3 Compilers

Interpreters and compilers:

∗ Assume a high-level program P takes input I and produces output O.

∗ An interpreter takes P and I and produces O.

∗ Compiling breaks this process into two parts.

∗ The compiler takes P and produces low-level machine code M.

∗ The processor takes M and I and produces O.

Remarks:

∗ The processor is an interpreter for a low-level language.

∗ The processor might be a virtual machine; see Java.

∗ Libraries may be part of the run-time environment or linked after compiling.

Benefits of compiling:

∗ The compiler performs tasks once, which would have to be repeated during interpretation.

∗ Analyse programs to check aspects of correctness.

∗ Optimise programs so they use less time and space.

Structure of a compiler:

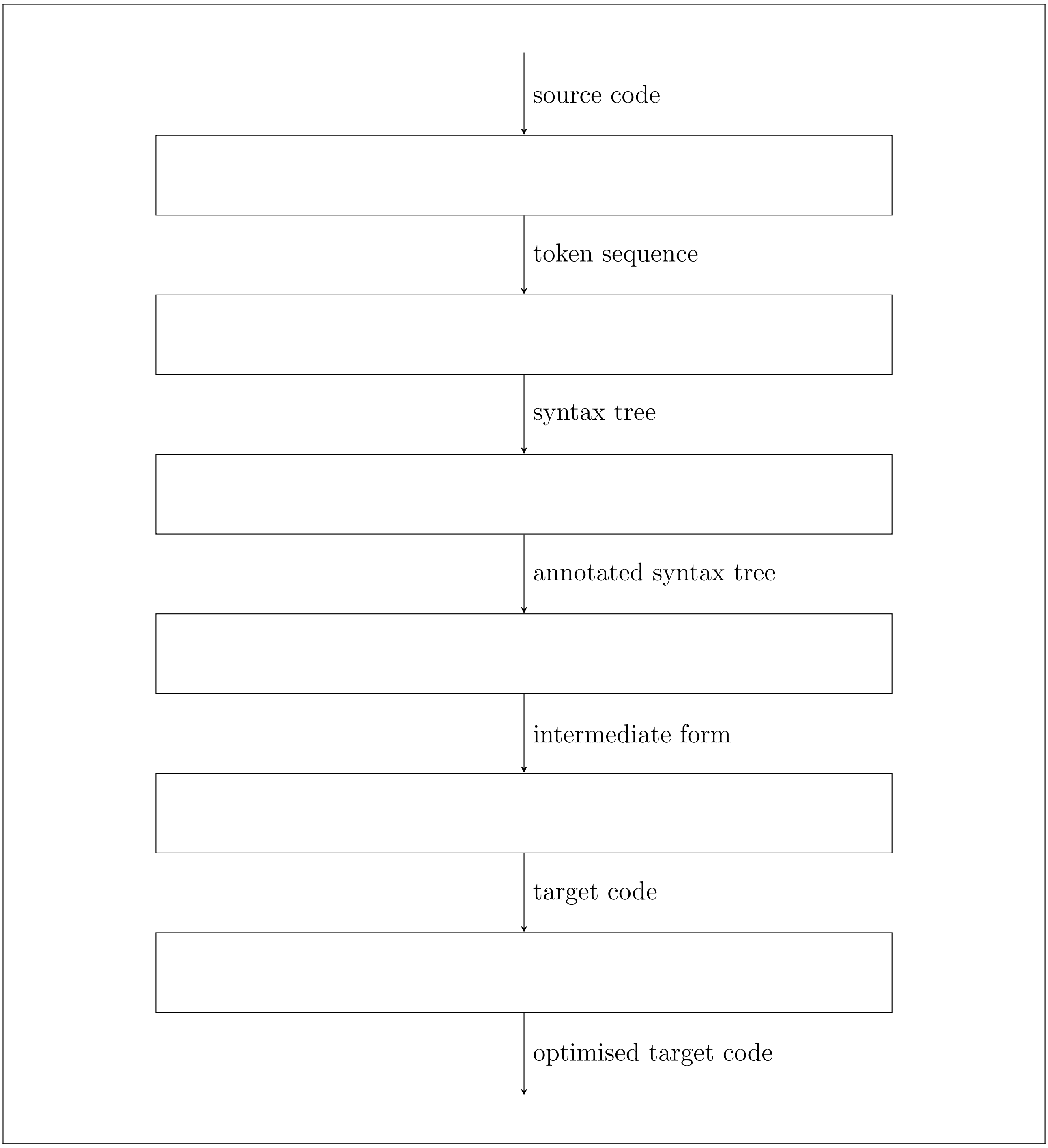
∗ The input of a compiler is a sequence of characters: the source-level program.

∗ The *syntax tree* constructed from this sequence reflects the structure of the program.

∗ Analyses such as type checking and some optimisations can be performed on this tree.

∗ The tree is ultimately converted to a low-level instruction sequence such as assembly code.

∗ A compiler is typically divided into phases.



Topics discussed in the following:

* lexical analysis
* syntax analysis
* semantic analysis
* optimisation
* code generation for virtual machines

Topics not discussed are:

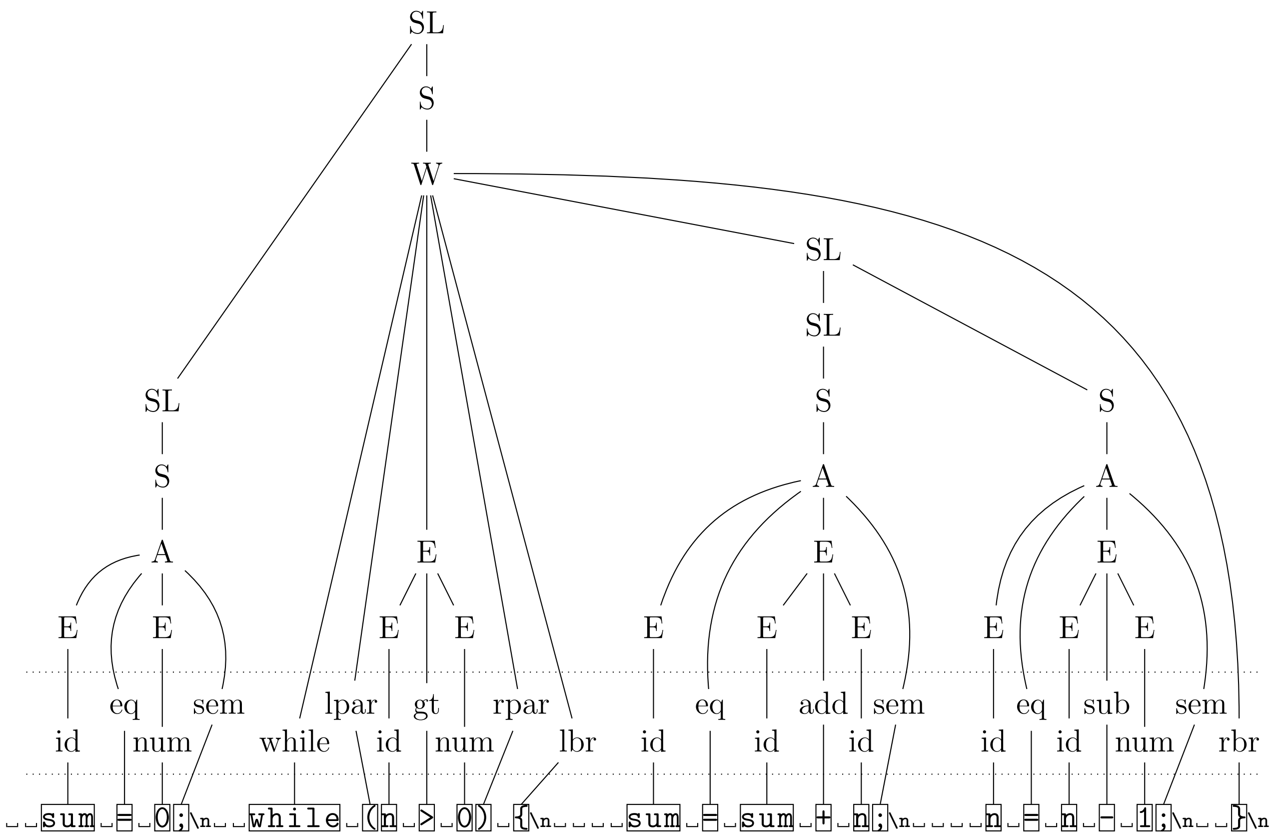
* code generation for real machines
* error-handling

The diagram below exemplifies the analysis:

* Consider the following fragment of a C program or a Java program:

```
sum = 0;
while (n > 0) {
  sum = sum + n;
  n = n - 1;
}
```

* At the bottom of the diagram is the sequence of characters of this program.
* Above that is the sequence of *tokens* generated by the scanner.
* On top is the syntax tree generated by the parser.



The nodes of the syntax tree have the following meaning:

* SL: statement list
* S: statement
* W: while-loop
* A: assignment
* E: expression
* id, eq, num, sem, while, lpar, gt, rpar, lbr, add, sub, rbr: tokens of the source program
* Tokens can be annotated with their position in the source code.
* White-space characters are eliminated by the scanner.

The syntax tree is translated to code for a low-level machine:

* The machine holds the values of variables in a small number of *registers*.
* Calculations take place on a *stack*.
* The contents of registers are pushed to and popped from the stack.
* The processor executes the code step-by-step unless instructed to jump:

```
          sipush 0      -- constant 0
          istore 0      -- register 0 holds the value of sum
     l1:  iload 1       -- register 1 holds the value of n
          sipush 0
          if_icmple l2  -- jump to l2 if n <= 0
          iload 0
          iload 1
          iadd
          istore 0      -- completes sum = sum + n
          iload 1
          sipush 1
          isub
          istore 1      -- completes n = n - 1
          goto l1
     l2:
```

3.1 Lexical Analysis

The first step of compiling is performed by the *scanner*:

* It reads the source code character-by-character.
* It produces a sequence of *tokens* to be fed to the parser.
* White-space and comments are typically discarded.
* It keeps track of source-code line and column for layout checking and error messages.

There are several kinds of token:

* reserved words, such as
* symbols, such as
* identifiers, such as variable names
* number constants, such as
* string constants, such as
* Identifiers and constants have the concrete value attached.

Use of the scanner:

* The scanner will not convert the entire source code at once.
* Typically, it is called by the parser to deliver the next token in the source.
* The parser needs *lookahead* that shows the next token without consuming it.

Each kind of token is described by a regular expression:

∗ For reserved words and symbols, this might be just a simple string.

∗ For other kinds, a sequence of definitions might be used:

$$
\begin{aligned}
\mathit{digit} &= \texttt{0}-\texttt{9} \\
\mathit{lower} &= \texttt{a}-\texttt{z} \\
\mathit{upper} &= \texttt{A}-\texttt{Z} \\
\mathit{letter} &= \mathit{lower}|\mathit{upper} \\
\mathit{identifier} &= \\[1em]
\mathit{nonzerodigit} &= \texttt{1}-\texttt{9} \\
\mathit{decinteger} &= \\
\mathit{hexdigit} &= \mathit{digit}|\texttt{a}-\texttt{f}|\texttt{A}-\texttt{F} \\
\mathit{hexinteger} &= \\
\mathit{integer} &= \mathit{decinteger}|\mathit{hexinteger} \\[1em]
\mathit{stringchar} &= \text{any character except } \texttt{"} \text{ and } \texttt{\textbackslash} \text{ and newline} \\
\mathit{escape} &= \texttt{\textbackslash}(\text{any character}) \\
\mathit{stringitem} &= \mathit{stringchar}|\mathit{escape} \\
\mathit{string} &=
\end{aligned}
$$

∗ Such a sequence of definitions must not be cyclic.

∗ It can be converted to a flat regular expression by repeated substitution.

∗ *Character classes* abbreviate choices: $\texttt{a}-\texttt{z}$ amounts to $\texttt{a}|\texttt{b}|\ldots|\texttt{y}|\texttt{z}$.

∗ The resulting regular expression is converted to a DFA that is used for matching.

3.1.1 Scanners and Automata

The scanner recognises several kinds of token at the same time.

∗ For example, there might be four tokens for integers in different bases:

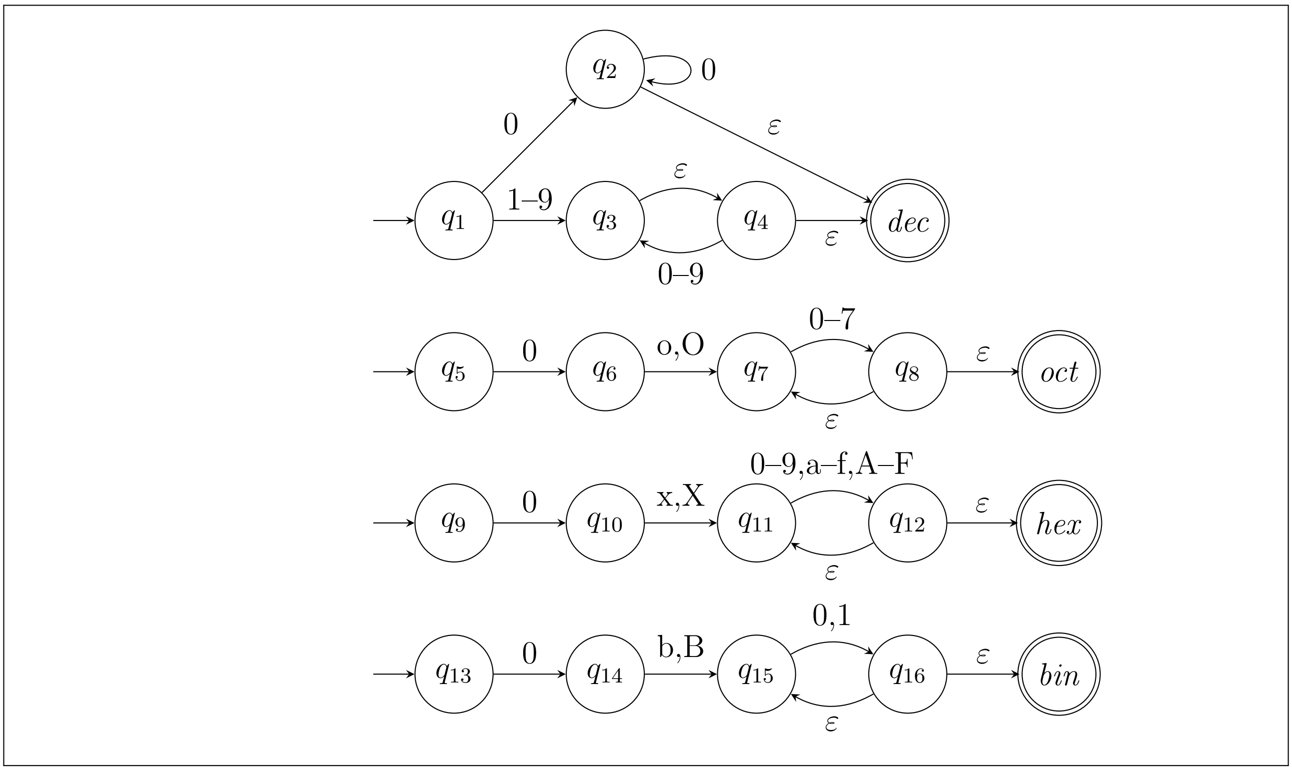
$$
\begin{aligned}
\mathit{decinteger} &= \texttt{0}^+|\mathit{nonzerodigit}\ \mathit{digit}^* \\
\mathit{octinteger} &= \texttt{0}(\texttt{o}|\texttt{O})\mathit{octdigit}^+ \\
\mathit{hexinteger} &= \texttt{0}(\texttt{x}|\texttt{X})\mathit{hexdigit}^+ \\
\mathit{bininteger} &= \texttt{0}(\texttt{b}|\texttt{B})\mathit{bindigit}^+
\end{aligned}
$$

$$
\begin{aligned}
\mathit{octdigit} &= \texttt{0}-\texttt{7} \\
\mathit{bindigit} &= \texttt{0}|\texttt{1}
\end{aligned}
$$

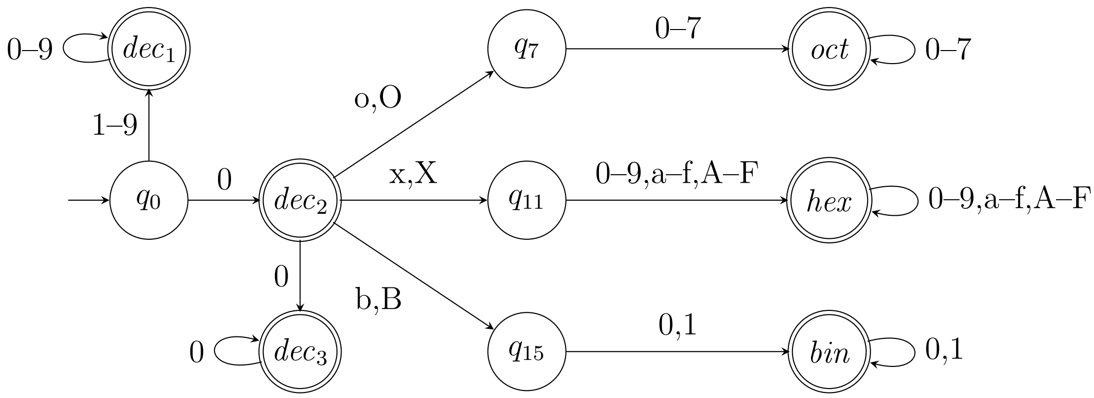
∗ An NFA is constructed for each kind of token that needs to be recognised.

∗ Each accept state is marked with the respective token.

* A new initial node is connected by ε-transitions to the initial node of each NFA.



* The resulting NFA is converted to a DFA and minimised.



* Marks on the accept states are preserved by the subset construction.
* They identify which token has been recognised.

Operation of the scanner:

* In each step, the scanner tries to match as much of the input as possible.
* The DFA consumes input as long as there is more input and a transition is possible.
* If the resulting state is accepting, the corresponding token is returned.
* The user specifies which token to return if more than one corresponds to a state.
* If the resulting state is not accepting, the DFA returns to the last accept state.
* To this end, transitions are undone and corresponding symbols are put back to the input.
* If no accept state is reached, the scanner reports a lexical error.
* The DFA is repeatedly restarted on the remaining input to find further tokens.

3.1.2 Extensions

Extended regular expressions:

* Eliminate at the regular expression stage:
* Reducing p^+ to pp^* can cause
* Special automata may be devised in such cases; for p^+ construct
* Eliminate at the automaton stage: for $\overline{p}$,
* Some constructs require more complex constructions.

Consider C-style comments `/* ... */`:

* Unlike typical strings, they are delimited by multiple characters.
* The characters in the comment cannot be described using character classes.
* A regular expression for such comments is $\texttt{/*}\overline{\Sigma^*\texttt{*/}\Sigma^*}\texttt{*/}$.

Constructing a minimal DFA for

The above procedure can be automated.

* Describe lexical structure by a sequence of definitions using extended regular expressions.
* A *scanner generator* takes such a description and produces a scanner.
* Examples of scanner generators are Lex, Flex, JFlex, PLY.
* Each kind of token may have an action which is performed when the token is found.

3.2 Syntax Analysis

The second step of compiling is performed by the *parser*:

∗ It calls the scanner to deliver tokens as required.

∗ It produces a *syntax tree* for further analysis and code generation.

∗ Lookahead tokens are used to efficiently recognise the structure.

∗ The output is typically an *abstract syntax tree* that omits irrelevant details.

∗ The parser may generate code on-the-fly, without constructing the full syntax tree.

The syntactic structure of a program is described in *Backus-Naur form* (BNF):

∗ A BNF description is a set of grammar rules.

∗ Each rule describes the structure of a program fragment.

∗ Unlike sequences of definitions using regular expressions, the rules may be *recursive*.

∗ Examples of fragments are statements, expressions, declarations, parameter lists.

∗ Rules for expressions, comparisons and statements might read as follows:

$$
\begin{array}{rcl}
\textit{Expression} & = & \\
\textit{Arithmetic} & = & \\
 & & \\
\textit{Comparison} & = & \textit{Expression Relation Expression} \\
\textit{Relation} & = & \texttt{=} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
 & & \\
\textit{Statements} & = & \\
\textit{Statement} & = & \textit{If} \mid \textit{While} \mid \textit{Assignment} \\
\textit{Assignment} & = & \\
\textit{While} & = & \texttt{while}\ \textit{Comparison}\ \texttt{do}\ \textit{Statements}\ \texttt{end} \\
\textit{If} & = &
\end{array}
$$

∗ An item on the left-hand side of a rule is a *non-terminal*.

∗ It may occur any number of times on the right-hand side of any rule.

∗ Several rules for the same non-terminal are abbreviated by | on the right-hand side:

$$
\begin{array}{rcl}
\textit{Expression} & = & \textit{Expression Arithmetic Expression} \\
\textit{Expression} & = & \texttt{(}\textit{Expression}\texttt{)} \\
\textit{Expression} & = & \texttt{number} \\
\textit{Expression} & = & \texttt{identifier}
\end{array}
$$

∗ This represents a choice, as for regular expressions.

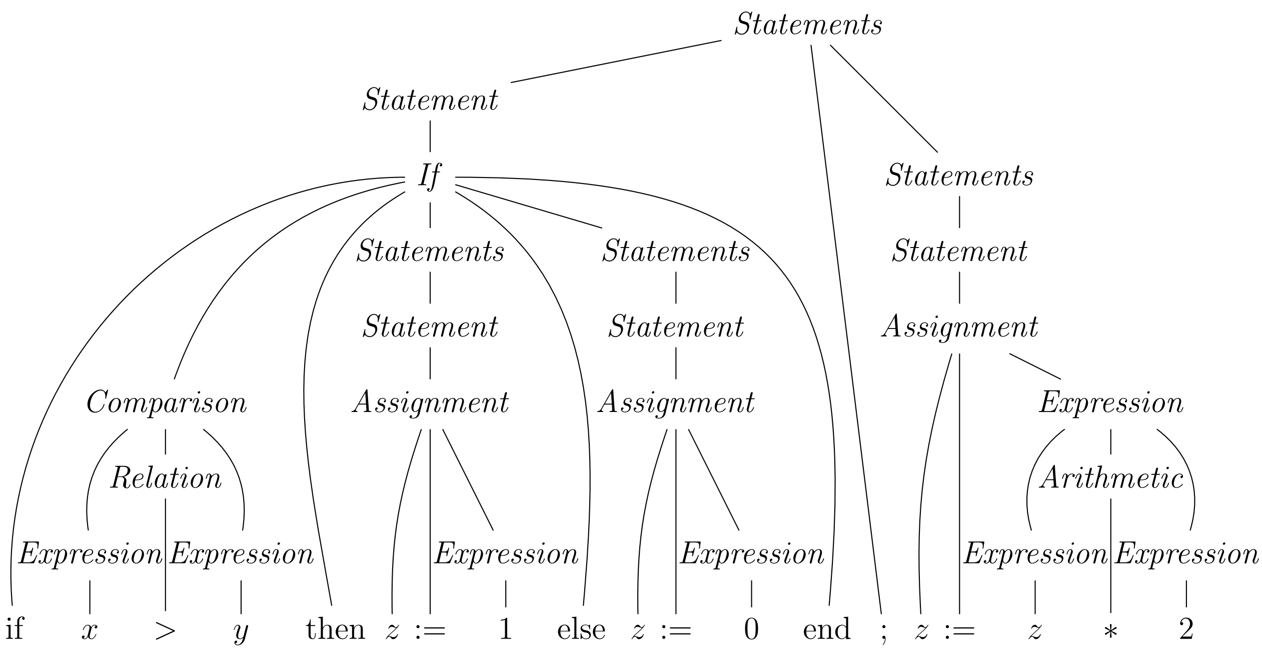
∗ Any other item on the right-hand side is a *terminal*.

∗ Terminals are matched by the tokens delivered from the scanner.

3.2.1 Syntax Trees

The parser constructs a syntax tree from a sequence of tokens.

* Every leaf of the syntax tree is labelled with a terminal.
* Every inner node of the syntax tree corresponds to the application of a rule.
* The inner node is labelled with the non-terminal on the left-hand side.
* Its children are labelled with the items on the right-hand side.
* The tree is *ordered*: the sequence of children matters.
* An example of a syntax tree is:



Consider the input `2 + 8 * 5`.

* The scanner delivers the token sequence `number(2)`, `+`, `number(8)`, `*`, `number(5)`.
* A syntax tree for this token sequence is:

∗ Another syntax tree for the same token sequence is:

∗ The BNF grammar is *ambiguous* for the given token sequence.

∗ Evaluation of the expression gives different values depending on the syntax tree.

∗ The programmer might use parentheses to specify precedence.

∗ The language might specify precedence rules that need to be reflected in the BNF:

$$
\begin{array}{rcl}
\textit{Expression} & = & \textit{Term} \mid \textit{Expression Additive Term} \\
\textit{Additive} & = & \texttt{+} \mid \texttt{-} \\
\textit{Term} & = & \textit{Factor} \mid \textit{Term Multiplicative Factor} \\
\textit{Multiplicative} & = & \texttt{*} \mid \texttt{/} \\
\textit{Factor} & = & \texttt{(}\textit{Expression}\texttt{)} \mid \texttt{number} \mid \texttt{identifier}
\end{array}
$$

∗ With this BNF the token sequence has just one syntax tree:

∗ Intermediate nodes, which help parsing, might be omitted in an abstract syntax tree.

Parsers construct the syntax tree

* top-down: LL, recursive-descent;
* bottom-up: LR, LALR, SLR, shift-reduce.
* LR is more expressive than LL by deferring decisions until more information is available.
* Recursive-descent parsers are easier to understand than shift-reduce parsers.

3.2.2 Extensions

Extended BNF has regular expressions on the right-hand side.

* The rules for expressions and terms can be written as:

$$\begin{array}{rcl} \mathit{Expression} & = & (\varepsilon \mid \mathit{Expression}\ \mathit{Additive})\ \mathit{Term} \\ \mathit{Term} & = & (\varepsilon \mid \mathit{Term}\ \mathit{Multiplicative})\ \mathit{Factor} \end{array}$$

* Using $[p]$ for optional p, this is:

$$\begin{array}{rcl} \mathit{Expression} & = & [\mathit{Expression}\ \mathit{Additive}]\ \mathit{Term} \\ \mathit{Term} & = & [\mathit{Term}\ \mathit{Multiplicative}]\ \mathit{Factor} \end{array}$$

* An alternative extended BNF is:

$$\begin{array}{rcl} \mathit{Expression} & = & \mathit{Term}\ (\mathit{Additive}\ \mathit{Term})^* \\ \mathit{Term} & = & \mathit{Factor}\ (\mathit{Multiplicative}\ \mathit{Factor})^* \end{array}$$

Syntax diagrams are a graphical representation of extended BNF:

* The rule $\mathit{Factor} = \texttt{(}\mathit{Expression}\texttt{)} \mid \texttt{number} \mid \texttt{identifier}$ is represented as:

* The rule $\mathit{Term} = [\mathit{Term}\ \mathit{Multiplicative}]\ \mathit{Factor}$ is represented as:

* The rule *Expression* = *Term* (*Additive Term*)* is represented as:

* Rectangles represent non-terminals and ovals represent terminals.
* Arrows represent sequence, choice and iteration.

3.2.3 Recursive-Descent Parsers

A parser can be implemented as a set of mutually recursive functions.

* Consider the extended BNF for expressions:

$$\begin{array}{rcl} \mathit{Expression} & = & \mathit{Term}\,((\texttt{+} \mid \texttt{-})\ \mathit{Term})^* \\ \mathit{Term} & = & \mathit{Factor}\,((\texttt{*} \mid \texttt{/})\ \mathit{Factor})^* \\ \mathit{Factor} & = & \texttt{(}\mathit{Expression}\texttt{)} \mid \texttt{number} \mid \texttt{identifier} \end{array}$$

* It results in the following parser:

```
def expression():
    term()
    while lookahead() in [ADD, SUB]:
        consume(ADD, SUB)
        term()

def term():
    factor()
    while lookahead() in [MUL, DIV]:
        consume(MUL, DIV)
        factor()

def factor():
    if lookahead() == LPAR:
        consume(LPAR)
        expression()
        consume(RPAR)
    elif lookahead() == NUM:
        consume(NUM)
    elif lookahead() == ID:
        consume(ID)
    else:
        raise Exception
```

The parser calls the scanner to obtain tokens.

* `ADD`, `SUB`, `MUL`, `DIV`, `LPAR`, `RPAR`, `NUM`, `ID` are tokens.
* Two functions form the interface to the scanner:

```
def lookahead():
    '''Returns the next token without consuming it.'''

def consume(*expected_tokens):
    '''Consumes the next token, if it is in expected_tokens.
       Raises an exception otherwise.'''
```

Extended BNF can be translated to a *recursive-descent* parser:

* Every non-terminal n is translated to a function.
* The function's body is obtained from the right-hand side r of the BNF rule $n = r$.

```
def n():
    parse(r)
```

Let $parse(r)$ be the parser code for regular expression r.

* $parse(r)$ is defined by induction over the structure of r.
* $parse(q|r)$ is the conditional

* $parse(r^*)$ is the while-loop

* *parse*(qr) is the sequence

* If r is a non-terminal, then *parse*(r) is the function call

  r()

* If r is a terminal, then *parse*(r) is a call to the scanner to consume token r

  consume(r)

The scanner needs to look ahead to inform the decisions in recursive-descent parsing:

* The scanner might provide a lookahead of one token.
* It is checked whether a string described by the regular expression begins with that token.
* The *First* set of r comprises the symbols that can begin a string described by r.
* If $\varepsilon \in L(r)$, then any symbol possibly following r must be considered.
* The *Follow* set of an occurrence of r is the symbols that can follow a string described by r.
* There are algorithms to compute the First and Follow sets from an extended BNF.

3.2.4 Abstract Syntax Trees

An abstract syntax tree can be constructed while parsing takes place.

* The interface to the scanner is extended thus:

```
def consume(*expected_tokens):
    '''Returns the next token and consumes it, if it is in
       expected_tokens. Raises an exception otherwise.
       If the token is a number or an identifier, not just the
       token but a pair of the token and its value is returned.'''
```

* Expressions can be represented by the following classes:

```
class Expression_AST:
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right

class Number_AST:
    def __init__(self, number):
        self.number = number

class Identifier_AST:
    def __init__(self, identifier):
        self.identifier = identifier
```

* The parser constructs and returns objects of the above classes:

```
operator = { ADD:'+', SUB:'-', MUL:'*', DIV:'/' }

def expression():
    result = term()
    while lookahead() in [ADD, SUB]:
        op = consume(ADD, SUB)
        tree = term()
        result = Expression_AST(result, operator[op], tree)
    return result

def term():
    result = factor()
    while lookahead() in [MUL, DIV]:
        op = consume(MUL, DIV)
        tree = factor()
        result = Expression_AST(result, operator[op], tree)
    return result

def factor():
    if lookahead() == LPAR:
        consume(LPAR)
        result = expression()
        consume(RPAR)
        return result
    elif lookahead() == NUM:
        value = consume(NUM)[1]
        return Number_AST(value)
    elif lookahead() == ID:
        value = consume(ID)[1]
        return Identifier_AST(value)
    else:
        raise Exception
```

* There is just one kind of arithmetic expression node in the abstract syntax tree.
* Irrelevant details, such as parenthesis tokens, are omitted from the tree.

3.3 Semantic Analysis

The third step of compiling is performed by the *semantic analyser*.

* It traverses the abstract syntax tree constructed by the parser.
* Nodes are annotated with information about the program.
* This information can be used for optimisation, code generation and error handling.
* The *semantics* of a program is its meaning.
* Semantic analysis covers just a few aspects of the meaning.
* *Static semantics* are those aspects that can be checked at compile-time.

Examples of semantic analyses are:

* type checking: are functions applied to arguments of matching type?
* type inference: what is the type of an expression?
* declaration: is every variable declared exactly once?
* definite assignment: is a variable assigned before it is used?
* binding: to which declaration does the use of a variable belong?

3.3.1 Type Systems

Many functions are partial, that is, they cannot be applied to all arguments:

* Add two numbers, but not two lists.
* Access the first element of a list, but not of a set.
* Compare two integers for $\leq$, but not two complex numbers.
* Find the shortest path in a graph, but not in a string.
* Sort a list of integers, but not a list that mixes integers and strings.
* Convert an integer to a string, but not a function.

The *type* of a variable or expression is the set of possible values it can take.

* A function might declare the types of its parameters and the type of its result.
* Such a function can only be applied to expressions with matching type.
* Its result can only be used in a context with matching type.

Low-level processors do not know types.

* They treat all data as bits.
* Operations apply to bits, but their result is meaningless if types do not match.

Type checking can be performed at different times.

* *Dynamic typing*: check types at *run-time*; raise exception on a mismatch.
* *Static typing*: check types before the program is run.
* A type mismatch indicates an error which the programmer needs to correct.
* Static typing allows the early detection of such errors.
* Type information can also be used for optimisation.

A *type system* defines types and typing rules for a programming language.

* Assembler: no types.
* C: few types, uncontrolled type casts.
* C++, Java: more types, type polymorphism.
* Haskell: expressive type system, no type errors at run-time.
* Python: dynamic typing.

3.3.2 Attribute Grammars

Simple type system for numbers:

The type system is implemented by an *attribute grammar*.

* This is a CFG extended by attributes and rules.
* An attribute stores information associated with non-terminals and terminals.
* The rules describe how to calculate this information.

Attribute grammar for typing arithmetic expressions:

* attribute
* attribute

$$
\begin{aligned}
E &= T \\
E &= E\;A\;T \\
A &= \texttt{+} \\
A &= \texttt{-} \\
T &= F \\
T &= T\;M\;F \\
M &= \texttt{*} \\
M &= \texttt{/} \\
F &= \texttt{(}E\texttt{)} \\
F &= \texttt{num} \\
F &= \texttt{id}
\end{aligned}
$$

These are *synthesised* attributes.

* The value on the left-hand side depends on the values of the right-hand side.
* Information is propagated bottom-up in the syntax tree.
* The type of a number constant is determined by the scanner.
* The type of an identifier is determined by its declaration or by assignments.

Type checking uses the calculated attribute values.

* The types of identifier and expression have to match in an assignment.
* Arguments must be integers for integer division.
* If a mismatch is found, a type error is reported or automatic conversions are added.

There are also *inherited* attributes.

* The value for a non-terminal on the right-hand side depends on the other values.
* Information is propagated top-down in the syntax tree.
* In combination with synthesised attributes information is passed around in the syntax tree.
* Attributes are often calculated during parsing without a separate pass over the tree.

3.4 Machine-Independent Optimisation

The fourth step of compiling is performed by the *machine-independent optimiser*.

* Optimisation takes place on the syntax tree or another intermediate form.
* It does not need to know the target processor.

The following example shows constant propagation, constant folding and dead code elimination.

* Constants assigned to a variable can be propagated to uses of the variable and substituted.
* Constant expressions can be evaluated and the result substituted (folded).
* Code that does not affect the result can be removed.

```
x := 11;
y := 4;
z := 8 * x;
z := x * y;
z := z * 2 + x;
if z > 99 then
    z := 0
else
    z := 1
```

Optimisations often lead to further optimisations.

* Constant folding enables constant propagation.
* Constant propagation enables further constant folding.
* Constant folding enables dead-code elimination.

Constant folding is the simplification of constant expressions at *compile-time*.

* Evaluate arithmetic expressions that use only constants.
* Values of variables are typically unknown, but might be known from preceding assignments.
* Constants might also be inserted by a pre-processor.
* Let ? denote an unknown value.

* The evaluation must implement the semantics of the program that is compiled.
* This may differ from operations of the machine on which the compiler runs.

Attribute grammar for evaluation of constant expressions:

* attribute

$$
\begin{aligned}
E &= T \\
E &= E\ A\ T \\
T &= F \\
T &= T\ M\ F \\
F &= (E) \\
F &= \texttt{num} \\
F &= \texttt{id}
\end{aligned}
$$

Compilers perform many kinds of optimisation leading to much faster running times:

* move calculations out of loops;
* reorder calculations;
* inline code;
* unroll loops;
* eliminate common subexpressions;
* remove tail-recursion.

3.5 Code Generation

The fifth step of compiling is performed by the *code generator*.

∗ It traverses the abstract syntax tree or another intermediate form.

∗ It emits code as soon as a sufficient portion is processed.

∗ The output is a data structure representing code for a virtual machine.

∗ It can be further analysed and optimised.

∗ The program might also be translated to binary code for a real machine.

∗ Thus compilation is *translation* from one language to another.

Programs have different parts, which are translated in different ways.

∗ *Expressions* are program fragments that yield a value.

∗ *Statements* are fragments that modify the values of variables.

∗ *Declarations* provide information for type checking and other analyses.

3.5.1 A Virtual Machine

In the following, the target code runs on the Java virtual machine.

∗ The machine features a *stack* to perform calculations.

∗ Values of numbers and identifiers are pushed on the stack.

∗ Operations remove their operands from the stack and push the result back.

Consider the expression `2 + 8 * 5`.

∗ The code generated for this expression is:

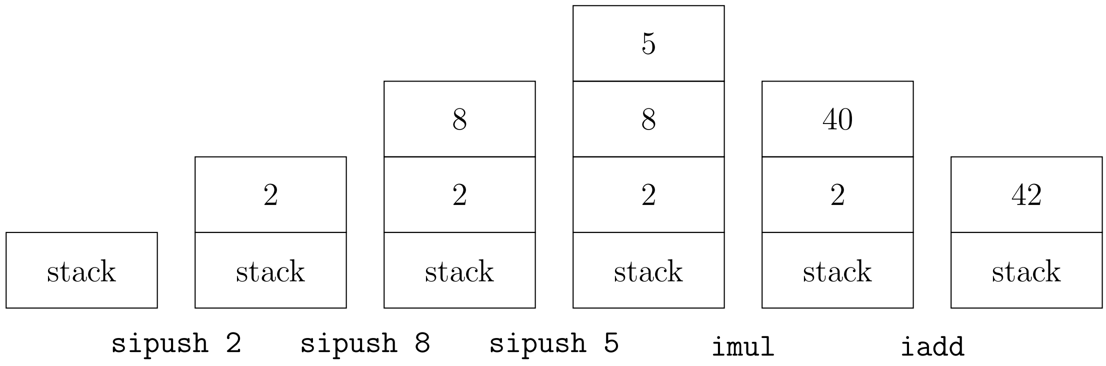
∗ `sipush`, `imul`, `iadd` are instructions of the Java virtual machine.

∗ The instruction `sipush` n pushes the integer constant n on the stack.

∗ The instruction `imul` pops two integers from the stack and pushes their product on the stack.

∗ The instruction `iadd` similarly adds the two top integers.

* The steps of running this code are:



* Running the code generated for an expression leaves its value on top of the stack.

3.5.2 Code Generation for Expressions

Code can thus be generated recursively.

* Let *code*(e) be the code generated for expression e.
* If n is a number, *code*(n) is just

* *code*($e_1 + e_2$) is the sequence

* Similar code is generated for $e_1 - e_2$ and $e_1 * e_2$ and e_1/e_2, using `isub`, `imul` and `idiv`.
* The code generator is implemented by methods of the abstract syntax tree classes.

```
class Number_AST:
    def code(self):
        return 'sipush ' + self.number + '\n'

class Expression_AST:
    def code(self):
        op = { '+':'iadd', '-':'isub', '*':'imul', '/':'idiv' }
        return self.left.code() + self.right.code() + \
               op[self.op] + '\n'
```

* These methods return a string representation of the generated code.
* The resulting code is fed to a Java assembler to produce a Java class file.
* The Java class file can be executed using the Java virtual machine.

3.5.3 Identifiers in the Virtual Machine

Another kind of expression is an identifier.

∗ Identifiers refer to variables.

∗ Variables are like numbers: their value should be pushed on the stack.

∗ Unlike constant numbers, the value of a variable can be changed by assignments.

∗ Therefore the value of a variable is stored in the memory of the machine.

∗ To change it, the *location* of that piece of storage is remembered.

Variables have two kinds of associated information:

A compiler needs to know the L-value.

∗ The R-value can change, but the L-value is fixed at compile-time.

∗ Obtain the R-value from the L-value by looking up the memory contents at that location.

A *frame* contains arguments, local data and the calculation stack of a method call.

∗ Locations of *local variables* or statically allocated data are offsets relative to the frame.

∗ Locations of *global variables* or dynamically allocated data are absolute memory addresses.

∗ The following assumes just a single frame.

Consider the assignment `x := y + 2`.

∗ Assume relative location 0 for `y` and 1 for `x`.

∗ The code generated for this assignment is:

∗ The instruction `iload` n pushes the value stored at relative location n on the stack.

∗ The instruction `istore` n pops a value from the stack and stores it at relative location n.

3.5.4 Code Generation for Identifiers

During compilation, the *symbol table* keeps track of the location of each variable.

∗ Upon encountering a new identifier, it is entered into the symbol table with a new location.

∗ Relative locations are just consecutive numbers $0, 1, 2, \ldots$

* If the identifier is encountered subsequently, its location is looked up in the symbol table.

```
class Symbol_Table:
    def __init__(self):
        self.symbol_table = {}
    def location(self, identifier):
        '''Returns the location of an identifier.
           A new identifier is entered with a new location.'''
        if identifier in self.symbol_table:
            return self.symbol_table[identifier]
        index = len(self.symbol_table)
        self.symbol_table[identifier] = index
        return index
```

* *code*(v) for a variable v with location $l(v)$ is just

<table><tr><td> </td></tr></table>

* The following class implements code generation for identifiers:

```
class Identifier_AST:
    def code(self):
        loc = symbol_table.location(self.identifier)
        return 'iload ' + str(loc) + '\n'
```

<table><tr><td>Using relative locations 0 and 1 for <code>y</code> and <code>x</code>, the code generated for <code>y + 8 * x</code> is:</td></tr></table>

3.5.5 Code Generation for Assignments

An assignment statement changes the value of a variable.

* The left-hand side of an assignment is an identifier.
* The right-hand side of an assignment is an expression.
* The generated code evaluates the expression and stores its value at the variable's location.
* *code*$(v := e)$ for expression e and variable v with location $l(v)$ is:

<table><tr><td> </td></tr></table>

∗ The code generated for the assignment `x := y + 8 * x` is:

∗ The following class implements assignments and their code generation:

```
class Assign_AST:
    def __init__(self, identifier, expression):
        self.identifier = identifier
        self.expression = expression
    def code(self):
        loc = symbol_table.location(self.identifier.identifier)
        return self.expression.code() + 'istore ' + str(loc) + '\n'
```

∗ The function for parsing assignments is:

```
def assignment():
    value = consume(ID)[1]
    ident = Identifier_AST(value)
    consume(BEC)
    expr = expression()
    return Assign_AST(ident, expr)
```

∗ The token `BEC` represents `:=`.

3.5.6 Control flow in the Virtual Machine

Conditionals and loops involve a change of *control flow*.

∗ Virtual machine instructions are executed sequentially.

∗ This matches a sequence of assignments.

∗ For other statements, the executed code depends on conditions known only at run-time.

∗ This is implemented by *labels* and *jumps*.

∗ Labels mark positions in the virtual machine code.

```
l1:
```

∗ An *unconditional jump* transfers control to a given label.

```
goto l1
```

∗ A *conditional jump* does the same, but only if a given condition holds.

```
if_icmpeq l1
```

* `if_icmpeq` l pops the top two values from the stack, and jumps to l if they are equal.
* If they are not equal, execution continues with the next instruction.

Consider the following program:

```
n := 2;
while n > 0 do
  n := n - 1
end
```

The code generated for this program is:

3.5.7 Code Generation for While-Loops

A condition compares two expressions.

* The values of both expressions are calculated first.
* The comparison is performed similarly to an arithmetic operation.
* Unlike arithmetic, which leaves the result on the stack, a conditional jump is emitted.
* $\mathit{false\_code}(e_1 < e_2, l)$ is the sequence

* The additional label l is the destination of the jump in case the comparison is *false*.

∗ `if_icmpge` l jumps to l if $e_1 \geq e_2$, otherwise continues with the next instruction.

∗ `if_icmpne`, `if_icmple`, `if_icmpgt`, `if_icmpeq`, `if_icmplt` are similar comparisons.

∗ The following class implements comparisons and their code generation:

```
class Comparison_AST:
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right
    def false_code(self, label):
        op = { '<':'if_icmpge', '<=':'if_icmpgt',
               '=':'if_icmpne', '!=':'if_icmpeq',
               '>':'if_icmple', '>=':'if_icmplt' }
        return self.left.code() + self.right.code() + \
               op[self.op] + ' ' + label + '\n'
```

A while-loop has a condition and a body, which is a sequence of statements.

∗ The condition is evaluated first.

∗ If it is true, the body is executed; this needs a conditional jump.

∗ After execution of the body, the above is repeated; this needs an unconditional jump.

∗ *code*(*while c do s end*) is the sequence

∗ This uses two new labels l_1 and l_2.

∗ The following class generates code for while-loops:

```
class While_AST:
    def __init__(self, condition, body):
        self.condition = condition
        self.body = body
    def code(self):
        l1 = new_label()
        l2 = new_label()
        return l1 + ':\n' + self.condition.false_code(l2) + \
               self.body.code() + 'goto ' + l1 + '\n' + l2 + ':\n'
```

∗ `new_label()` returns the string representation of a new label.

3.6 Machine-Dependent Optimisation

The sixth step of compiling is performed by the *machine-dependent optimiser*.

* It modifies the generated code.
* The optimisations are specific for each target processor.
* *Peephole optimisation* looks at a small part of generated instructions.
* They might be replaced with a shorter or faster sequence of instructions.

Consider the following fragment of a Java program:

```
int x, y, z;
z = 7;
x = y = z = z + 1;
```

The unoptimised code is:

```
sipush 7
istore 2
iload 2
sipush 1
iadd
dup       -- duplicate top element of stack
istore 2
dup
istore 1
dup
istore 0
pop       -- remove top element of stack
```

Realistic machines are more detailed than discussed above.

* They have stores for program, stack, heap.
* Their stacks are divided into frames, each of which belongs to a method call.
* They have registers such as program counter, stack pointer, frame pointer.
* The underlying interpreter loop fetches, decodes and executes instructions.
* The benefit of a virtual machine is that it abstracts from these specific details.
* The virtual machine can be implemented on a variety of platforms.
* This is an instance of adding a *level of indirection*.