Programming Languages and Concepts

Siegfried Benkner

Research Group Scientific Computing

Universität Wien

Compilers

- Language Processors
- Structure of a Compiler
- Overview of Compiler Phases
- ANTLR
- Assignment 4

Literature

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey Ullman.

Compilers: Principles, Techniques, and Tools,

2nd Edition, Addison-Wesley, 2006.

Terence Parr. **The Definitive ANTLR 4 Reference**, 2013.

https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/

Goals

- Basic understanding of (formal) language processing systems.
- Understand theoretical foundations, main concepts and tools for compiler construction.
- Practical work: build a working compiler/interpreter for computing with numbers of arbitrary size.

Language Processor

A language processor is a program that processes programs/code written in a formal language:

- Compiler
- Interpreter
- Translator
- Parser
- Syntax-Checker
- …

Example: Bank Account Search

Query: `"billa > 20 last month"`

Example: Bank Account Search

Query: `"billa > 20 last month"`

Grammar

```
query: Id amountSearch timeSearch;
amountSearch: (gt | lt | eq);
gt: '>' Number;
lt: '<' Number;
eq: '=' Number;
...
```

Example: Bank Account Search

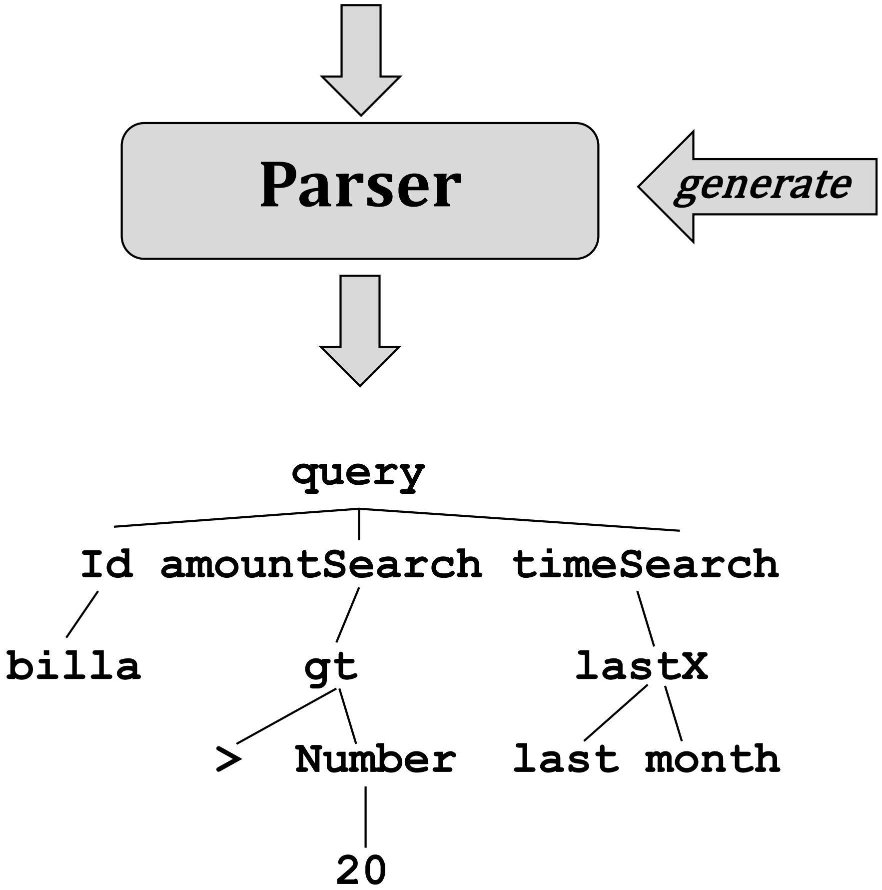
Query: **"billa > 20 last month"**

Grammar


```
query: Id amountSearch timeSearch;
amountSearch: (gt | lt | eq);
gt: '>' Number;
lt: '<' Number;
eq: '=' Number;
...
```

Example: Bank Account Search

Query: `"billa > 20 last month"`



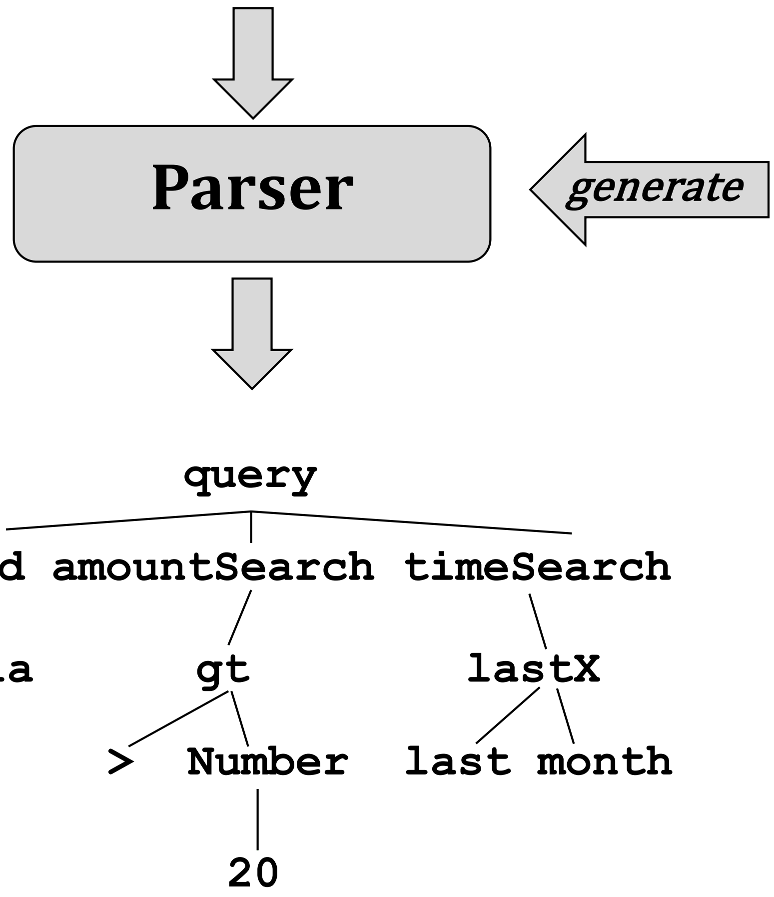
Abstract Syntax Tree (AST)

Grammar

```
query: Id amountSearch timeSearch;
amountSearch: (gt | lt | eq);
gt: '>' Number;
lt: '<' Number;
eq: '=' Number;
...
```

Example: Bank Account Search

Query: **"billa > 20 last month"**

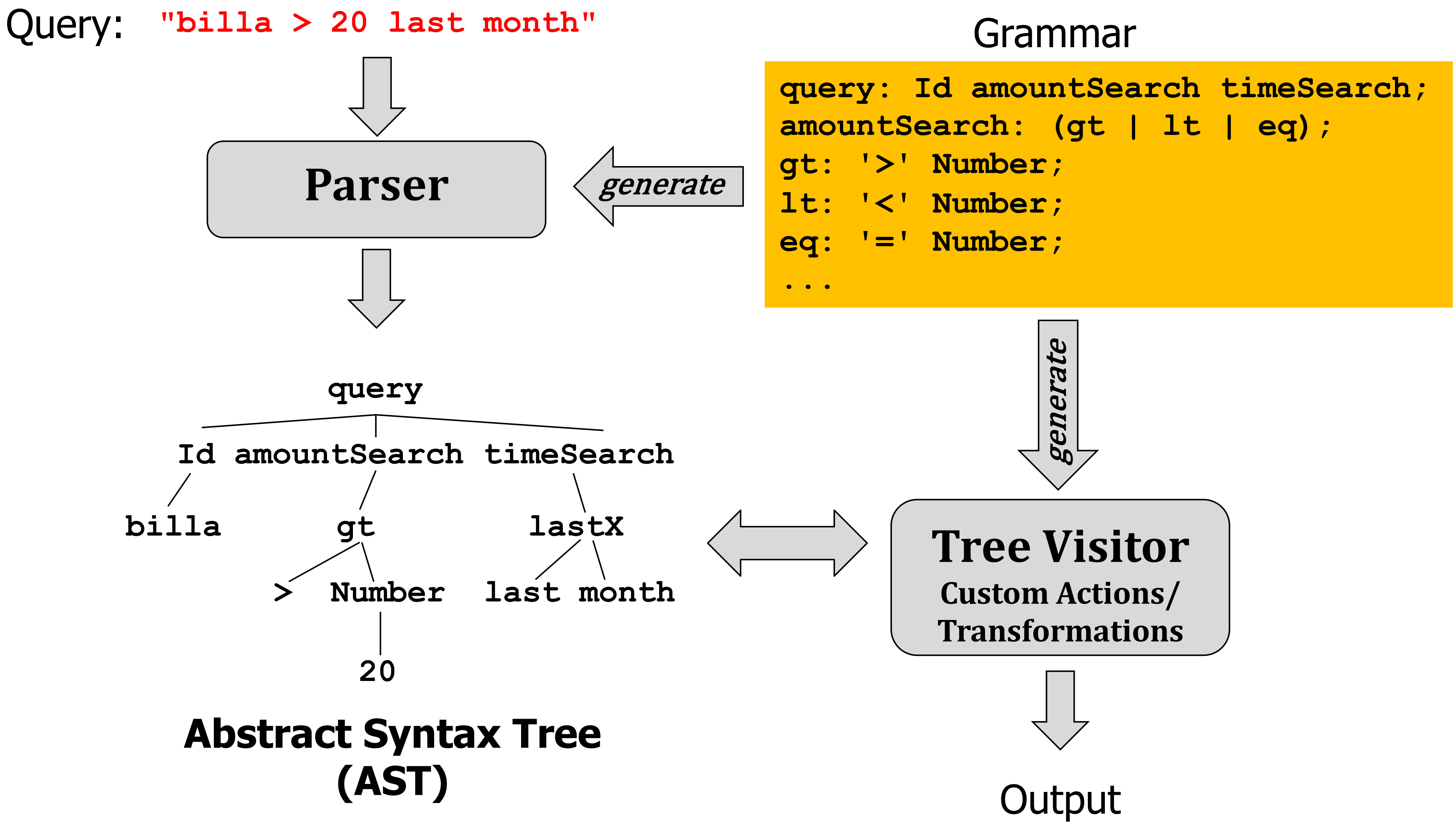


Abstract Syntax Tree (AST)

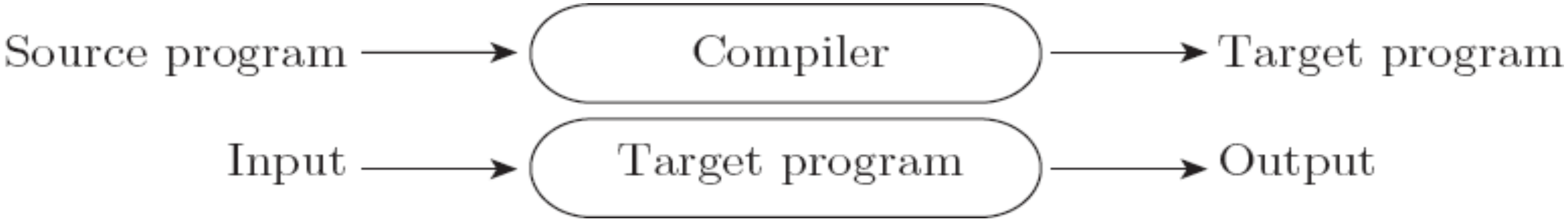
Grammar

```
query: Id amountSearch timeSearch;
amountSearch: (gt | lt | eq);
gt: '>' Number;
lt: '<' Number;
eq: '=' Number;
...
```

Example: Bank Account Search



Compiler

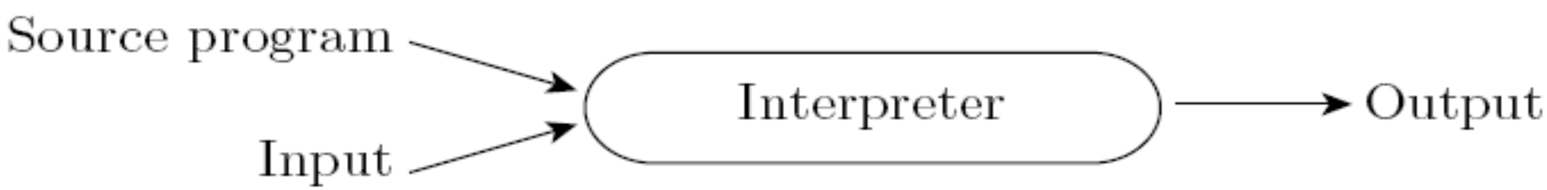


- A compiler is a program that translates a source program written in one language (source language) into an equivalent target program in another language (target language).

- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce output.

- A Source-to-Source compiler translates a target program into another high-level language (e.g., C++ → C).

Interpreter



- Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

- The interpreter is the locus of control during execution and stays around for the execution of the program.

Compilation vs. Interpretation

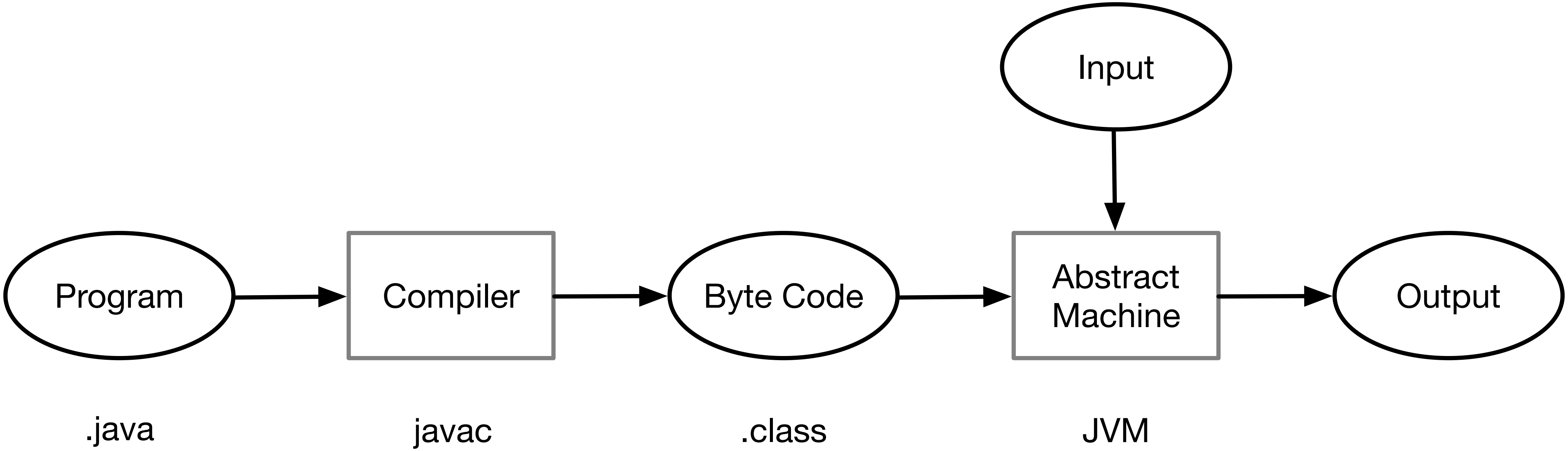
- The machine-language program produced by a compiler is usually much faster than an interpreted program because a compiler can do more optimizations.

- Compiled code usually exhibits better safety/correctness
  - check all names are defined (classes, methods, fields, variables, ...)
  - check that names have correct type
  - check visibility rules (public, private), …

- Compilation may reduce flexibility through static bindings and static type checks. Web programming often requires more flexibility (JavaScript, PHP, …)

- An interpreter can usually give better error diagnostics, because it executes the source program statement by statement.

Compilation vs. Interpretation - Java



A Java source program is compiled into an intermediate form called bytecode.

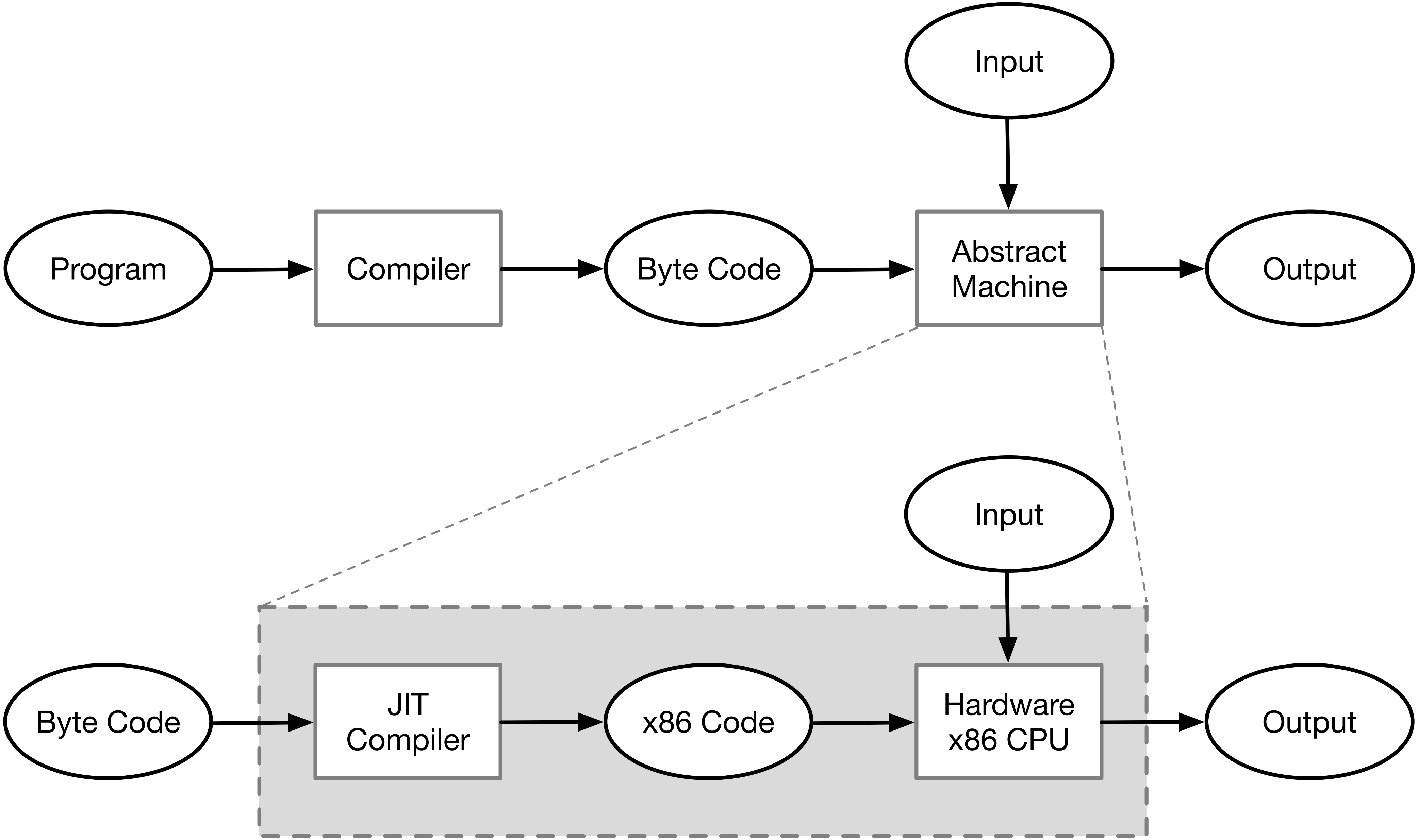
The bytecode is then interpreted by a virtual machine.

Dynamic and Just-in-Time Compilation

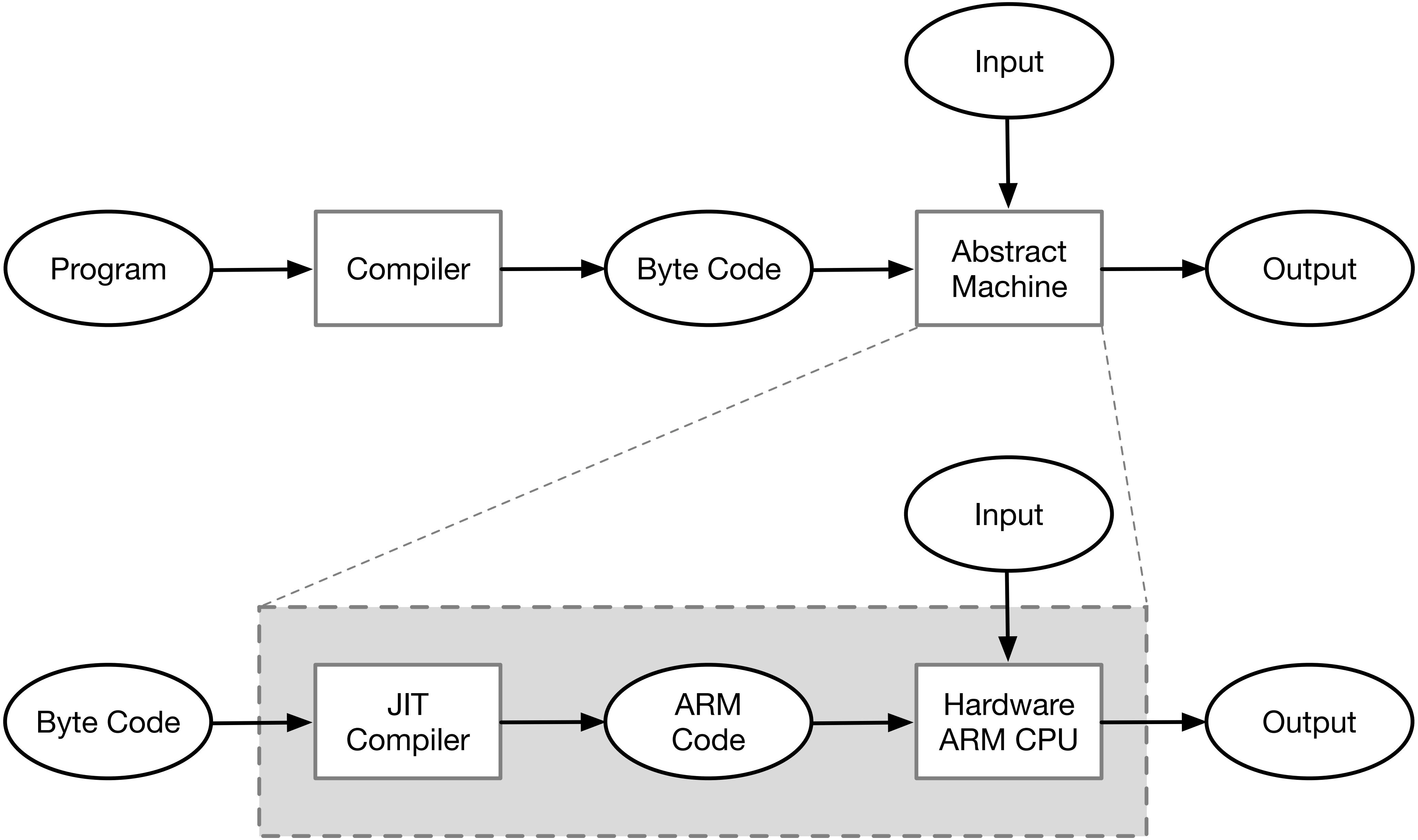
- Lisp or Prolog invoke the compiler on the fly, to dynamically translate newly created source code into machine language, or to optimize the code for a particular input set.
- The main C# compiler produces .NET Common Intermediate Language (CIL), which is then translated into machine code immediately prior to execution.
- The Java compiler generates platform-independent byte code. The Java Just-in-Time (JIT) compiler may compile byte code into machine code at runtime when the program is executed.

See: Programming Language Pragmatics, Fourth Edition, Michael L. Scott

Just-in-Time Compilation

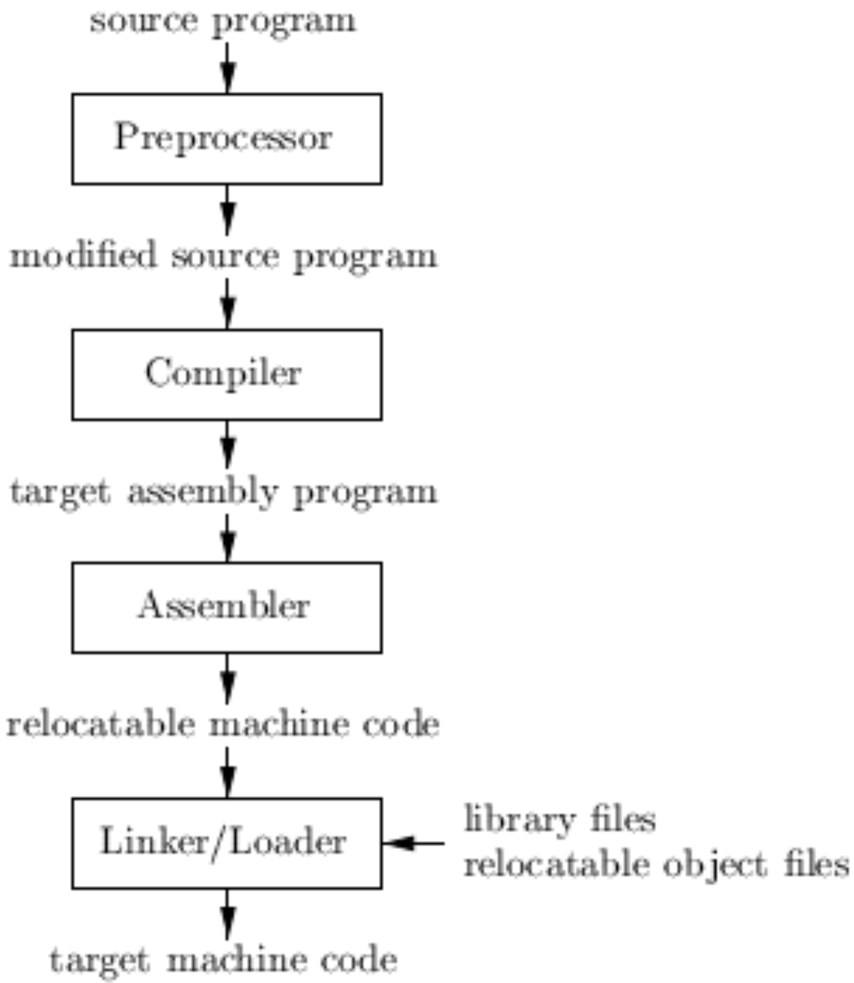


Just-in-Time Compilation



Language Processing System (e.g. C/C++)

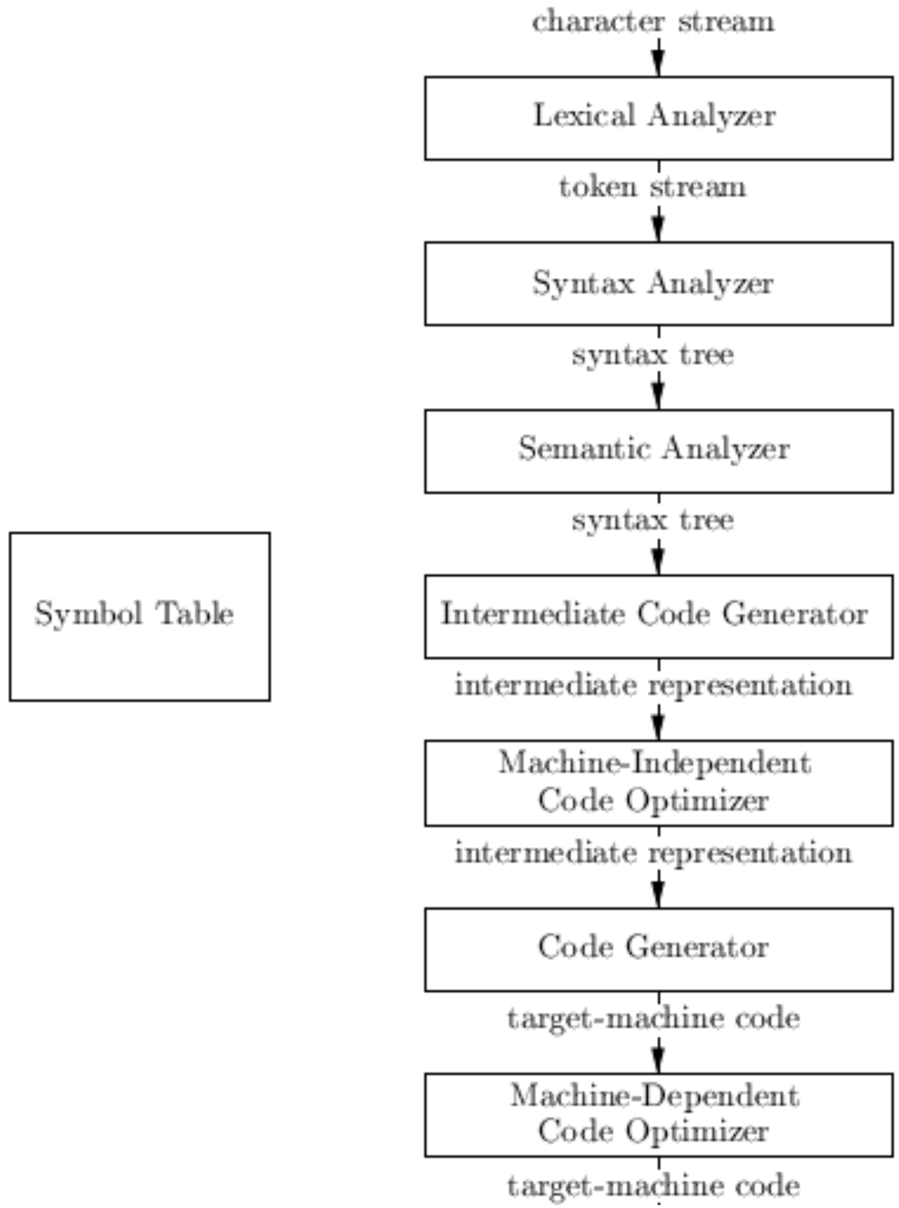
- **Preprocessor**
  - preprocesses source files; expands macros;
- **Compiler**
  - produces assembly language code
- **Assembler**
  - produces relocatable machine code
- **Linker/Loader**
  - links together relocatable object files and libraries; resolves external memory addresses
  - loads all executable object files into memory for execution



Structure of a Compiler

- **Frontend Phases**
  - Lexical Analysis (Scanning)
  - Syntax Analysis (Parsing)
  - Semantic Analysis

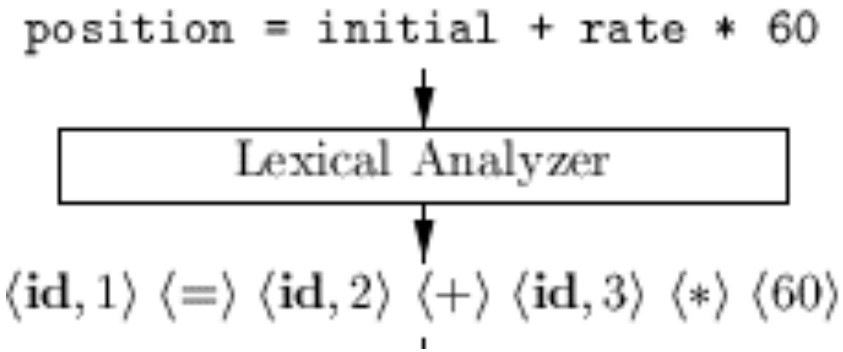
- **Backend Phases**
  - Intermediate Code Generation
  - Machine-Independent Optimizations
  - Code Generation
  - Machine Dependent Optimization



Lexical Analysis

- The lexical analyzer (scanner, lexer) reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

- For each lexeme, the lexical analyzer produces as output a token, e.g., of the form **<token-name, attribute-value>**



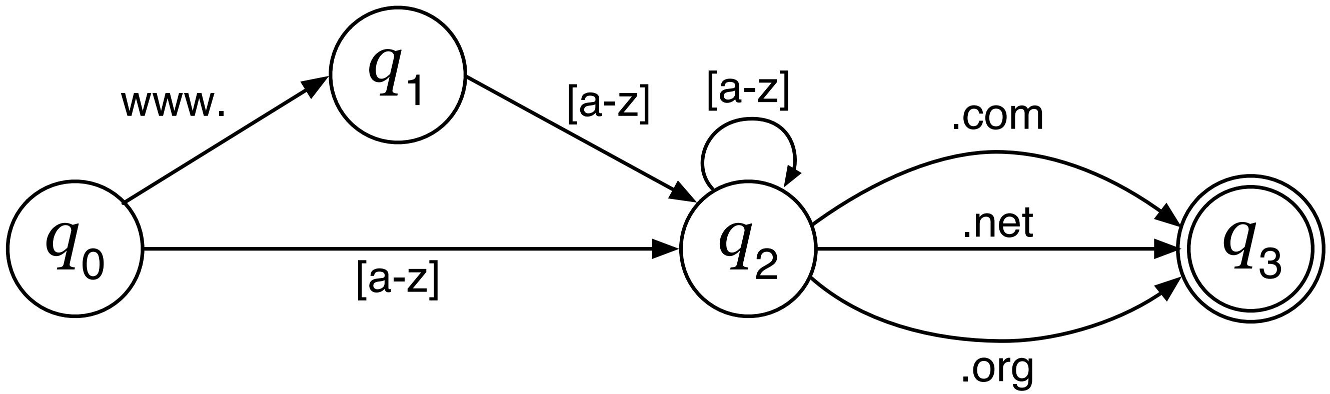
Regular Grammars/Languages

- Defined via regular expressions or finite state automatons.

- Scanners (lexical analyzers) are usually generated/constructed based on a regular-expression description of the tokens of a language.

- Can be handled very efficiently (linear time) but are quite limited.

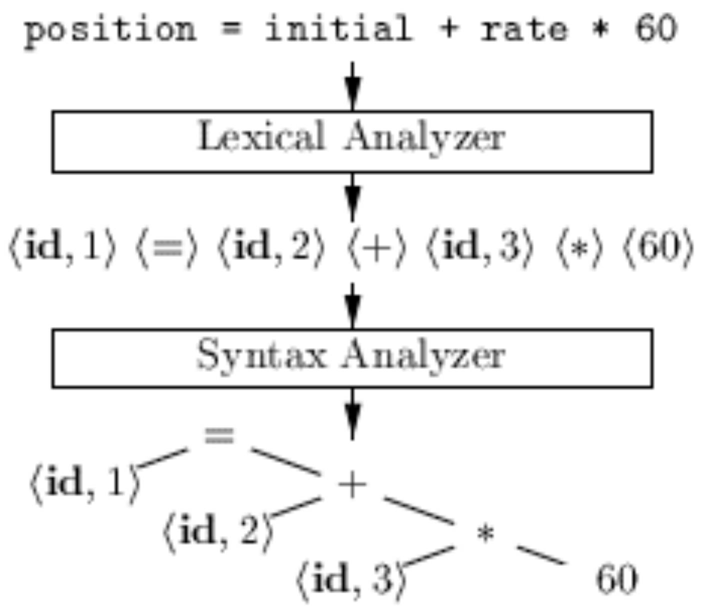
(www.)?[a-z]+(.com|.net|.org)



Syntax Analysis

- Based on the tokens produced by the lexical analyzer, the parser creates an abstract syntax tree (AST) that represents the grammatical structure of the program.

- In the AST each interior node represents an operation and the children of the node represent the arguments of the operation.



Context-Free Grammars/Languages

- The syntax of programming languages is usually specified with a (sub-class) of a context-free grammar, for which a parser can be constructed or generated automatically.

- Context-free grammars can be handled efficiently (polynomial time).

Context-Free Grammars/Languages

- **Top-Down Parsers**: LL(1), LL(k), ALL(*)
    - Left-to-right processing of input & Leftmost derivation; 1/k look ahead
    - Predictive, recursive-descendant; cannot handle left recursion;
    - Construct parse tree from root to leaves
    - Example: ANTLR (LL(k), k arbitrary)

- **Bottom-Up/Shift-Reduce Parsers**: LR(1), LALR(1)
    - Left-to-right processing of input & Rightmost derivation;
    - More powerful that LL parsers; No problem with left recursion;
    - Construct parse tree from leaves to root
    - Example: Yacc (LALR(1))

Leftmost vs. Rightmost Derivation

Grammar:

```
E : E * E
  | E + E
  | num
```

Note: grammar is ambiguous

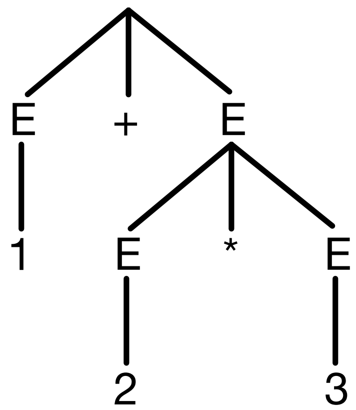
Note: grammar has left recursion

Leftmost Derivation

Derivation

```
E
E + E
1 + E
1 + E * E
1 + 2 * E
1 + 2 * 3
```

Parse Tree



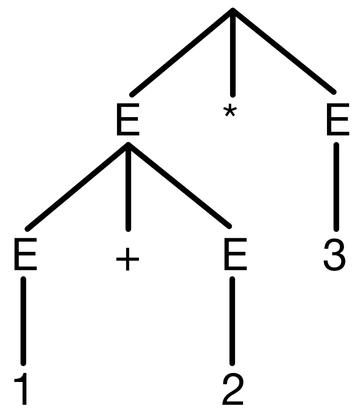
expanding the leftmost non-terminal at each step

Rightmost Derivation

Derivation

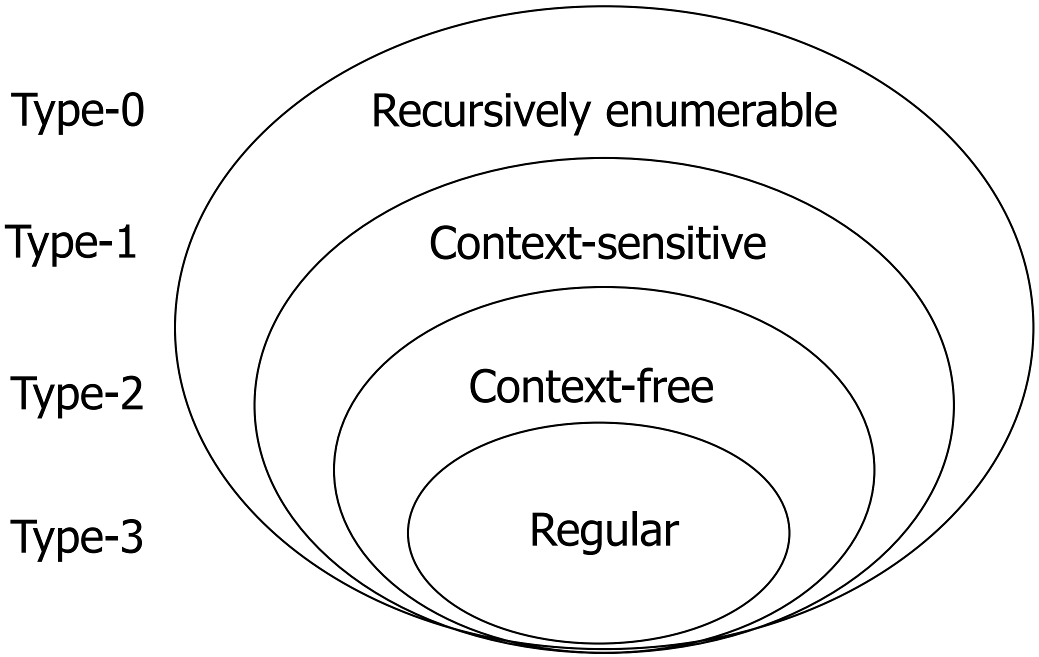
```
E
E * E
E * 3
E + E * 3
E + 2 * 3
1 + 2 * 3
```

Parse Tree

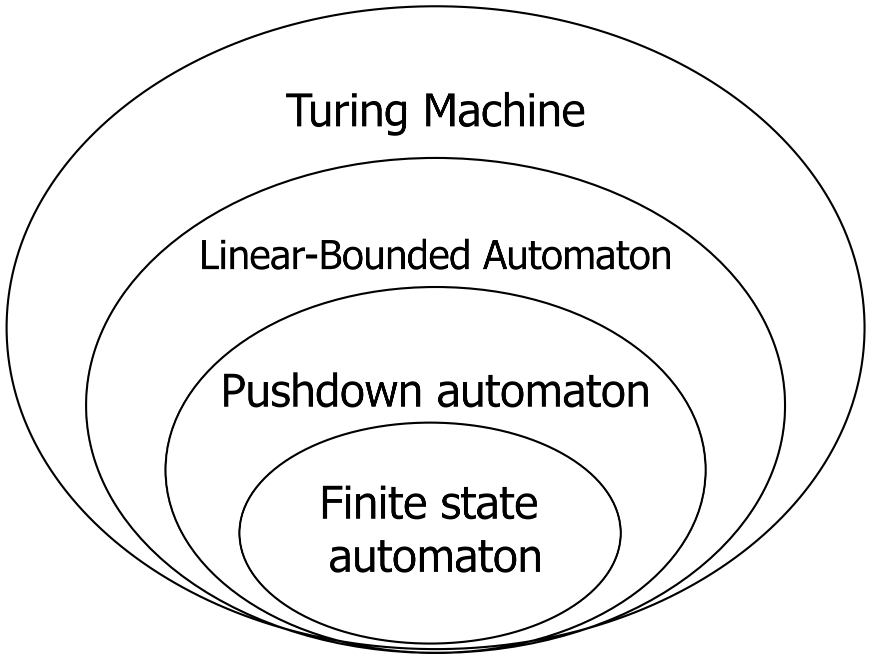


expanding the rightmost non-terminal at each step

Formal Grammars and Languages



Chomsky Hierarchy



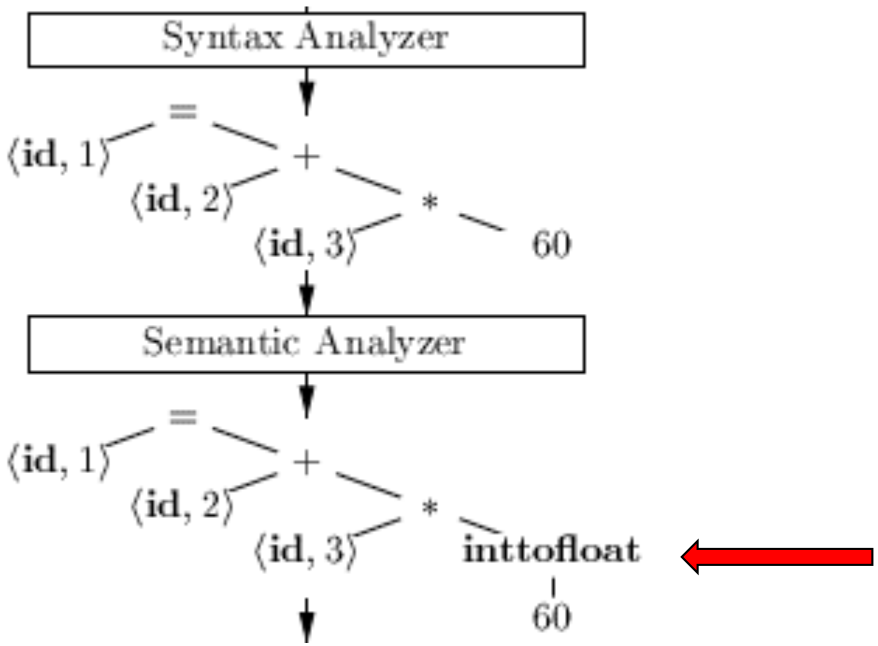
Accepting Automatons

Class	*Required form of rules*	*Examples*
Type-0	$\gamma \to \alpha$ (unrestricted)	$L = \{w \mid w \text{ accepted by a Turing machine}\}$
Type-1	$\alpha A \beta \to \alpha \gamma \beta$	$L = \{a^n b^n c^n \mid n > 0\}$
Type-2	$A \to \alpha$	$L = \{a^n b^n \mid n > 0\}$
Type-3	$A \to a \quad A \to aB$	$L = \{a^n \mid n \geq 0\}$

A, B non-terminal
α, β, γ string (terminals, non–terminals)
α, β may be empty

Semantic Analysis

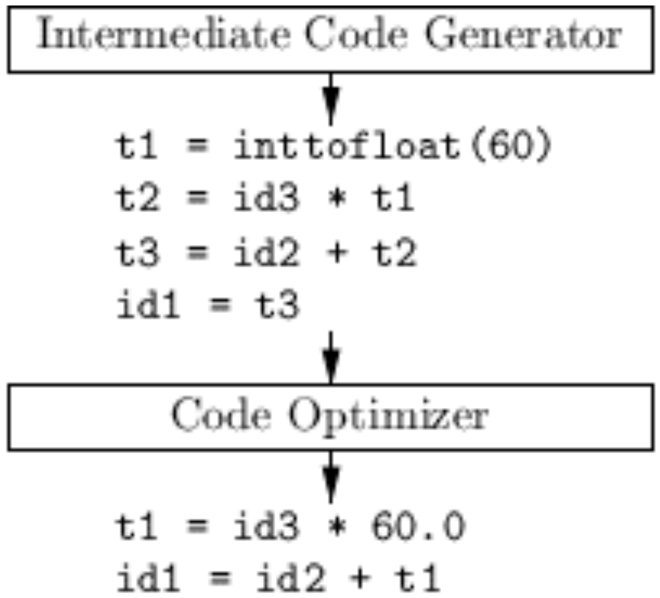
- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.



Code Optimization

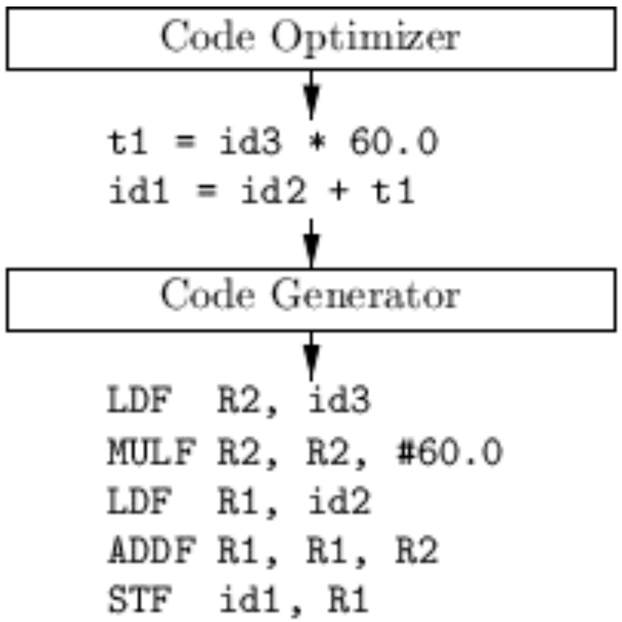
- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

- Usually better means faster, but other objectives may be desired, such as shorter code, less memory requirements, or target code that consumes less power.



Code Generation

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.



Symbol Table Management

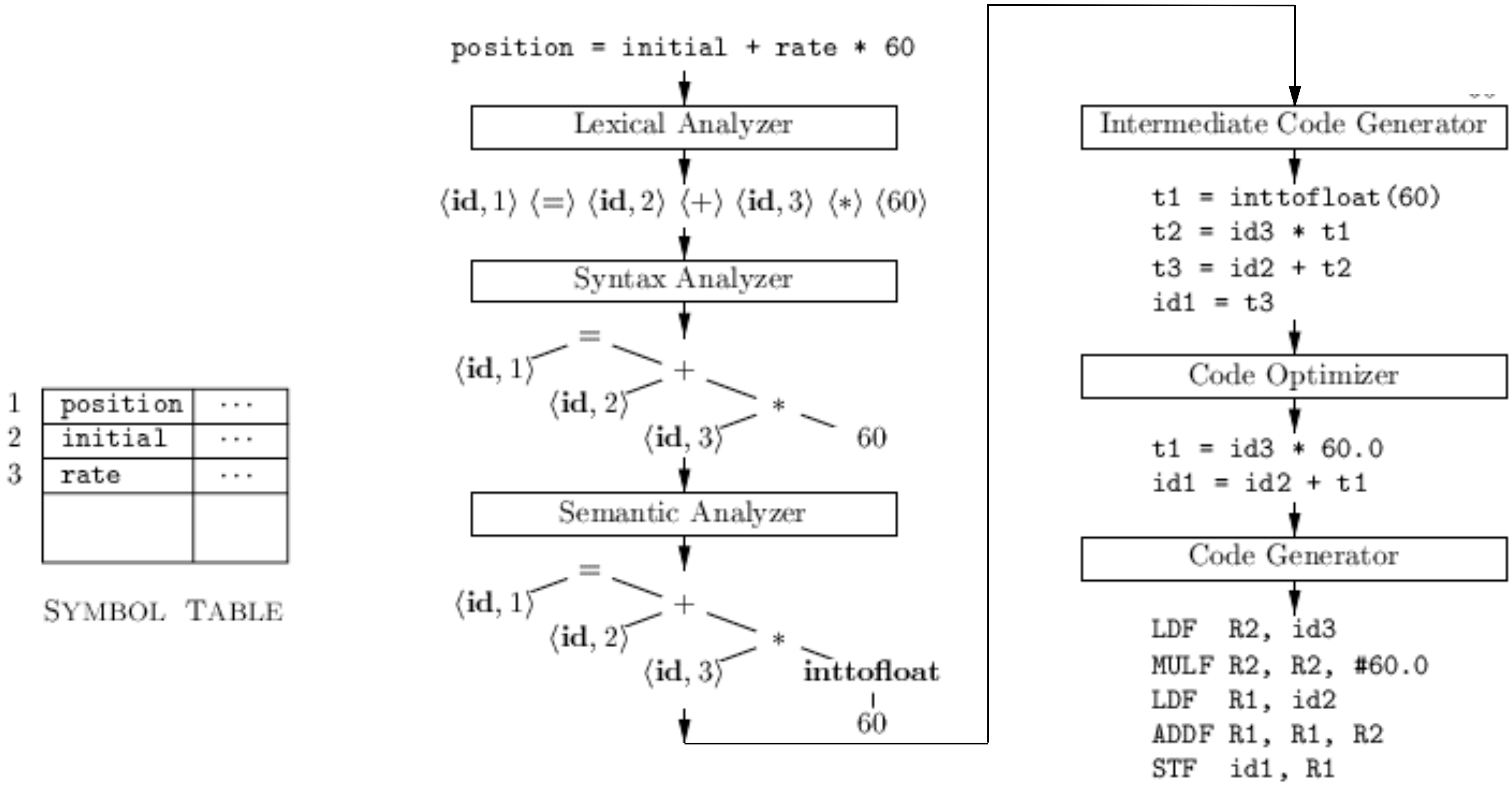
- An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

- These attributes may provide information about the storage allocated for a name, its type, its scope, and in the case of procedure names, the number and types of its arguments, the method of passing each argument (e.g., by value or by reference), and the type returned.

1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

Translation of an assignment statement



Compiler Construction Tools

- Scanner generators produce lexical analyzers from a regular-expression description of the tokens of a language.

- Parser generators automatically produce syntax analyzers from a grammatical description of a programming language.

- Syntax-directed translation engines produce collections of routines for walking a parse tree and generating intermediate code.

Compiler Construction Tools

- Code-generator generators produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

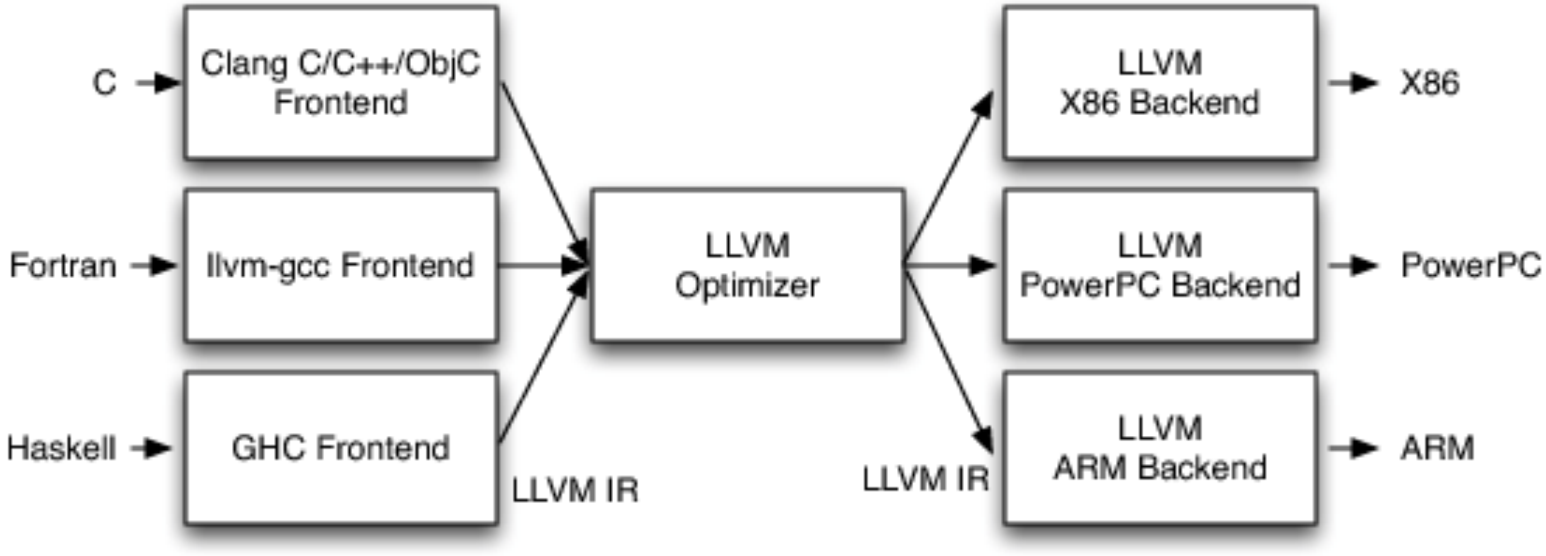
- Data-flow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to others.

- Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Compiler Infrastructures

- Some compiler collections have been created around carefully designed **intermediate representations** that allow combining front ends and back ends for different languages.
- With these collections, compilers can be built that accept different source languages and generate code for different target machines.

Example: **LLVM-based Compilers**



Applications of Compiler Technologies

- Implementation of High-Level Languages and Domain-Specific Languages

  Mitigate tradeoff between high level of abstraction and efficiency

- Optimizations for computer architectures

  memory hierarchy (caches), parallelism, heterogeneous cores (big.LITTLE, Alder Lake)

- Design of new computer architectures

  RISC, VLIW, SIMD, GPU, TPU, NPU, …

- Program translation

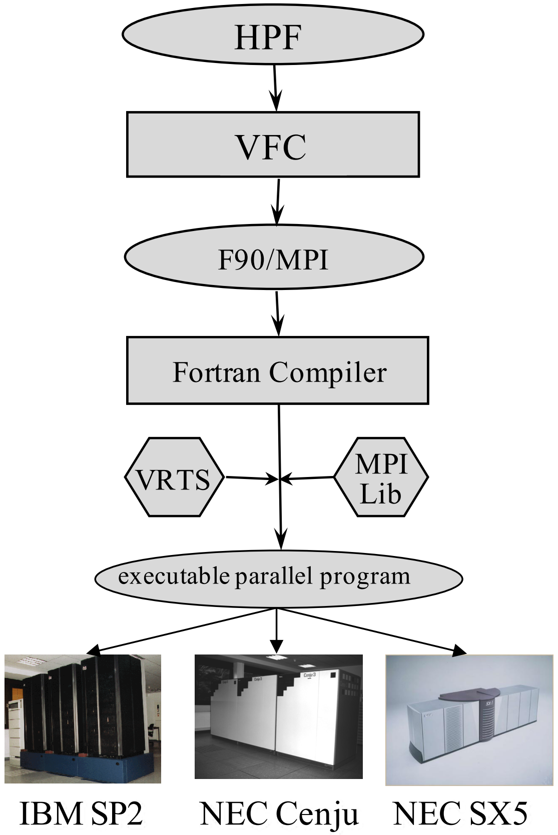
  binary translation, hardware synthesis, database queries, …

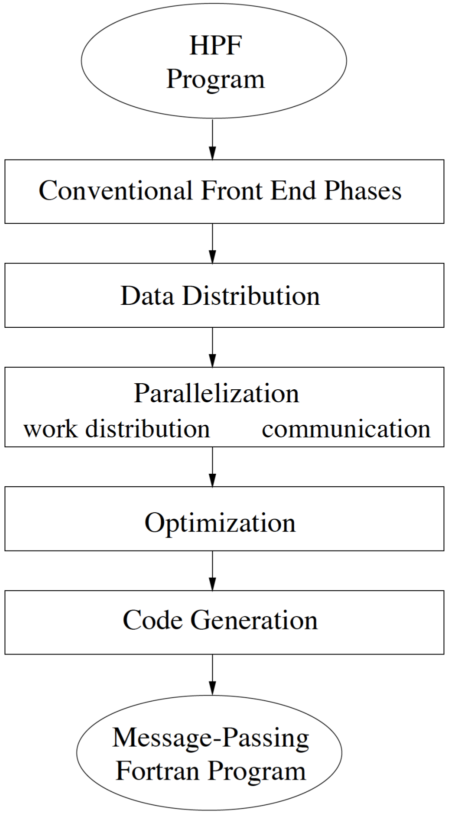
- Software productivity tools

  error checkers, type checking, memory management, …

VFC - Parallelizing Compiler

Compiling sequential Fortran (HPF) programs into parallel message-passing programs (MPI) for clusters and supercomputers.





Siegfried Benkner, Hans Zima. Compiling High Performance Fortran for distributed-memory architectures, Parallel Computing, 1999. https://www.sciencedirect.com/science/article/abs/pii/S0167819199000745

Some Remarks on Compilers

- Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages.

- Compilers are also critical in making (high-performance) computer architectures effective on users' applications.

- Compilers are used as a tool in evaluating architectural concepts before a computer/processor is built.

Some Remarks on Compilers

- Compiler writing is challenging. A compiler by itself is a large program possibly consisting of millions of lines of code. (GCC: ~ 15 MLOCs)

- Models used in compilers comprise regular expression, finite-state machines, context-free grammars, trees, graphs, matrices, linear programs, …

Some Remarks on Compilers

- The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code.

- A compiler must translate correctly the potentially infinite set of programs that could be written in the source language.

- However, the problem of generating optimal target code from a source program is undecidable in general.

ANTLR

ANTLR (ANother Tool for Language Recognition) is a parser generator for reading, processing, executing, or translating structured text or binary files.

It's widely used to build languages, tools, and frameworks.

From a grammar, ANTLR generates a parser that can build and walk parse trees.

ANTLR generates parsers in different languages including Java, Python, JavaScript, Go.

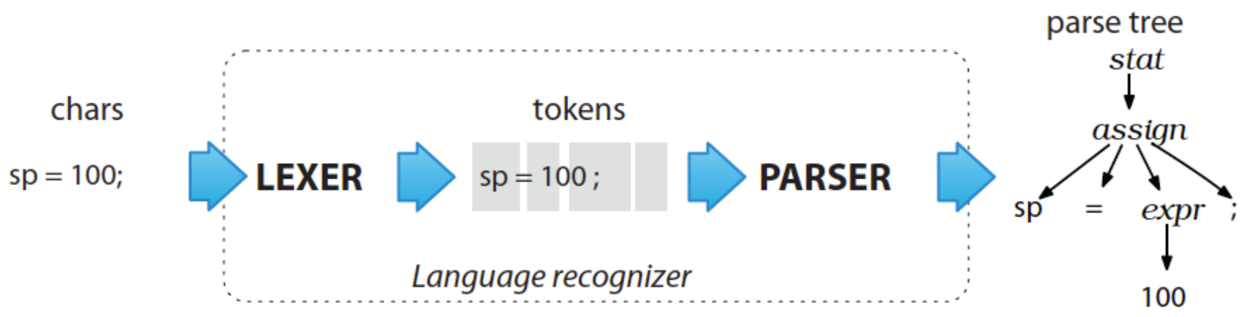
https://www.antlr.org/

ANTLR – Theoretical Background

- ANTLR can generate top-down (recursive descendant) LL(k) parsers, with an arbitrary k.
- An LL(k) parser uses k tokens of lookahead.
- An LL parser parses the input from Left to right, and constructs a Leftmost derivation of the sentence (as opposed to an LR parser which constructs a Rightmost derivation).
- An LL parser can parse a subset of context-free languages.

ANTLR

- ANTLR generates lexers, parsers and tree walkers.



- From grammar rules ANTLR generates recursive-descent parsers, which are a collection of recursive methods, one per rule.

```
assign : ID '=' expr ';' ;
```

```
void assign() {   // method generated from rule assign
   match(ID);     // compare ID to current input symbol then consume
   match('=');
   expr();         // match an expression by calling expr()
   match(';');
}
```

ANTLR – Recursive Descendent Parser

```
stat: assign
    | ifstat
...
;
```

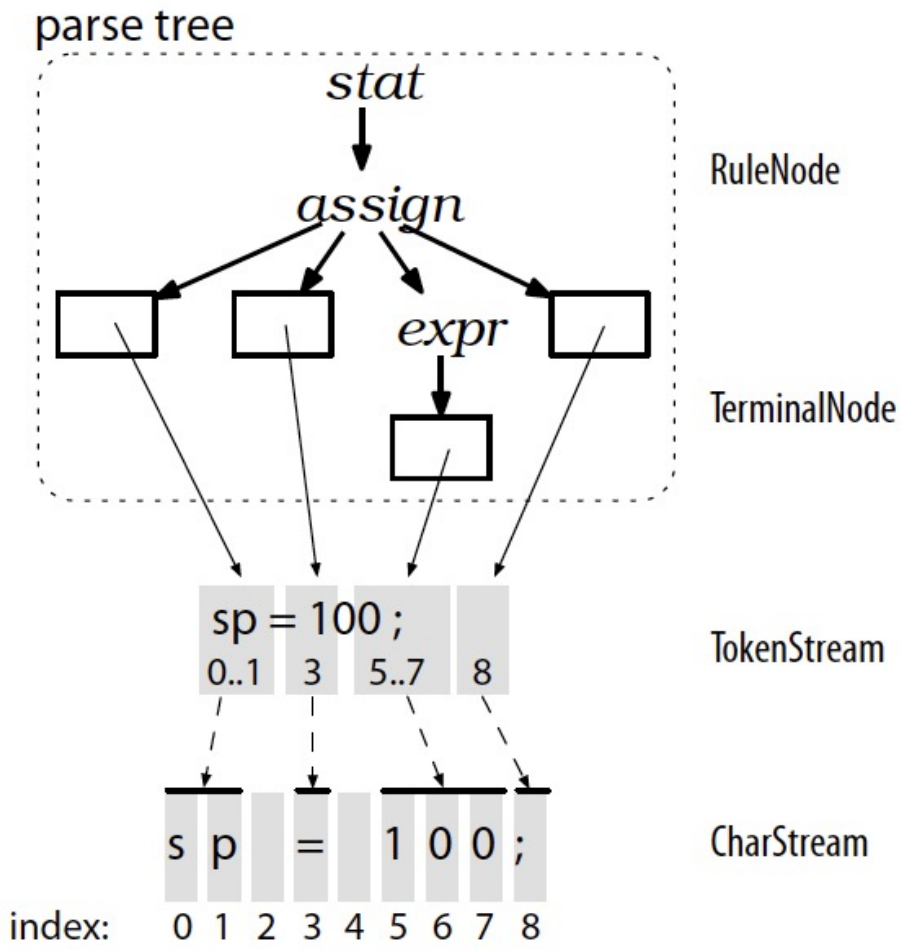
```
void stat() {
  switch ( «current input token» ) {
    CASE ID : assign(); break;
    CASE IF : ifstat(); break; // IF is token type for keyword 'if'
    CASE WHILE : whilestat(); break;
    ...
    default : «raise no viable alternative exception»
  }
}
```

Lexer, Parser

- The lexer processes characters and passes tokens to the parser via a token stream.

- The parser checks syntax and creates a parse tree.

- The corresponding ANTLR classes are: **`CharStream`**, **`Lexer`**, **`Token`**, **`TokenStream`**, **`Parser`**, and **`ParseTree`**.

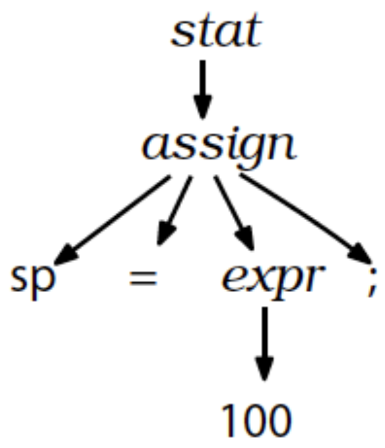


- **`RuleNode`** and **`TerminalNode`** are subclasses of **`ParseTree`** that correspond to subtree roots and leaf nodes.

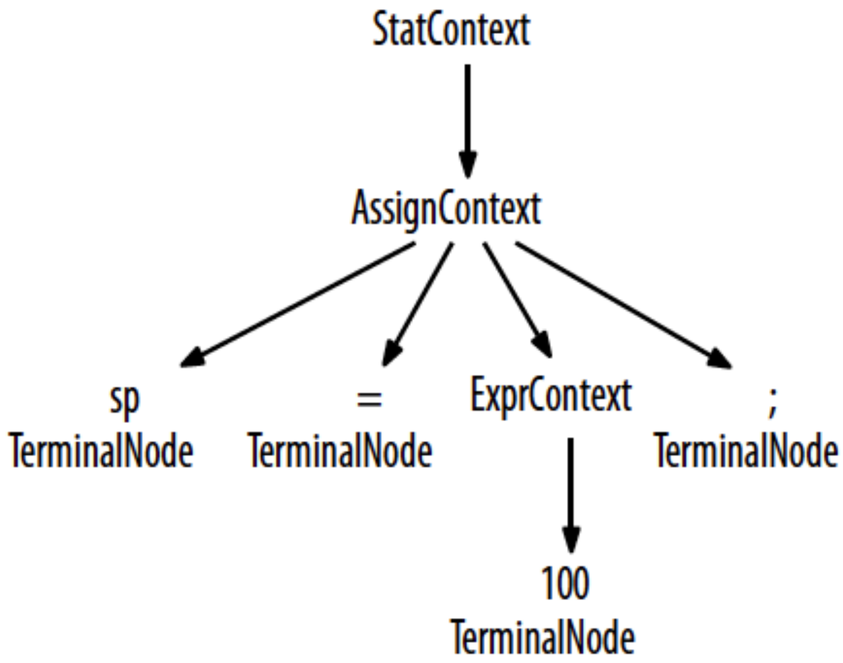
Parse Tree

- Classes of the subtree roots for assignment statement example:

  `StatContext`, **`AssignContext`**, and **`ExprContext`**:



Parse tree



Parse tree node class names

Assignment 4 - BigCalc

- BigCalc is an interpreter that can evaluate expressions with decimal numbers of arbitrary length/precision.

File: expr.bc

```
1234567890123456789 0.0987654321 +
99999999988888888777777766666.7777777 / 333.333;
```

```
$ java BigCalc expr.bc
result: 300000301201534523423423389.7650984311
$
```

Assignment 4 – ANTLR Grammar

File: BigCalc.g4 (Parser rules)

```
grammar BigCalc;

expressionStatement
        : expression ';' EOF
        ;
```

Label
ctx.op().getText()

Alternative
Labels

```
expression
        : expression op=('*' | '/') expression  # mulDiv
        | expression op=('+' | '-') expression  # addSub
        | Number                                # num
        ;

...
```

Note: **EOF** should be used at the end of the entry rule to ensure that the whole input file will be parsed.

ANTLR – Parse-Tree Visitors and Listeners

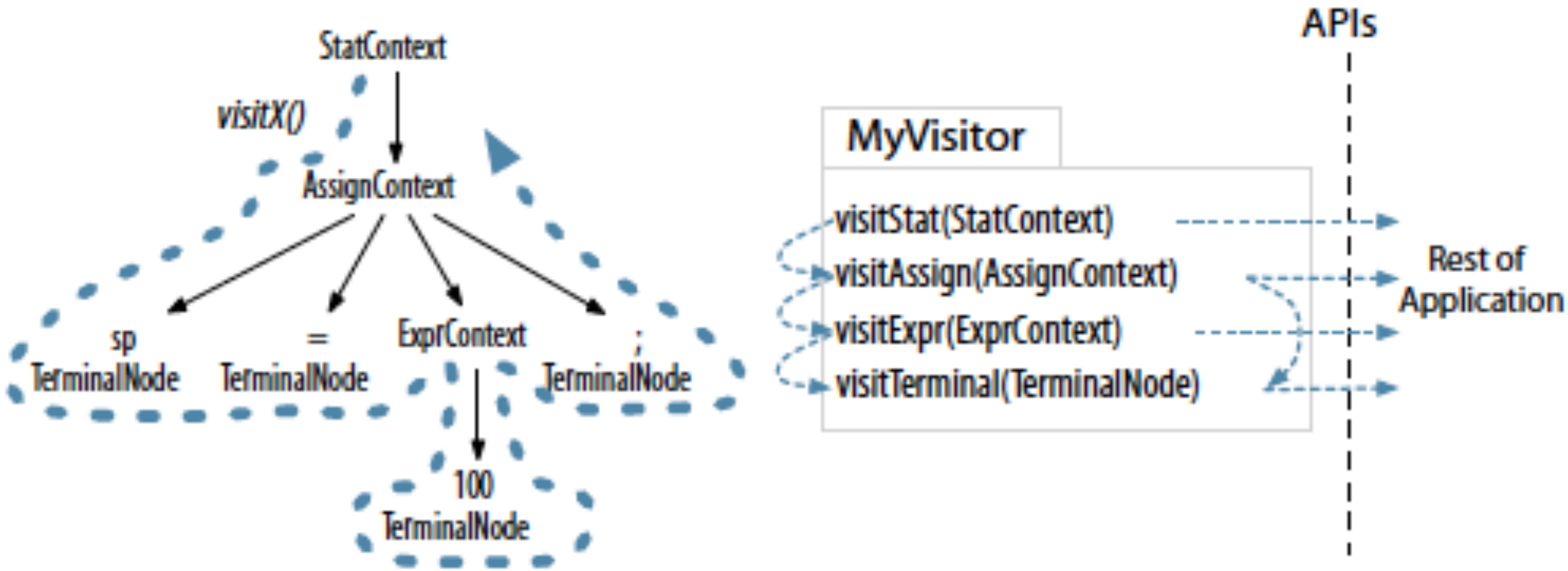
ANTLR provides support for two tree-walking mechanisms:

1. The visitor interfaces and corresponding classes in order to realize user-defined tree walkers (based on the visitor design pattern).

2. The parse-tree listener interface provides callbacks to events triggered by the built-in tree walker.

Parse-Tree Visitors

- If option **`-visitor`** is specified, ANTLR generates a visitor interface with a corresponding visit method for each grammar rule.



- ANTLR parse-tree visitors follow the visitor design pattern, and allow the user to control the tree walk.

Parse-Tree Visitors

- ANTLR provides a class (…**`BaseVisitor`**) with default implementation methods, which then can be overwritten by the user as needed.

- To initiate a walk of the tree, the application-specific code needs to create an implementation (e.g., **`BigCalcVisitorImpl`**) of the visitor interface (e.g., **`BigCalcVisitor`**) and call **`visit()`**.

```
...
ParseTree tree = parser.expressionStatement();

BigCalcVisitor<BigDecimal> visitor = new BigCalcVisitorImpl();

BigDecimal result = visitor.visit(tree);
...
```

Parse-Tree Visitors

- For each rule a corresponding visitor method is generated (if no Alternative Labels are present).
- If Alternative Labels (#) are present, a different visitor method for each alternative is generated.

```
grammar BigCalc;
expressionStatement
        : expression ';' EOF
        ;
expression
        : expression op=('*' | '/') expression  # mulDiv
```

```
public interface BigCalcVisitor<T> extends ParseTreeVisitor<T> {
  T visitExpressionStatement(
                   BigCalcParser.ExpressionStatementContext ctx);
  ...
  T visitMulDiv(BigCalcParser.MulDivContext ctx);
  ...
```

Parse-Tree Visitors

- The user needs to provide an application-specific parse-tree visitor which implements the generated visitor interface.

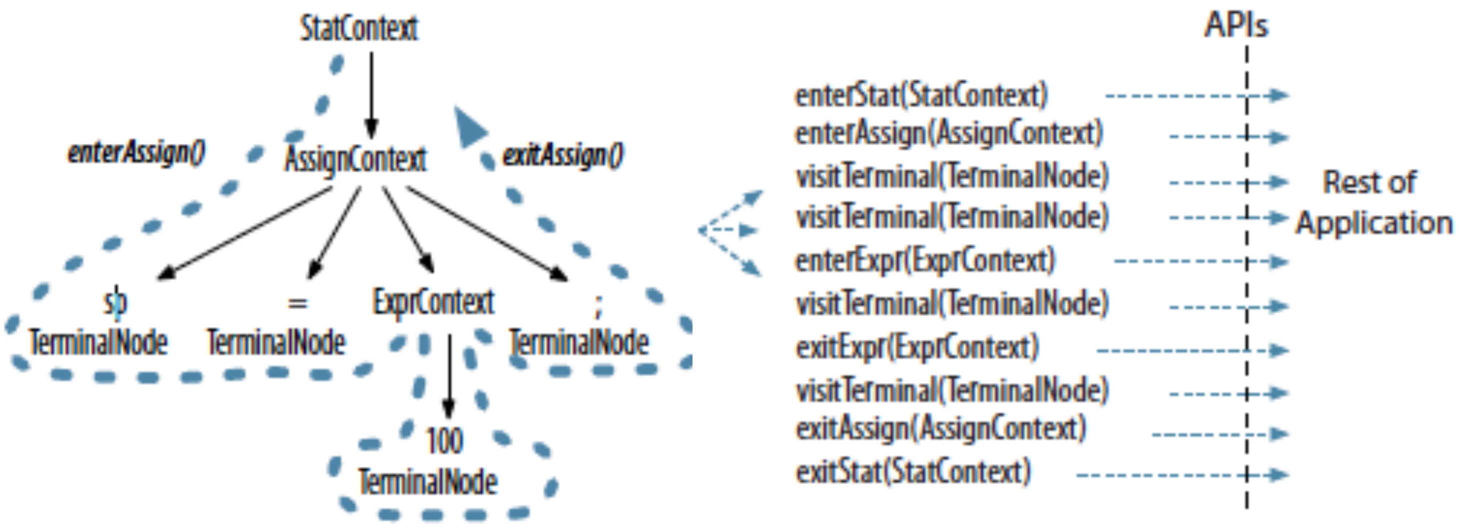
```
public class BigCalcVisitorImpl extends
                                 BigCalcBaseVisitor<BigDecimal> {

  @Override
  public BigDecimal visitExpressionStatement(
                    BigCalcParser.ExpressionStatementContext ctx) {
      ...
  }

  @Override
  public BigDecimal visitMulDiv(BigCalcParser.MulDivContext ctx) {
      ...
  }
```

Parse-Tree Listeners

- To walk a tree and trigger calls into a listener, ANTLR provides the class **`ParseTreeWalker`**.

- The user needs to provide an application-specific implementation of the generated **`ParseTreeListener`** interface.

- ANTLR generates a **`ParseTreeListener`** subclass specific to each grammar with enter and exit methods for each rule (or each Alternative Label).



ANTLR – Parse-Tree Listeners and Visitors

- A difference between listeners and visitors is that listener methods aren't responsible for explicitly calling methods to walk their children.

- Visitors, on the other hand, must explicitly trigger visits to child nodes to keep the tree traversal going.

- Using parse-tree listeners and visitors allows decoupling a grammar from application-specific code, unlike methods where application specific-code is directly embedded in a grammar.

```
grammar PropertyFile;
file : { «start file» } prop+ { «finish file» } ;
prop : ID '=' STRING '\n' { «process property» } ;
```

Example: Java 8 - ANTLR Grammar

Excerpt from Java8Parser.g4 https://github.com/antlr/codebuff/blob/master/grammars/org/antlr/codebuff/Java8.g4

```
normalClassDeclaration
    : classModifier* 'class' Identifier typeParameters?
      superclass? superinterfaces? classBody
    ;

classModifier
    : annotation
    | 'public'
    | 'protected'
    | 'private'
    | 'abstract'
    | 'static'
    | 'final'
    | 'strictfp'
    ;
```

Precedence

- ANTLR resolves ambiguities based on the order of the given alternatives, implicitly allowing to specify operator precedence.

```
expr : expr '*' expr
     | expr '+' expr
     ;
```

Associativity

- By default, ANTLR associates operators left to right.

- For operators like exponentiation that group right to left, the associativity has to be specified explicitly using option `assoc`.

```
expr : expr '^'<assoc=right> expr // ^operator is right associative
     | INT
     ;
```

Left Recursion

- Conventional top-down parser generators, cannot handle left-recursive rules.
- Since version 4, ANTLR can handle direct left recursion.

```
expr : expr '^'<assoc=right> expr
     | expr '*' expr
     | expr '+' expr
     | INT
     ;
```

- Indirect left recursion cannot be handled.

```
expr : expo // indirectly invokes expr left recursively via expo
     | ...
     ;
expo : expr '^'<assoc=right> expr ;
```

Assignment 4 - BigCalc

- BigCalc is an interpreter that can evaluate expressions with decimal numbers of arbitrary length/precision.

File: expr.bc

```
12345678901234567890.0987654321 +
99999999988888888777777766666.7777777 / 333.333;
```

```
$ java BigCalc expr.bc
result: 3000003012015345234234233889.7650984311
$
```

Assignment 4 – ANTLR Grammar

File: BigCalc.g4 (Parser rules)

```
grammar BigCalc;

expressionStatement
        : expression ';' EOF
        ;
```

Label
ctx.op().getText()

Alternative
Labels

```
expression
        : expression op=('*' | '/') expression   # mulDiv
        | expression op=('+' | '-') expression   # addSub
        | Number                                 # num
        ;

...
```

Note: **EOF** should be used at the end of the entry rule to ensure that the whole input file will be parsed.

Assignment 4 – ANTLR Grammar

File: BigCalc.g4 ctd. (Lexer rules)

```
...
Number
          : Digit* '.' Digit+
          | Digit+
          ;

Digit
          : [0-9]
          ;

WS        : [ \t\r\n\u000C]+ -> skip
          ;

COMMENT
          :   '/*' .*? '*/' -> skip
          ;

LINE_COMMENT
          : '//' ~[\r\n]* -> skip
          ;
```

Assignment 4

```
...

public class BigCalc {
   public static void main(String[] args) {
      try {
         final CharStream input = CharStreams.fromFileName(args[0]);
         final BigCalcLexer lexer = new BigCalcLexer(input);
         final CommonTokenStream tokens = new CommonTokenStream(lexer);
         final BigCalcParser parser = new BigCalcParser(tokens);

         final ParseTree tree = parser.expressionStatement();

         final BigCalcVisitor<BigDecimal> visitor = new
                                             BigCalcVisitorImpl();
         final BigDecimal result = visitor.visit(tree);

         if (result != null)
            System.out.println("result: " + result.setScale(10,
                                            RoundingMode.HALF_UP));
      }
      ...
   }
}
```

Assignment 4 - Visitor Implementation

```
public class BigCalcVisitorImpl extends BigCalcBaseVisitor<BigDecimal> {
```

```
expressionStatement
     : expression ';'
```

```
    @Override
    public BigDecimal visitExpressionStatement(
                               BigCalcParser.ExpressionStatementContext ctx) {
        return visit(ctx.expression());
    }
```

```
expression
     : expression op=('*' | '/') expression  # mulDiv
```

```
    @Override
    public BigDecimal visitMulDiv(BigCalcParser.MulDivContext ctx) {
        final BigDecimal left = visit(ctx.expression(0));
        final BigDecimal right = visit(ctx.expression(1));
        if (ctx.op.getText().equals("*")) {
            return left.multiply(right);
        } else {
            return left.divide(right, 10, RoundingMode.HALF_UP);
        }
    }
    ...
```

Assignment 4 - Visitor Implementation ctd.

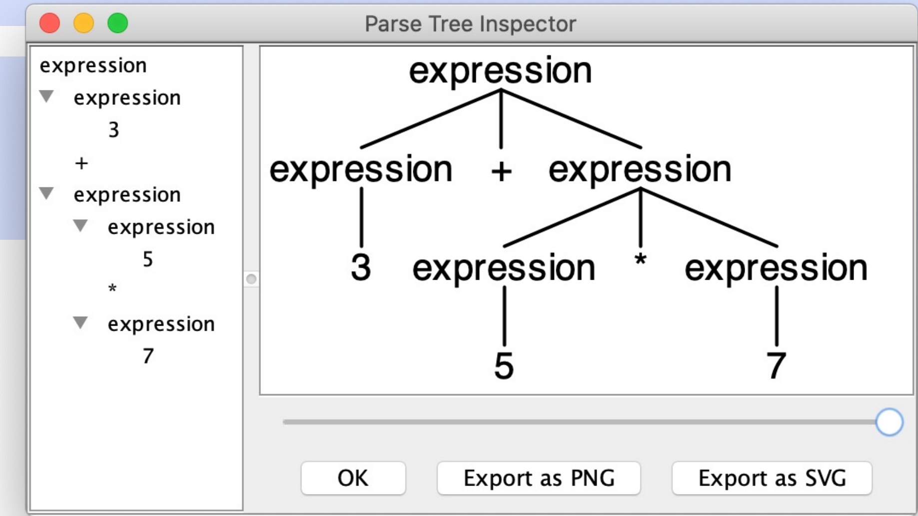
```
...

    @Override
    public BigDecimal visitAddSub(BigCalcParser.AddSubContext ctx) {
        final BigDecimal left = visit(ctx.expression(0));
        final BigDecimal right = visit(ctx.expression(1));
        if (ctx.op.getText().equals("+")) {
            return left.add(right);
        } else {
            return left.subtract(right);
        }
    }

    @Override
    public BigDecimal visitNum(BigCalcParser.NumContext ctx) {
        return new BigDecimal(ctx.Number().getText());
    }

}
```

Assignment 4

```
compile:
   java -jar antlr-4.7.1-complete.jar -visitor BigCalc.g4
   javac -cp antlr-4.7.1-complete.jar:. *.java

run:
   java -cp antlr-4.7.1-complete.jar:. BigCalc $(file)

viz:
   java -cp antlr-4.7.1-complete.jar:. org.antlr.v4.gui.TestRig BigCalc
expression -gui
```

```
$ make run file=expr1.bc
java -cp antlr-4.7.1-complete.jar:. BigCalc expr1.bc
result: 3000003012015345234234233889.7650984311
```

```
$ make viz
3+5*7
CTRL-D
```



Assignment 4 - Extensions

- Extend the interpreter BigCalc such that it can handle programs not just a single expression.
- The following functionality has to be provided:
    - A program is comprised of one or more statements. Each statement is terminated with a ";"
    - A statement is either an assignment statement (e.g., `t = 7;`) or an expression statement (e.g., `1 + 2 * s / u;`).
    - Expressions may contain parentheses and variables (e.g., `(1+x)*3`).
    - Names of variables start with a letter and zero or more digits.
    - Undefined variables have the value 0.
    - When program execution finishes, the result of the last statement is printed on the console.