecCode = new String(); // moon instructions part
    public String        moonDataCode = new String(); // moon data part

    public CodeGenerationVisitor() {
    // create a pool of registers as a stack of Strings
    // assuming only r1, ..., r12 are available
    for (Integer i = 12; i>=1; i--)
    registerPool.push("r" + i.toString());
    }

    public String getNewTempVarName(){
    tempVarNum++;
    return "t" + tempVarNum.toString();
    }
…
}
```

- These facilities are local to this visitor and are hidden from other visitors. Which is good modularity.

Conclusions

- Most compilers build an intermediate representation of the parsed program, normally as an abstract syntax tree.
- These will allow high-level optimizations to occur before the code is generated.
- In the project, we are outputting Moon code, which is an intermediate language.
- Moon code could be the subject of low-level optimizations.

- Semantic actions are composed of *semantic checking*, and *semantic translation* counterparts.
- Semantic actions are triggered upon reaching certain node types when the AST is traversed. This generally requires a double dispatch mechanism, which is not available in most languages.
- The Visitor pattern is a very appropriate solution to:
  - Achieve double dispatch
  - Allow great modularity by grouping semantic actions in different phases.
  - Decorate the AST nodes with accumulated information used by further phases.
  - Store information internally to the current visitor, so that other visitors are not cluttered with information that is not pertinent to them.

References

- Fischer, Cytron, LeBlanc. Crafting a Compiler. Chapter 7, 8, 9, 10, 11. Addison-Wesley, 2010.
- Wikipedia. Reverse Polish notation.
- Wikipedia. Directed Acyclic Graph.
- Dijkstra, E.W. (1961). Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60. Stichting Mathematisch Centrum. Rekenafdeling. Stichting Mathematisch Centrum.