Compiler Compiler Tutorial

CSA2010 – Compiler Techniques

Gordon Mangion

Introduction

- With so many Compilers around, do we need to write parsers/compilers in industry?

- FTP interface
- Translator from VB to Java
- EPS Reader
- NLP

Topics

- Prerequisites
- Compiler Architecture/Modules
- JavaCC
- Virtual Machines
- Semantic Analysis
- Code Generation and Execution
- Examples
- The assignment (SL Compiler)

Prerequisites

- Java
- Regular Expressions
  - ? + * …
- Production rules and EBNF
- Semantics

Regular Expressions

- Repetitions
  - \+ = 1 or more
  - * = 0 or more
  - ? = 0 or 1
- Examples
  - a+ - a, aa, aaa
  - b* - , b, bb
  - c? - , c

- Alternative
  - a | b = a or b
- Examples
  - a | b - a, b

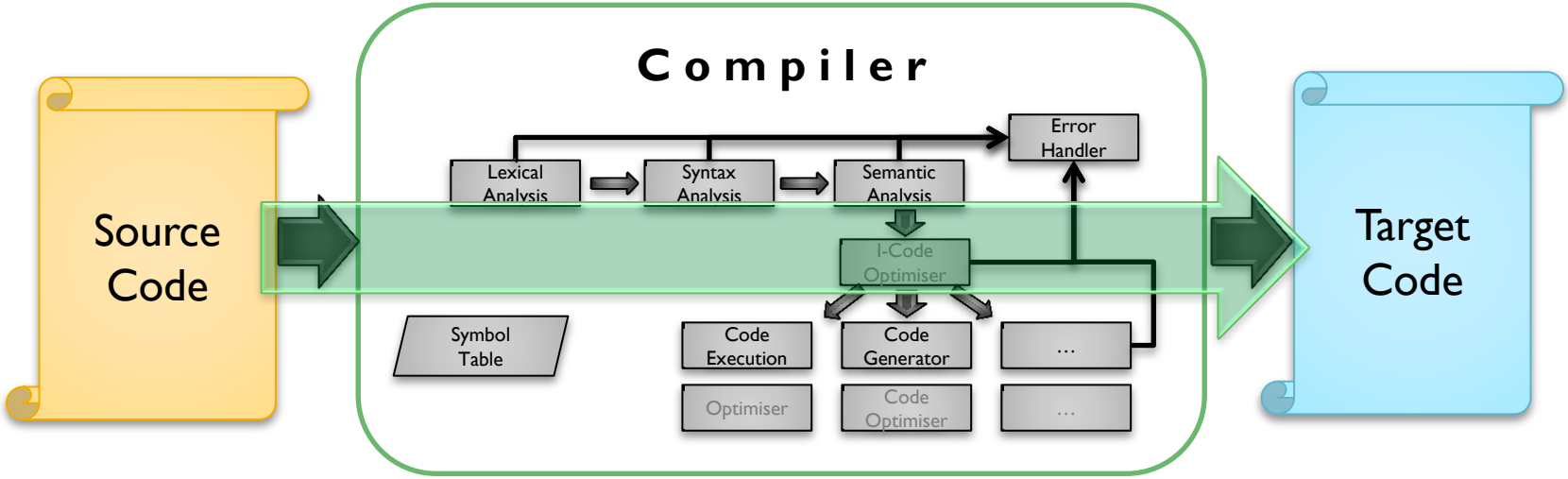
- Ranges
  - [a-z] = a to z - a,b,c,d,e,f,g,…,y,z
  - [fls] = f, l and s - f,l,s
  - [^cde] = not c,d,e - a,b,f,g,…,y,z

Compiler

- What is a compiler?
  - Where to start from?
  - Design / Architecture of a Compiler

Compiler

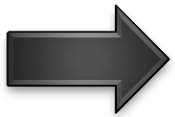
- Is essentially a complex **function** which **maps** a program in a **source** language onto a program in the **target** language.



Translation

Source Code

```
int Sqr( int x )
{
    var int n = x;
    n = n * n;
    return n;
}
```



Target Code

```
push        ebp
mov         ebp,esp
sub         esp,0CCh
push        ebx
push        esi
push        edi
lea         edi,[ebp-0CCh]
mov         ecx,33h
mov         eax,0CCCCCCCCh
rep stos    dword ptr es:[edi]
mov         eax,dword ptr [x]
mov         dword ptr [n],eax
mov         eax,dword ptr [n]
imul        eax,dword ptr [n]
mov         dword ptr [n],eax
mov         eax,dword ptr [n]
pop         edi
pop         esi
pop         ebx
mov         esp,ebp
pop         ebp
ret
```

Translation

Source Code

```
int Sqr( int x )
{
    var int n = x;
    n = n * n;
    return n;
}
```

Compiler

Translation

Source Code

```
int Sqr( int x )
{
    var int n = x;
    n = n * n;
    return n;
}
```

Target Code

```
push        ebp
mov         ebp,esp
sub         esp,0CCh
push        ebx
push        esi
push        edi
lea         edi,[ebp-0CCh]
mov         ecx,33h
mov         eax,0CCCCCCCCh
rep stos    dword ptr es:[edi]

mov         eax,dword ptr [x]
mov         dword ptr [n],eax

mov         eax,dword ptr [n]
imul        eax,dword ptr [n]
mov         dword ptr [n],eax

mov         eax,dword ptr [n]

pop         edi
pop         esi
pop         ebx
mov         esp,ebp
pop         ebp
ret
```

Production Rules

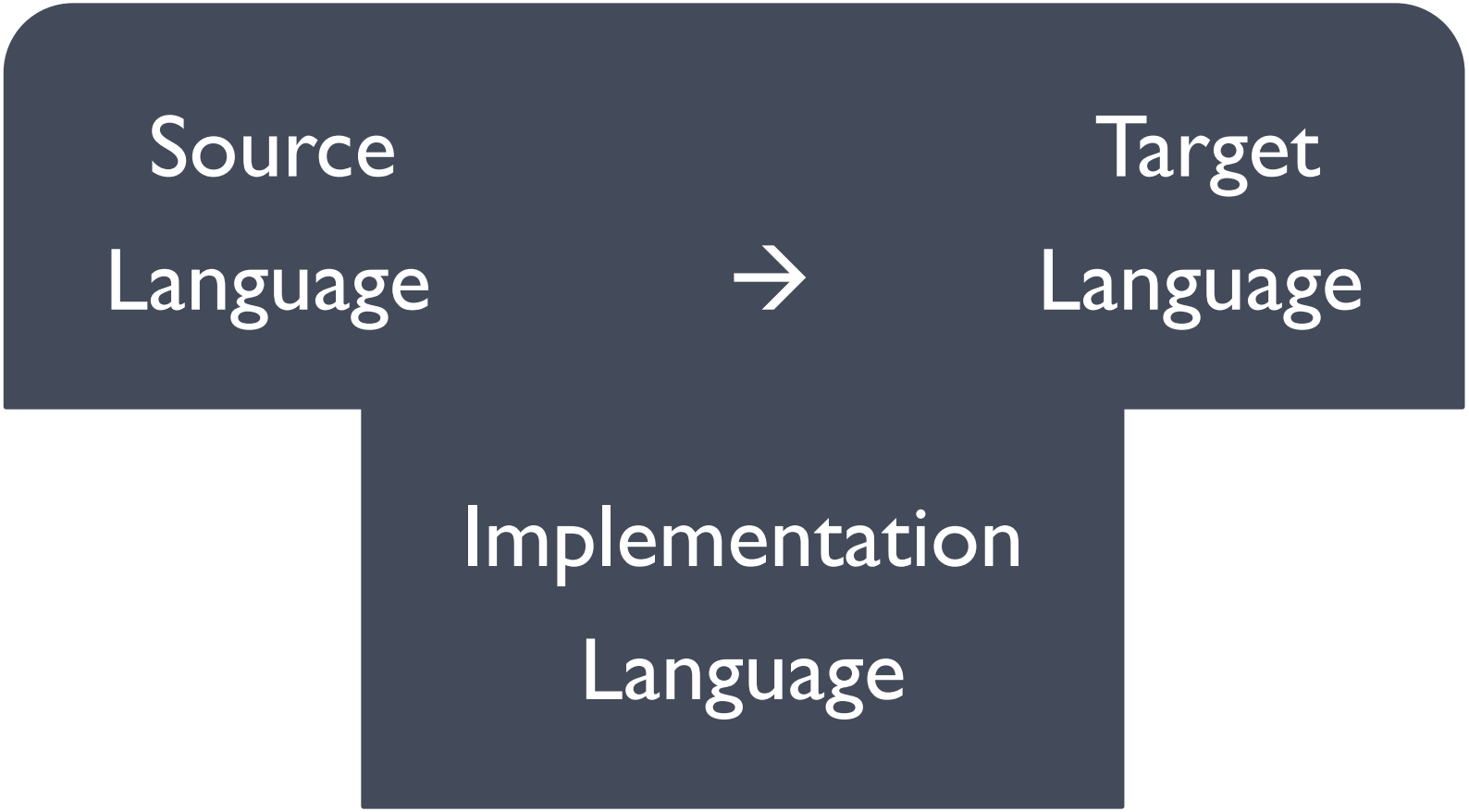
- Context Free Grammars
  - A → B “c” D
- BNF
  - A ::= B “c” D
- EBNF
  - A ::= B* “c”? D+

The Language game

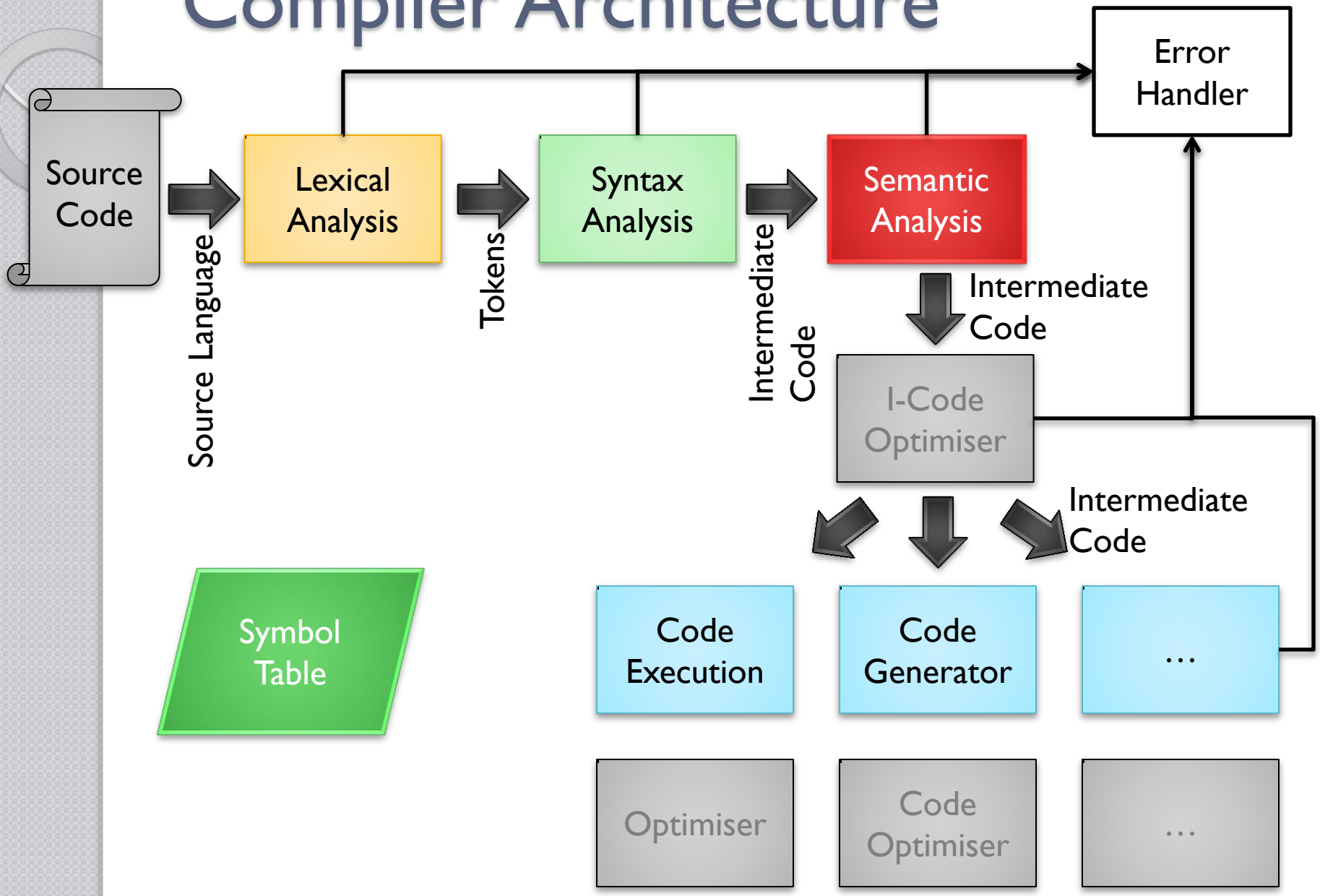
- Source Language
- Target Language
- Implementation Language

…

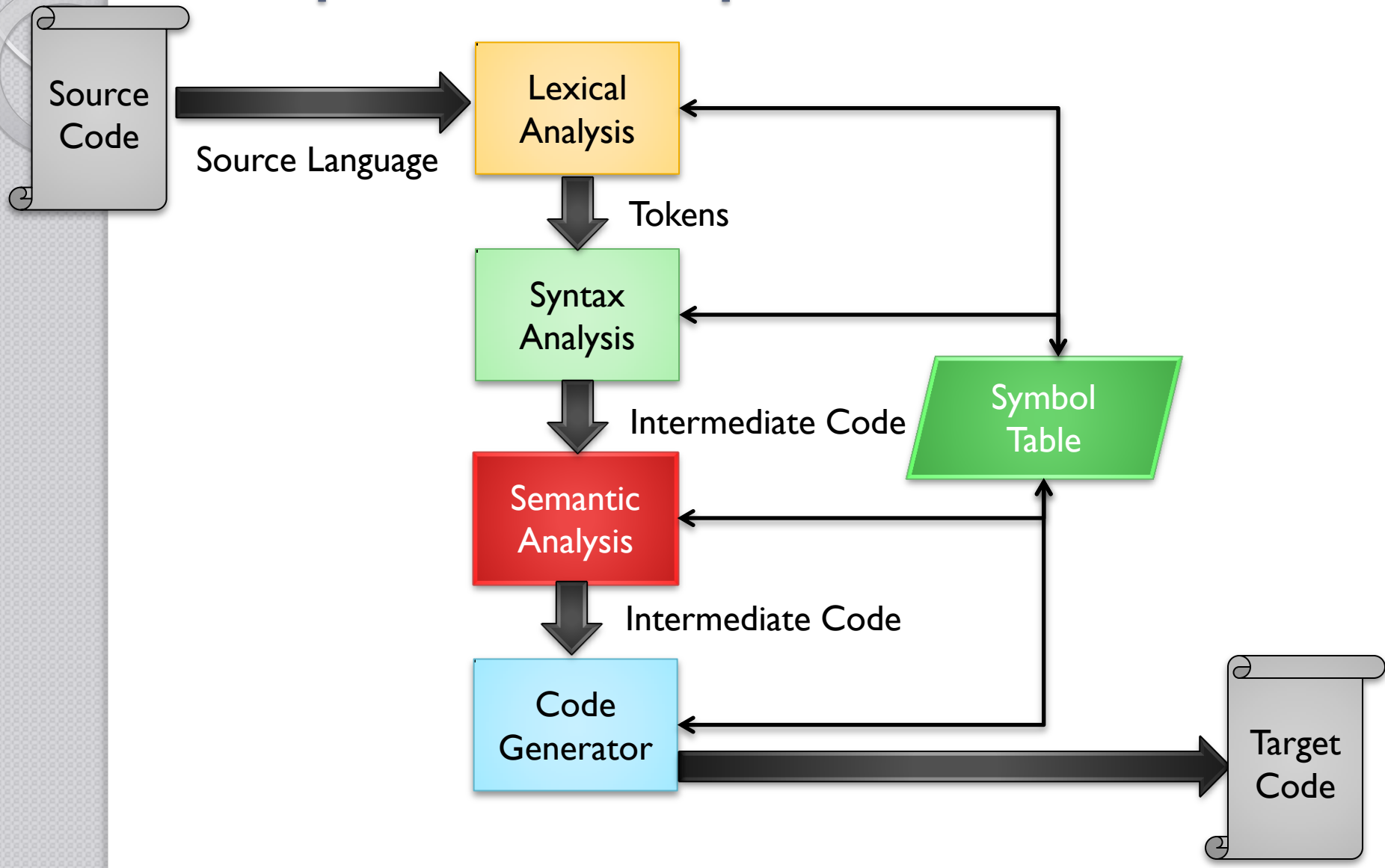
Tombstone Diagram



Compiler Architecture



Simplified Compiler Architecture



Lexical Analysis

- Tokenises the source file
  - Removes whitespace
  - Tokens
  - Keywords

Source Code

```
function Sqr( x : int )
: int
{
    let n : int = x;
    n <- n * n;
    return n;
}
```

```
Function    (Keyword)
Sqr         (Identifier)
(           (Lparen)
X           (Identifier)
:           (Colon)
Int         (Keyword)
)           (Rparen)
:           (Colon)
Int         (Keyword)
{           (Lbrace)
Let         (Keyword)
N           (Identifier)
:           (Colon)
Int         (Keyword)
=           (Eq)
X           (Identifier)
;           (Semicolon)
…
```

Syntax Analysis (Parser)

- Checks that the source file conforms to the language grammar
  - E.g. FunctionDecl ::= "function" identifier "(" …

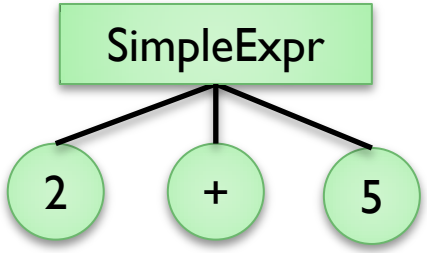
Source Code ✓

```
function Sqr( x : int ) …
```

Source Code ✗

```
function 1.5( x : int )
…
```

- Creates intermediate code
  - ParseTree

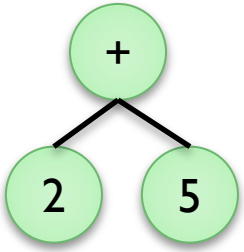


Semantic Analysis

- Checks that the meaning behind the syntax makes sense, according to the type-system
  - E.g. var int x = 3;
    x:int, 3:int 
  - E.g. var int x = 3.2;
    x:int, 3.2:Real 

Code Generation / Execution

- Traverses the Intermediate Code representation and generates or executes the code




```
PlusNode.Generate()
{
        leftChild.Generate() ;
        rightChild.Generate();
        EmitCode(Add);
}
```

```
IntLiteralNode.Generate()
{
        EmitCode ( Push, Integer.parseInt(this.getValue()) );
}
```

Symbol Table

- Important Data Structure
  - Keeps information about every identifier in the source
    - Variables
    - functions
    - Labels …
  - Keeps the scope information
  - Entry Properties
    - Name
    - Type
    - Value
    - Scope

Symbol Table

A data structure that maps an Identifier's name onto a structure that contains the identifier's information

Name → (Name, Type, Value, Scope)

Symbol Table

- Easiest way is to make use of a hash-table / hash-map
- Create a new Hash-map for each new scope

Java Interfaces

- Interface is a contract
- If a Class implements interface
  - Honours the contract
  - Implements the methods of the interface
- Interface type variable can hold instance of a class that implements that interface

Java Interfaces

```
public interface IAdder
{
    int add(int n, int m);
}
```

```
public interface IMinus
{
    int subtract(int n, int m);
}
```

Java Interfaces

```
public interface IAdder
{
    int add(int n, int m);
}
```

```
public interface IMinus
{
    int subtract(int n, int m);
}
```

```
public class SimpleMath
    implements IAdder, IMinus
{
    public int add(int n, int m)
    { … }

    public int subtract(int n, int m)
    { … }
}
```

Java Interfaces

```
public interface IAdder
{
    int add(int n, int m);
}
```

```
public interface IMinus
{
    int subtract(int n, int m);
}
```

```
public class SimpleMath
implements IAdder, IMinus
{
    public int add(int n, int m)
    { … }

    public int subtract(int n, int m)
    { … }
}
```

```
public static void main(…)
{
    IAdder a = new SimpleMath();
    IMinus s = new SimpleMath();

    a.add(1,2);
    s.subtract(4,2);
}
```

Java Interfaces

```
public interface ICalculator
{
    int Calculate(int n, int m);
}
```

Java Interfaces

```
public class Mul
    implements ICalculator
{
    public int Calculate(int n, int m)
    { … n * m … }
}
```

```
public class Div
    implements ICalculator
{
    public int Calculate(int n, int m)
    { … n / m …}
```

```
public interface ICalculator
{
    int Calculate(int n, int m);
}
```

Java Interfaces

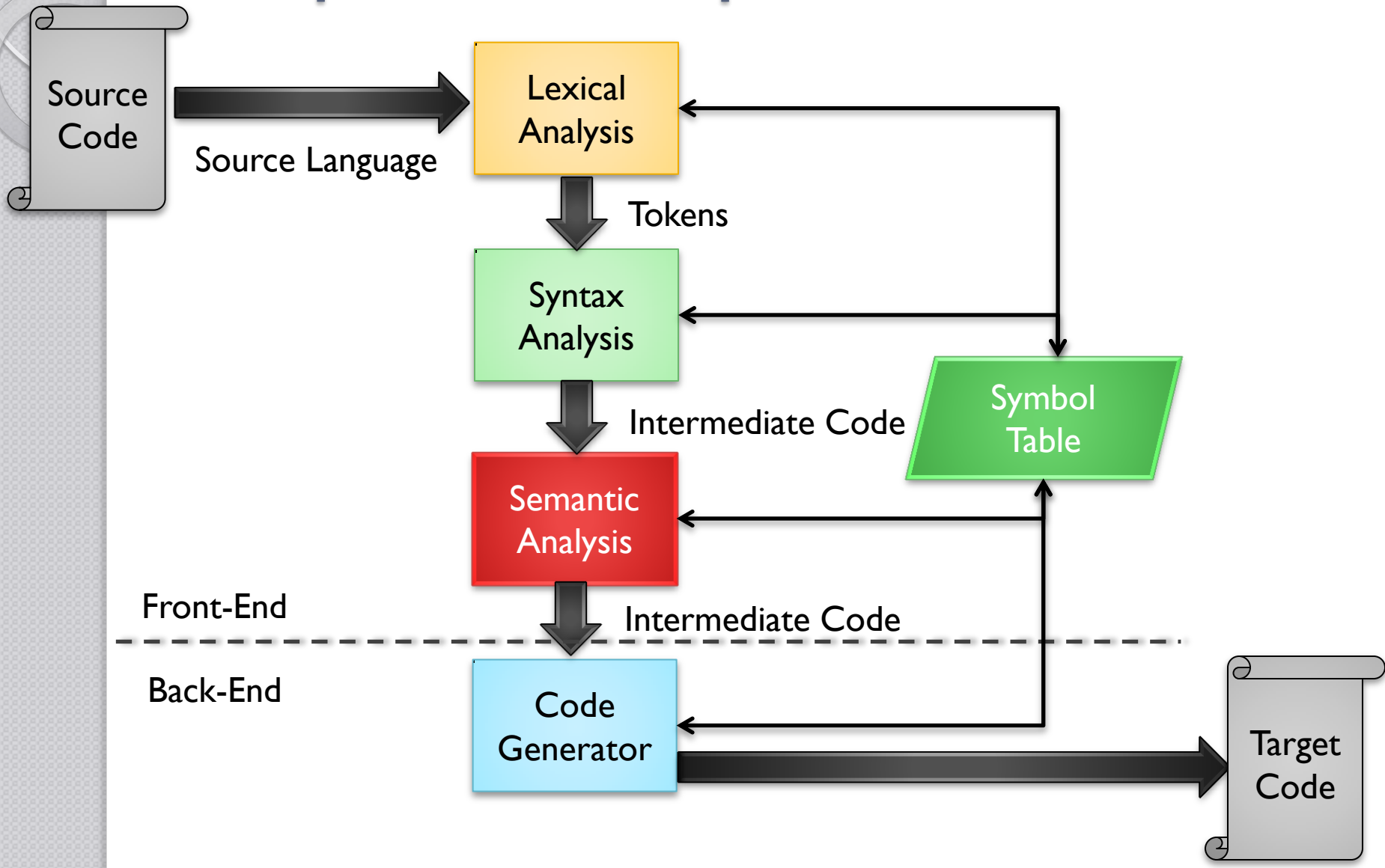
```
public class Mul
    implements ICalculator
{
    public int Calculate(int n, int m)
    { … n * m … }
}
```

```
public class Div
    implements ICalculator
{
    public int Calculate(int n, int m)
    { … n / m …}
```

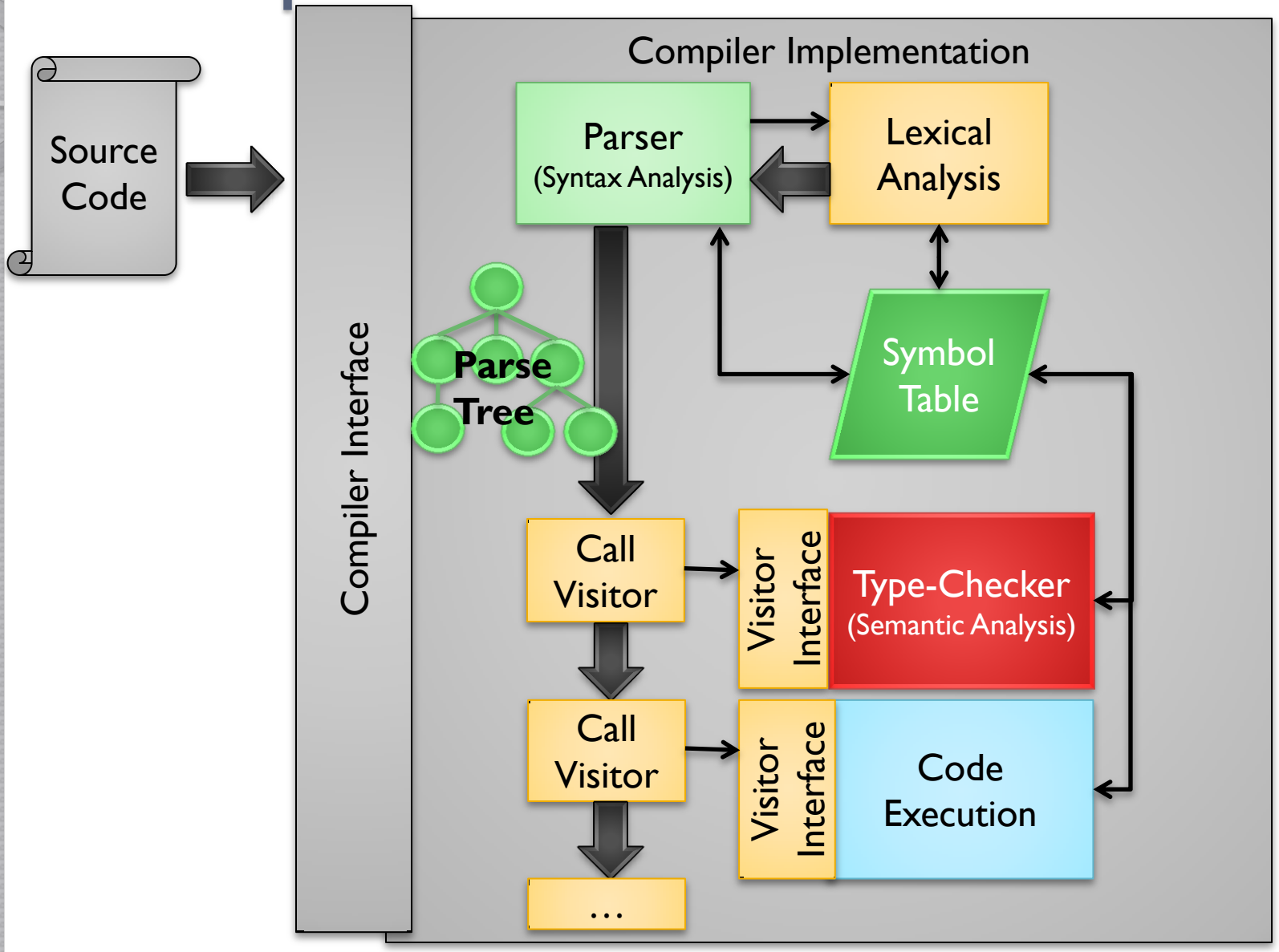
```
public interface ICalculator
{
    int Calculate(int n, int m);
}
```

```
public static void main(…)
{
    ICalculator c1 = new Mul();
    ICalculator c2 = new Div();

    c1.Calculate(4,2);
    c2.Calculate(4,2);
}
```

Simplified Compiler Architecture



Compiler Architecture



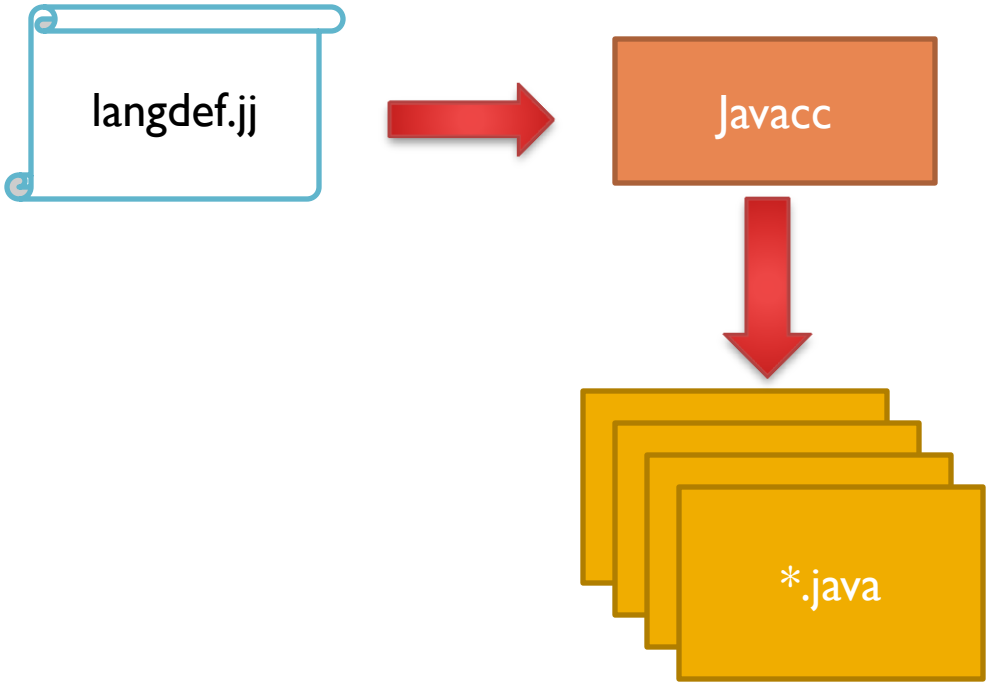
Program Generators?

- Source-to-Executable
  - Compiler
- Reverse Polish Notation
  - 2 + 3 - 4 → 2 3 + 4 -
- Source-to-Source
  - Preprocessors

Javacc

- A program that creates parsers
- The source code(input) is a definition file
  - Grammar definition
  - Inline Java code
- The target(output) is java-based parser
- Recognizer?

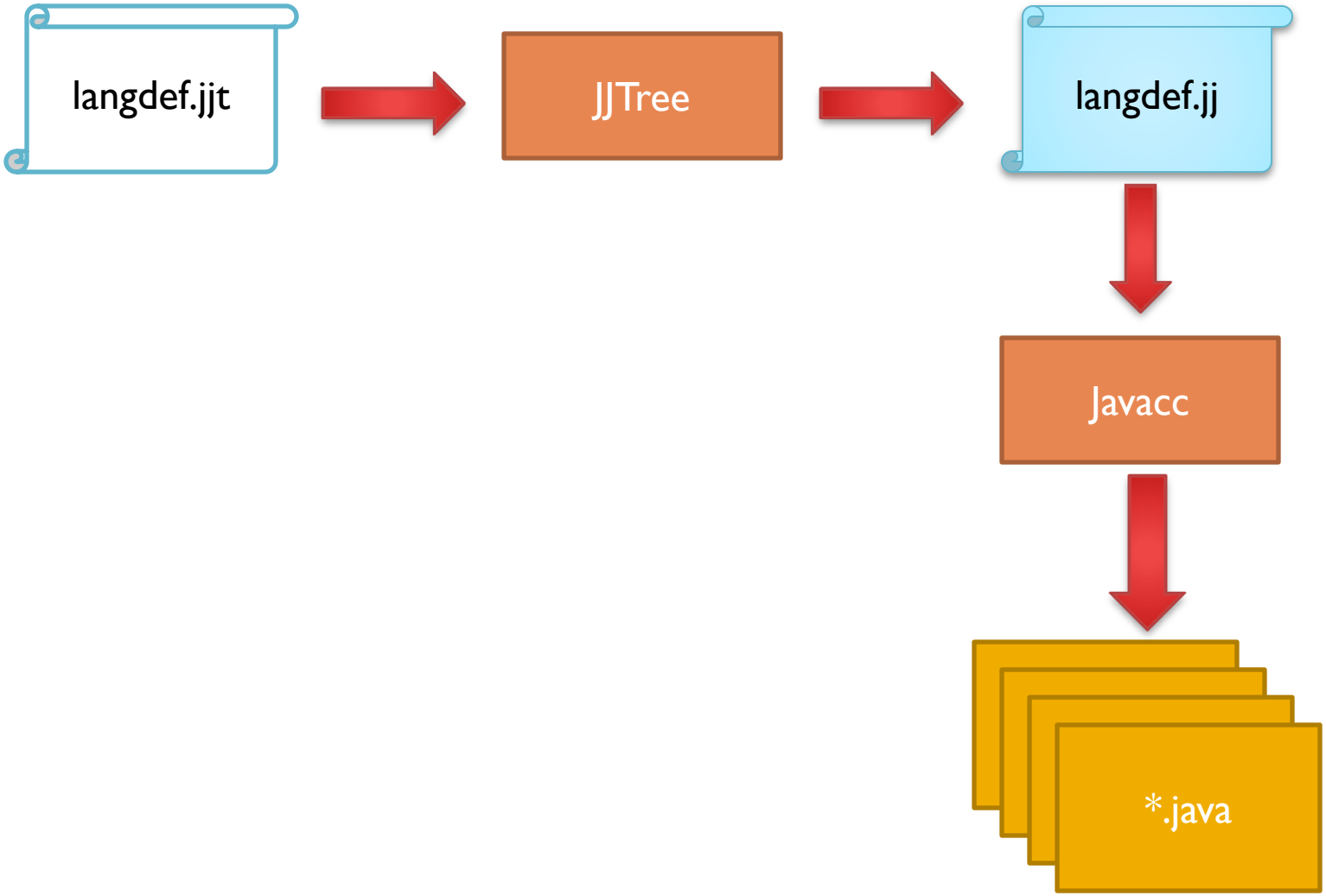
Javacc



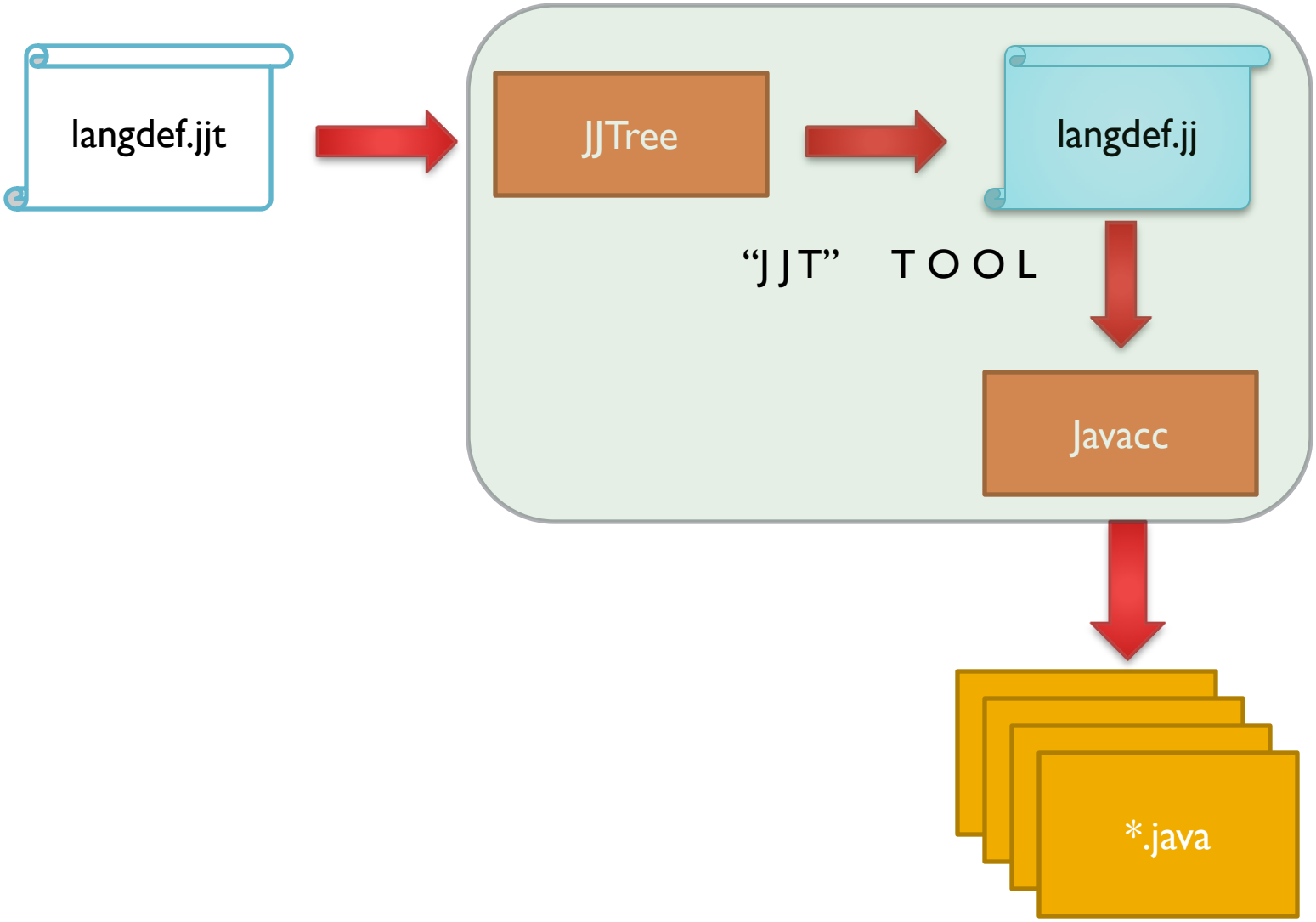
JJTree

- Preprocessor to Javacc
- Creates “.jj” files from “.jjt” files
- From a recognizer to a proper parser
- Produces Parse Tree building actions

JJTree



JJTree



Installing Javacc

- https://javacc.dev.java.net/
  - Javacc.zip / Javacc.tar.gz
- Eclipse plugin
  - Preferences->Install/Update->Add
    - http://eclipse-javacc.sourceforge.net/
  - Help->New Software
  - Newer versions
    - Eclipse Marketplace

.jjt Grammar Files

- Four (4) sections
  - Options
  - Parser
  - Tokens
  - Production Rules

.jjt Options

```
options
{
   BUILD_NODE_FILES=false;
   STATIC=false;
   MULTI=true;
   …
}
```

.jjt Parser block

PARSER_BEGIN(**parser_name**)

…

public class **parser_name**

{

…

}

…

PARSER_END(**parser_name**)

.jjt Tokens

- Lexeme to ignore

```
SKIP :
{
  " "
  | "\t"
  | "\n"
  | "\r"
  | <"//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

.jjt Tokens

- Tokens that are to be returned

```
TOKEN :
{
  <PLUS:“+”>
  |<TRUE: "true" >
  | <FALSE: "false" >
  | <LETTER: (["A"-"Z"] | ["a"-"z"]) >
  | <STRING_LITERAL: "\"" (~["\"","\r","\n"])* "\"" >
  | <#DIGIT: [“0”-”9”]>
}
```

.jjt Production Rules

- Assume the following:

Header → "**Script**" <STRING_LITERAL>

or

Header ::= "**Script**" <STRING_LITERAL>

.jjt Production Rules

- Assume the following:

Header ::= "**Script**" <STRING_LITERAL>

```
void Header():
{
  <SCRIPT> <STRING_LITERAL>
}
```

.jjt Production Rules

Header ::= “Script” <STRING_LITERAL>

```
void Header(): { int n = 0; n++;}
{
  <SCRIPT> <STRING_LITERAL>
}
```

.jjt Production Rules

- Token matching - Actions

```
void Header(): { int n = 0; n++; }
{
  <SCRIPT>{ n = 1; }
  <STRING_LITERAL>
  { n++; }
}
```

.jjt Production Rules

- Calling Non-Terminals

```
void Integer(): {}
{ … }

void Header(): { }
{
   <SCRIPT>
   Integer()
}
```

.jjt Production Rules

- Getting Token value

```
void Number()
{ Token t;
  int n = 0; }
{
  t=<DIGIT>{ n = integer.parseInt(t.image);
  }
}
```

Lookahead

- Consider the following java statements
  - public int name;
  - public int name[];
  - public int name() { … }

Building the AST (Abstract Syntax Tree)

```
options
{
   STATIC=false;
   MULTI=true;
   BUILD_NODE_FILES=true;
   NODE_USES_PARSER=false;
   NODE_PREFIX="";          // default "AST"
   …
}
```

Building the AST

```
SimpleNode Start() : {}
{
  Expression()
  ";"
  <EOF>                    // Special Token
  {
      return jjtThis;
  }
}
```

Parsing

```
PARSER_BEGIN(MyParser)
...
MyParser parser = new MyParser(System.in);
try {
   SimpleNode n = parser.Start();
   System.out.println(“Parsed!");
} catch (Exception e) {
      System.out.println("Oops!");
      System.out.println(e.getMessage());
}
...
PARSER_END(MyParser)
```

Example

Digit ::= [“**0**” - “**9**”]

Number ::= Digit+

Plus ::= “+”

Minus ::= “-”

Op ::= Plus | Minus

Expression ::= Number { Op Number }

Example

```
TOKEN:
{
< NUMBER: <DIGIT>+ >
| < Digit: [“0” - “9”] >
| < PLUS: “+” >
| < MINUS: “-” >
}
```

Digit ::= [“**0**” - “**9**”]
Number ::= Digit+
Plus ::= “+”
Minus ::= “-”

Example

```
void Op() :{}
{
   < PLUS > | < MINUS >
}

void Expression(): {}
{
   < Number >
   (Op() < Number >)*
}
```

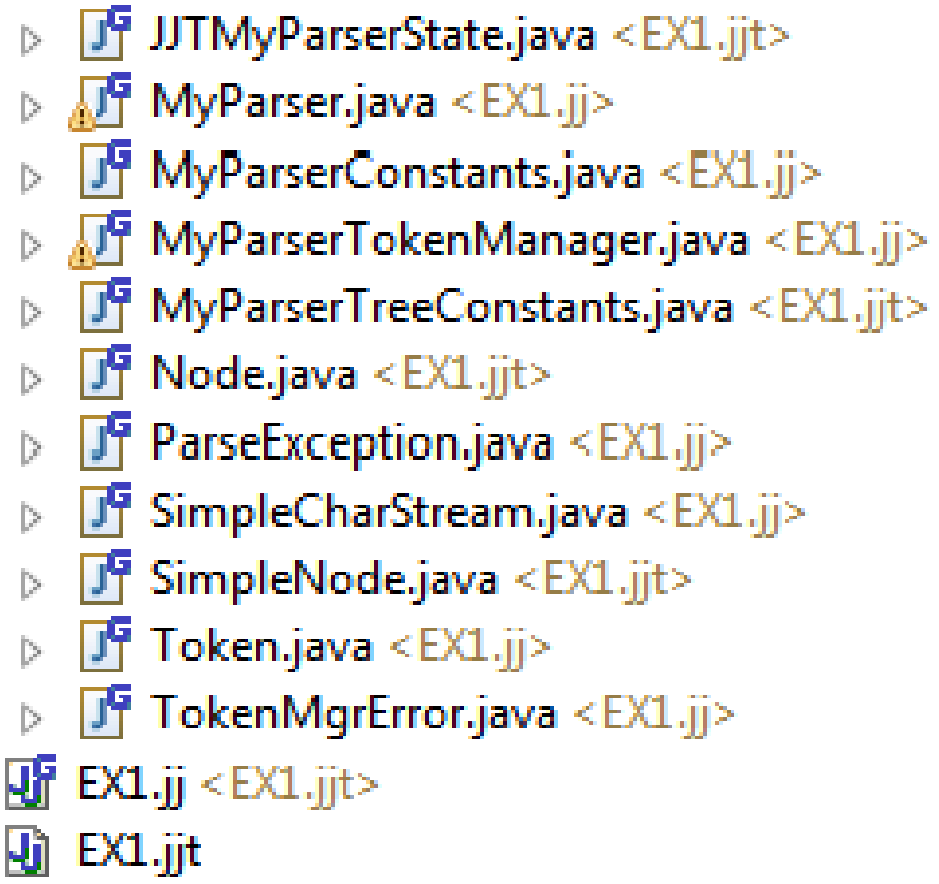
Op ::= Plus | Minus

Expression ::=
Number { Op Number }

Example

```
PARSER_BEGIN(MyParser)
...
MyParser parser = new MyParser(System.in);
try {
   parser.Expression();
   System.out.println(“Parsed!");
} catch (Exception e) {
      System.out.println("Oops!");
      System.out.println(e.getMessage());
}
...
PARSER_END(MyParser)
```

Generated Sources



Example – Evaluating

```
Token Op() :
{ Token t; }
{
   (t = < PLUS > | t = < MINUS >)
   { return t; }
}
```

Example

```
int Expression(): { Token t, op; int n;}
{
    t = < NUMBER >
    {
        n = Integer.parseInt( t.image );
    }
    ( op=Op()
    t = < NUMBER > {
    if(op.image.equals("+"))
            n += Integer.parseInt( t.image );
    else
            n -= Integer.parseInt( t.image ); }
    )*
    { return n; }
}
```

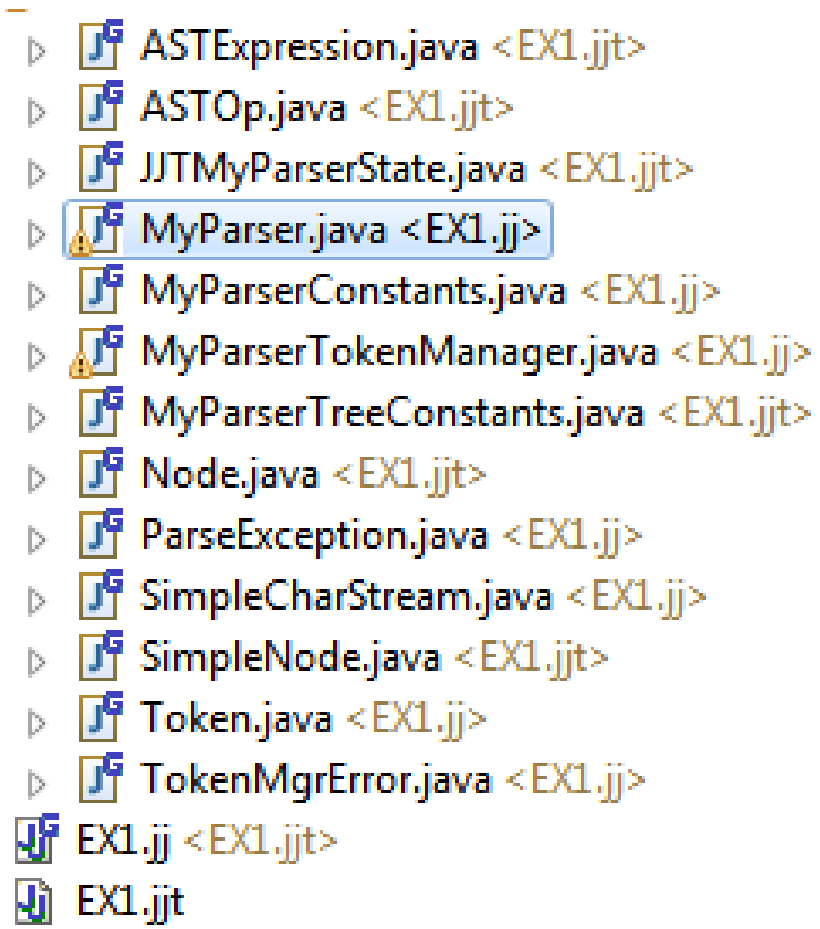
Example

```
PARSER_BEGIN(MyParser)

public class MyParser
{
...
    MyParser parser = new MyParser(System.in);
    try
    {
      int n = parser.Expression();
...

PARSER_END(MyParser)
```

Example - Building the AST

```
options
{
   STATIC=false;
   MULTI=true;
   BUILD_NODE_FILES=true;
   NODE_USES_PARSER=false;
   NODE_PREFIX="AST";
   …
}
```

Generated Code



Example

```
void Op() :{}
{
   < PLUS > | < MINUS >
}

SimpleNode Expression(): {}
{
   < Number >
   (Op() < Number >)*
   { return jjtThis; }
}
```

Op ::= Plus | Minus

Expression ::=
Number { Op Number }

Example

```
PARSER_BEGIN(MyParser)

public class MyParser
{
...
    MyParser parser = new MyParser(System.in);
    try
    {
      SimpleNode rootNode = parser.Expression();
...

PARSER_END(MyParser)
```

Example Grammar 2

Digit ::= [“**0**” - “**9**”]

Number ::= Digit+

Factor ::= Expression | Number

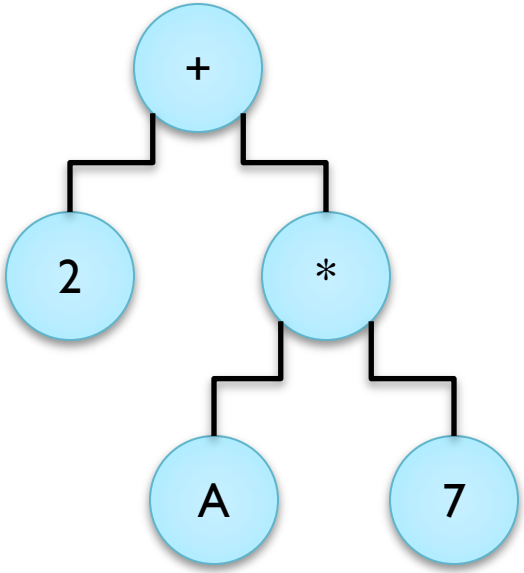
Term ::= Factor [“*” | “/” Factor]

Expression ::= Term { “**+**” | “**-**” Term }

Start ::= Expression

The Visitor Design Pattern

- The Problem



  - Number of operations to be performed on each node

  - Options

- Implement each operation inside each node
- Make use of visitor pattern

The Visitor Design Pattern

- Consider one Node
  - Printing Operation

- We can implement the operation in a separate class e.g. PrintVisitor

```
PrintVisitor

 void PrettyPrint( Plus )
{
   …
}
```

- This can be done for all type of nodes

The Visitor Design Pattern

- Modification on node

```
Plus

void Print( Visitor  p_visitor)
{
    p_visitor.PrettyPrint( this );
}
```

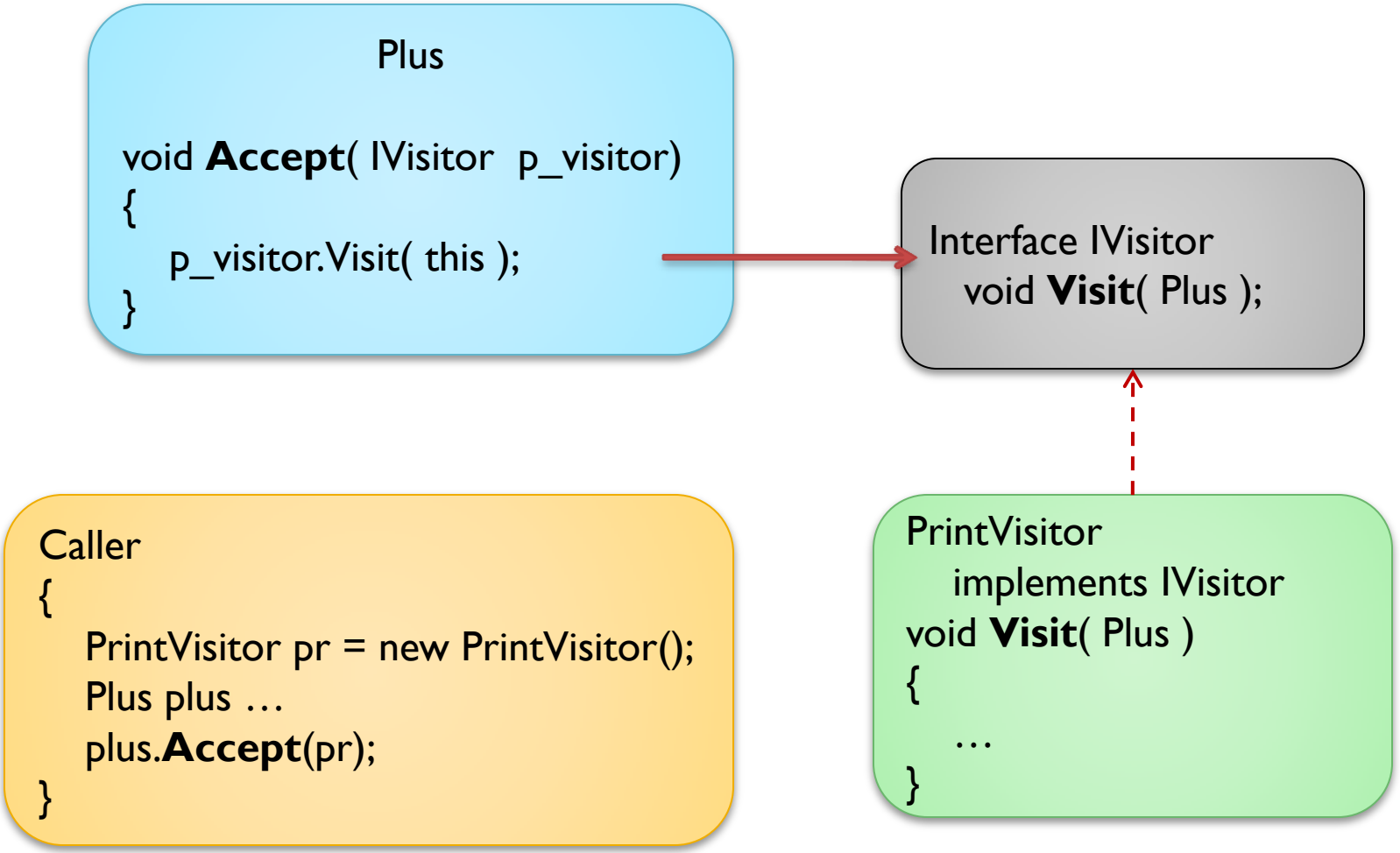
```
PrintVisitor

void PrettyPrint( Plus )
{
    …
}
```

- Client

```
Caller
{
    PrintVisitor pr = new PrintVisitor();
    Plus plus …
    plus.Print(pr);
}
```

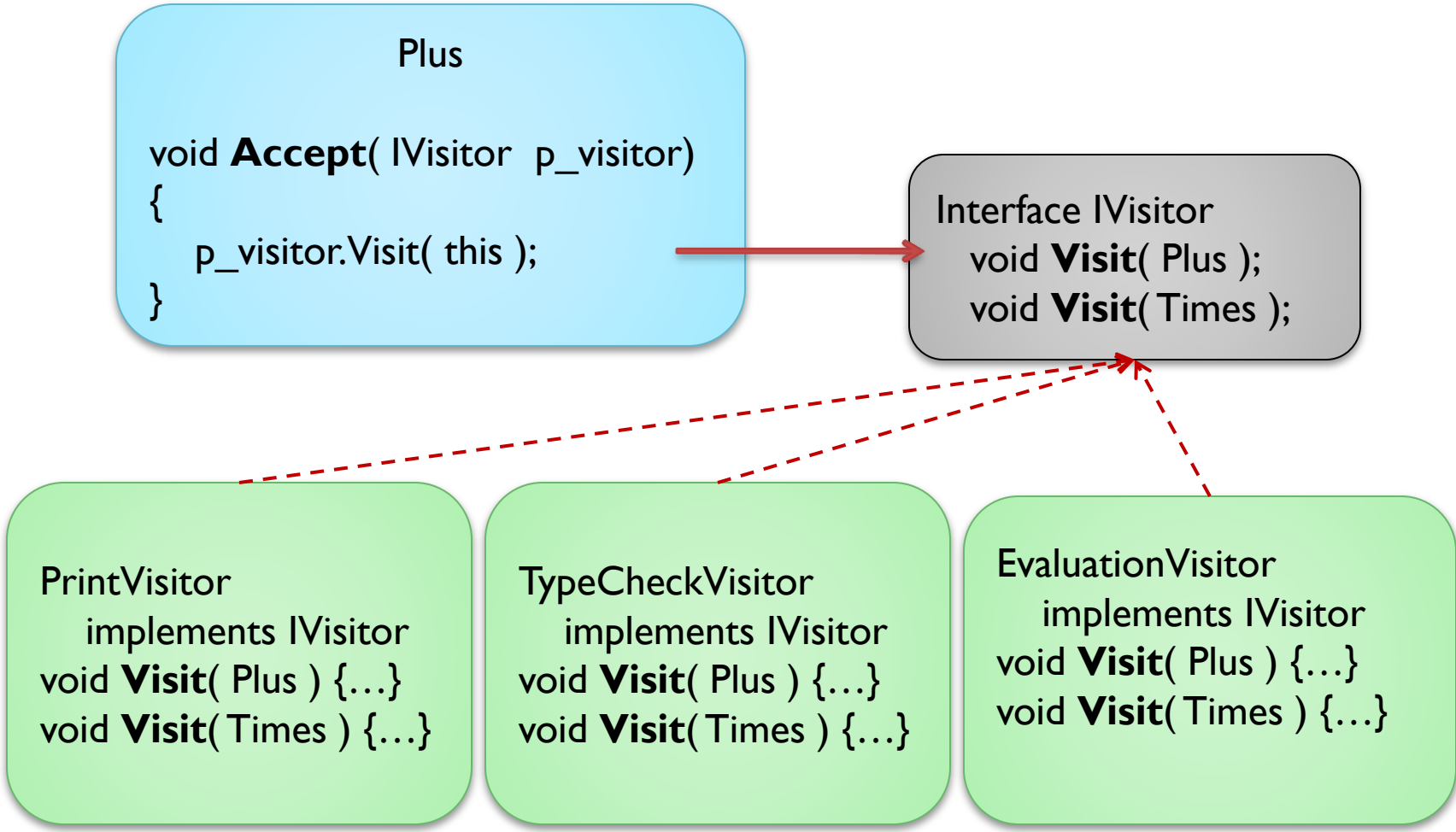
The Visitor Design Pattern

- Finally

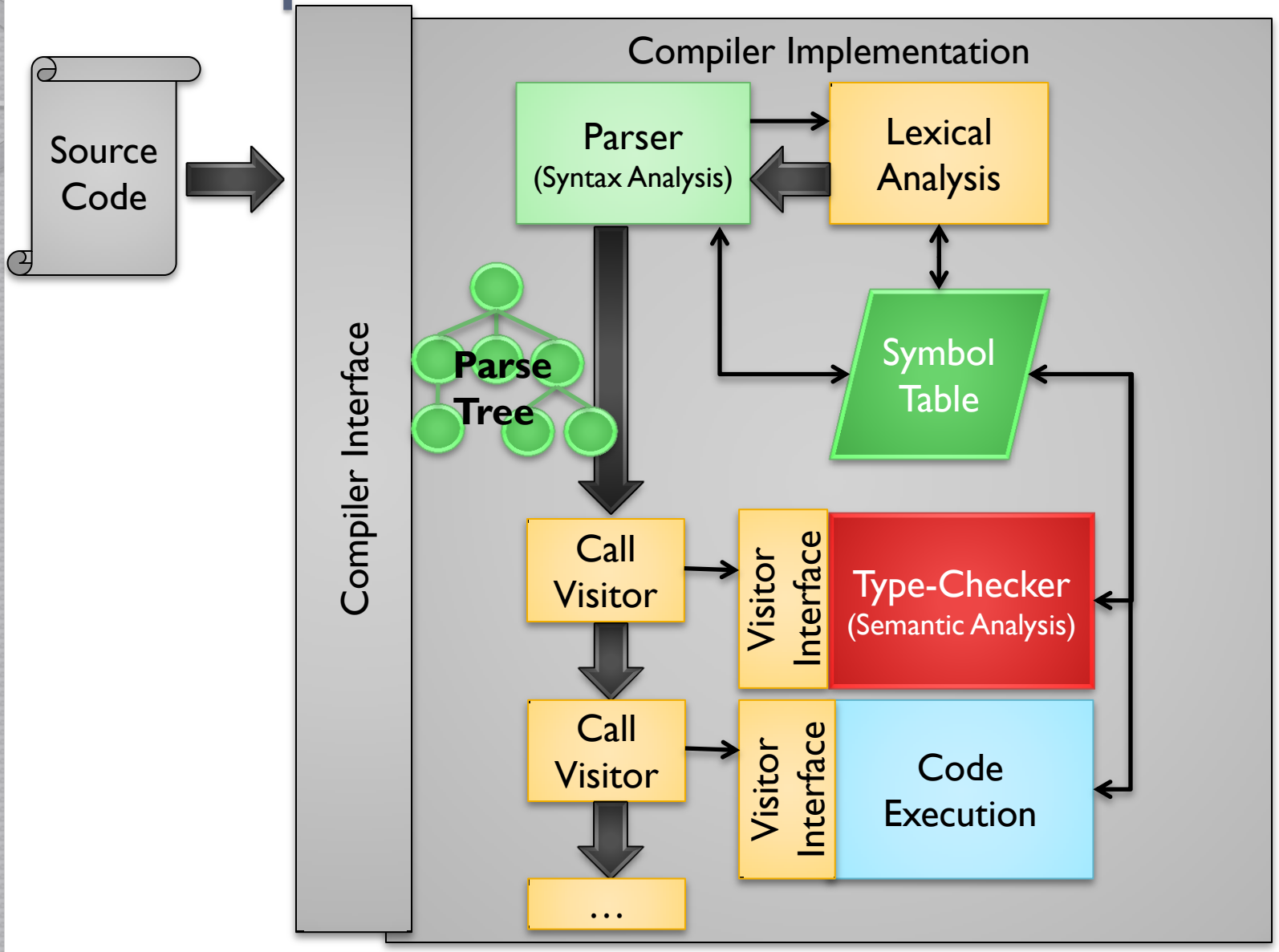


The Visitor Design Pattern

- Benefits



Compiler Architecture

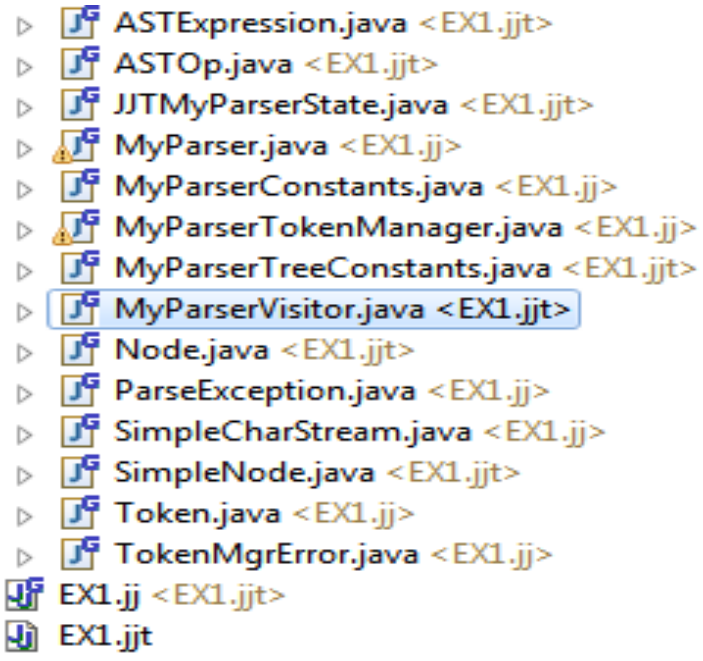


JJTree and the Visitor Pattern

```
Options
{
…
VISITOR=true;
…
}
```

Visitor interface

```
public interface MyParserVisitor
{

  public Object visit(SimpleNode node, Object data);

  public Object visit(ASTOp node, Object data);

  public Object visit(ASTExpression node, Object data);

}
```



Visitor - Node modification

```
Public class ASTExpression extends SimpleNode {
  public ASTExpression(int id) {
    super(id);
  }

  public ASTExpression(MyParser p, int id) {
    super(p, id);
  }

  /** Accept the visitor. **/
  public Object jjtAccept(MyParserVisitor visitor, Object data) {
    return visitor.visit(this, data);
  }
}
```

JJTree and the Visitor Pattern

```
Options
{
…
VISITOR=true;
VISITOR_DATA_TYPE="SymbolTable";
VISITOR_RETURN_TYPE="Object";
…
}
```

Visitor - return type

- We need a return-type generic enough to accommodate the results returned by all the visitor implementations
- By default jjt uses the Object class
  - Can easily be used however
    - class-casts
    - instanceof
- A generic container can be designed to hold the results

Visitor - return type

- Types –
  - Create an enumeration containing the types of the language's type-system.
- This should always be created

```
enum DataType
{
    None (/ Unknown),
    Error,
    Skip (/ Command),
    Bool,
    Int,
    Function,
    …
}
```

Visitor - return type

- Result class

```
Class Result
{
    DataType   type;
    Object     value;

    …
    Getter & Setter
    Conversion
    …
}
```

Visitor - return type

- We can make use of:
  - Object
    - Most generic return type
    - Have to cast to proper instance
  - DataType
    - When the Type of the node suffices
  - Result class
    - When we need to return some data

SymbolTable

- Entry
  - String Name
  - DataType Type
  - Value?

Name → Entry

SymbolTable

- Entry
  - String Name
  - DataType Type
  - Int ScopeLevel
  - Var / Param

  - DataType ReturnType
  - Int VarCount
  - Int ParamCount
  - (Entry Containter)

Type-Checking – (Semantic Analysis)

- SemanticAnalyser
  - (implements Visitor)

- Consider
  - BooleanLiteral()

```
public DataType visit(
  ASTBooleanLiteral node,
  SymbolTable data)
{
  return DataType.Bool;
}
```

Type-Checking – (Semantic Analysis)

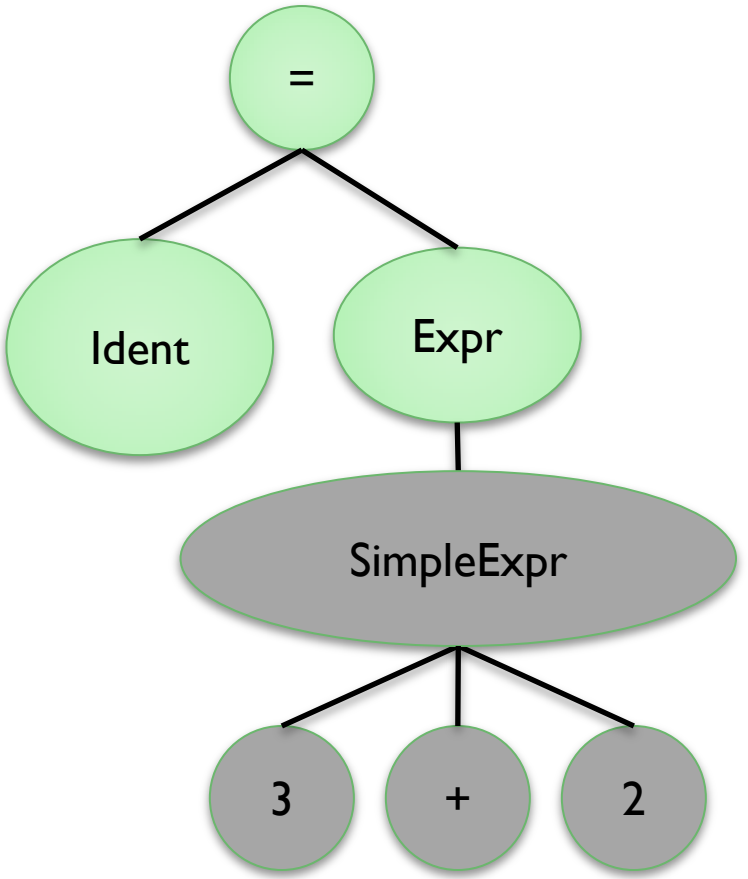
- Consider
  - Identifier()

```
public Object visit(
  ASTIdentifier node,
  SymbolTable data )
{
  // Get name from the node
  String name = (String)node.jjtGetValue();

  Consult symboltable, check if name exist
         if yes return its DataType

  println("Error ASTIdentifier : " + name);
  return DataType.Error;
}
```

Type-Checking – (Semantic Analysis)

- Consider

```
Assignment():
{
    Identifier()
    <ASSIGN>
    Expression()
}
```



Type-Checking – (Semantic Analysis)

- Consider
  - Assignment(): { Identifier() <ASSIGN> Expression() }

```
public Object visit(ASTAssignment node,
SymbolTable data )
{
     // Check identifier
     DataType identType =
          node.jjtGetChild(0).visit(this,
data);

     if(identType == DataType.Error)
          return DataType.Error;
```

Type-Checking – (Semantic Analysis)

- Consider
  - Assignment(): { Identifier() <ASSIGN> Expression() }

```
        …
        // Check expresion
        DataType exprType =
                node.jjtGetChild(1).visit(this, data);

        if(identType != exprType)
        {       println("Error ASTAssignment");
                return DataType.Error;     }

        return DataType.Skip;
}
```

Type-Checking – (Semantic Analysis)

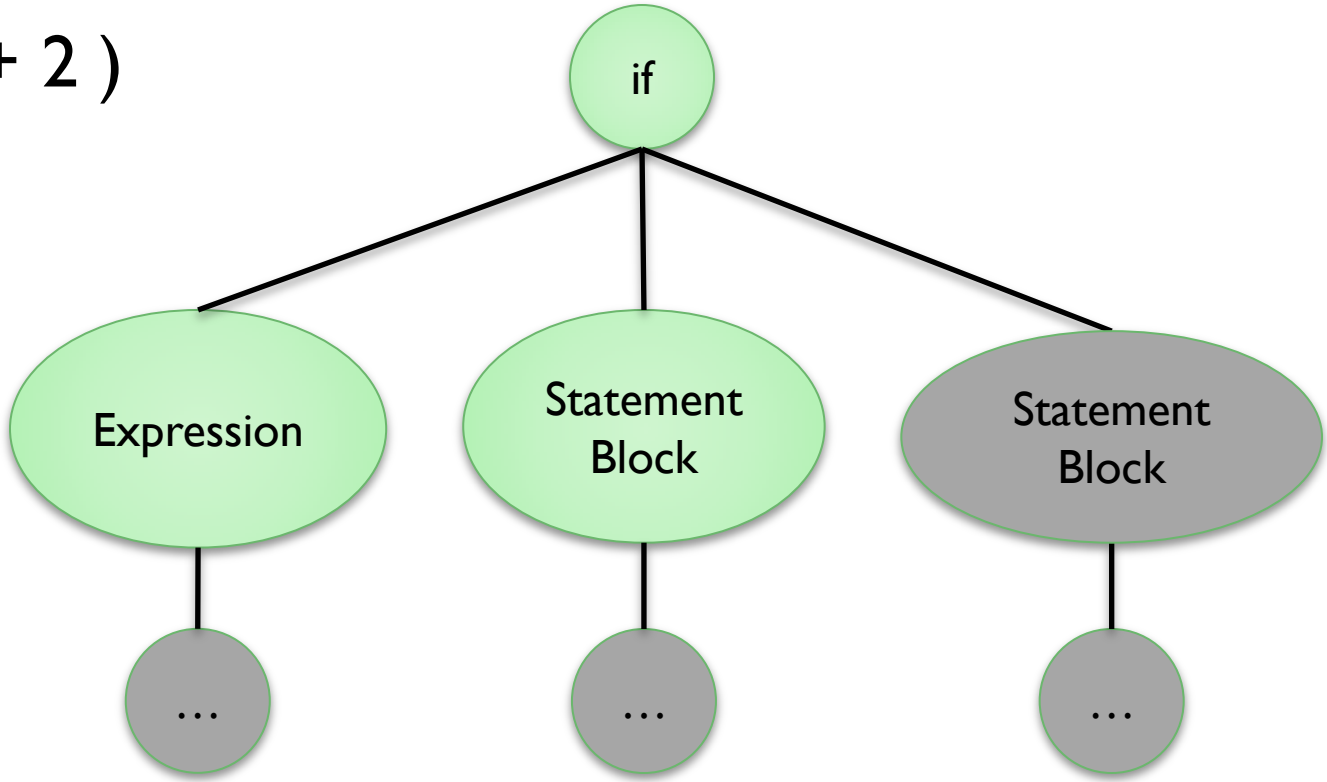
Note the difference in the use of identifiers when they are:

- L-value [a = 2 + 3]
- R-value [x = a + 4]

Type-Checking – (Semantic Analysis)

- Consider the following decision rule
  - IfStatement ::= “if” “(“ Expression “)” …

```
if( 3 + 2 )
{
   …
}
```



Type-Checking – (Semantic Analysis)

- Use node.jjtGetNumChildren() to get the number of children the node has.
- This determines the shape of the tree
- examples are
  - If Statement
  - Variable Declaration Initialiser (var int n = 0;)
  - Function Delaration
  - Statement Block

Type-Checking – (Semantic Analysis)

- Implement the call to the visitor:

```
Public class MyParser {
... main ...
Try
    {
      SimpleNode root = parser.Expression();

      MyParserVisitor visitor = new TypeChecker();
      Result result = root.jjtAccept(visitor, null );

      System.out.println("DataType is " +
           result .getType().toString());
...
```

Interpreter – Code Execution

- In a similar way to Semantic Analysis, we can make use of the **visitor** pattern
- This time, we return values as well rather than type-information only
- We have to take special care about the difference in the semantics of the languages involved

Code Generation

- Once again we make use of the visitor pattern
- In this stage we traverse the parse-tree and emit target language code in our case S-Machine instructions
- In particular, we traverse nodes and emit code patterns

SymbolTable

- Entries require an "Offset" variable
  - For data related nodes, "Offset" is the actual offset in the Stack.
  - For flow control nodes, "Offset" holds the location in the Code Area

Code Generation

- Since we are emitting S-Machine instructions, it is wise to create an enum of the instructions

```
public enum SMInstruction
{
        NOP,
        LDC,
        LD,
        STORE,
        DUP,
        …
}
```

Code Generation

- At the end of the whole process we are writing a binary file which is the compiled program.
- It is a good idea to create a buffer and perhaps a class that takes care of it
- When the process finished, the buffer is then written to disk

CodeBuilder

- class CodeBuilder
  - Byte[] codeBuffer
  - Int GetCodePosition()
  - void AddInstruction(SMInstruction inst)
  - void AddInstruction(SMInstruction inst, int nScope, int nArg)
  - void WriteCode(String p_strPath)

CodeGenerator

```
Class CodeGenerator
Implements Visitor

CodeGenerator(CodeBuilder cb)

Int GetCodeOffset()

Void Emit( SMInstruction inst, int nScope, int
nArg)

Void Emit( SMInstruction inst, int nArg)

Void Emit( SMInstruction inst )
```

CodeGenerator

- Endianess
  - CodeBuffer[Offset+2] = (nArg & 0xFF00)>>8
  - CodeBuffer[Offset+3] = (nArg & 0xFF)

CodeGenerator

- Consider Literal

```
public Object visit(
      ASTIntegerLiteral node,
      SymbolTable data)
{
      // LDC n
      Emit( SMInstruction.LDC, 0,
            (Integer)node.jjtGetValue() );
      return SLType.Skip;
}
```

CodeGenerator

- Consider Identifier

```
public DataType visit(
        ASTIdentifier node,
        SymbolTable data)
{
        //Get name from node
        //Get Entry from symbolTable
        //If Param …
        //Else if Var …

        Emit( SMInstruction.LD, …entry.Scope… , entry.Offset );

        return DataType.Skip;
}
```

CodeGenerator

- Consider Assignment

```
public DataType visit(
      ASTAssignment node,
      SymbolTable data )
{
      name = get name from node 0
      entry = data.get(name);

      node.jjtGetChild(1).jjtAccept(this, data);

      Emit(  SMInstruction.STORE, entry.Scope,
             entry.Offset);

      return DataType.Skip;
}
```

CodeGenerator

- Consider While
  - Can you identify a potential problem?

Location_of_Condition:
00 Code for Condition
01 JZ **Out_Of_While**
02 Code for While_Block
03 JMP **Location_of_Condition**
Out_Of_While:
04

CodeGenerator

- Consider While

```
public Object visit(ASTWhileStatement node, SymbolTable data)
{
        int addrCond = GetCodeOffset();
        node.jjtGetChild(0).jjtAccept(this, data);
        int addrJump = GetCodeOffset();
        Emit(SMInstruction.JZ, 0, 0);
        node.jjtGetChild(1).jjtAccept(this, data);

        Emit(SMInstruction.JMP, 0, addrCond);
        int addrEnd = m_codeBuilder.GetCodePos();

        BackPatch(addrJump, addrEnd);

        return SLType.Skip;
}
```

CodeGenerator

- BackPatch(
  address of instruction,
  new address);

- Modifies the address argument of an instruction

CodeGenerator

- Consider While

```
public Object visit(ASTWhileStatement node, SymbolTable data)
{
        int addrCond = GetCodeOffset();
        node.jjtGetChild(0).jjtAccept(this, data);
        int addrJump = GetCodeOffset();
        Emit(SMInstruction.JZ, 0, 0);
        node.jjtGetChild(1).jjtAccept(this, data);

        Emit(SMInstruction.JMP, 0, addrCond);
        int addrEnd = m_codeBuilder.GetCodePos();

        BackPatch(addrJump, addrEnd);

        return SLType.Skip;
}
```

CodeGenerator

- Consider FunctionDecl

```
public DataType visit(ASTFunctionDecl node, SymbolTable data)
{
   Get Entry from symboltable for node 0
   Set symboltable Scope to that of entry;
   entry.Offset = GetCodeOffset();

   Emit(SMInstruction.ENTER, 0, entry.VarNum);

   if(node.jjtGetNumChildren() == 4)
        node.jjtGetChild(3).jjtAccept(this, data);
   else
        node.jjtGetChild(2).jjtAccept(this, data);

   restore symboltable scope

   return DataType.Skip;
}
```

CodeGenerator

- Consider Program
  - Function Declarations are generated first
  - Start code with a jmp instruction
  - Enter a new scope

Questions?

The End