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.

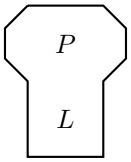


Figure 2.1: A tombstone diagram for a program P in language L.

- The following figure is used to denote a translator.

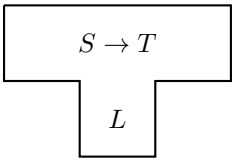


Figure 2.2: An $S \rightarrow T$ translator expressed in the language L.

- The following figure is used to denote an interpreter

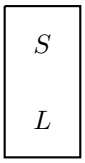


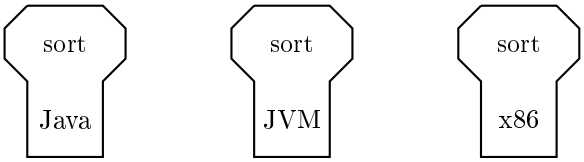
Figure 2.3: An S interpreter in the language L.

- The following figure is used to denote a hardware.

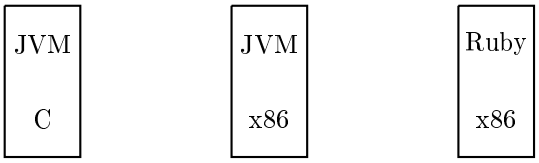
Figure 2.4: A machine M which can only execute M's machine code.

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

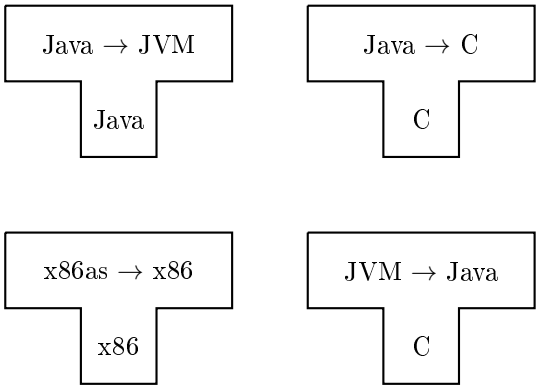
- We can denote the the program `sort` written in 3 different languages- Java, JVM and x86.



- 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.

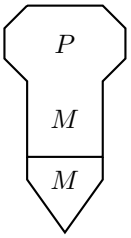
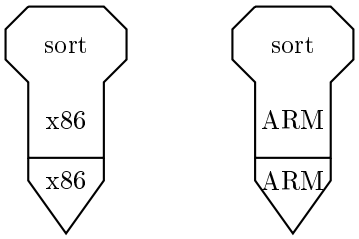
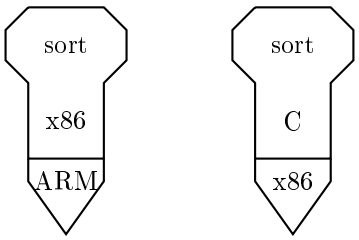


Figure 2.5: Running a program P written in M.

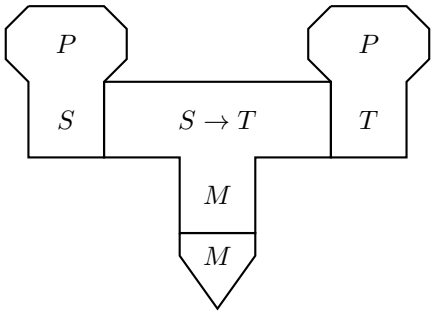
Note that the two M's must match for the program to run. So, we can run the sort program in x86 and ARM given the right hardware.



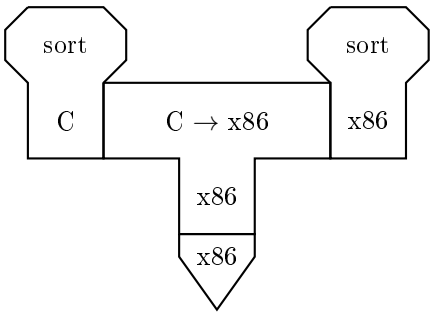
However, the following are not possible since the language of the program doesn't match the hardware.



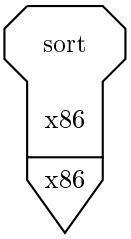
We can also show the translation of a program via tombstone diagrams. Given an $S \to T$ translator, expressed in M machine code, and a program P expressed in language S, we can translate P to language T.



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.

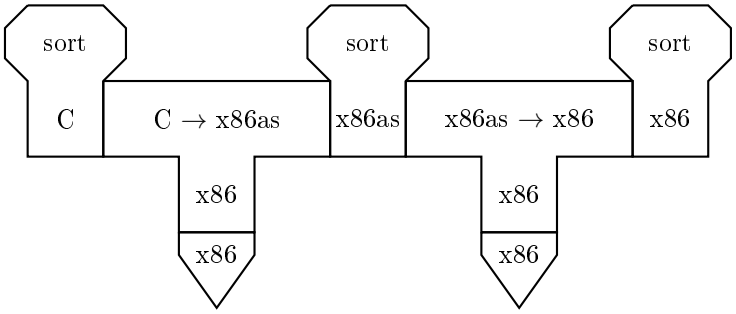


This represents the compilation of the program. We can later run the object program on an x86, which is depicted by the tombstone diagram below.

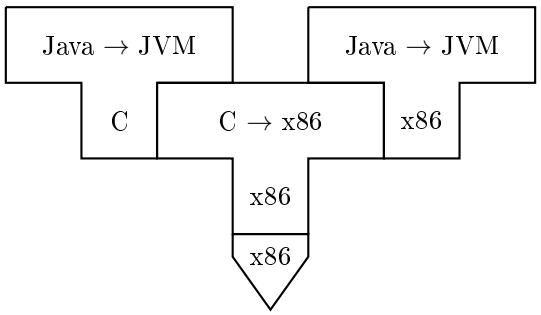


This represents the runtime of the program.

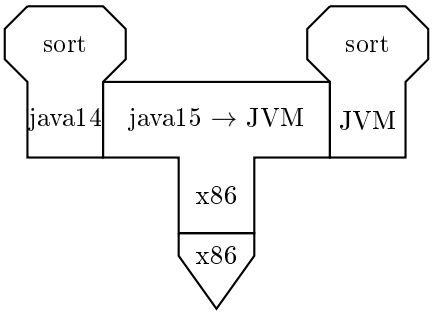
We could break the compilation into 2 steps. For example, we can have a C → x86as compiler, and an x86 assembler. Then, we can use them to compile a program in C into x86 machine code, in 2 stages.



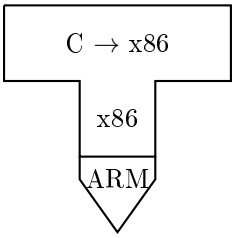
We can compile a compiler. Given a C → x86 compiler, we can use it to compile any C program into x86 machine code. In particular, we can compile a compiler expressed in C, e.g. compiling a Java → JVM compiler written in C to a Java → JVM compiler in x86.



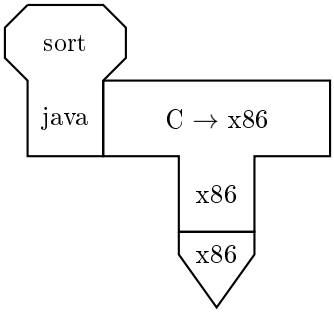
For this to be possible, we require the relevant languages to match. So, the following is possible.



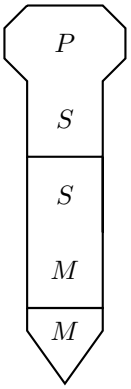
This is because Java14 is a subset of Java15. However, we cannot run a C → x86 compiler in an ARM hardware.



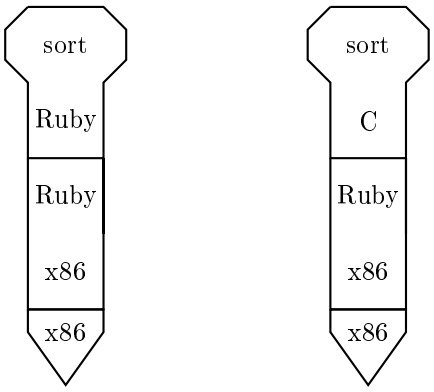
Similarly, we cannot compile a Java program using a C → x86 compiler.



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.



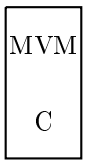
The language in which the program is written (i.e. S) must match the language that is being interpreted. Moreover, the language in which the interpreter is written (i.e. M) must match the language of the hardware. So, it is possible to interpret a sort program in Ruby with a Ruby interpreter, but not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



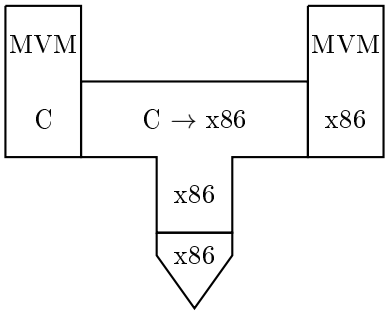
Virtual machines

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.

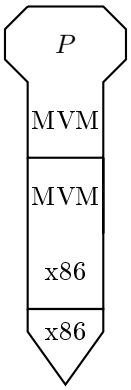
We can first write an interpreter for MVM machine code (an emulator) expressed in C, for example. This is given below.



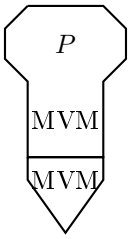
We can then compile it on a real machine, e.g. x86.



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

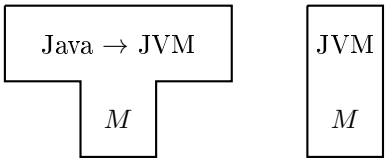
A compiler takes quite a long time to translate the source program to native machine code, but the subsequent execution is fast. On the other hand, an interpreter starts executing the source program immediately, but the execution is slow. We can combine the two and get an interpretive compiler. It translates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

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

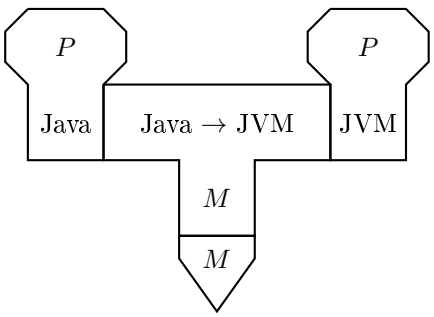
- the VM code is intermediate-level (lower-level than the source code but higher-level than native machine code);
- translation from the source language to VM code is easy and fast;
- the VM instructions have simpler formats (so can be analysed quickly by an interpreter).

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.

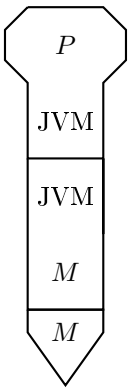
JDK comprises a Java $\rightarrow$ JVM compiler and a JVM interpreter. Once JDK has been installed on a real machine, we have the following components:



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



Later, the object program is interpreted.

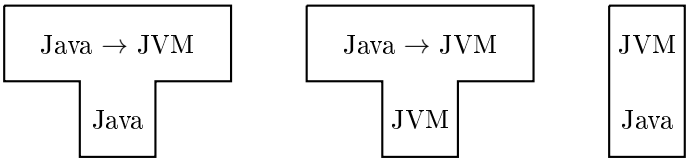


The bottom two tiles are called the Java Virtual Machine.

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.

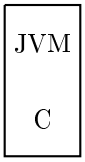
A program is portable if it can be made to run on different machines with minimal change. So, a program P written in Java is portable, but the same program in x86 is not. A compiler that generates native machine code is unportable- if it must be changed to target a different machine, its code generator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

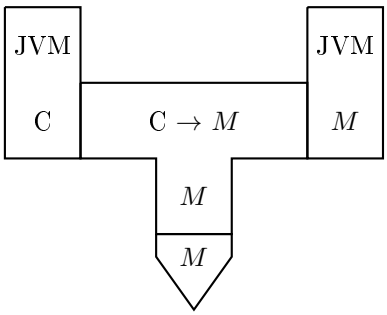


We can install this kit on machine M. But, we cannot run the JVM interpreter until we have a running Java compiler. Similarly, we cannot run the Java compiler until we have a running JVM interpreter.

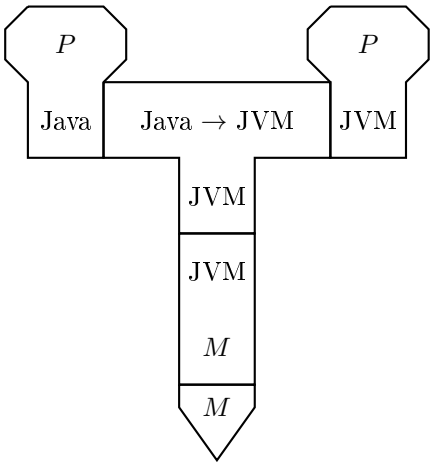
To progress, we first rewrite the JVM interpreter, e.g. in C.



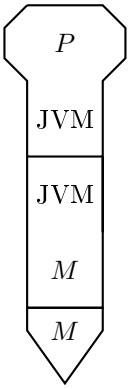
Then, we compile the JVM interpreter on M.



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

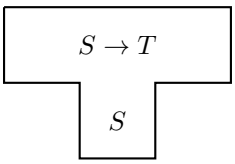


Using the result, we can interpret the program P using the JVM.



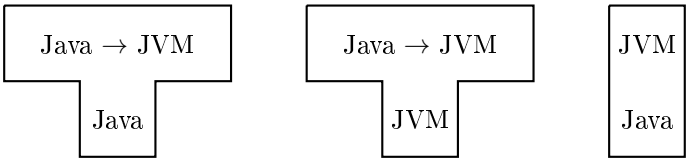
Here, the compilation stage is very slow, but it can be improved by bootstraping.

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

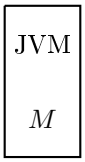


We can use this translator to translate itself. This process is called bootstraping. It is a useful tool for improving an existing compiler. This is because it makes the compilation process faster, which makes it generate faster object code. We can bootstrap a portable compiler to make a true compiler,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

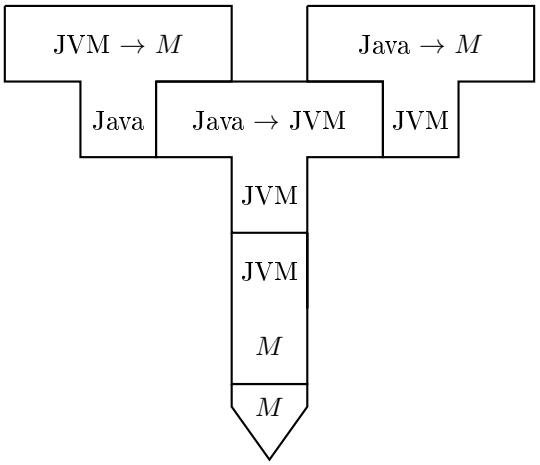
For example, consider the Java portable compiler kit.



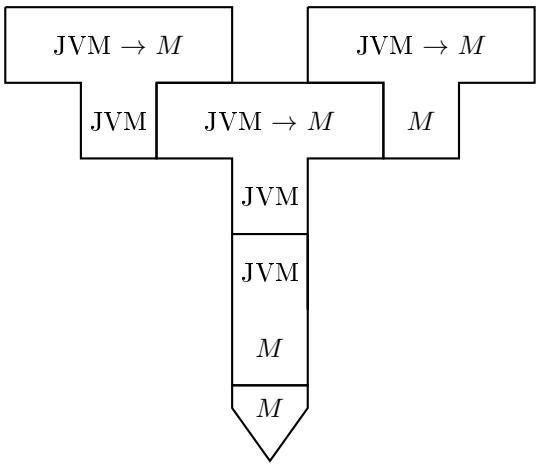
We also constructed a JVM interpreter in machine language M.



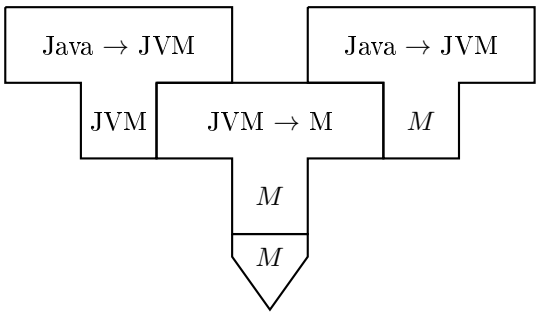
We can write a JVM → M translator in Java itself. We then compile it into JVM using the existing, slow compiler.



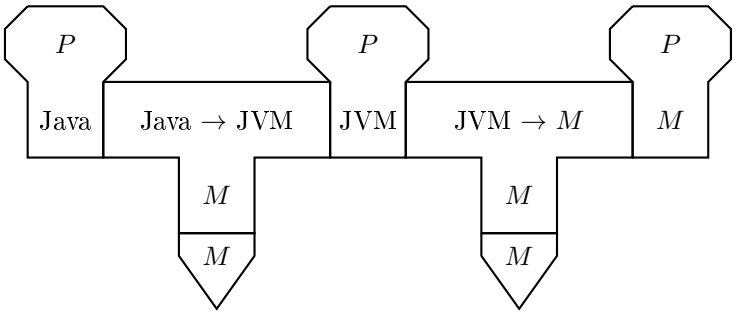
Next, we can use this JVM → M translator to translate itself.



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



Now, we have a 2-stage Java $\rightarrow M$ 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 words) 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.

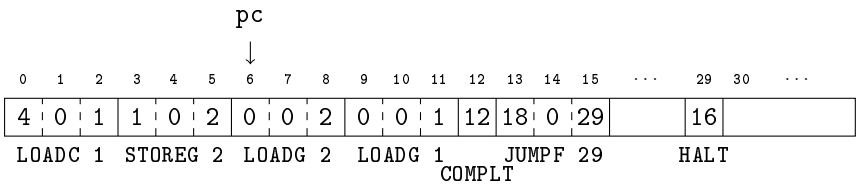


Figure 2.6: An illustration of the code source.

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.

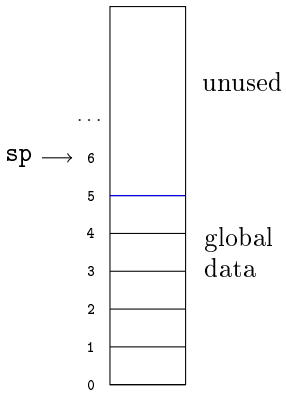


Figure 2.7: An illustration of the data store at the start.

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.

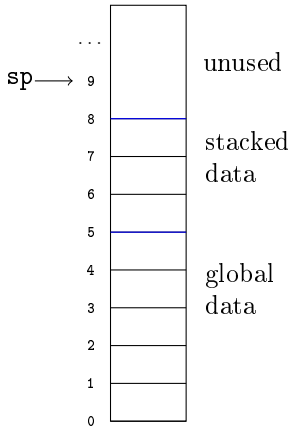


Figure 2.8: An illustration of the data store during execution.

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

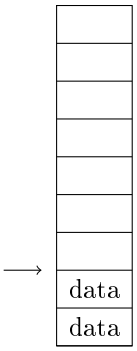
Table 2.1: The SVM Instruction Set

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:

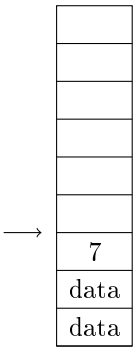
```
// load constant 7
LOADC 7
// load constant 3
LOADC 3
// add 7 and 3 = 10
ADD
// load constant 3
LOADC 5
```

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

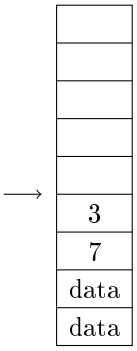
We will illustrate the execution with the data store. Assume that the initial state of the stack is the following.



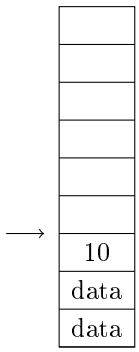
The arrow denotes the stack pointer `sp`. We first execute `LOADC 7`, which will add 7 to the stack.



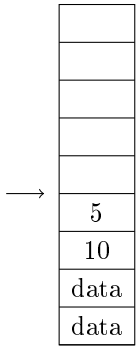
Next, we execute `LOADC 3`, which will add 3 to the stack.



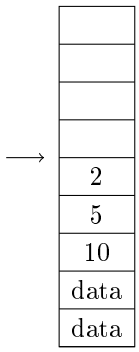
The next command is `ADD`. By its specification, we pop $w_2 = 3$ and $w_1 = 7$, and push $w_1 + w_2 = 10$.



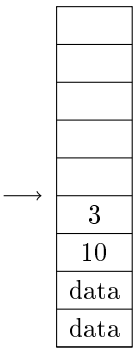
The next command is `LOADC 5`. So, we push 5 to the stack.



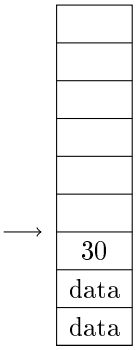
Now, we perform `LOADC 2`, when we push 2 to the stack.



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 () {
    // Initialise 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:

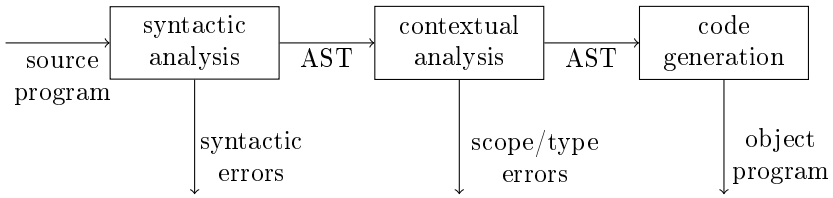


Figure 2.9: The data flow between the compilation phases.

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

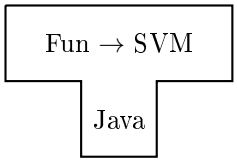
```
5           f = f * n
6           n = n - 1
7       .
8       return f
9   .
10
11  proc main ():
12      int num = read()
13      write(num)
14      write(fact(num))
15  .
```

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:

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

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

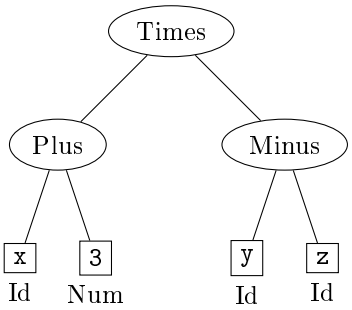
An abstract syntax tree (AST) is a convenient way to represent a source program's phrase structure. The structure of an AST is the following:

- 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:

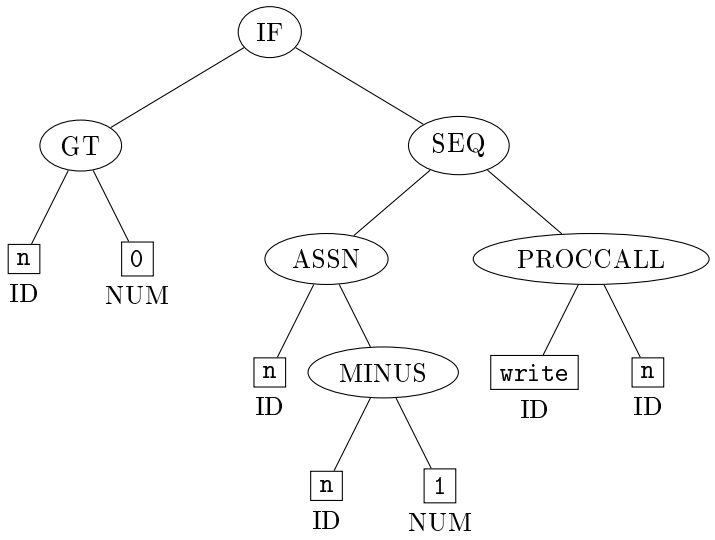


In the AST, we make no distinct between *expr*, *sec-expr*, etc. They are all treated as expressions. This is one of the ways an AST is compact.

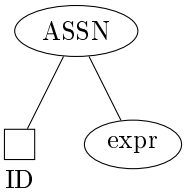
Now, consider the following program in Fun:

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

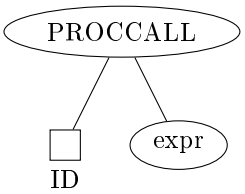
The corresponding AST is the following:



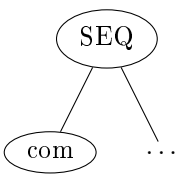
Now, we will look at all the Fun ASTs. For assignment, the AST is the following:



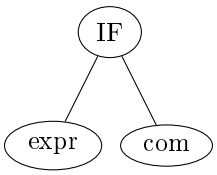
The assignment is given by an identifier, which is given the value of some expression. The procedure call AST is the following:



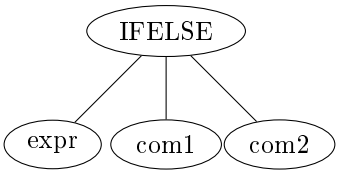
The expression is the same as assignment- we have an identifier for the procedure name, and its parameter is some expression. Next, the AST for a sequence of commands is the following:



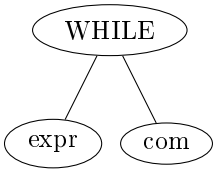
It is composed of a sequence of commands, and we must have at least one command. Now, the if AST is given below:



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:



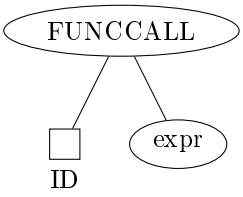
Here, we have an expression (that evaluates to true or false), and if it evaluates to true, we run the command com1, and otherwise, com2. The AST for a while loop is given below.



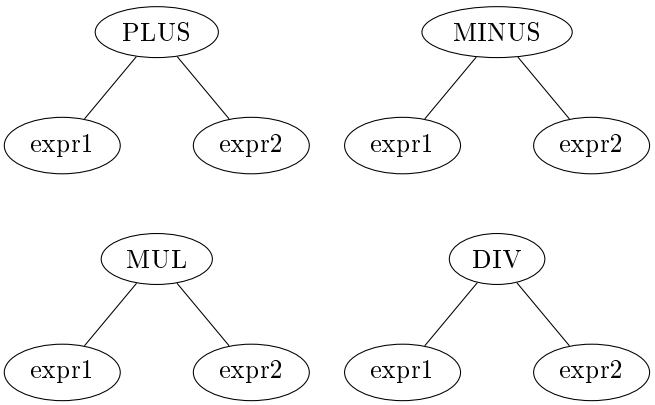
Here too, We have an expression (that evaluates to true or false), and if it evaluates to true, we run the command com1, and we rerun the expression and so on.

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

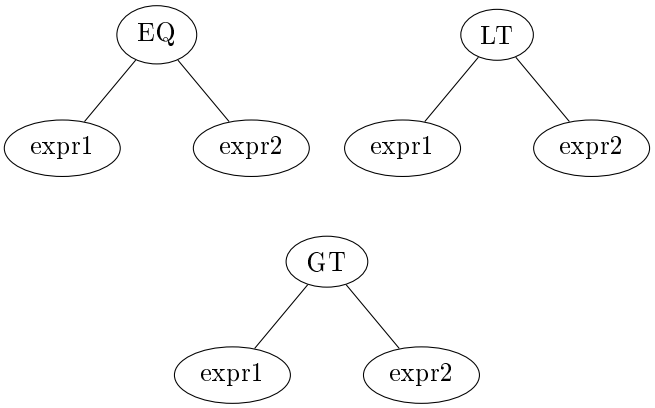
The literals are the boolean values `true` and `false`, numbers and identifiers. The AST for the function call expression is the following:



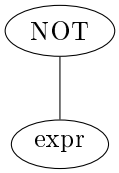
We have a name for the function, and the value of the parameter is some expression. Moreover, the arithmetic operations (plus, minus, times, div) are also expressions, with AST given below:



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:

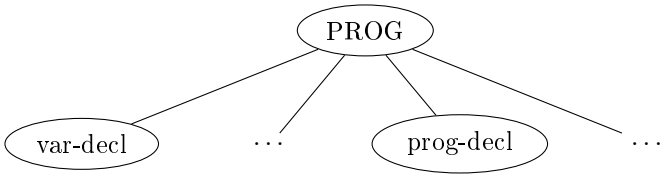


We compare two expressions (for equality, less than or greater than). Next, the AST for the not expression is given below.

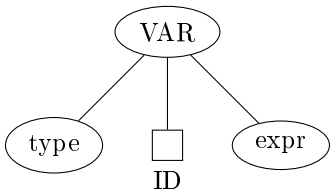


The expression evaluates to some boolean value, so the result will be the opposite boolean.

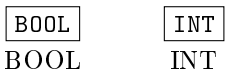
The AST for a Fun program is given below.



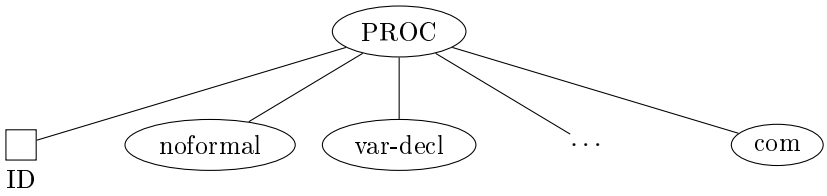
So, a program is a sequence of variable declarations and a sequence of program declarations. The AST for variable declarations is given below.



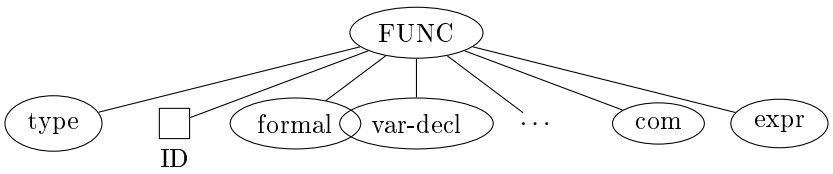
We specify the type of a variable, call it an identifier, and assign it the value of some expression. The ASTs for Fun types is given below:



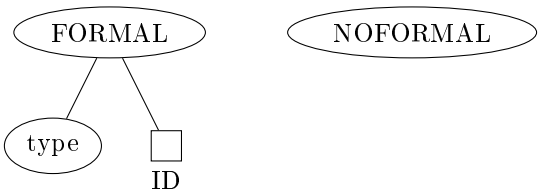
In Fun, there are only 2 types- bool and int. The AST for a procedure declaration is given below.



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.

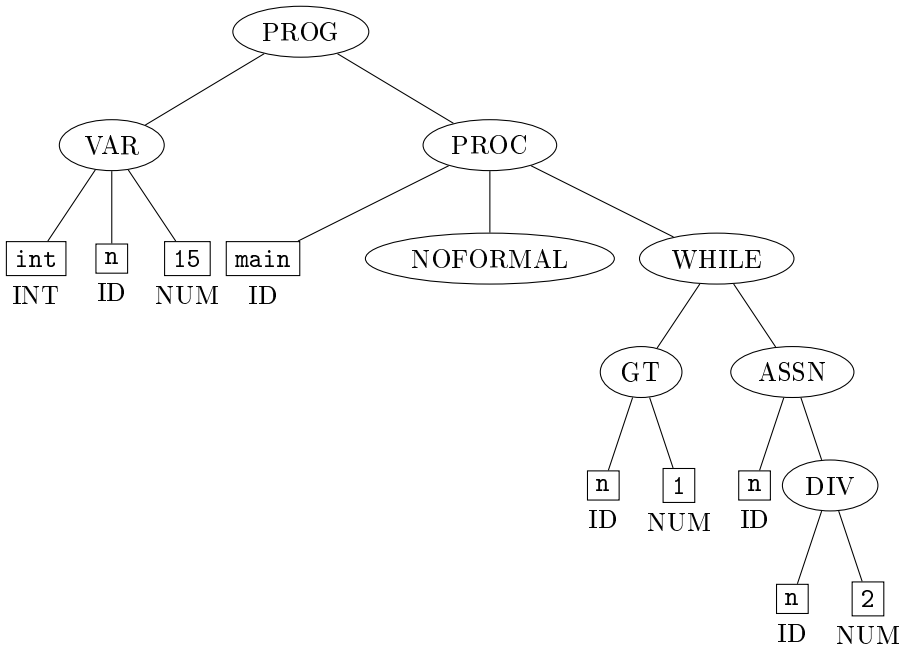


A formal parameter represents a single parameter (for functions) and no parameters (for procedures).

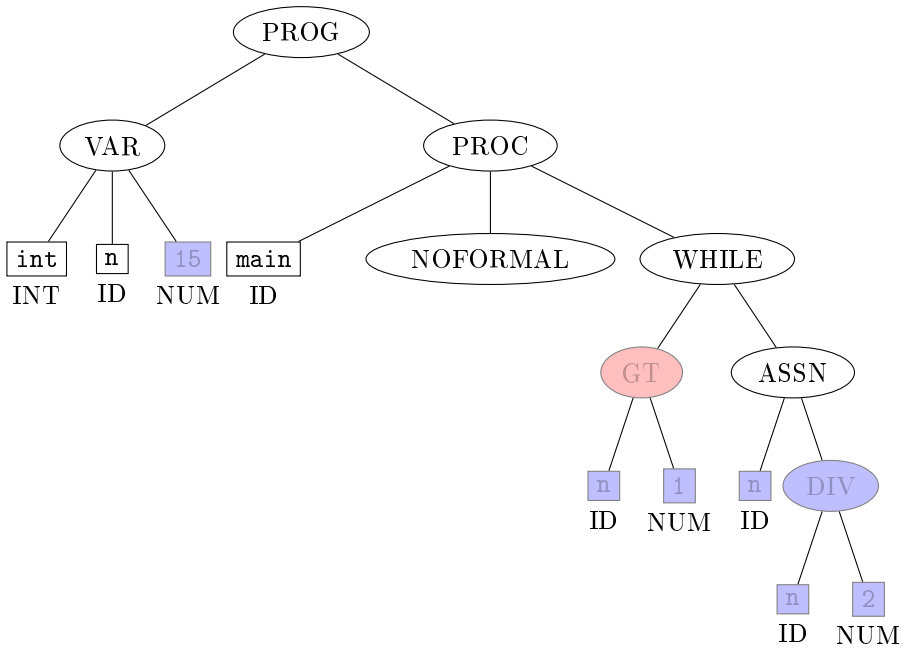
We will now create an AST from a program. So, consider the following program.

```
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
```