CHAPTER 2

IMPLEMENTATION

2.1 Compilers and Interpreters

An $S \to T$ translator accepts code expressed in one language S (the source language), and translates it to equivalent code expressed in another language T (the target language). Examples of translators include:

- compilers, which translate high-level PL code to low level code, e.g. Java $\to$ JVM, C $\to$ x86as (x86 assembly code) and C $\to$ x86 (machine code).
- assemblers, which translate assembly language to the corresponding machine code, e.g. x86as $\to$ x86.
- high-level translators, or transpilers, which translate code in one high-level PL to code in another high level PL, e.g. Java $\to$ C.
- decompilers, or disassemblers, which translate low-level code to high(er)-level PL code, e.g. JVM $\to$ Java and x86 $\to$ x86as.

An S interpreter accepts code expressed in language S, and immediately executes that code. That is, there is no intermediate object code generated. It works by fetching, analysing and executing one instruction at a time. If an instruction is fetched repeatedly, it will be analysed repeatedly. This is time consuming unless instructions have very simple formats.

Interpreting a program is slower than executing native machine code. Moreover, interpreting a high-level language is much slower than interpreting an intermediate-level language, such as JVM code. On the other hand, interpreting a program cuts out compile-time.

Interpretation is sensible when:

- a user is entering instructions interactively and wishes to see the results of each instruction before entering the next one;
- the program is to be used once and then discarded (meaning that the execution speed doesn't matter);
- each instruction will be executed only once or a few times;
- the instructions have very simple formats;
- the program code is required to be highly portable.

Some interpreters are:

- Unix command language interpreter (shell). Here, the user enters one command at a time. The shell reads the command, parses it to determine the command name and arguments, and executes it.

- JVM (Java virtual machine) interpreter. A JVM program consists of bytecodes. The interpreter fetches, decodes and executes one bytecode at a time.

There is a big difference between compilers and interpreters. A compiler translates source code to object code. It does not execute the source or object code. On the other hand, an interpreter executes source code one instruction at a time. It does not translate the source code.

Tombstone diagrams

We can use tombstone diagrams to represent programs, interpreters, compilers and hardware. We have the following symbols available:

- The following figure is used to denote a program.

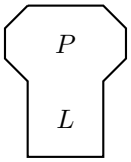


Figure 2.1: A tombstone diagram for a program P in language L.

- The following figure is used to denote a translator.

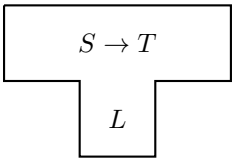


Figure 2.2: An $S \rightarrow T$ translator expressed in the language L.

- The following figure is used to denote an interpreter

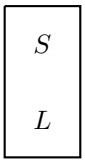


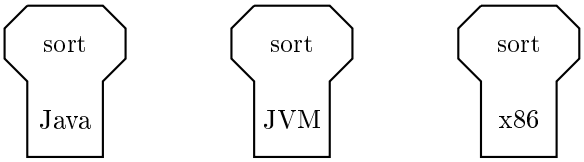
Figure 2.3: An S interpreter in the language L.

- The following figure is used to denote a hardware.

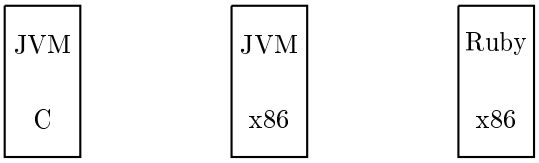
Figure 2.4: A machine M which can only execute M's machine code.

We will now look at some concrete examples of tombstone diagrams.

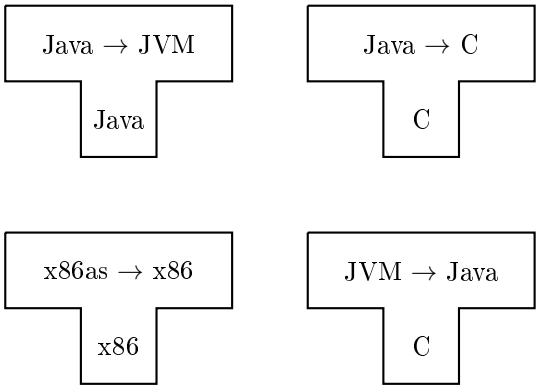
- We can denote the the program `sort` written in 3 different languages- Java, JVM and x86.



- We can denote the interpreters: a JVM interpreter in C, a JVM interpreter in x86 and a Ruby interpreter in x86.



- We can denote the translators: a Java $\rightarrow$ JVM compiler in Java, a Java $\rightarrow$ C transpiler, an x86as $\rightarrow$ x86 assembler in x86, and a JVM $\rightarrow$ Java decompiler in C.



We can use tombstone diagrams to run programs. Given a program P expressed in M machine code, we can run P on machine M.

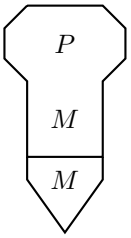
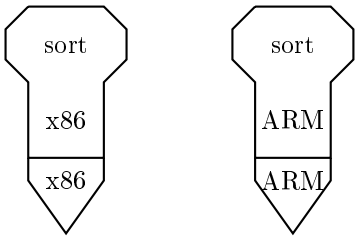
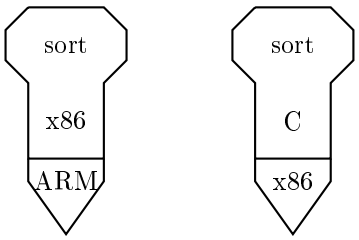


Figure 2.5: Running a program P written in M.

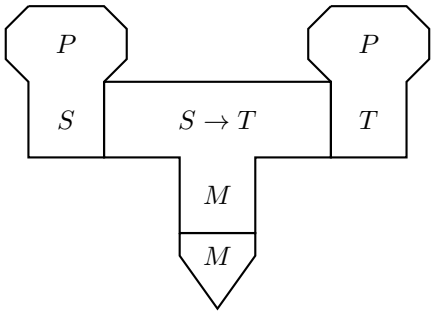
Note that the two M's must match for the program to run. So, we can run the sort program in x86 and ARM given the right hardware.



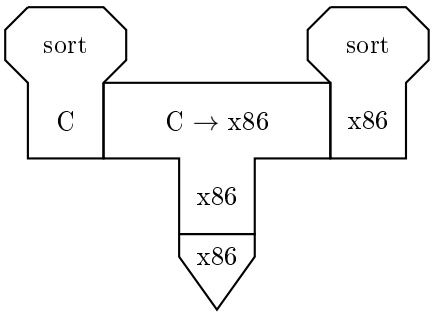
However, the following are not possible since the language of the program doesn't match the hardware.



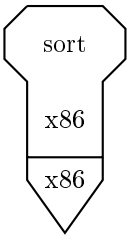
We can also show the translation of a program via tombstone diagrams. Given an $S \to T$ translator, expressed in M machine code, and a program P expressed in language S, we can translate P to language T.



Note that the PLs must match the source and the target languages of the translator. Also, the language of the translator must match the language of the hardware. An example of this is given below- compiling the sort program from C to x86.

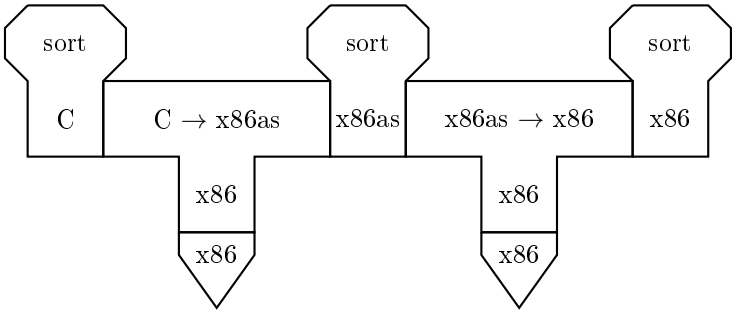


This represents the compilation of the program. We can later run the object program on an x86, which is depicted by the tombstone diagram below.

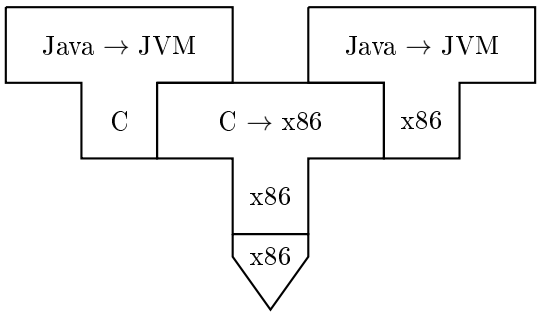


This represents the runtime of the program.

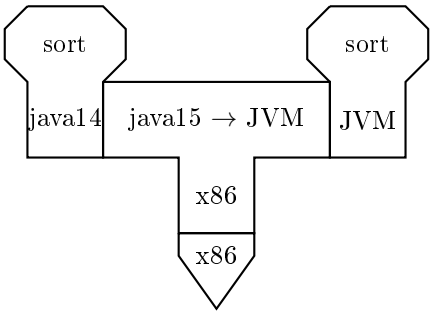
We could break the compilation into 2 steps. For example, we can have a C → x86as compiler, and an x86 assembler. Then, we can use them to compile a program in C into x86 machine code, in 2 stages.



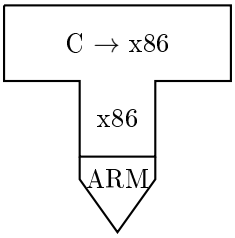
We can compile a compiler. Given a C → x86 compiler, we can use it to compile any C program into x86 machine code. In particular, we can compile a compiler expressed in C, e.g. compiling a Java → JVM compiler written in C to a Java → JVM compiler in x86.



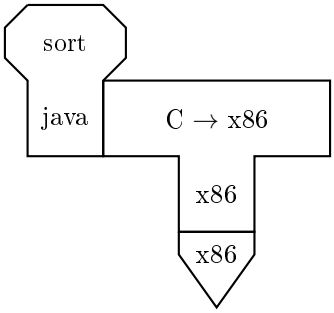
For this to be possible, we require the relevant languages to match. So, the following is possible.



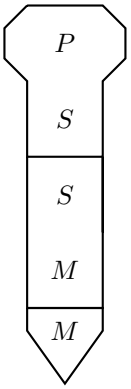
This is because Java14 is a subset of Java15. However, we cannot run a C → x86 compiler in an ARM hardware.



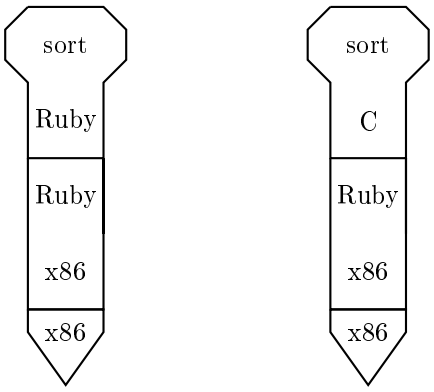
Similarly, we cannot compile a Java program using a C → x86 compiler.



Now, we will consider representing interpretation via tombstone diagrams. Given an S interpreter expressed in M machine code and a program P expressed in language S, we can interpret P. This is denoted as follows.



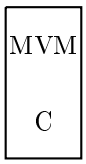
The language in which the program is written (i.e. S) must match the language that is being interpreted. Moreover, the language in which the interpreter is written (i.e. M) must match the language of the hardware. So, it is possible to interpret a sort program in Ruby with a Ruby interpreter, but not a sort program in C.



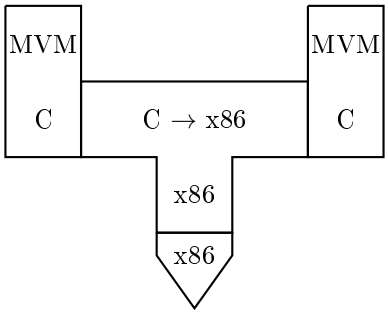
Virtual machines

A real machine is one whose machine code is executed by hardware. A virtual machine (or an abstract machine) is one whose machine code is executed by an interpreter. We can use tombstone diagrams to denote virtual machines. For example, assume that we designed the architecture and instruction set of a new machine called MVM. Building a hardware prototype would be expensive, and even more to modify.

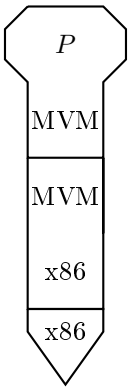
We can first write an interpreter for MVM machine code (an emulator) expressed in C, for example. This is given below.



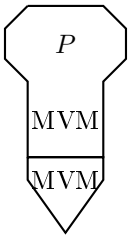
We can then compile it on a real machine, e.g. x86.



Now, we can use the emulator to execute programs P expressed in MVM machine code.



The bottom two layers are called the MVM virtual machine. This has the same effect as the MVM real machine, given by the following.



However, using the MVM real machine would be much faster.

Interpretive Compilers

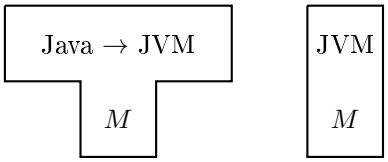
A compiler takes quite a long time to translate the source program to native machine code, but the subsequent execution is fast. On the other hand, an interpreter starts executing the source program immediately, but the execution is slow. We can combine the two and get an interpretive compiler. It translates the source program into virtual machine (VM) code which is subsequently interpreted.

An interpretive compiler combines fast translation with moderately fast execution, provided that:

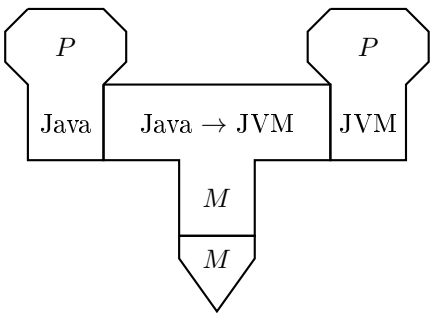
- the VM code is intermediate-level (lower-level than the source code but higher-level than native machine code);
- translation from the source language to VM code is easy and fast;
- the VM instructions have simpler formats (so can be analysed quickly by an interpreter).

For example, JDK (Java development kit) provides an interpretive compiler for Java. This is based on the JVM (Java virtual machine) that was designed to run Java programs. JVM provides powerful instructions that implement object creation, method calls, array indexing, etc. However, the instructions (called 'bytecodes') are similar in format to native machine code, i.e. opcode + operand.

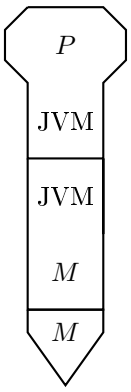
JDK comprises a Java $\rightarrow$ JVM compiler and a JVM interpreter. Once JDK has been installed on a real machine, we have the following components:



Using these, we can translate Java source program P into JVM code.



Later, the object program is interpreted.

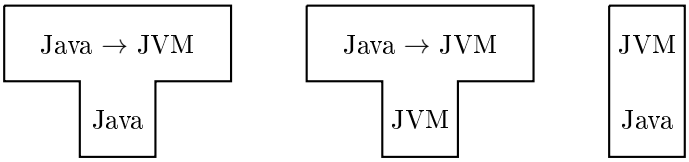


The bottom two tiles are called the Java Virtual Machine.

A just-in-time (JIT) compiler translates virtual machine code to native machine code just prior to execution. More usually, a Java JIT compiler translates JVM code selectively. The interpreter and the JIT compiler work together. The interpreter is instrumented to count method calls. When the interpreter discovers that a method is 'hot' (i.e. it is called frequently), it tells the JIT compiler to translate the particular method into native code.

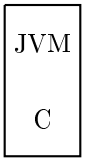
A program is portable if it can be made to run on different machines with minimal change. So, a program P written in Java is portable, but the same program in x86 is not. A compiler that generates native machine code is unportable in that if it must be changed to target a different machine, its code generator must be replaced. However, a compiler that generates suitable virtual machine code can be portable.

So, a portable compiler kit for Java is composed of the following tiles:

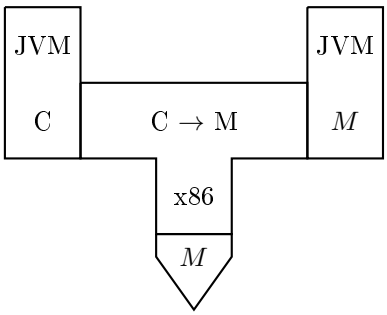


We can install this kit on machine M. But, we cannot run the JVM interpreter until we have a running Java compiler. Similarly, we cannot run the Java compiler until we have a running JVM interpreter.

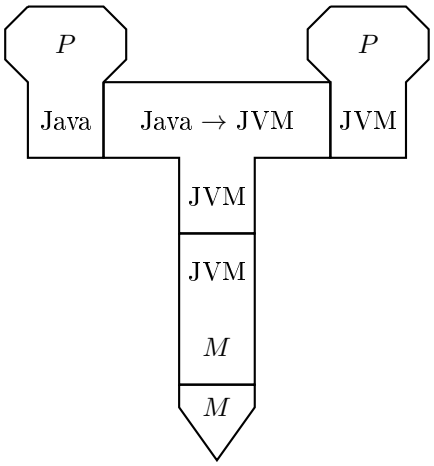
To progress, we first rewrite the JVM interpreter, e.g. in C.



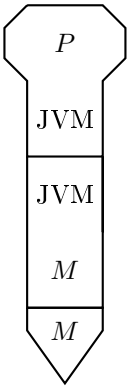
Then, we compile the JVM interpreter on M.



Now, we have an interpretive compiler. But, the compiler itself must be interpreted in this case.

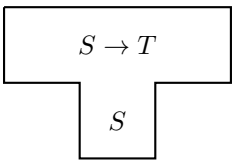


Using the result, we can interpret the program P using the JVM.



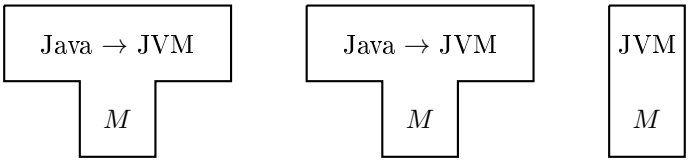
Here, the compilation stage is very slow, but it can be improved by bootstraping.

Assume that we have an $S \rightarrow T$ translator expressed in language S.

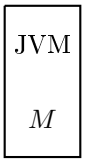


We can use this translator to translate itself. This process is called bootstraping. It is a useful tool for improving an existing compiler. This is because it makes the compilation process faster, which makes it generate faster object code. We can bootstrap a portable compiler to make a true compiler, by translating virtual machine code into native machine code.

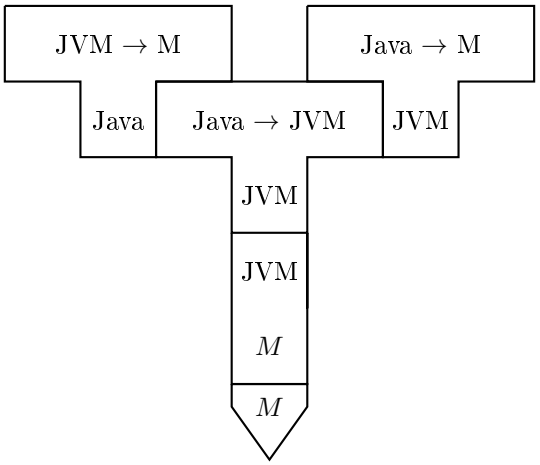
For example, consider the Java portable compiler kit.



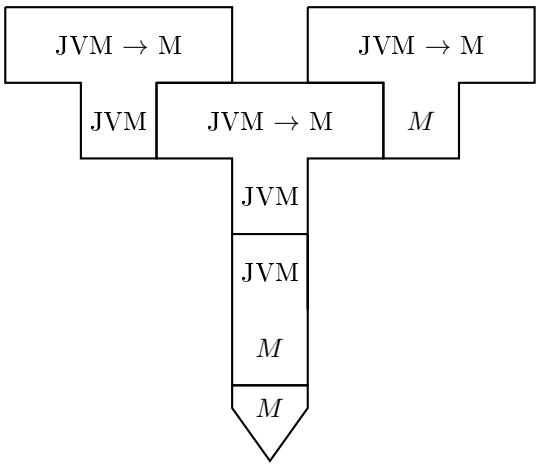
We also constructed a JVM interpreter in machine language M.



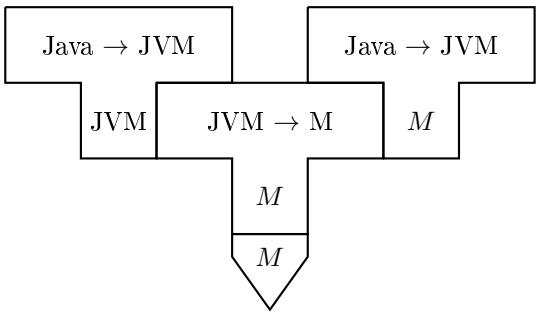
We can write a JVM → M translator in Java itself. We then compile it into JVM using the existing, slow compiler.



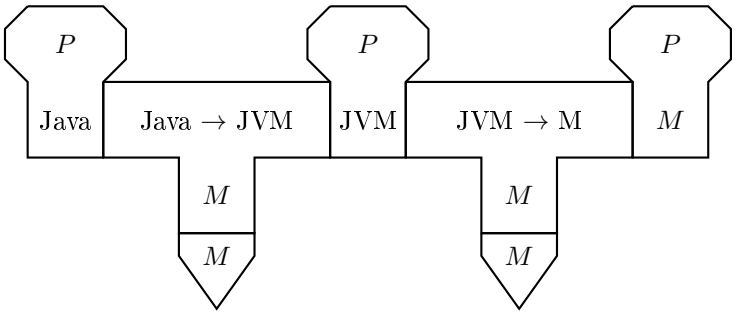
Next, we can use this JVM → M translator to translate itself.



This is the actual bootstrap. It generates the JVM → M translator, expressed in M machine code. Finally, we can translate the Java → JVM compiler into M machine code.



Now, we have a 2-stage Java → M compiler.



This Java compiler is improved in two respects. It compilers faster since it is expressed in native machine code. It also generates faster object code.

2.2 Interpretation

An S interpreter accepts code expressed in language S and immediately executes that code. Assuming that the code to be interpreted is just a sequence of simple instructions (with conditional and unconditional jumps), the interpreter:

- first initialises the state,
- then repeatedly fetches, analyses and executes the next instruction, and
- updates the state as an instruction gets updated, as required.

Virtual machine code typically consists of:

- load and store instructions,
- arithmetic and logical instructions,
- conditional and unconditional jumps,
- call and return instructions, etc.

The virtual machine state typically consists of storage (code and data) along with registers (status, program counter, stack pointer, etc.).

SVM

Simple Virtual Machine (SVM) is suitable for executing programs in simple imperative PLs. Consider we have the following source code in a C-style language:

```
p = 1;
while (p < n) {
    p = 10 * p;
}
```

The corresponding SVM code for it is the following:

```
// load constant 1
LOADC 1
// store the constant 1 at address 2 (variable p)
STOREG 2
// load from address 2 (variable p)
LOADG 2
// load from address 1 (variable n)
LOADG 1
// compare p < n
COMPLT
// if false, jump to 29 = line 24 (and halt)
JUMPF 29
// load constant 10
LOADC 10
// load from address 2 (variable p)
LOADG 2
// multiply p and 10
MUL
// store p*10 to address 2
STOREG 2
// jump to 6 = line 6
```

```
22 JUMP 6
23 // halt
24 HALT
```

The SVM storage is composed of the code store and the data store. The code store is a fixed array of bytes (32 768 bytes) providing space for instructions. The data store is a fixed array of words (32 768 bytes) providing a stack to contain global and local data. The main registers for SVM are:

- `pc` (program counter) points to the next instruction to be executed
- `sp` (stack pointer) points to the top of the stack
- `fp` (frame pointer) points to the base of the topmost frame
- `status` indicates whether the program is running, failed or halted.

The following image illustrates the code store of the SVM code above.

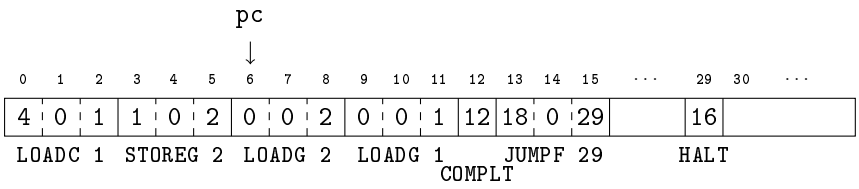


Figure 2.6: An illustration of the code source.

Each instruction occupies 1, 2 or 3 bytes, e.g. `STOREG 2` occupies 3 bytes, while `COMPLT` occupies only 1. The register `pc` is pointing to 6, meaning that is the next instruction to be executed. The final part (after 30) is the unused part of the code source.

Next, the following figure illustrates the data store.

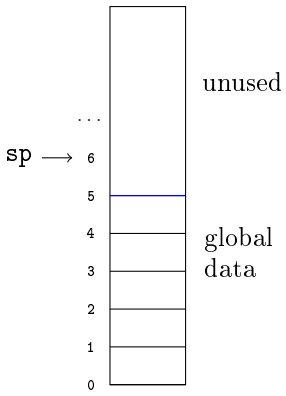


Figure 2.7: An illustration of the data store at the start.

The register `sp` is pointing to 6, meaning that all the positions above 6 (including it) is unused and empty. At the bottom, there is only global data on the

stack. As the program executes, we load further data, so we might end up at the following state.

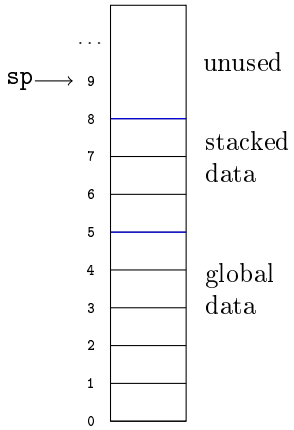


Figure 2.8: An illustration of the data store during execution.

The following is a simplified version of the SVM instruction set.

Opcode	Mnemonic	Behavior
6	`ADD`	pop w_2; pop w_1; push $(w_1 + w_2)$
7	`SUB`	pop w_2; pop w_1; push $(w_1 - w_2)$
8	`MUL`	pop w_2; pop w_1; push $(w_1 * w_2)$
9	`DOV`	pop w_2; pop w_1; push (w_1/w_2)
10	`CMPEQ`	pop w_2; pop w_1; push (if $w_1 = w_2$ then 1 else 0)
11	`CMPLT`	pop w_2; pop w_1; push (if $w_1 < w_2$ then 1 else 0)
14	`INV`	pop w; push (if $w = 0$ then 1 else 0)
0	`LOADG d`	$w \leftarrow$ word at address d; push w
1	`STOREG d`	pop w; word at address $d \leftarrow w$
4	`LOADC v`	push v
16	`HALT`	status $\leftarrow$ halted
17	`JUMP c`	pc $\leftarrow$ c
18	`JUMPF c`	pop w; if $w = 0$ then pc $\leftarrow$ c
19	`JUMPT c`	pop w; if $w \neq 0$ then pc $\leftarrow$ c

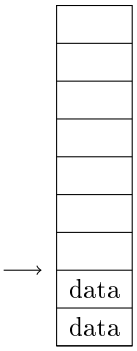
Table 2.1: The SVM Instruction Set

We will illustrate how the data store changes as we execute the program. So, consider we are evaluating the following expression: `(7 + 3)*(5 - 2)`. In SVM, it is the following:

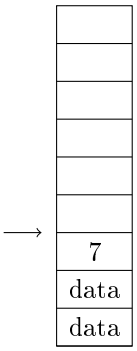
```
// load constant 7
LOADC 7
// load constant 3
LOADC 3
// add 7 and 3 = 10
ADD
// load constant 3
LOADC 5
```

```
// load constant 2
LOADC 2
// subtract 2 from 5 = 3
SUB
// multiply 10 and 3 = 30
MUL
```

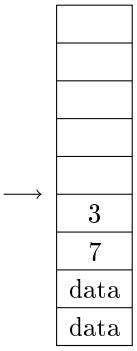
We will illustrate the execution with the data store. Assume that the initial state of the stack is the following.



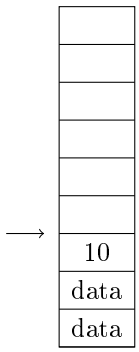
The arrow denotes the stack pointer `sp`. We first execute `LOADC 7`, which will add 7 to the stack.



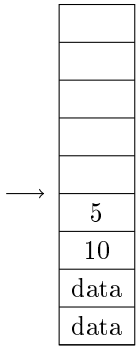
Next, we execute `LOADC 3`, which will add 3 to the stack.



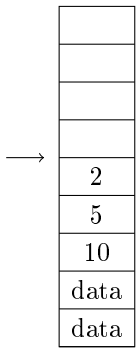
The next command is `ADD`. By its specification, we pop $w_2 = 3$ and $w_1 = 7$, and push $w_1 + w_2 = 10$.



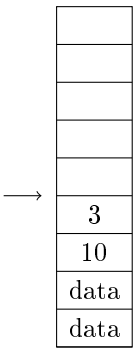
The next command is `LOADC 5`. So, we push 5 to the stack.



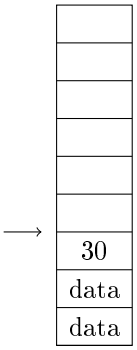
Now, we perform `LOADC 2`, when we push 2 to the stack.



The next command is `SUB`. So, we pop $w_2 = 2$ and $w_1 = 5$, and push $w_1 - w_2 = 3$.



The final command is `MUL`. So, we pop $w_2 = 3$ and $w_1 = 10$, and push $w_1 * w_2 = 30$.



Writing interpreters

Interpreters are commonly written in C or Java. In such an interpreter, the virtual machine state is represented by a group of variables. Each instruction is executed by inspecting and/or updating the virtual machine state.

For the SVM interpreter in a Java, the following is the representation of the instructions.

```
final byte
    LOADG = 0, STOREG = 1,
    LOADL = 2, STOREL = 3,
    LOADC = 4,
    ADD = 6, SUB = 7,
    MUL = 8, DIV = 9,
    CMPEQ = 10,
    CMPLT = 12, CMPGT = 13,
    INV = 14, INC = 14,
    HALT = 16, JUMP = 17,
    JUMPF = 18, JUMPT = 19,
```

Each instruction is a byte, and given a value, e.g. `ADD` is 6, and so on. The virtual machine state is given by the following.

```
byte[] code; // code store
int[] data; // data store
int pc, cl, sp, fp, status; // registers
final byte
    RUNNING = 0,
```

```
    FAILED = 1,
    HALTED = 2;
```

The interpreter initialises the state, then repeatedly fetches and executes the instructions. The interpret method is outlined below:

```
void interpret () {
    // Initialize the state:
    status = RUNNING;
    sp = 0; fp = 0; pc = 0;
    do {
        // Fetch the next instruction:
        byte opcode = code[pc++];
        // Execute this instruction:
        ...
    } while (status == RUNNING);
}
```

To execute an instruction, we first inspect its opcode, which is given below.

```
// Execute this instruction:
switch (opcode) {
    case LOADG: ...
    case STOREG: ...
    ...
    case ADD: ...
    case CMPLT: ...
    ...
    case HALT: ...
    case JUMP: ...
    case JUMPT: ...
    ...
}
```

Within each case, we have the instructions to execute it, e.g. `ADD` and `CMPLT` are given below.

```
case ADD: {
    int w2 = data[--sp];
    int w1 = data[--sp];
    data[sp++] = w1 + w2;
    break;
}
case CMPLT: {
    int w2 = data[--sp];
    int w1 = data[--sp];
    data[sp++] = w1 < w2 ? 1 : 0;
    break;
}
```

The load and store instructions are given below.

```
case LOADG: {
    // gives the address d for the data store (2-byte operand)
    int d = code[pc++] << 8 | code[pc++];
    data[sp++] = data[d];
    break;
}
case STOREG: {
    int d = code[pc++] << 8 | code[pc++];
    data[d] = data[--sp];
    break;
}
```

The following are halting and jumping instructions.

```
case HALT: {
    status = HALTED;
    break;
}
case JUMP: {
    // fetch 2-byte operand
    int c = ...;
    pc = c;
    break;
}
case JUMPT: {
    // fetch 2-byte operand
    int c = ...;
    int w = data[--sp];
    if (w != 0) pc = c;
    break;
}
```

2.3 Compilation

An $S \to T$ compiler translates a source program in S to object code in T given that it conforms the source language's syntax and scope/type rules. This suggests that the compilation stage should be decomposed into three phases: syntactic analysis, contextual analysis and code generation.

In syntactic analysis, we parse the source program to check whether it is well-formed, and to determine its phrase structure, in accordance with the source language's syntax. In contextual analysis, we analyse the parsed program to check whether it conforms the source language's scope and type rules. In code generation, we translate the parsed program to object code, in accordance with the source language's semantics.

The following figure illustrates data flow between the phases:

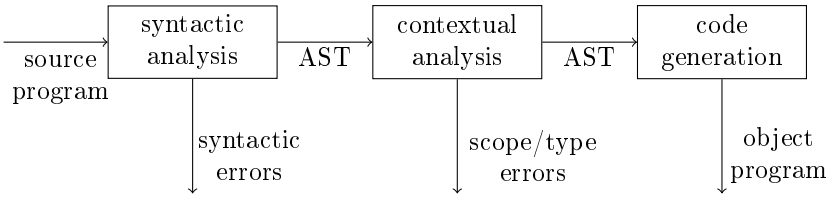


Figure 2.9: The data flow between the compilation phases.

Source program is given as input to the syntactic analysis phase. If there are no syntactic errors, we can transform it into an abstract syntax tree (AST). This is then analysed contextually. If there are no type or scope errors, then the AST will get annotated. This is then used to generate the object program. An AST is a convenient way to represent a source program after syntactic analysis.

Fun

Fun is a simple imperative language. A Fun program declares some global variables and some procedures/functions, always including a procedure named `main()`. A Fun procedure/function may have a single parameter. It may also declare local variables. A function returns a result, but a procedure does not. Fun has two data types- `bool` and `int`. Commands in Fun can be:

- assignment,
- procedure/function call,
- if command,
- while command, and
- sequential command.

The following is a simple Fun program.

```
1 func int fact (int n): # returns n!
2     int f = 1
3     while n > 1:
```

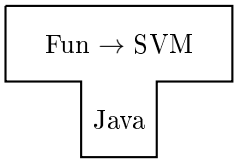
```
4           f = f*n
5           n = n-1 .
6       return f .
7
8   proc main ():
9       int num = read()
10      write(num)
11      write(fact(num)) .
```

Fun programs are free-format, i.e. spaces, tabs and EOLs are not significant.

The following is an extract of the Fun grammar, in EBNF form:

$$
\begin{aligned}
\textit{prog} &= \textit{var-decl}^{*}\ \textit{proc-decl}^{+}\ \textit{eof} \\
\textit{var-decl} &= \textit{type}\ \textit{id}\ \texttt{=}\ \textit{expr} \\
\textit{type} &= \texttt{bool} \\
&\mid \texttt{int} \\
\textit{com} &= \textit{id}\ \texttt{=}\ \textit{expr} \\
&\mid \texttt{if}\ \textit{expr}\texttt{:}\ \textit{seq-com}\texttt{;} \\
&\mid \ldots \\
\textit{seq-com} &= \textit{com}^{*} \\
\textit{expr} &= \textit{sec-expr}\ldots \\
\textit{sec-expr} &= \textit{prim-expr}\ ((\texttt{+}\mid\texttt{-}\mid\texttt{*}\mid\texttt{/})\ \textit{prim-expr})^{*} \\
\textit{prim-expr} &= \textit{num} \\
&\mid \textit{id} \\
&\mid \texttt{(}\ \textit{expr}\ \texttt{)} \\
&\mid \ldots
\end{aligned}
$$

The Fun compiler generates SVM code. It is expressed in Java. The tombstone is given below.



The compiler contains the following classes:

- syntactic analyser (`FunLexer`, `FunParser`),
- contextual analyser (`FunChecker`),
- code generator (`FunEncoder`).

The compiler calls each of these in turn:

- The syntactic analyser lexes and parses the source program, printing any error messages and generates an AST. Then, the AST is printed.
- The contextual analyser performs scope and type checking, printing any error messages.

- The code generator emits object code into the SVM code store. Then, the object code is printed.

Compilation is terminated after syntactic or contextual analysis if any errors are detected.

The driver `FunRun` compiles the source program into an SVM object program. If no errors are detected, it calls the SVM interpreter to run the object program.

AST

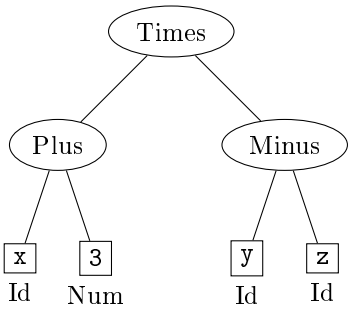
An abstract syntax tree (AST) is a convenient way to represent a source program's phrase structure. The structure of an AST is the following:

- Each leaf node represents an identifier or literal.

- Each internal node corresponds to a source language construct (e.g. a variable declaration or while-command). The internal node's subtrees represent the parts of that construct.

ASTs are much more compact than syntax trees.

For example, the AST for the expression `(x + 13) * (y - z)` in Fun is the following:

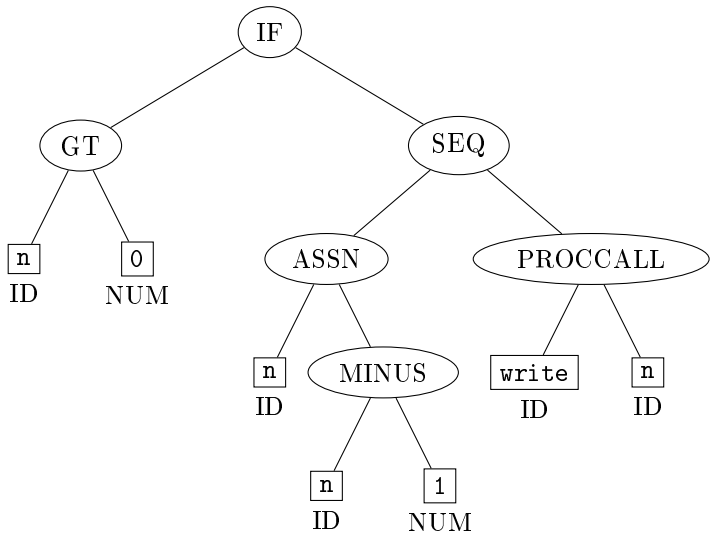


In the AST, we make no distinct between expr, sec-expr, etc. They are all treated as expressions. This is one of the ways an AST is compact.

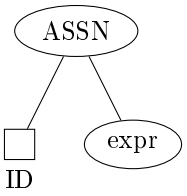
Now, consider the following program in Fun:

```
1 if n > 0:
2     n = n - 1
3     write(n); .
```

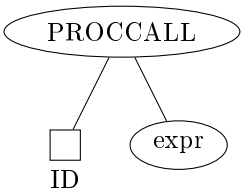
The corresponding AST is the following:



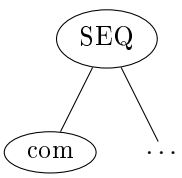
Now, we will look at all the Fun ASTs. For assignment, the AST is the following:



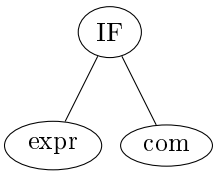
The assignment is given by an identifier, which is given the value of some expression. The procedure call AST is the following:



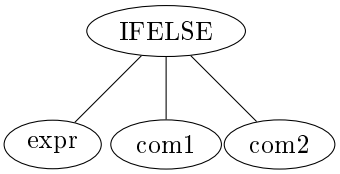
The expression is the same as assignment- we have an identifier for the procedure name, and its parameter is some expression. Next, the AST for a sequence of commands is the following:



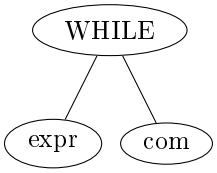
It is composed of a sequence of commands, and we must have at least one command. Now, the if AST is given below:



We have an expression (that evaluates to true or false), and if it evaluates to true, we run the command. The AST for ifelse is given below:



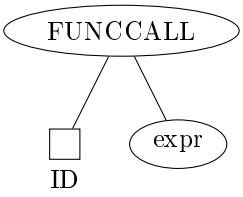
Here, we have an expression (that evaluates to true or false), and if it evaluates to true, we run the command com1, and otherwise, com2. The AST for a while loop is given below.



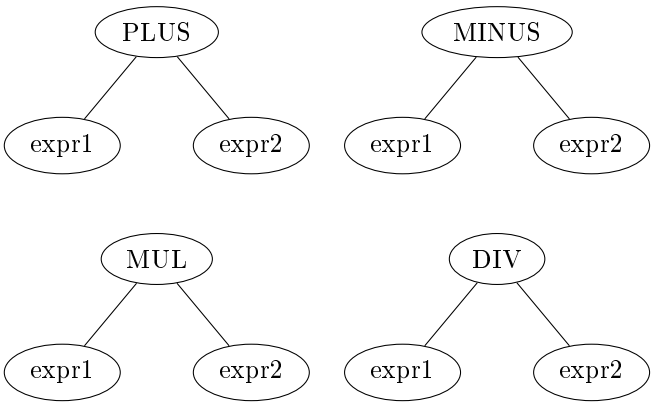
Here too, We have an expression (that evaluates to true or false), and if it evaluates to true, we run the command com1, and we rerun the expression and so on.

Now, we will look at ASTs for Fun expressions. There are 4 literal expressions, shown below.

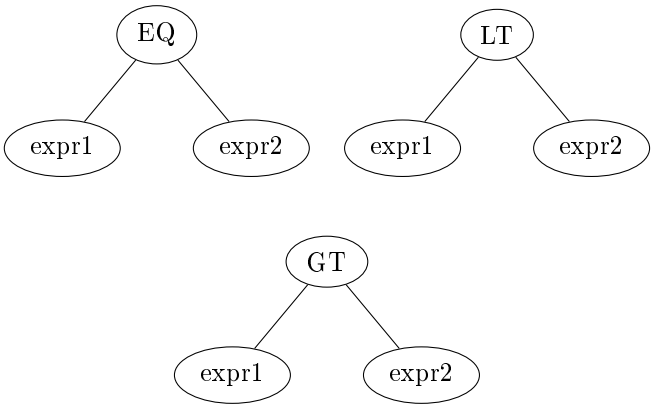
The literals are the boolean values `true` and `false`, numbers and identifiers. The AST for the function call expression is the following:



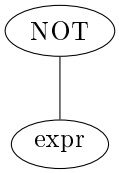
We have a name for the function, and the value of the parameter is some expression. Moreover, the arithmetic operations (plus, minus, times, div) are also expressions, with AST given below:



In each case, we have two expressions, and the operation is applied in the natural way. We also have comparison operations, whose AST is given below:

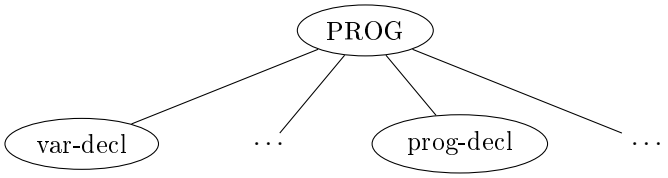


We compare two expressions (for equality, less than or greater than). Next, the AST for the not expression is given below.

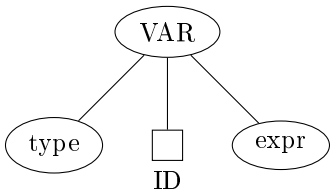


The expression evaluates to some boolean value, so the result will be the opposite boolean.

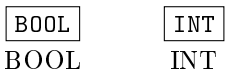
The AST for a Fun program is given below.



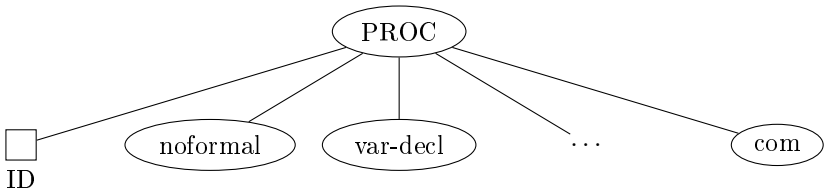
So, a program is a sequence of variable declarations and a sequence of program declarations. The AST for variable declarations is given below.



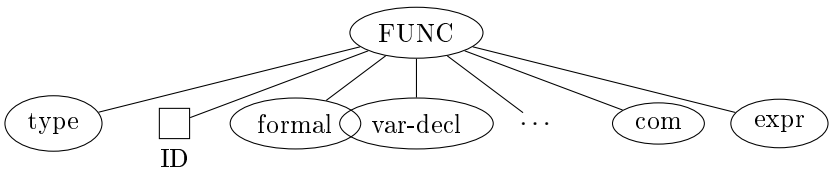
We specify the type of a variable, call it an identifier, and assign it the value of some expression. The ASTs for Fun types is given below:



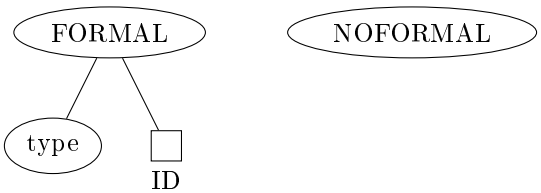
In Fun, there are only 2 types- bool and int. The AST for a procedure declaration is given below.



A procedure has an identifier, a noformal (parameter), followed by some variable declarations and command for the body. The AST for a function declaration is the following.



We have the return type, identifier, the parameter formal, command and expression. The AST for formal and nonformal parameters is given below.

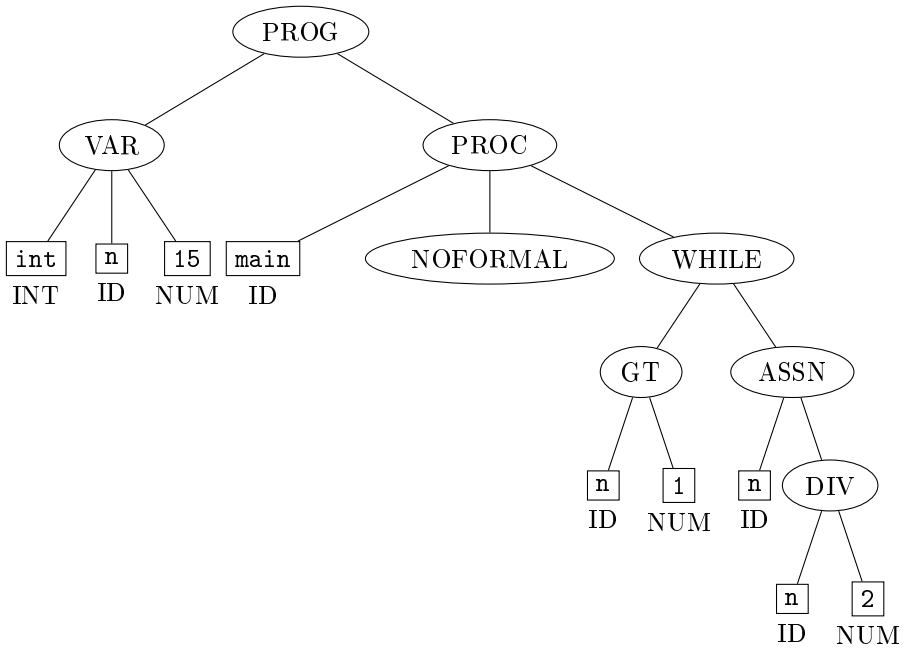


A formal parameter represents a single parameter (for functions) and no parameters (for procedures).

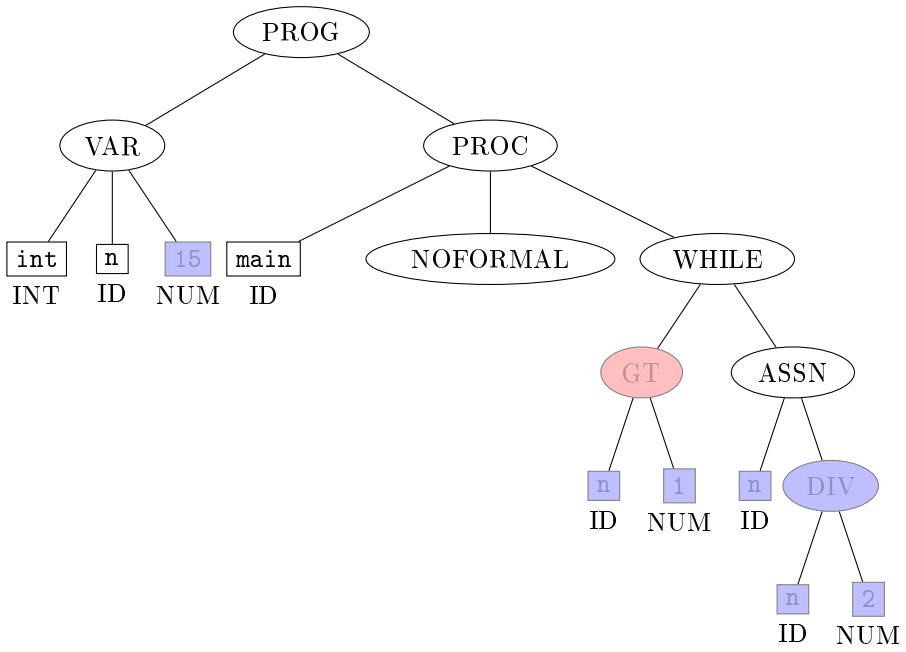
We will now create an AST from a program. So, consider the following program.

```
int n = 15
# div program
proc main ():
    while n > 1:
        n = n/2 .
.
```

Then, its AST is the following, after syntactical analysis of the program.



During contextual analysis, we do a type check and a scope check. Within the program, we find that we have a global variable `n:INT` and a global procedure `main:VOID` $\rightarrow$ `VOID`. This is stored in a type table, specifying the name and the type. Using this, we can check the type and infer the type of each expression. This is given in the annotated AST below.



A red value represents a `BOOL`, while a blue value represents an `INT`. Finally, we can create the SVM object code.

```
// load constant 15
LOADC 15
// go to 7 = line 8
CALL 7
// stop
HALT
// load the variable at address 0 (variable n)
LOADG 0
// load constant 1
LOADC 1
// compare n > 1
COMPGT
// if false, go to 30 = line 26 (and halt)
JUMPF 30
// load the variable at address 0 (variable n)
LOADG 0
// load constant 2
LOADC 2
// compute n/2
DIV
// store the result at address 0
STOREG 0
// go to 7 = line 8
JUMP 7
// return the value at address 0 (variable n)
RETURN 0
```

2.4 Syntactic Analysis

Syntactic analysis checks that the source program is well-formed and determines its phrase structure. The process of syntactic analysis can be broken into 2 further subphrases:

- a lexer, which breaks the source program down into tokens; and
- a parser, which determines the phrase structure of the program.

The syntactic analyser inputs a source program and outputs an AST. Inside the syntactic analyser, the lexer channels a stream of tokens to the parser, as shown below.

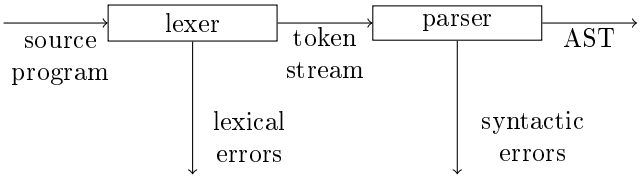


Figure 2.10: The data flow within the syntactic analysis phase.

Tokens

Tokens are textual symbols that influence the source program's phrase structures, such as literals, identifiers, operators, keywords and punctuation (e.g. commas, colons, parentheses). Each token has a tag and a text. For example, the addition operator might have tag PLUS and text `+`; a numeral might have tag NUM and text `1`; and an identifier might have tag ID and text `x`. These will become the leaves of the AST.

Separators are pieces of text that do not influence the phrase structure, such as spaces and comments. An end-of-line is a separator in most PLs, but a token in Python.

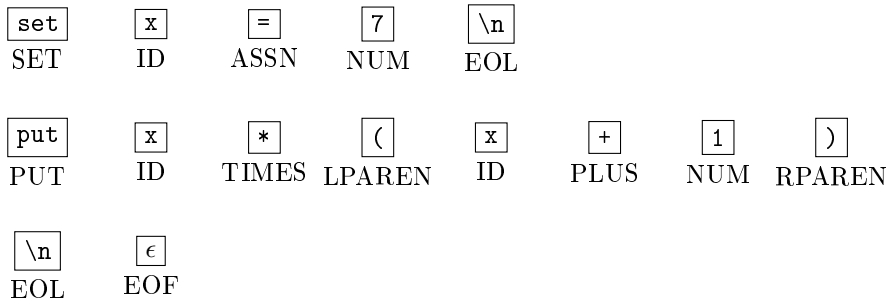
The following are tokens in Calc:

	`put` PUT	`set` SET	
`=` ASSN	`+` PLUS	`-` MINUS	`*` TIMES
	`(` LPAR	`)` RPAR	
	`...` NUM	`...` ID	
	`\n` EOL	ϵ EOF	

Now, assume we have the following Calc program:

```
set x = 7
put x * (x + 1)
```

We get the following token stream from the lexer:



The lexer converts source source code into a token stream. At each step, the lexer inspects the next character of the source code and acts accordingly. When there is no source code left, the lexer outputs an EOF token. Depending on the next character, we choose what to do as follows:

- space- we discard it
- the start of a comment- we scan the rest of the comment and then discard it
- punctuation mark- we output the corresponding token
- digit- we scan the remaining digits and output the corresponding token (a NUM)
- letter- we scan the remaining letters and output the corresponding token (either an ID or a keyword)

Parser

The parser converts a token stream into an AST. Parsing is divided into two kinds, reflected in how the parse tree is constructed- top-down (recursive-descent or backtracking), and bottom-up. Recursive-descent (RD) parsing is common and particularly simple. It uses recursive procedures to process the stream of tokens. Give a suitable grammar for the source language, we can systematically write a RD parser for it.

A recursive-descent parser consists of:

- a family of parsing methods, one for each non-terminal symbol of the source language's grammar; and
- an auxiliary method `match`.

We say that these methods consume the token stream from left to right. Moreover, in recursive-descent parser, these families of methods perform the following checks/parsing.

- The method `match(t)` checks whether the next token tag `t`. If yes, it consumes the token. Otherwise, it reports a syntactic error.

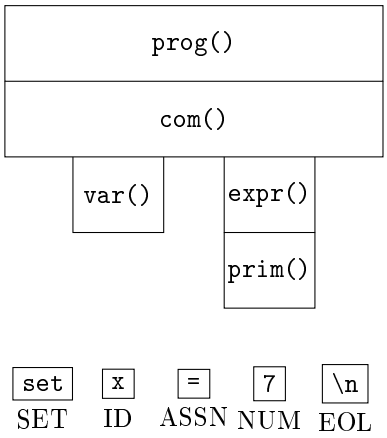
- For each non-terminal symbol N, the method `N()` checks whether the next few tokens constitute a phrase of class N. If yes, it consumes those tokens (and returns an AST representing the parsed phrase). Otherwise, it reports a syntactic error.

Now, we consider a Calc parser. The parsing methods for Calc are:

- `prog()`, which parses a program;
- `com()`, which parses a command;
- `expr()`, which parses an expression;
- `prim()`, which parses an primary expression;
- `var()`, which parses a variable.

Assume that we have the following token stream.

Then, the following figure illustrates the recursive nature of the parsing method.



We say that the entire token stream is consumed by `com()`, the ID token is consumed by `var()`, and so on.

In Calc, the EBNF production rule for command is:

$$\begin{aligned} com &= \texttt{put}\ expr\ eol \\ &\mid \texttt{set}\ var\ \texttt{=}\ expr\ eol \end{aligned}$$

So, the parsing method for the command (in Java) is given by the following:

```
public void com() {
    // if the next token is 'put'
    if (tokenStream.next() == "put") {
        match(PUT);
        expr();
        match(EOL);
```

```
    } else if (tokenStream.next() == "set") {
        match(SET);
        var();
        match(ASSN);
        expr();
        match(EOL);
    } else {
        // syntactic error
    }
}
```

This is the way we convert EBNF rules into parsing methods. The EBNF production rule for program is:

$$prog \;=\; com^{*}\; eof$$

So, its parsing method is given below.

```
public void prog() {
    // while the next token matches a command token,
    while (tokenStream.next().isCom()) {
        com();
    }
    match(EOF);
}
```

In general, if we have the rule $N = t$, where t is a terminal system, we produce the following method.

```
public void N() {
    match(t);
}
```

Instead, if we have the rule $N = M$, where M is a non-terminal system, we produce the following method.

```
public void N() {
    M();
}
```

If we have the rule $N = RE_1RE_2$, where RE_1 and RE_2 are two regular expressions, we produce the following method.

```
public void N() {
    // match RE1
    // match RE2
}
```

Next, if we have the rule $N = RE^{*}$, where RE is a regular expression, we produce the following method.

```
public void N() {
    // while the next token can be RE,
    while (tokenStream.next().isRE()) {
        // match RE
    }
}
```

Now, if we have the rule $N = RE_1|RE_2$, where RE_1 and RE_2 are two regular expressions, we produce the following method.

```
public void N() {
    // while the next token can be RE1,
    if (tokenStream.next().isRE1()) {
```

```
4           // match RE1
5       }
6       // if the next token can be RE2,
7       else if (tokenStream.next().isRE2()) {
8           // match RE2
9       }
10      // while the next token can be RE,
11      else {
12          // syntactic error
13      }
14  }
```

This only works if no token can start both RE_1 and RE_2. In particular, it does not work if a product rule is left-recursive, i.e. it is of the form $N = X|N\ Y$.

Syntactic analysis has a variety of applications.

- It is used in compilers, as we have been used to compile Calc.
- It is used in XML applications to convert them to tree form.
- It is used in web browsers to parse and render HTML documents.
- It is used in natural language applications to parse and translate NL documents.

Compiler generator tools

A compiler generation tool automates the process of building compiler components. It takes as input the specification of what the compiler component is supposed to do, e.g. the input to a parser generator is a grammar. Examples of compiler generator tools include lex and yacc, JavaCC, ANTLR, etc.

We will look at ANTLR (Another tool for language recognition). It automatically generates a lexer and a recursive-descent parser, starting from a grammar file (.g4). It allows to build and walk a parse tree. We start by expressing the source language's grammar in ANTLR notation, which is very close to the EBNF notation.

ANTLR is composed of two main parts:

- the tool, which is used to generate the lexer and the parser, and
- the runtime, which is used to run them.

The tool is a Java program, whether the runtime is different for every language targeted. When we run ANTLR tool, we can specify the target language (Java by default) to generate a parser, e.g. in Python. ANTLR can generate listeners (by default) and visitors, which are the basis for implementing a contextual analyser and a code generator.

The ANTLR notation is very similar to EBNF. Lexer rules (at the end of the file) are all uppercase, and parser rules are all lowercase. The syntax of a rule is- name, a colon, the definition of the rule, and a terminating semicolon. The Calc grammar in ANTLR is shown below.

```
grammar Calc;
prog
        :   com* EOF
```

```
        ;
com
        :   PUT expr EOL            # put
        |   SET var ASSN expr EOL   # set
        ;
expr
        :   prim
            (operator +=
                (PLUS | MINUS | TIMES)
            prim)*                  # op
prim
        :   NUM                     # num
        |   ID                      # id
        |   LPAR expr RPAR          # parens
        ;
var
        : ID
        ;
PUT     : "put";
SET     : "set";
ASSN    : "=";
PLUS    : "+";
MINUS   : "-";
TIMES   : "*";
LPAR    : "(";
RPAR    : ")"
ID      : "a" .. "z";
NUM     : "0".."9"+;
EOL     : "\r"? "\n";
SPACE   : (" "| "\t")+ -> skip;
```

If we run the ANTLR tool on the grammar above, ANTLR will generate the following classes:

- The class `CalcLexer`, which contains methods that convert an input stream (source code) to a token stream.
- The class `CalcParser`, which contains parsing methods `prog()`, `com()`, etc. that consume the token stream.

We then write a driver program that runs `CalcParser`'s method `prog()`:

```
1  public class CalcRun {
2      public static void main(String[] args) {
3          // create the input stream
4          InputStream source = new FileInputStream(args[0]);
5          // create the lexer
6          CalcLexer lexer = new CalcLexer(
7              new ANTLRInputStream(source)
8          );
9          // run the lexer (creates a token stream)
10         CommonTokenStream tokens = new CommonTokenStream(lexer);
11         // create a parser
12         CalcParser parser = new CalcParser(tokens);
```

```
13          // run the parser
14          ParseTree tree = parser.prog();
15      }
16  }
```

When compiled and run, `CalcRun` performs syntactic analysis on the source program and reports any syntactic errors. It also constructs a syntax tree. The syntax tree can be viewed with the ANTLR `TestRig` tool. However, `CalcRun` does not execute the program. The program can be executed by traversing the syntax tree, left to right, depth first, and interpreting each command.

We get the syntax tree automatically by ANTLR. But, to implement an interpreter or compile, we need to walk over the tree and analyse its structure. We will use ASTs to explain, but ANTLR makes use of the full syntax tree with all the tokens. ANTLR uses the Visitor pattern to define tree walkers. It is also possible to use the Listener pattern to do so.

It is possible to configure ANTLR so that we visit the AST. Using a different command, ANTLR generates two further files:

- the interface `CalcVisitor`, which specifies a `visit` method for each type of syntax tree node, e.g. `visitProg`, `visitPut`, etc. The visitor methods have a single parameter `context`, which has the syntax tree node we are visiting.
- the class `CalcBaseVisitor`, which implements `CalcVisitor`. Initially, it is formatted to just walk over the syntax tree. We will need to adapt it so that it does what we want it to do.

Assume that we want `CalcRun` to perform calculations, i.e. `put` *expr* should evaluate the expression and print the result, and `set` *var* = *expr* should evaluate the expression and then store the result in the given variable. For this, we define the class `ExecVisitor` that extends `CalcBaseVisitor` and overrides the `visit` methods so that they interpret the program while walking over the syntax tree.

The following is the class `ExecVisitor`:

```
1  class ExecVisitor extends CalcBaseVisitor<Integer> {
2      // variables
3      int[] store = new int[26];
4
5      public Integer visitProg(CalcParser.ProgContext ctx) {
6          // interpret every command in the program
7          return visitChildren(ctx);
8      }
9
10     public Integer visitPut(CalcParser.PutContext ctx) {
11         // visit the relevant expression
12         int value = visit(ctx.expr());
13         System.out.println(value);
14         // no true value to return
15         return 0;
16     }
17
18     public Integer visitSet(CalcParser.SetContext ctx) {
19         // visit the relevant expression
20         int value = visit(ctx.expr());
21         // get the index corresponding to the variable
22         int address = ctx.var().ID().getText().charAt(0) - "a";
23         // add the value to the index
```

```
24          store[address] = value;
25          return 0;
26      }
27
28      public Integer visitOp(CalcParser.OpContext ctx) {
29          // get the values and the operations
30          List<CalcParser.PrimContext> prims = ctx.prim();
31          List<Token> ops = ctx.operator;
32          // visit the first value
33          int value = visit(prims.get(0));
34          // apply the operation to each of the next values
35          for (int i = 1; i < prims.size(); i++) {
36              switch (ops.get(i-1).getType()) {
37                  case CalcParser.PLUS;
38                      value = value + visit(prims.get(i));
39                      break;
40                  case CalcParser.MINUS;
41                      value = value - visit(prims.get(i));
42                      break;
43                  ...
44              }
45          }
46      return value;
47      }
48
49      public Integer visitNum(CalcParser.NumContext ctx) {
50          return Integer.valueOf(ctx.NUM().getText());
51      }
52
53      public Integer visitId(CalcParser.IdContext ctx) {
54          int address = ctx.ID().getText().charAt(0) - "a";
55          return store[address];
56      }
57
58      public Integer visitParens(CalcParser.ParensContext ctx) {
59          return visit(ctx.expr());
60      }
61  }
```

The driver program needs to be extended to create and call an `ExecVisitor`.

```
1  public class CalcRun {
2      public static void main(String[] args) {
3          InputStream source = new FileInputStream(args[0]);
4          CalcLexer lexer = new CalcLexer(
5              new ANTLRInputStream(source)
6          );
7          CommonTokenStream tokens = new CommonTokenStream(lexer);
8          CalcParser parser = new CalcParser(tokens);
9          ParseTree tree = parser.prog();
10         ExecVisitor exec = new ExecVisitor();
11         exec.visit(tree);
12     }
13 }
```

When compiled and run, `CalcRun` performs syntactic analysis on the source program and then interprets it.

Below is the Fun grammar in the ANTLR notation.

```
grammar Fun;

prog
```

```
        :   var_decl* proc_decl+ EOF        # prog
        ;
var_decl
        :   type ID ASSN expr               # var
        ;
type
        :   BOOL                            # bool
        |   INT
        ;
com
        :   ID ASSN expr                    # assn
        |   WHILE expr COLON seq_com DOT    # while
seq_com
        :   com*                            # seq
        ;
expr    :   e1=sec_expr ...
sec_expr
        :   e1=prim_expr
            ( op = (PLUS | MINUS | TIMES | DIV)
                e2 = sec_expr
            ) ?
prim_expr
        :   NUM                             # num
        |   ID                              # id
        |   LPAR expr RPAR                  # parens
        |   ...
```

Running this with the ANTLR tool generates a lexer and a parser. This creates the classes `FunLexer` and `FunParser`. The `prog()` method returns a full syntax tree.

2.5 Contextual Analysis

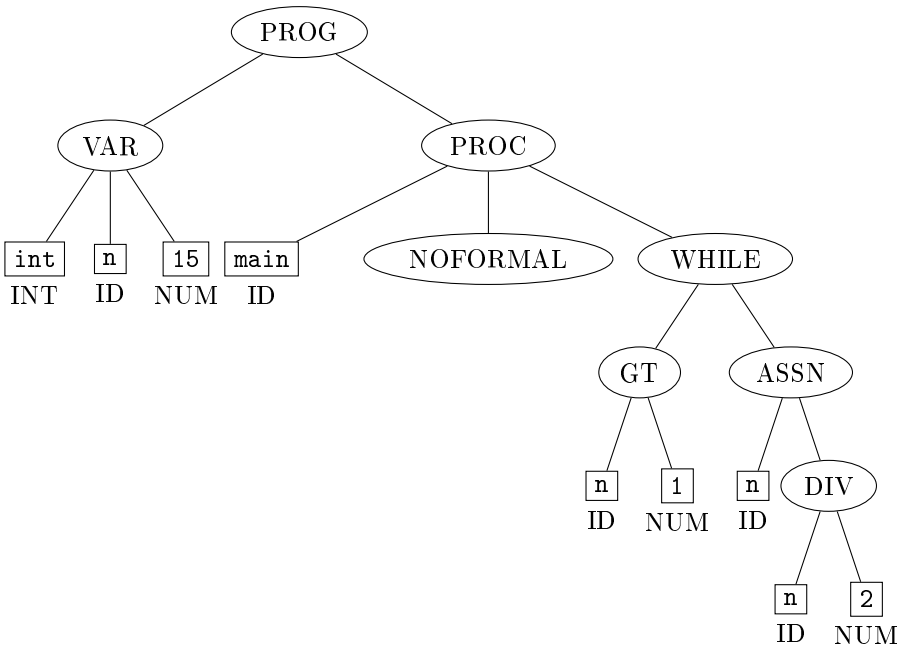
Contextual analysis checks whether the source program (represented by an AST or a syntax tree) satisfies the source language's scope rules and type rules. Logically, there are 2 phases to contextual analysis:

- scope checking, which ensures that every identifier used in the source program is declared, and
- type checking, which ensures that every operation has operands with the expected types.

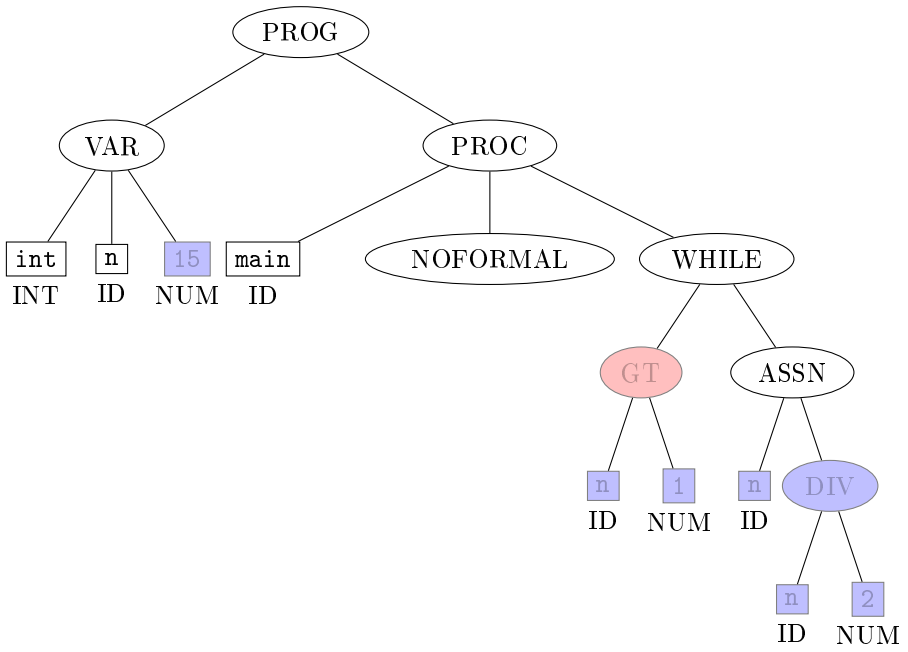
For example, consider the following source program in Fun:

```
int n = 15
# div program
proc main ():
    while n > 1:
        n = n/2 .
.
```

After syntactic analysis, we get the following AST.



During contextual analysis, we will infer the types of the variables and functions, and get the following enhanced AST:



A red value represents a `BOOL`, while a blue value represents an `INT`. We want to keep a type table for each variable. For example, the identifier `n` is known to be an integer. When we assign `n = n/2`, this is valid since the division operation returns an integer. In contextual analysis, we check whether the variable exists (scope) and whether it matches the expected type. We also check that the division operator receives 2 integers, and the condition on a while statement is of type bool.

Scope checking

Scope checking is the collection and dissemination of information about declared identifiers. The contextual analyser employs a type table. This contains the type of each declared variable, e.g.

`n`	`BOOL`
`fac`	`INT` → `INT`
`main`	`INT` → `VOID`

Table 2.2: A type table

Whenever an identifier is declared, we put the identifier and its type into the type table. If the identifier is already in the type table (and in the same scope), we report a scope error. Whenever an identifier is used, we check that it is in the type table, and retrieve its type. If the identifier is not in the type table, we report a scope error.

We illustrate the process with Fun scope checking. In Fun, a declaration of a variable identifier is of the form

type id = *expr*

So, we put the identifier value into the type table, along with its type. If the identifier already exists, we return a scope error. Similarly, a use of a variable identifier in Fun is of the form

$$id \texttt{ = } expr$$

In this case, we lookup the identifier in the type type and retrieve its type. If the identifier does not exist, we return a scope error.

Type checking

Type checking is the process of checking that every command and expression is well-typed, meaning that it does not contain any errors. The compiler only performs type checking if the source language is statically typed, i.e. the type of an identifier is known at compile-time.

For example, at each expression, we check the type of each sub-expression. We then infer the type of the expression as a whole. If a sub-expression has unexpected type, we report a type error. Similarly, at each command, we check the type of any constituent expression. If an expression has unexpected type, we report a type error. We do not return a type at the end.

Now, we consider some Fun type checking. The greater than expression is of the form

$$expr_1 > expr_2$$

We walk $\texttt{expr}_1$ and check that its type is `INT`, and do the same thing for $\texttt{expr}_2$. We then infer that the type of the whole expression is `BOOL`. An assignment command is of the form

$$id \texttt{ = } expr$$

We lookup the identifier and find its type. We then walk the expression and note its value. At the end, we require that the two types are equivalent. An if-command is of the form

$$\texttt{if } expr\ com$$

We first walk expression, and check that its type is `BOOL`. We then walk the command to ensure that there are no type errors within the command.

Contextual Analysis in ANTLR

Contextual analysis is done by walking over the syntax tree. We created a base visitor class during syntactical analysis, so we create another class that extends this class. For contextual analysis, we require also `SymbolTable`. We use the class `Type` to representation the type of a value in Fun.

Visting an expression returns the type of the expression, if any. For example, the `visitNum` method is of the form:

```
Type visitNum(FunParser.NumContext ctx) {
    return Type.INT;
}
```

The `visitID` method, that visits an identifier is given below.

```
Type visitID(FunParser.IdContext ctx) {
    return retrieve(ctx.ID().getText(), ctx);
}
```

To get the type of an identifier, we use the `retrieve` method. This is returned from the type table. We also pass in the parameter `ctx` to extract the line number in the case of a type/scope error.

In Fun, an expression is of the form

$$e_1 = \ sec\_expr \ (op = (= \mid < \mid >) \ e_2 = \ sec\_expr)?$$

So, the `visitExpr` method is the following.

```
Type visitExpr(FunParser.ExprContext ctx) {
    Type t1 = visit(ctx.e1);
    if (ctx.e2 != null) {
        Type t2 = visit(ctx.e2);
        return checkBinary(COMPTYPE, t1, t2, ctx);
    }
    else {
        return t1;
    }
}
```

Every expression has e_1, so we visit it. If e_2 does not exist, then the type of the expression is the same as the type of e_1. Otherwise, we visit the second expression and ensure that both the types are `INT`. The function `checkBinary` returns the type `BOOL` after checking the types. The type `COMPTYPE` is the type representing $\texttt{INT} \times \texttt{INT} \to \texttt{BOOL}$.

In Fun, an assignment command is of the form

$$ID \ = \ expr$$

So, the `visitAssn` method is the following.

```
Type visitAssn(FunParser.AssnContext ctx) {
    Type tvar = retrieve(ctx.ID().getText(), ctx);
    Type t = visit(ctx.expr());
    checkType(tvar, t, ctx);
    return null;
}
```

We first try to find the type of the identifier. Then, we visit the expression and check that the type of the expression matches the type of the identifier. We return `null` at the end since a command does not have a type.

In Fun, an if command is of the form

$$\texttt{IF} \ \ expr\texttt{:} \ c_1 = seq\_com(\texttt{.} \mid \texttt{ELSE:} \ c_2 = seq\_com \ \texttt{.})$$

So, the `visitIf` method is the following.

```
Type visitIf(FunParser.IfContext ctx) {
    Type t = visit(ctx.expr());
    visit(ctx.c1);
    if (ctx.c2 != null) {
        visit(ctx.c2);
    }
    checkType(Type.BOOL, t, ctx);
    return null;
}
```

We first visit the expression, and then the commands. We require the type of the expression to be `BOOL`.

A sequence of commands is given by

$$com^*$$

So, the `visitSeq` method is the following.

```
Type visitSeq(FunParser.SeqContext ctx) {
    visitChildren(ctx);
    return null;
}
```

We are just visiting all the children nodes in the tree and not type checking anything at this level.

A variable declaration is given by

$$type\ id\ =\ expr.$$

So, the `visitVar` method is the following.

```
Type visitVar(FunParser.VarContext ctx) {
    Type t1 = visit(ctx.type());
    Type t2 = visit(ctx.expr());
    define(ctx.ID().getText(), t1, ctx);
    checkType(t1, t2, ctx);
    return null;
}
```

We define the identifier to be of the given type. Then, we check that the type of the expression matches the type of the identifier given.

Finally, a program is given by

$$var\_\ decl^*\ proc\_\ decl^+$$

So, the `visitProg` method is the following.

```
Type visitProg(FunParser.ProgContext ctx) {
    predefine();
    visitChildren(ctx);
    Type tmain = retrieve("main", ctx);
    checkType(MAINTYPE, tmain, ctx);
    return null;
}
```

The function `predefine` adds the read and write functions into the type table. We visit all the children to find any type errors within the program. We expect there to be a `main` procedure in each program, so we check that it exists and its type is `VOID` $\rightarrow$ `VOID` (called `MAINTYPE`).

The driver program visits the tree, as given below.

```
public static void main(String[] args) {
    ..
    // Syntactic analysis
    ParseTree tree = parser.program();
    // Contextual analysis
    FunCheckerVisitor checker = new FunCheckVisitor(tokens);
    checker.visit(tree);
}
```

Representing types

To implement type checking, we need a way to represent the source language's types. We know that the type of a variable/function can be primitive, Cartesian product, disjoint union or mapping. In the Fun language, we have primitive, Cartesian product and mapping.

We represent Fun primitive data types by `Type.BOOL` and `Type.INT` static variables. For a function, we use a mapping type, e.g. `T` $\rightarrow$ `T'` or `VOID` $\rightarrow$ `T'`. We can represent procedures in a similar way, e.g. `T` $\rightarrow$ `VOID` or `VOID` $\rightarrow$ `VOID`. We also use mapping types for operators, e.g. `+`, `-`, `*` and `\` are of type (`INT` $\times$ `INT`) $\rightarrow$ `INT`; `==`, `<` and `>` are of type (`INT` $\times$ `INT`) $\rightarrow$ `BOOL`; and `not` is of type `BOOL` $\rightarrow$ `BOOL`.

The class `Type` is of the following format.

```
public abstract class Type {
    public abstract boolean equiv(Type t);

    public static class Primitive extends Type {
        ...
    }

    public static class Pair extends Type {
        ...
    }

    public static class Mapping extends Type {
        ...
    }
}
```

The 3 subclasses are used for the 3 types of types. The subclass `Type.Primitive` has a field that distinguishes different primitive types.

```
public static final Type
    VOID = new Type.Primitive(0),
    BOOL = new Type.Primitive(1),
    INT = new Type.Primitive(2);
```

The subclass `Type.Pair` has two `Type` fields, which are the types of the pair components, e.g.

```
Type prod = new Type.Pair(Type.BOOL, Type.INT);
```

represents the type `BOOL` $\times$ `INT`. Similarly, the subclass `Type.Mapping` has two `Type` fields. These are the domain type and range of the mapping type, e.g.

```
Type proctype = new Type.Mapping(Type.INT, Type.VOID);
Type optype = new Type.Mapping(
    new Type.Pair(Type.INT, Type.INT), Type.BOOL
);
```

The type `proctype` corresponds to the map `INT` $\rightarrow$ `VOID`, while the type `optype` corresponds to the map (`INT` $\times$ `INT`) $\rightarrow$ `BOOL`.

2.6 Scopes

In Fun, declarations are global or local. A PL of this structure is said to have flat block structure. The same identifier can be declared both globally and locally, e.g.

```
1  int x = 1
2
3  proc main ():
4      int x = 2
5      write(x)
6  .
7
8  proc p (bool x):
9      if x:
10         write(9)
11     .
12 .
```

The variable `x` declared at line 1 is a global variable. The variables `x` at lines 4 and 9 are local variables, and do not affect the value of the global variable.

The type table must distinguish between global and local entries. The global entries are always present. Local entries are present only when analysing an inner scope. At any given point during analysis of the source program, the same identifier may occur in at most one global entry and at most one local entry. Most PLs have nested scopes instead of this flat global/local scope.

For the Fun program above, the type table of the program at line 1 is the following.

global	x	INT

At line 4, the type table is the following.

global	x	INT
global	main	VOID → VOID
local	x	INT

At line 9, the type table is the following.

global	x	INT
global	main	VOID → VOID
global	p	BOOL → VOID
local	x	BOOL

To implement the type table, we can use 2 hash tables- one for global variables and one for local. This is shown below.

```
1  public class SymbolTable<A> {
2      private HashMap<String, A> globals, locals;
3
4      public SymbolTable() {
5          globals = new HashMap<String, A>();
6          locals = null;
7      }
8  }
```

When the symbol table gets initialised, the local scope does not exist.

We can enter and leave the local scope whenever we want as follows.

```
1  public void enterLocalScope() {
2      locals = new HashMap<String, A>();
```

```
}

public void exitLocalScope() {
    locals = null;
}
```

So, we are at a local scope if and only if the variable `locals` is not null.

We can add variables and retrieve them to the symbol table as follows.

```
public void put (String id, A attr) {
    if (locals != null) {
        locals.put(id, attr);
    } else {
        globals.put(id, attr);
    }
}

public A get(String id) {
    if (locals != null && locals.containsKey(id)) {
        return locals.get(id);
    } else {
        return globals.get(id);
    }
}
```

Each time, we check whether we are in a local scope or a global scope. In the `put` case, we add the entry to the relevant map. In the `get` case, we first check whether we have the entry in the local scope. If so, we return that value; otherwise, we return the value at the global scope.

2.7 Code Generation

Code generation translates the source program (represented by an AST or a syntax tree) into equivalent object code. In general, code generation can be broken into:

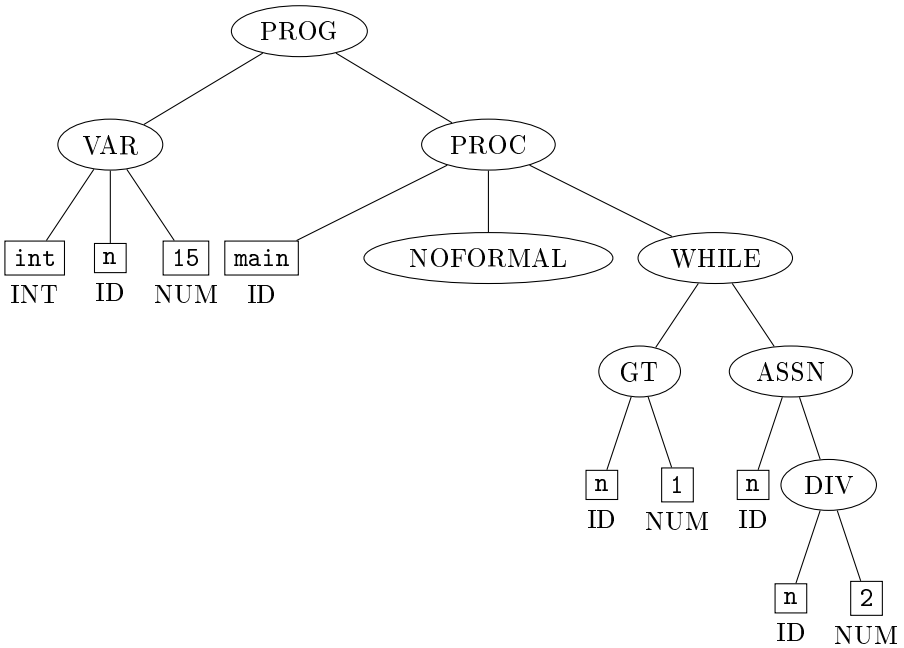
- address allocation, where we declare the representation and address of each variable in the source program;
- code selection, where we select and generate object code; and
- register allocation, where we assign registers to local and temporary variables, if applicable.

We will focus on stack-based virtual machines. In that case, address allocation and code generation are straightforward, and we do not have register allocation. For real machines, register allocation is somewhat complicated.

The following is a program in Fun.

```
1 int n = 15
2 proc main():
3     while n > 1:
4         n = n/2
5     .
6 .
```

The corresponding AST for it that we generate after syntactic analysis is given below.



In code generation, we take this AST and convert it into the following SVM object code.

```
1 // load constant 15
2 LOADC 15
3 // go to 7 = line 8
4 CALL 7
5 // stop
6 HALT
7 // load the variable at address 0 (variable n)
8 LOADG 0
9 // load constant 1
10 LOADC 1
11 // compare n > 1
12 COMPGT
13 // if false, go to 30 = line 26 (and halt)
14 JUMPF 30
15 // load the variable at address 0 (variable n)
16 LOADG 0
17 // load constant 2
18 LOADC 2
19 // compute n/2
20 DIV
21 // store the result at address 0
22 STOREG 0
23 // go to 7 = line 8
24 JUMP 7
25 // return the value at address 0 (variable n)
26 RETURN 0
```

The address table is:

n	0 (global)
main	7 (code)

Address allocation

Address allocation requires collection and dissemination of information about declared variables, procedures, etc. The code generator employs an address table. This contains the address of each declared variable, procedure, etc.

At each variable declaration, we allocate a suitable address, and put the identifier and address into address table. Wherever a variable is used (e.g. in a command or expression), we retrieve its address. At each procedure declaration, we note the address of its entry point, and put the identifier and address into the address table. Wherever a procedure is called, we retrieve its address.

We typically allocate consecutive addresses to variables, taking account of their size, e.g. we can have a variable of size 4, 2 or so on. In Fun, all variables are of size 1.

Code selection

The code generator will walk the AST. For each construct (e.g. an expression or a command) in the AST, the code generator must emit suitable object code. We must plan what object code will be selected by the code generator.

For each construct in the source language, we devise a code template. This specifies what object code will be selected. The code template to evaluate an expression should include code to evaluate any sub-expressions, together with any other necessary instructions. The code template to execute a command

should include code to evaluate any sub-expressions amd code to execute any sub-commands, together with any other necessary instructions.

We will now look at some SVM code templates for different expressions. The addition operation is of the form

$$expr_1 \texttt{ + } expr_2.$$

The code template here is:

- we evaluate $expr_1$;
- we evaluate $expr_2$; and
- we run the `ADD` operation.

For example, if we have `m + (7 * n)`, then the SVM code is:

```
// load variable m
LOADG 3
// load constant 7
LOADC 7
// load variable n
LOADG 4
// compute 7 * n
MUL
// compute m + (7 * n)
ADD
```

Because of the stack nature of data store, this SVM code gets implemented in the expected manner.

The code template specifies what code should be selected. The action specifies what the code generator will actually do to generate the selected code. A template can have several actions, depending on the object code language. For example, the action would be different if we were not using SVM, but the template remains the same.

The assignment command is of the form

$$ID \texttt{ = } expr$$

The code template here is:

- we evaluate *expr*
- we run the `STOREG` or `STOREL` operation to store the value at a specified address offset.

For example, if we have `m = n - 9`, then the SVM code is:

```
// load the variable n
LOADG 4
// load the constant 9
LOADC 9
// compute n - 9
SUB
// store the result as variable m
STOREG 3
```

The code generator action for the assignment command is:

- walk `expr` and generate the code;
- lookup the identifier and retrieve its address d;
- emit instruction `STOREG` d (if global variable), or `STOREL` d (if local variable).

The code generator emits instructions one by one. When an instruction is emitted, it is added to the end of the object code. At the destination of a jump instruction, the code generator must note the destination address and incorporate it into the jump instruction.

For a backward jump, the destination address is already known when the jump instruction is emitted. On the other hand, for a forward jump, the destination address is unknown when the jump instruction is emitted. In that case, we emit an incomplete jump address (with 0 in its address field) and note its address. When the destination address becomes known later, we patch that address into the jump instruction.

We can see this in the if command. An if command is of the form

`if` *expr com*

The code template is:

- evaluate the *expr*
- `JUMPF` to the end of *com*
- evaluate the *com*
- patch the location of `JUMPF` with the end of *com* evaluation.

For example, if we want to execute `if m > n: m = n.`, then we get the following SVM code:

```
// load variable m
LOADG 3
// load variable n
LOADG 4
// compute m > n
COMPGT
// if false, jump to the end
JUMPF *
// load variable m
LOADG 4
// store as variable n
STOREG 3
```

The code generator action for if-command is:

- walk *expr* and generate code;
- emit the instruction `JUMPF 0`;
- walk *com* and generate code;
- patch the correct address into the `JUMPF` instruction.

Next, we consider the while command. It is of the form

`while` *expr com*

The code template here is:

- evaluate the *expr*
- `JUMPF` to after the end of *com*
- evaluate the *com*
- `JUMP` to before *expr*

Code generation with ANTLR

The code generator is a visitor, with a similar structure to the contextual analysis visitor. For each type of syntax tree node, the visit method implements the code generation action. The preamble is given below:

```
// creates an instance of SVM. The code generator will emit
// instructions directly into its code store
SVM obj = new SVM();
int globalvaraddr = 0;
int localvaraddr = 0;
int currentLocale = Adress.GLOBAL;

SymbolTable<Address> addrTable = new SymbolTable<>();
```

We need to implement visit methods for each kind of node in the AST. The following is the method for visiting numbers.

```
void visitNum(FunParser.NumContext ctx) {
    int value = Integer.parseInt(ctx.NUM().getText());
    obj.emit12(SVM.LOADC, value);
}
```

The command `emit12` means 1 opcode and 2 byte operand.

The visit identifier method is given below.

```
void visitId(FunParser.IDContext ctx) {
    String id = ctx.ID().getText();
    Address varaddr = addrTable.get(id);
    switch (varaddr.locale) {
        case Address.GLOBAL:
            obj.emit12(SVM.LOADG, varaddr.offset);
            break;
        case Address.LOCAL:
            obj.emit12(SVM.LOADL, varaddr.offset);
            break;
    }
}
```

The visit expression method is given below.

```
void visitExpr(FunParser.ExprContext ctx) {
    // generate code to evaluate e1
    visit(ctx.e1);
    // if e2 exists, then generate code to evaluate e2
    if (ctx.e2 != null) {
        visit(ctx.e2);

```

```
        // generate an instruction for each operator
        switch (ctx.op.getType()) {
            case FunParser.EQ:
                obj.emit1(SVM.CMPEQ);
                break;
            case FunParser.LT:
                obj.emit1(SVM.CMPLT);
                break;
            case FunParser.GT:
                obj.emit1(SVM.CMPGT);
                break;
        }
    }
}
```

The command `emit1` means just 1 opcode.

The visit assignment method is given below.

```
void visitAssn(FunParser.AssnContext ctx) {
    // generate code to evaluate expr
    visit(ctx.expr());
    String id = ctx.ID().getText();

    // find the address of the variable
    // this always succeeds, because we added the address
    // during contextual analysis
    Address varaddr = addrTable.get(id);
    switch (varaddr.locale) {
        case Address.GLOBAL:
            obj.emit12(SVM.STOREG, varaddr.offset);
            break;
        case Address.LOCAL:
            obj.emit12(SVM.STOREL, varaddr.offset);
            break;

    }
}
```

The visit if method is given below.

```
void visitIf(Funparser.IfContext ctx) {
    visit(ctx.expr());
    int condaddr = obj.currentOffset();
    // This has to be patched later
    obj.emit12(SVM.JUMPF, 0);
    // IF without ELSE
    if (ctx.c2 == null) {
        visit(ctx.c1);
        int exitaddr = obj.currentOffset();
        obj.patch12(condaddr, exitaddr);
    } else {
        visit(ctx.c1);
        int jumpaddr = obj.currentOffset();
        obj.emit12(SVM.JUMP, 0);
        int elseaddr = obj.currentOffset();
        obj.patch12(conaddr, elseaddr);
        visit(ctx.c2);
        int exitaddr = obj.currentOffset();
        obj.patch12(jumpaddr, exitaddr);
    }
}
```

The visit var method is given below.

```
void visitVar(FunParser.VarContext ctx) {
    visit(ctx.expr());
    String id = ctx.ID().getText();
    switch (currentLocale) {
        // adding the variable to the address table always
        // succeeds because we have done contextual analysis
        case Address.LOCAL:
            addrTable.put(id, new Address(localvaraddr++,
                Address.LOCAL));
            break;
        case Address.GLOBAL:
            addrTable.put(id, new Address(globalvaraddr++,
                Address.GLOBAL));
            break;
    }
}
```

Storage organisation

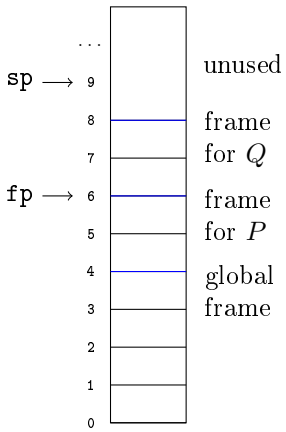
Each variable occupies storage space throughout its lifetime. This storage must be allocated at the start of the variable's lifetime and deallocated at the end of the variable's lifetime. For a statically typed PL, every variable's type is known to the compiler. Moreover, we assume that all variables of the same type occupy the same amount of storage space.

A global variable's lifetime is the program's entire runtime. For global variables, the compiler allocates fixed storage space. On the other hand, a local variable's lifetime is an activation of the block in which variable is declared. The lifetime of local variables are nested. For local variables, the compiler allocates storage space on a stack.

At any given time, the stack contains one or more activation frames. The frame at the base of the stack contains the global variables. For each active procedure P, there is a frame containing P's local variables. An active procedure is one that has been called but not yet returned. A frame for a procedure P is pushed onto the stack when P is called, and popped off the stack when P returns.

The compiler fixes the size and layout of each frame. The offset of each global and local variable (relative to the frame) is known to the compiler.

This can be seen in the figure below.



The diagram represents the SVM data store when the main program has called P, and P has called Q. A local frame contains the local variables, the return address and the dynamic link which points to the previous frame. The stack pointer (sp) points to the first cell above the top of the stack, and the frame pointer (fp) points to the first cell of the topmost frame.

We will use this for procedure declaration, whose visit method is given below.

```
void visitProc(FunParser.ProcContext ctx) {
    String id = ctx.ID().getText();
    Address procaddr = new Address(obj.currentOffset(),
        Address.CODE);
    addrTable.put(id, procaddr);
    addrTable.enterLocalScope();
    currentLocale = Address.LOCAL;
    localvaraddr = 2;

    FunParser.Formal_declContext fd = ctx.formal_decl();
    if (fd != null)
        visit(fd);
    List<FunParser.Var_declContext> var_decl = ctx.var_decl();
    for (FunParser.Var_declContext vd : var_decl) {
        visit(vd);
    }
    visit(ctx.seq_com());
    // 0 because there is no result
    obj.emit11(SVM.RETURN, 0);
    addrTable.exitLocalScope();
    currentLocale = Address.GLOBAL;
}
```

The visit formal method is given below.

```
void visitFormal(FunParser.FormalContext ctx) {
    FunParser.TypeContext tc = ctx.type();
    if (tc != null) {
        String id = ctx.ID().getText();
        // A parameter is like a local variable
        addrTable.put(id, new Address(localvaraddr++,
            Address.LOCAL));
        // copy arguments into the stack frame
        obj.emit11(SVM.COPYARG, 1);
    }
}
```

Finally, the visit program method is given below.

```
void visitProg(FunParser.ProgContext ctx) {
    // add read and write to the address table
    predefine();
    List<FunParser.Var_declContext> var_decl = ctx.var_decl();
    for (FunParser.Var_declContext vd: var_decl) {
        visit(vd);
    }
    int calladdr = obj.currentOffset();
    // call the main method (will patch)
    obj.emit12(SVM.CALL, 0);
    obj.emit1(SVM.HALT);
    List<FunParser.Proc_declContext> proc_decl = ctx.proc_decl();
    for (FunParser.Proc_declContext pd : proc_decl) {
        visit(pd);
    }
    int mainaddr = addrTable.get("main").offset;
```

```
17      obj.patch12(calladdr, mainaddr);
18  }
```

The code generator must distinguish between three kinds of addresses:

- code addresses, which refers to an instruction within the space allocated to the object code.
- global address, which refers to a location within the space allocated to global variables.
- local address, which refers to a location within a space allocated to a group of local variables.

The implementation of address in Java is given below.

```
1  public class Address {
2      public static final int
3          CODE = 0, GLOBAL = 1, LOCAL = 2
4      public int offset;
5      // CODE, GLOBAL or LOCAL
6      public int locale;
7
8      public Address(int off, int loc) {
9          offset = off;
10         locale = loc;
11     }
12 }
```