CS 153: Concepts of Compiler Design

October 31 Class Meeting

Department of Computer Science
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

ANTLR Pcl

- Pcl, a tiny subset of Pascal.
- Use ANTLR to generate a Pcl parser and lexer and integrate them with our Pascal interpreter's symbol table code.
  - ANTLR doesn't do symbol tables
- Parse a Pcl program and print the symbol table.
- Sample program **`sample.pas`**:

sample.pas

```
PROGRAM sample;

VAR
    i, j : integer;
    alpha, beta5x : real;

BEGIN
    REPEAT
        j := 3;
        i := 2 + 3*j
    UNTIL i >= j + 2;

    IF i <= j THEN i := j;

    IF j > i THEN i := 3*j
    ELSE BEGIN
        alpha := 9;
        beta5x := alpha/3 - alpha*2;
    END
END.
```

ANTLR Pcl, *cont'd*

- Strategy:
  - Write the grammar file **`Pcl.g4`**.
  - Generate the parser and lexer.
  - Generate a parse tree with the visitor interface.

- Use the following Pascal interpreter code:
  - Java
    - package **`wci.intermediate`**
    - package **`wci.util`**
  - C++
    - namespace **`wci::intermediate`**
    - namespace **`wci::util`**

ANTLR Pcl, *cont'd*

- Override visitor functions to:
  - Create a symbol table stack.
  - Create a symbol table for the program.
  - While parsing variable declarations, enter the variables into the program's symbol table.
  - After parsing the program, print the symbol table.

Pcl.g4

```
grammar Pcl;  // A tiny subset of Pascal

program : header block '.' ;
header  : PROGRAM IDENTIFIER ';' ;
block   : declarations compound_stmt ;

declarations : VAR decl_list ';' ;
decl_list    : decl ( ';' decl )* ;
decl         : var_list ':' type_id ;
var_list     : var_id ( ',' var_id )* ;
var_id       : IDENTIFIER ;
type_id      : IDENTIFIER ;

compound_stmt : BEGIN stmt_list END ;

stmt : compound_stmt    # compoundStmt
     | assignment_stmt  # assignmentStmt
     | repeat_stmt      # repeatStmt
     | if_stmt          # ifStmt
     |                  # emptyStmt
     ;
```

Pcl.g4

Pcl.g4, *cont'd*

Pcl.g4

```
stmt_list        : stmt ( ';' stmt )* ;
assignment_stmt : variable ':=' expr ;
repeat_stmt      : REPEAT stmt_list UNTIL expr ;
if_stmt          : IF expr THEN stmt ( ELSE stmt )? ;

variable : IDENTIFIER ;

expr : expr mul_div_op expr      # mulDivExpr
     | expr add_sub_op expr      # addSubExpr
     | expr rel_op expr          # relExpr
     | number                    # numberConst
     | IDENTIFIER                # identifier
     | '(' expr ')'              # parens
     ;

number : sign? INTEGER ;
sign   : '+' | '-' ;

mul_div_op : MUL_OP | DIV_OP ;
add_sub_op : ADD_OP | SUB_OP ;
rel_op     : EQ_OP | NE_OP | LT_OP | LE_OP | GT_OP | GE_OP ;
```

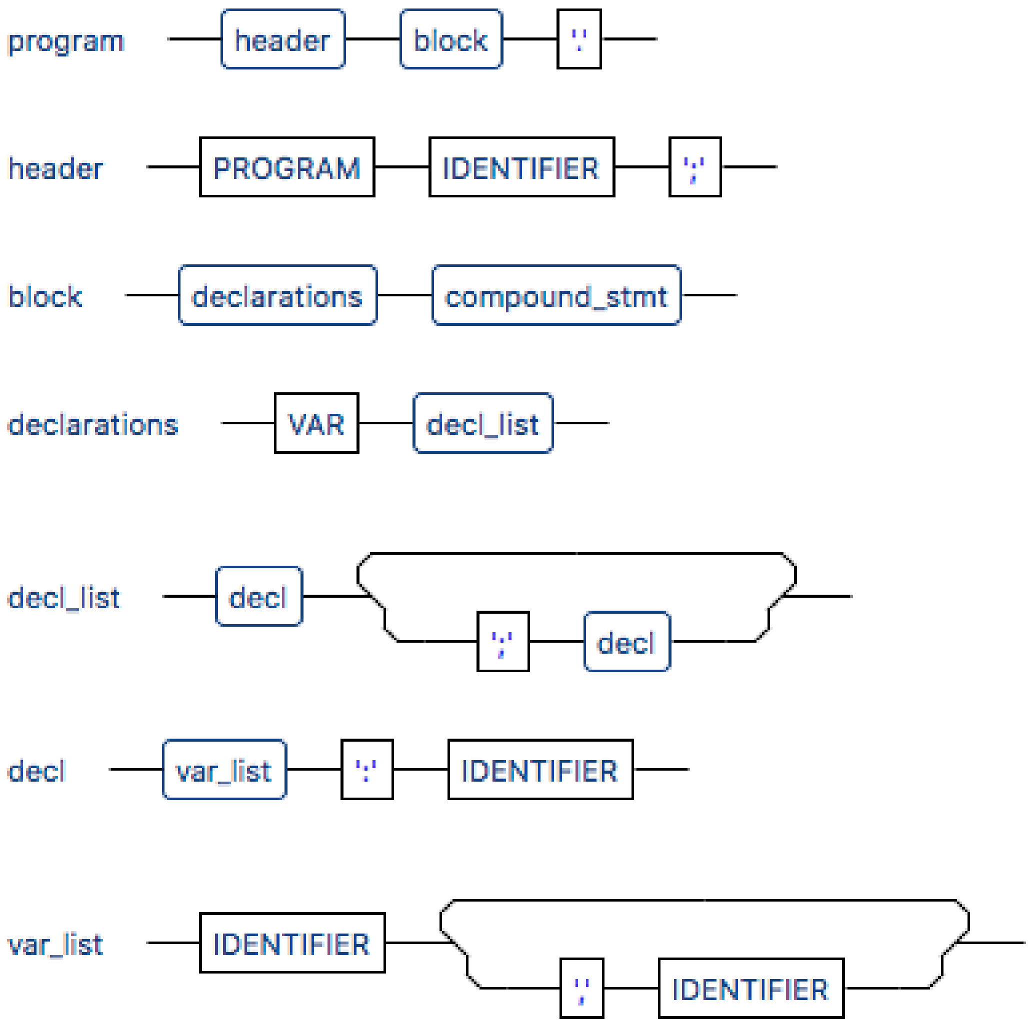
`Pcl.g4`, *cont'd*

```
PROGRAM : 'PROGRAM' ;
BEGIN   : 'BEGIN' ;
END     : 'END' ;
VAR     : 'VAR' ;
REPEAT  : 'REPEAT' ;
UNTIL   : 'UNTIL' ;
IF      : 'IF' ;
THEN    : 'THEN' ;
ELSE    : 'ELSE';

IDENTIFIER : [a-zA-Z][a-zA-Z0-9]* ;
INTEGER    : [0-9] ;

MUL_OP :   '*' ;
DIV_OP :   '/' ;
ADD_OP :   '+' ;
SUB_OP :   '-' ;
```

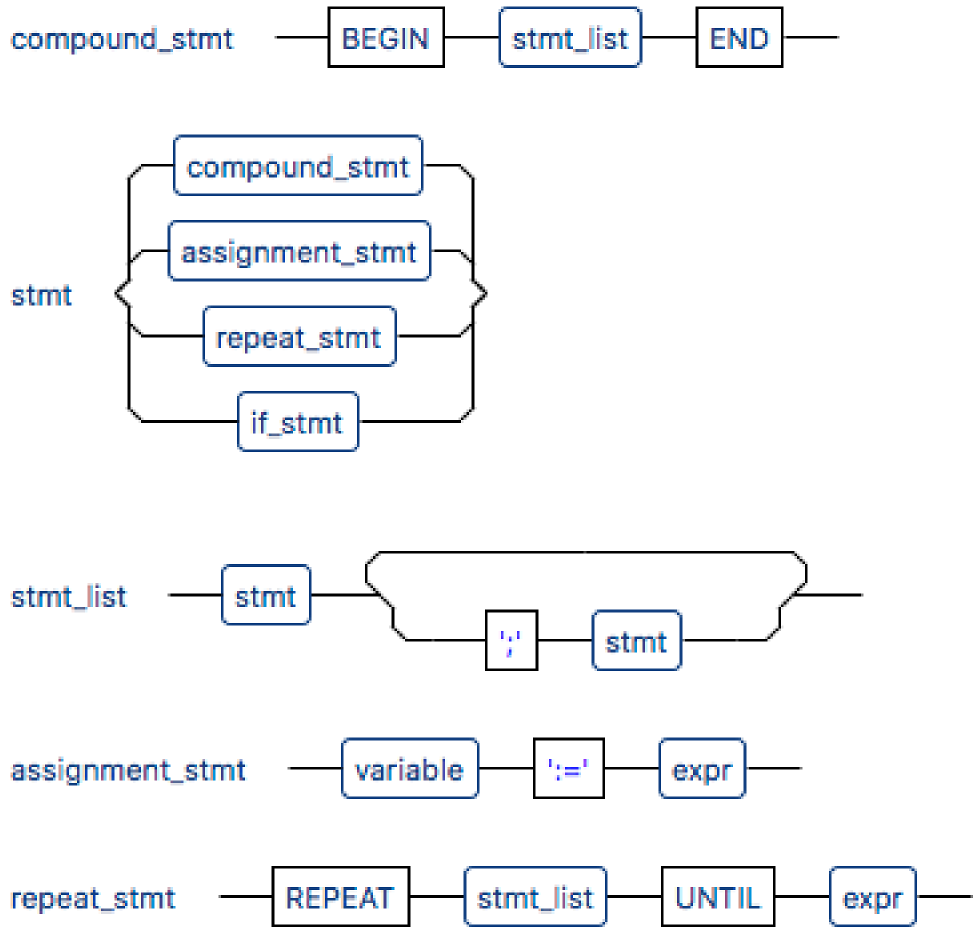
```
MUL_OP :    '*' ;
DIV_OP :    '/' ;
ADD_OP :    '+' ;
SUB_OP :    '-' ;

EQ_OP : '=' ;
NE_OP : '<>' ;
LT_OP : '<' ;
LE_OP : '<=' ;
GT_OP : '>' ;
GE_OP : '>=' ;

NEWLINE : '\r'? '\n' -> skip  ;
WS      : [ \t]+ -> skip ;
```

Pcl.g4

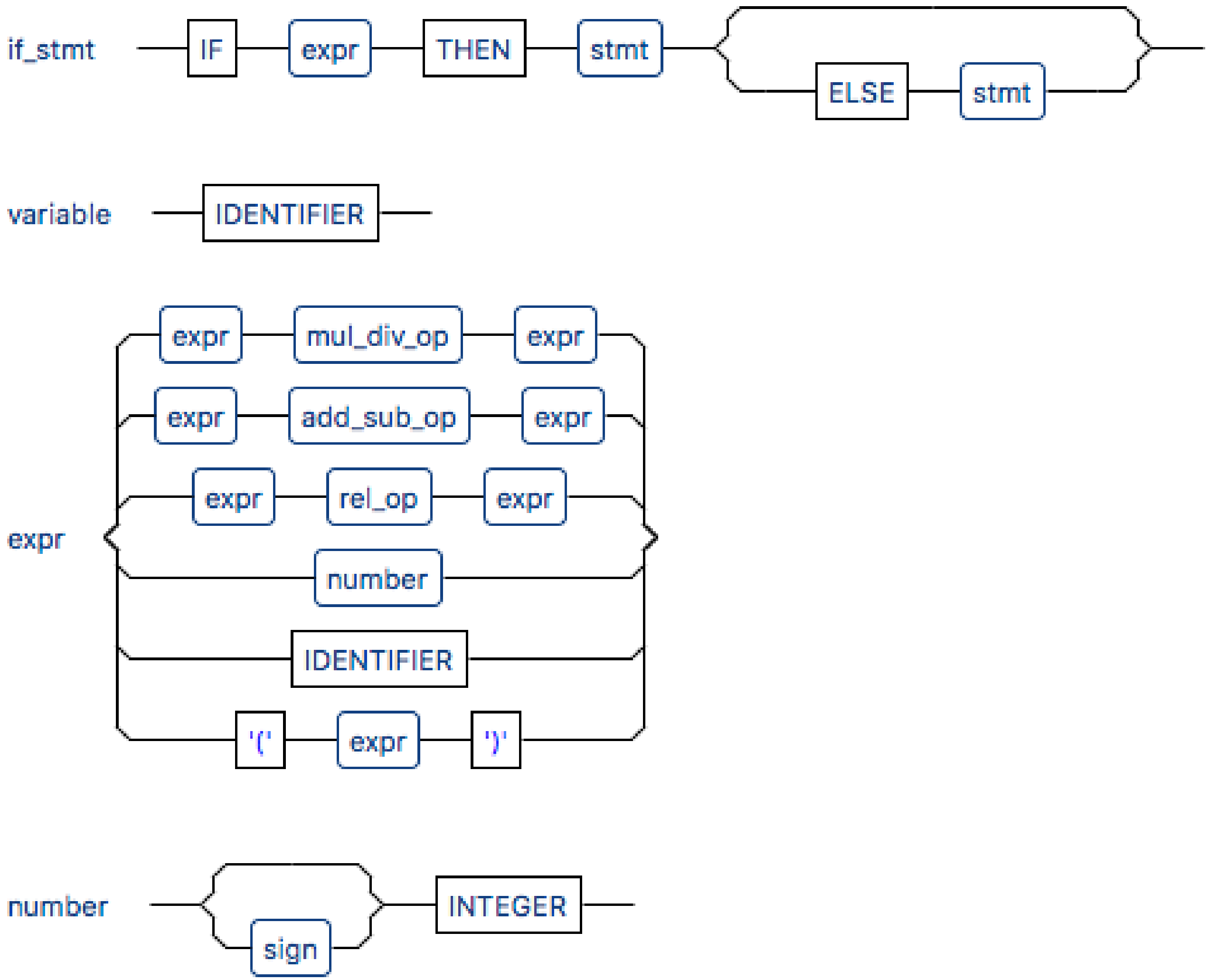
Pcl Syntax Diagrams



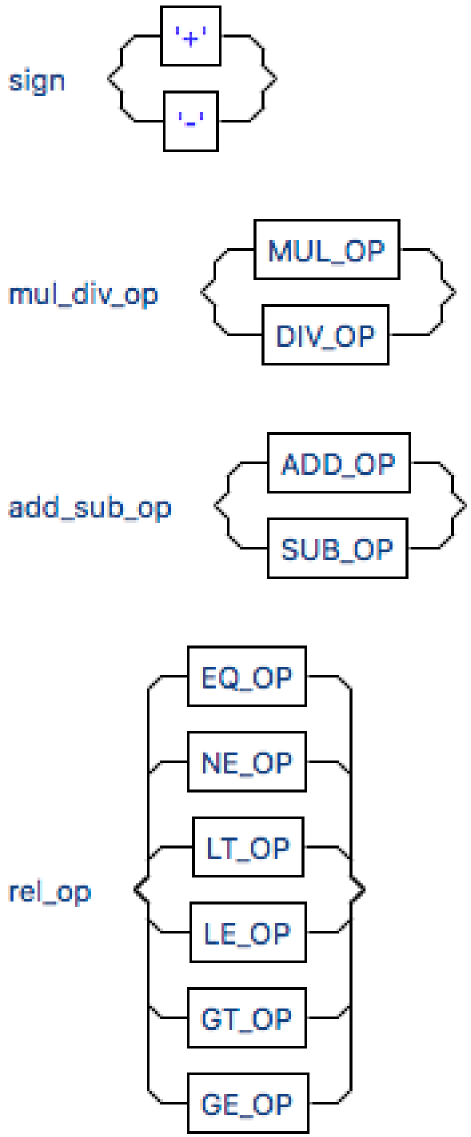
Pcl Syntax Diagrams, *cont'd*

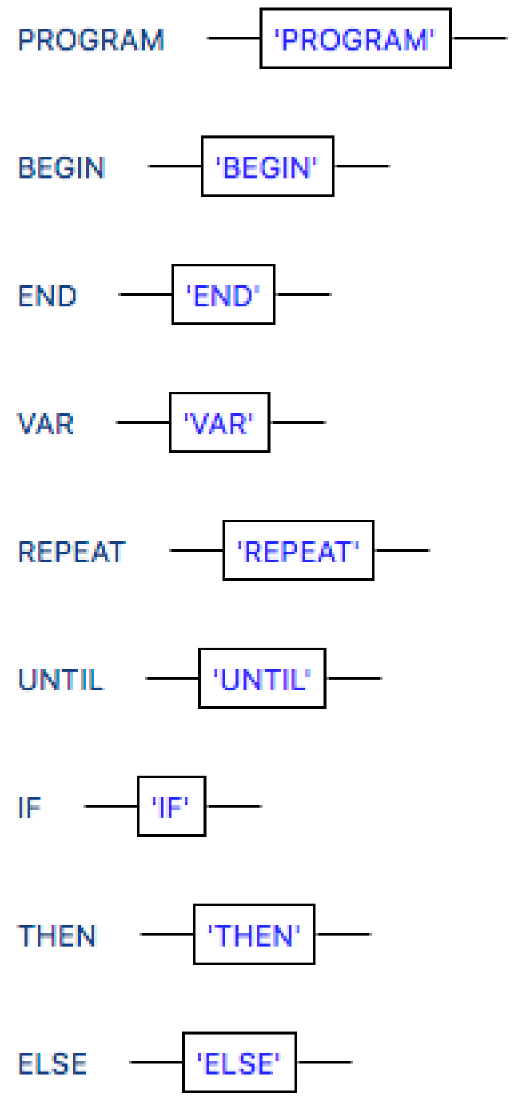


Pcl Syntax Diagrams, *cont'd*

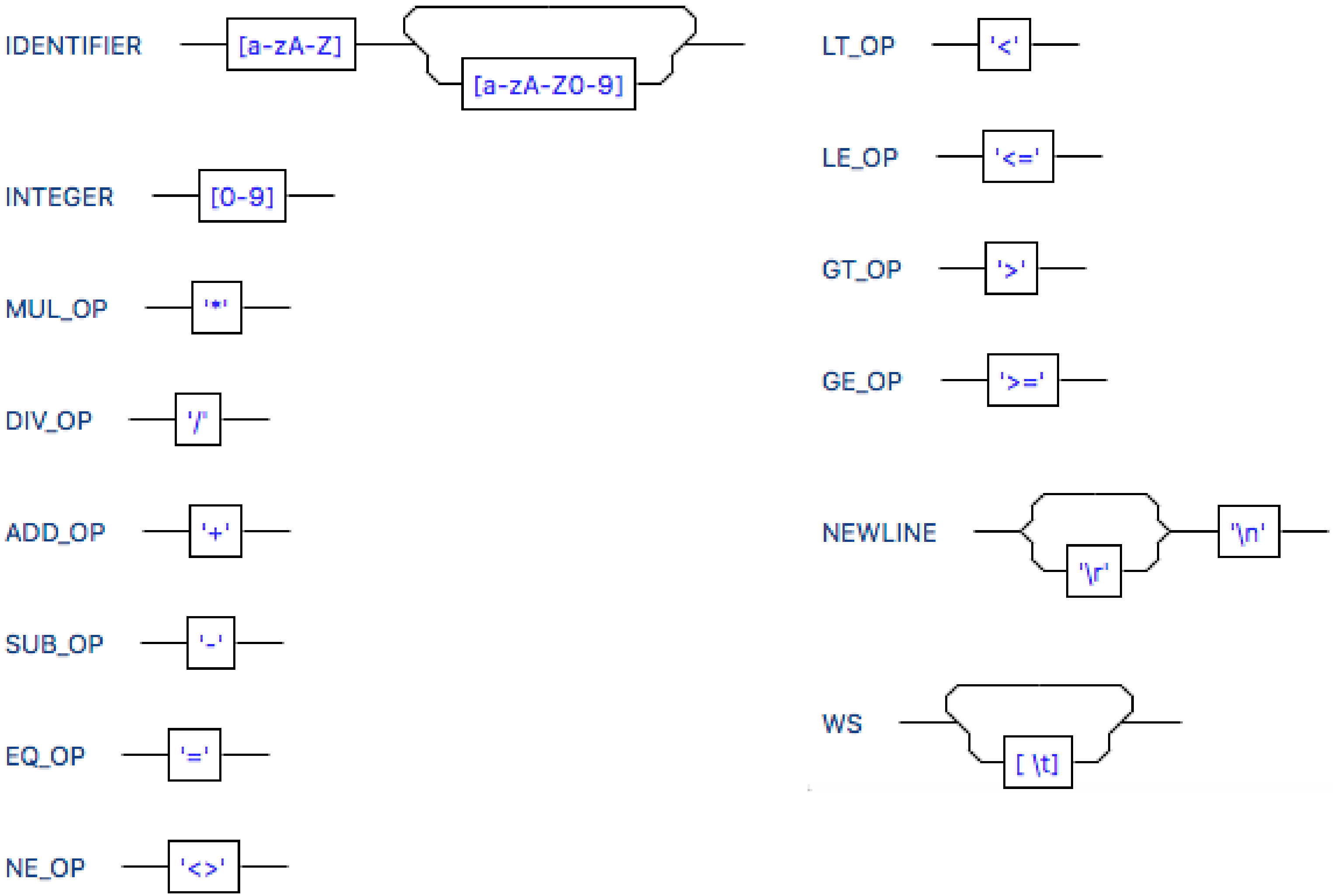


Pcl Syntax Diagrams, *cont'd*





Pcl Syntax Diagrams, *cont'd*



Pcl Compiler Java Code

```
public class PclBaseVisitor<T> extends AbstractParseTreeVisitor<T> implements PclVisitor<T>
{
    /**
     * {@inheritDoc}
     *
     * <p>The default implementation returns the result of calling
     * {@link #visitChildren} on {@code ctx}.</p>
     */
    @Override public T visitProgram(PclParser.ProgramContext ctx) { return visitChildren(ctx); }

    /**
     * {@inheritDoc}
     *
     * <p>The default implementation returns the result of calling
     * {@link #visitChildren} on {@code ctx}.</p>
     */
    @Override public T visitHeader(PclParser.HeaderContext ctx) { return visitChildren(ctx); }

    /**
     * {@inheritDoc}
     *
     * <p>The default implementation returns the result of calling
     * {@link #visitChildren} on {@code ctx}.</p>
     */
    @Override public T visitBlock(PclParser.BlockContext ctx) { return visitChildren(ctx); }
    ...
}
```

Pcl Compiler Java Code, *cont'd*

```
import java.util.ArrayList;

import wci.intermediate.*;
import wci.intermediate.symtabimpl.*;
import wci.util.*;

import static wci.intermediate.symtabimpl.SymTabKeyImpl.*;
import static wci.intermediate.symtabimpl.DefinitionImpl.*;

public class CompilerVisitor extends PclBaseVisitor<Integer>
{
    private SymTabStack symTabStack;
    private SymTabEntry programId;
    private ArrayList<SymTabEntry> variableIdList;
    private TypeSpec dataType;
```

Pcl Compiler Java Code, *cont'd*

```
public CompilerVisitor()
{
    // Create and initialize the symbol table stack.
    symTabStack = SymTabFactory.createSymTabStack();
    Predefined.initialize(symTabStack);
}

@Override
public Integer visitProgram(PclParser.ProgramContext ctx)
{
    System.out.println("Visiting program");
    Integer value = visitChildren(ctx);

    // Print the cross-reference table.
    CrossReferencer crossReferencer = new CrossReferencer();
    crossReferencer.print(symTabStack);

    return value;
}
```

Pcl Compiler Java Code, *cont'd*

```
@Override
public Integer visitHeader(PclParser.HeaderContext ctx)
{
    String programName = ctx.IDENTIFIER().toString();
    System.out.println("Program name = " + programName);

    programId = symTabStack.enterLocal(programName);
    programId.setDefinition(DefinitionImpl.PROGRAM);
    programId.setAttribute(ROUTINE_SYMTAB, symTabStack.push());
    symTabStack.setProgramId(programId);

    return visitChildren(ctx);
}

@Override
public Integer visitDecl(PclParser.DeclContext ctx)
{
    System.out.println("Visiting dcl");
    return visitChildren(ctx);
}
```

Program's name and symbol table.

Pcl Compiler Java Code, *cont'd*

```
@Override
public Integer visitVar_list(PclParser.Var_listContext ctx)
{
    System.out.println("Visiting variable list");
    variableIdList = new ArrayList<SymTabEntry>();

    return visitChildren(ctx);
}

@Override
public Integer visitVar_id(PclParser.Var_idContext ctx)
{
    String variableName = ctx.IDENTIFIER().toString();
    System.out.println("Declared Id = " + variableName);

    SymTabEntry variableId = symTabStack.enterLocal(variableName);
    variableId.setDefinition(VARIABLE);
    variableIdList.add(variableId);

    return visitChildren(ctx);
}
```

An array list to store variables being declared to be the same type.

Enter variables into the symbol table.

Pcl Compiler Java Code, *cont'd*

```
    @Override
    public Integer visitType_id(PclParser.Type_idContext ctx)
    {
        String typeName = ctx.IDENTIFIER().toString();
        System.out.println("Type = " + typeName);

        dataType = typeName.equalsIgnoreCase("integer")
                            ? Predefined.integerType
                  : typeName.equalsIgnoreCase("real")
                            ? Predefined.realType
                            : null;

        for (SymTabEntry id : variableIdList) id.setTypeSpec(dataType);

        return visitChildren(ctx);
    }
}
```

Set each variable's data type.

Pcl Compiler C++ Code

```
class PclBaseVisitor : public PclVisitor
{
public:

  virtual antlrcpp::Any visitProgram(PclParser::ProgramContext *ctx) override {
    return visitChildren(ctx);
  }

  virtual antlrcpp::Any visitHeader(PclParser::HeaderContext *ctx) override {
    return visitChildren(ctx);
  }

  virtual antlrcpp::Any visitBlock(PclParser::BlockContext *ctx) override {
    return visitChildren(ctx);
  }

  virtual antlrcpp::Any visitDeclarations(PclParser::DeclarationsContext *ctx) override {
    return visitChildren(ctx);
  }

  ...
}
```

Pcl Compiler C++ Code, *cont'd*

```
#include "wci/intermediate/SymTabStack.h"
#include "wci/intermediate/SymTabEntry.h"
#include "wci/intermediate/TypeSpec.h"

#include "PclBaseVisitor.h"
#include "antlr4-runtime.h"
#include "PclVisitor.h"

using namespace wci;
using namespace wci::intermediate;

class CompilerVisitor : public PclBaseVisitor
{
private:
    SymTabStack *symtab_stack;
    SymTabEntry *program_id;
    vector<SymTabEntry *> variable_id_list;
    TypeSpec *data_type;
```

Pcl Compiler C++ Code, *cont'd*

```
public:
    CompilerVisitor();
    virtual ~CompilerVisitor();

    antlrcpp::Any visitProgram(PclParser::ProgramContext *ctx) override;
    antlrcpp::Any visitHeader(PclParser::HeaderContext *ctx) override;
    antlrcpp::Any visitDecl(PclParser::DeclContext *ctx) override;
    antlrcpp::Any visitVar_list(PclParser::Var_listContext *ctx) override;
    antlrcpp::Any visitVar_id(PclParser::Var_idContext *ctx) override;
    antlrcpp::Any visitType_id(PclParser::Type_idContext *ctx) override;
};
```

Pcl Compiler C++ Code, *cont'd*

```
#include <iostream>
#include <string>
#include <vector>

#include "CompilerVisitor.h"
#include "wci/intermediate/SymTabFactory.h"
#include "wci/intermediate/symtabimpl/Predefined.h"
#include "wci/util/CrossReferencer.h"

using namespace std;
using namespace wci;
using namespace wci::intermediate;
using namespace wci::intermediate::symtabimpl;
using namespace wci::util;

CompilerVisitor::CompilerVisitor()
{
    // Create and initialize the symbol table stack.
    symtab_stack = SymTabFactory::create_symtab_stack();
    Predefined::initialize(symtab_stack);
}
```

Pcl Compiler C++ Code, *cont'd*

```
antlrcpp::Any CompilerVisitor::visitProgram(PclParser::ProgramContext *ctx)
{
    cout << "Visiting program" << endl;
    auto value = visitChildren(ctx);

    // Print the cross-reference table.
    CrossReferencer cross_referencer;
    cross_referencer.print(symtab_stack);

    return value;
}

antlrcpp::Any CompilerVisitor::visitHeader(PclParser::HeaderContext *ctx)
{
    string program_name = ctx->IDENTIFIER()->toString();
    cout << "Program name = " << program_name << endl;

    program_id = symtab_stack->enter_local(program_name);
    program_id->set_definition((Definition)DF_PROGRAM);
    program_id->set_attribute((SymTabKey) ROUTINE_SYMTAB,
                              new EntryValue(symtab_stack->push()));
    symtab_stack->set_program_id(program_id);

    return visitChildren(ctx);
}
```

Program's name and symbol table.

Pcl Compiler C++ Code, *cont'd*

```
antlrcpp::Any CompilerVisitor::visitDecl(PclParser::DeclContext *ctx)
{
    cout << "Visiting dcl" << endl;
    return visitChildren(ctx);
}

antlrcpp::Any CompilerVisitor::visitVar_list(PclParser::Var_listContext *ctx)
{
    cout << "Visiting variable list" << endl;
    variable_id_list.resize(0);

    return visitChildren(ctx);
}

antlrcpp::Any CompilerVisitor::visitVar_id(PclParser::Var_idContext *ctx)
{
    string variable_name = ctx->IDENTIFIER()->toString();
    cout << "Declared Id = " << variable_name << endl;

    SymTabEntry *variable_id = symtab_stack->enter_local(variable_name);
    variable_id->set_definition((Definition) DF_VARIABLE);
    variable_id_list.push_back(variable_id);

    return visitChildren(ctx);
}
```

A vector to store variables being declared to be the same type.

Enter variables into the symbol table.

Pcl Compiler C++ Code, *cont'd*

```
antlrcpp::Any CompilerVisitor::visitType_id(PclParser::Type_idContext *ctx)
{
    string type_name = ctx->IDENTIFIER()->toString();
    cout << "Type = " << type_name << endl;

    data_type = type_name == "integer" ? Predefined::integer_type
              : type_name == "real"    ? Predefined::real_type
                                       : nullptr;

    for (SymTabEntry *id : variable_id_list) id->set_typespec(data_type);

    return visitChildren(ctx);
}
```

Set each variable's data type.

Review: Interpreter vs. Compiler

- Same front end
  - parser, scanner, tokens
- Same intermediate tier
  - symbol tables, parse trees
- Different back end operations

Review: Interpreter vs. Compiler, *cont'd*

- Interpreter: Use the symbol tables and parse trees to execute the source program.
  - executor

- Compiler: Use the symbol tables and parse trees to generate an object program for the source program.
  - code generator

Target Machines

- A compiler's back end code generator produces object code for a target machine.
- Target machine: Java Virtual Machine (JVM)
- Object language: Jasmin assembly language
  - The Jasmin assembler translates assembly language programs into `.class` files.
  - The JVM loads and executes `.class` files.

Target Machines, *cont'd*

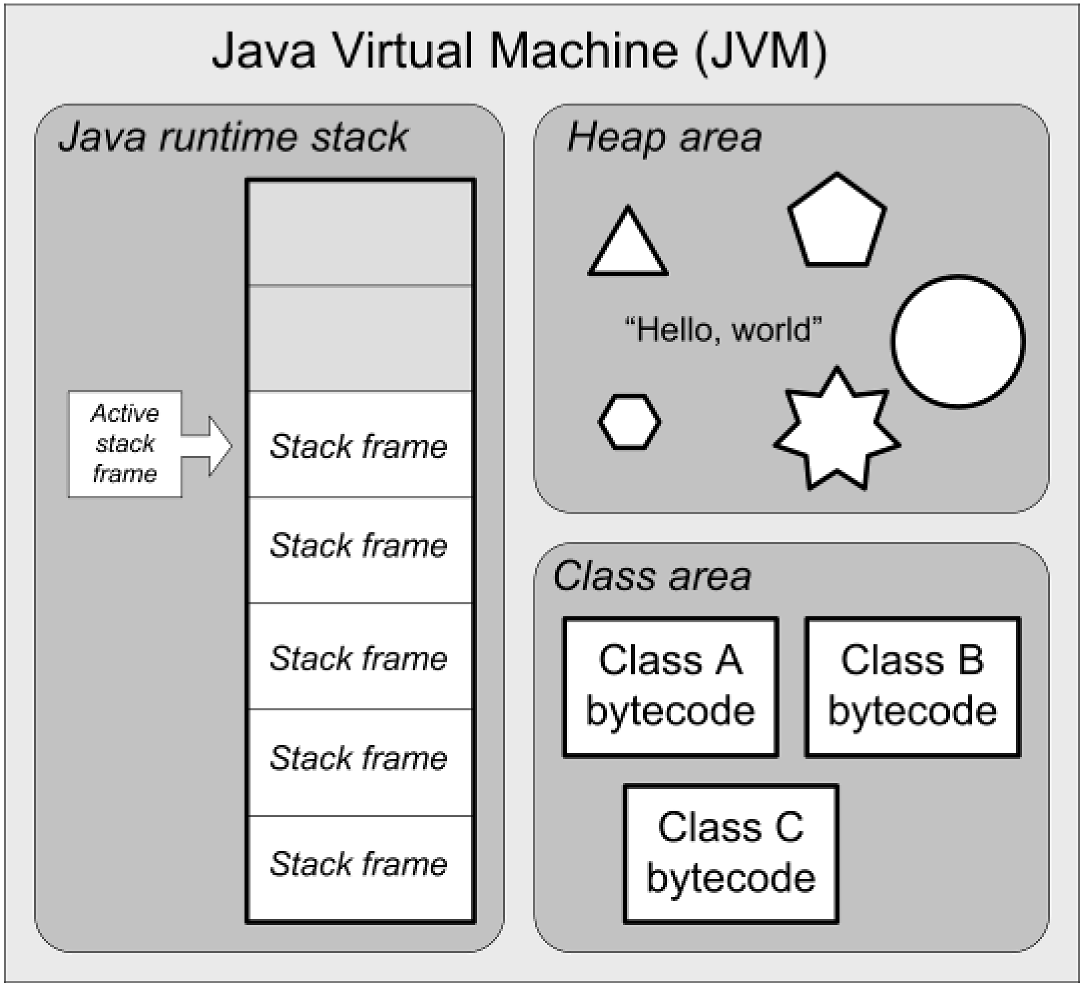
- Instead of using javac to compile a source program written in Java into a **`.class`** file ...
- Use <u>your compiler</u> to compile a source program written in your chosen language into a Jasmin object program.
- Then use the Jasmin assembler to create the **`.class`** file.

Target Machines, *cont'd*

- No matter what language the source program was originally written in, once it's been compiled into a `.class` file, <u>Java will be able to load and execute it</u>.

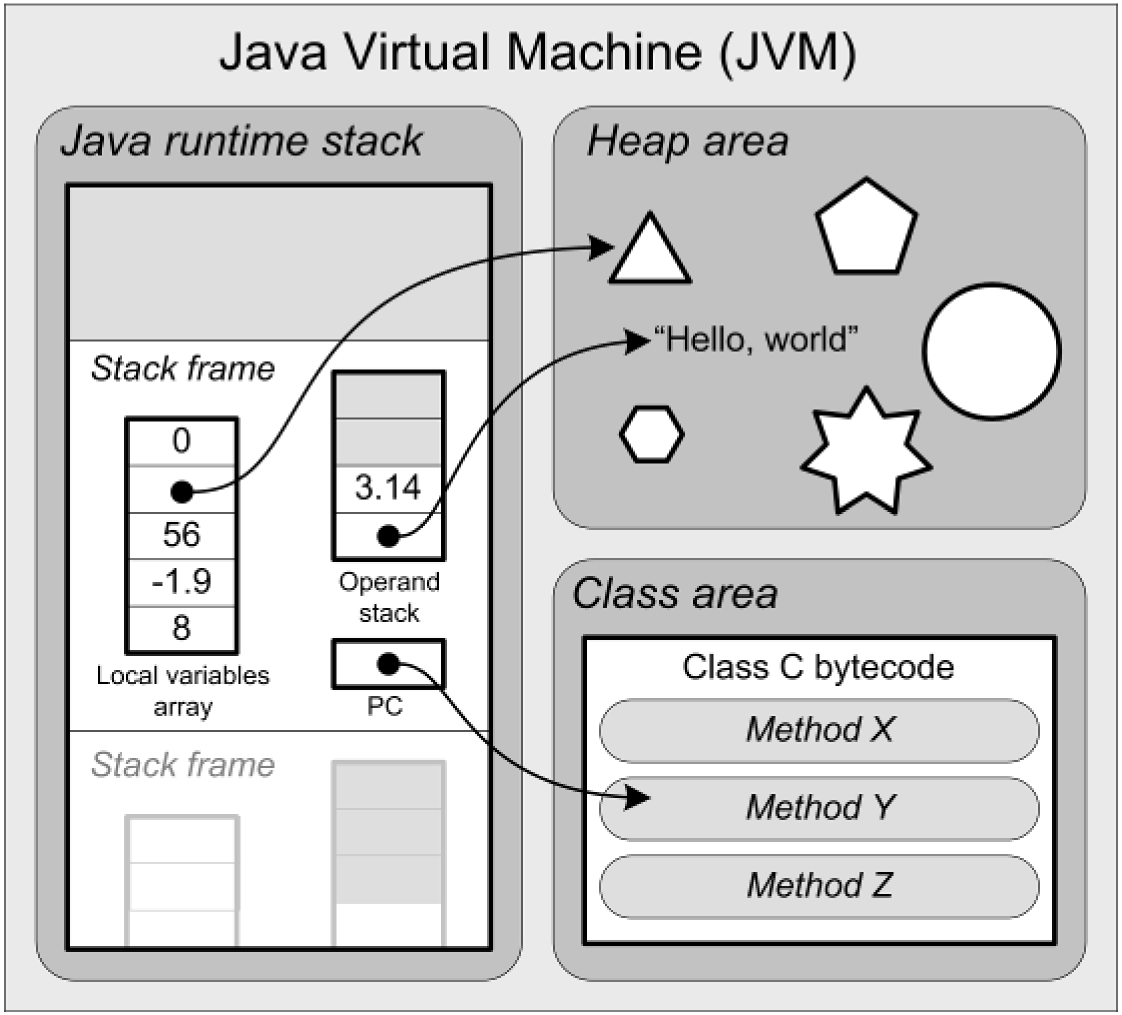
- The JVM runs on a wide variety of hardware platforms.

Java Virtual Machine (JVM) Architecture



- Java stack
  - runtime stack
- Heap area
  - dynamically allocated objects
  - automatic garbage collection
- Class area
  - code for methods
  - constants pool
- Native method stacks
  - support native methods, e.g., written in C
  - (not shown)

Java Virtual Machine Architecture, *cont'd*



- The runtime stack contains stack frames.
  - Stack frame = activation record.
- Each stack frame contains:
  - local variables array
  - operand stack
  - program counter (PC)

What is missing in the JVM that we had in our Pascal interpreter?

The JVM's Java Runtime Stack

- Each method invocation pushes a stack frame.
- Equivalent to the activation record of our Pascal interpreter.
- The stack frame currently on top of the runtime stack is the <u>active stack frame</u>.
- A stack frame is popped off when the method returns, possibly leaving behind a <u>return value</u> on top of the stack.

Stack Frame Contents

- Operand stack
  - For doing computations.
- Local variables array
  - Equivalent to the <u>memory map</u> in our Pascal interpreter's activation record.
- Program counter (PC)
  - Keeps track of the currently executing instruction.

JVM Instructions

- Load and store values
- Arithmetic operations
- Type conversions
- Object creation and management
- Runtime stack management (push/pop values)
- Branching
- Method call and return
- Throwing exceptions
- Concurrency

Jasmin Assembler

- Download from:
  - http://jasmin.sourceforge.net/
- Site also includes:
  - User Guide
  - Instruction set
  - Sample programs

Example Jasmin Program

hello.j

```
.class public HelloWorld
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit stack  2
.limit locals 1

  getstatic     java/lang/System/out Ljava/io/PrintStream;
  ldc           "Hello, world!"
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  return

.end method
```

- Assemble:
  - `java –jar jasmin.jar hello.j`
- Execute:
  - `java HelloWorld`

Demo