

THE COMPILATION PROCESS

3.1 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 subphases:

- 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.

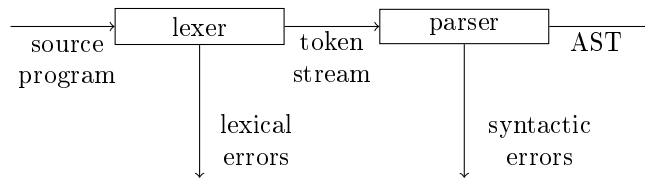


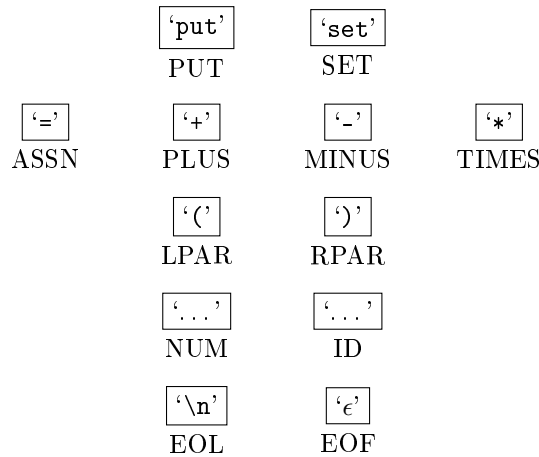
Figure 3.1: The data flow within the syntactic analysis phase.

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.

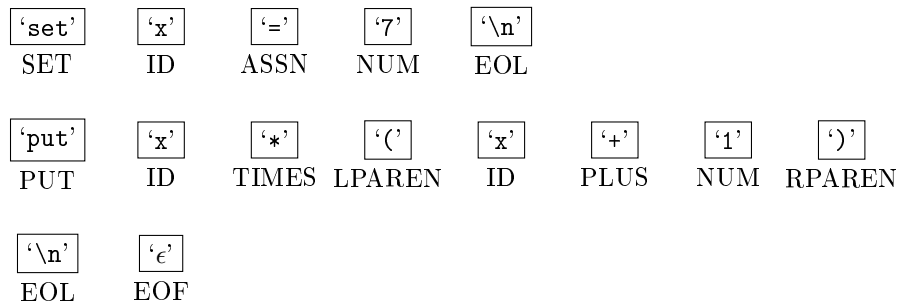
The following are tokens in Calc:



Now, assume we have the following Calc program:

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

We get the following token stream from the lexer:



The lexer converts 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. Given 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 `N()`, one for each non-terminal symbol `N` 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.

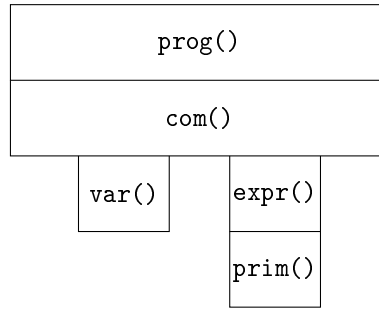
Now, we consider a Calc parser. The parsing methods for Calc are:

- `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.

<code>set</code>	<code>x</code>	<code>=</code>	<code>7</code>	<code>\n</code>
SET	ID	ASSN	NUM	EOL

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



set	x	=	7	\n
SET	ID	ASSN	NUM	EOL

In Calc, the EBNF production rule for command is:

$$\begin{aligned}
 com &= \text{'put' } expr \text{ } eol \\
 &| \text{'set' } var \text{'=' } expr \text{ } eol
 \end{aligned}$$

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

```

1 public void com() {
2     // if the next token is 'put'
3     if (tokenStream.next() == "put") {
4         match(PUT);
5         expr();
6         match(EOL);
7     } else if (tokenStream.next() == "set") {
8         match(SET);
9         var();
10        match(ASSN);
11        expr();
12        match(EOL);
13    } else {
14        // syntactic error
15    }
16 }
  
```

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.

```

1 public void prog() {
2     // while the next token matches a command token,
3     while (tokenStream.next().isCom()) {
4         com();
5     }
6     match(EOF);
7 }
  
```

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

```

1 public void N() {
2     match(t);
3 }
  
```

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

```
1 public void N() {
2     M();
3 }
```

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

```
1 public void N() {
2     // match RE1
3     // match RE2
4 }
```

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

```
1 public void N() {
2     // while the next token can be RE,
3     while (tokenStream.next().isRE()) {
4         // match RE
5     }
6 }
```

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.

```
1 public void N() {
2     // while the next token can be RE1,
3     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    else {
11        // syntactic error
12    }
13 }
```

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, even if 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.

```

1 grammar Calc;
2 prog
3     :    com* EOF
4     ;
5 com
6     :    PUT expr EOL          # put
7     |    SET var ASSN expr EOL # set
8     ;
9 expr
10    :    prim
11    |    (operator +=
12         (PLUS | MINUS | TIMES)
13         prim)*                # op
14    ;
15 prim
16    :    NUM                    # num
17    |    ID                     # id
18    |    LPAR expr RPAR        # parens
19    ;
20 var
21    :    ID
22    ;
23 PUT    : "put";
24 SET    : "set";
25 ASSN   : "=";
26 PLUS   : "+";
27 MINUS  : "-";
28 TIMES  : "*";
29 LPAR   : "(";
30 RPAR   : ")";
31 ID     : "a" .. "z";
32 NUM    : "0" .. "9"+;
33 EOL    : "\r"? "\n";
34 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.

```

1 grammar Fun;
2
3 prog
4     :   var_decl* proc_decl+ EOF           # prog
5     ;
6 var_decl
7     :   type ID ASSN expr                 # var
8     ;
9 type
10    :   BOOL                               # bool
11    |   INT
12    ;
13 com
14    :   ID ASSN expr                       # assn
15    |   WHILE expr COLON seq_com DOT       # while
16 seq_com
17    :   com*                               # seq
18    ;
19 expr
20    :   e1=sec_expr ...
21 sec_expr
22    :   e1=prim_expr
23        ( op = (PLUS | MINUS | TIMES | DIV)
24          e2 = sec_expr
25        ) ?
26 prim_expr
27    :   NUM                                # num
28    |   ID                                # id
29    |   LPAR expr RPAR                     # parens
30    |   ...

```

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.

3.2 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.

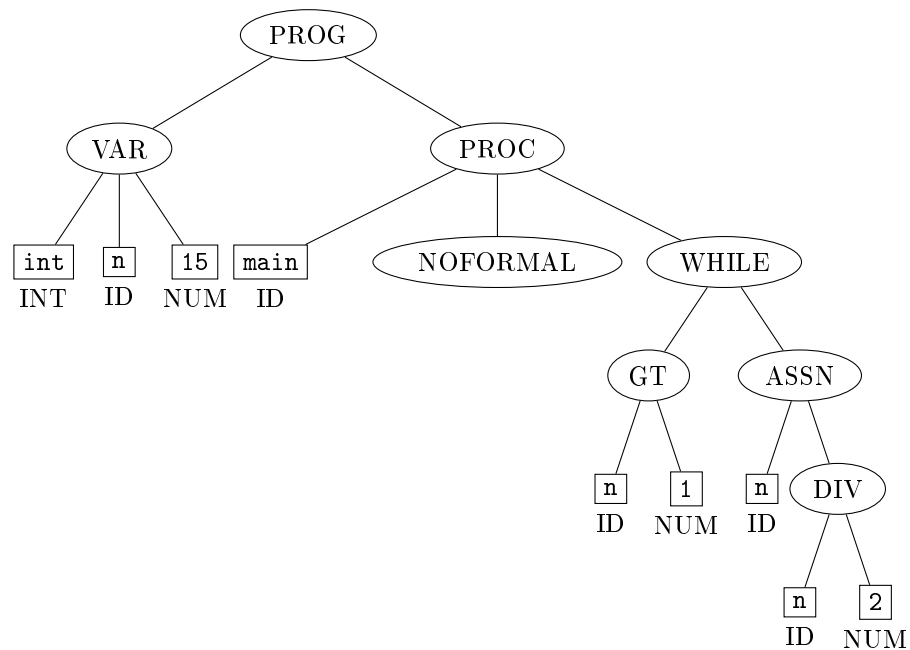
For example, consider the following source program in Fun:

```

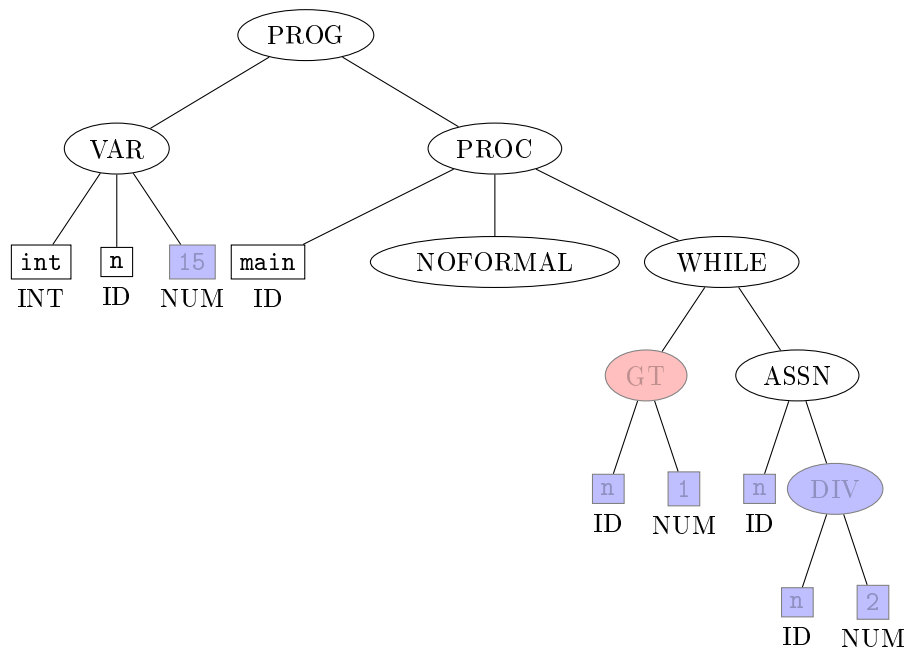
1 int n = 15
2 # div program
3 proc main ():
4     while n > 1:
5         n = n/2
6     .
7 .

```

After syntactic analysis, we have 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	VOID → VOID

Table 3.1: 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

$$id = expr$$

In this case, we lookup the identifier in the type table 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

$$expr_1 \text{ '>' } expr_2$$

We walk $expr_1$ and check that its type is INT, and do the same thing for $expr_2$. We then infer that the type of the whole expression is BOOL. An assignment command is of the form

$$id \text{ '=' } expr$$

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

$$\text{'if' } expr \text{ com}$$

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.

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

```
1 Type visitNum(FunParser.NumContext ctx) {
2     return Type.INT;
3 }
```

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

```
1 Type visitID(FunParser.IdContext ctx) {
2     return retrieve(ctx.ID().getText(), ctx);
3 }
```

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

$$e_1 = \text{sec_expr} (op = ('=' | '<' | '>') \ e_2 = \text{sec_expr})?$$

So, the `visitExpr` method is the following.

```

1 Type visitExpr(FunParser.ExprContext ctx) {
2     Type t1 = visit(ctx.e1);
3     if (ctx.e2 != null) {
4         Type t2 = visit(ctx.e2);
5         return checkBinary(COMPTYPE, t1, t2, ctx);
6     }
7     else {
8         return t1;
9     }
10 }

```

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 $\text{INT} \times \text{INT} \rightarrow \text{BOOL}$.

In Fun, an assignment command is of the form

$$ID \text{ '=' } \text{expr}$$

So, the `visitAssn` method is the following.

```

1 Type visitAssn(FunParser.AssnContext ctx) {
2     Type tvar = retrieve(ctx.ID().getText(), ctx);
3     Type t = visit(ctx.expr());
4     checkType(tvar, t, ctx);
5     return null;
6 }

```

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

$$\text{'if' expr ':' } c_1 = \text{seq_com('.' | 'else:' } c_2 = \text{seq_com '.')}$$

So, the `visitIf` method is the following.

```

1 Type visitIf(FunParser.IfContext ctx) {
2     Type t = visit(ctx.expr());
3     visit(ctx.c1);
4     if (ctx.c2 != null) {
5         visit(ctx.c2);
6     }
7     checkType(Type.BOOL, t, ctx);
8     return null;
9 }

```

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.

```
1 Type visitSeq(FunParser.SeqContext ctx) {
2     visitChildren(ctx);
3     return null;
4 }
```

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.

```
1 Type visitVar(FunParser.VarContext ctx) {
2     Type t1 = visit(ctx.type());
3     Type t2 = visit(ctx.expr());
4     define(ctx.ID().getText(), t1, ctx);
5     checkType(t1, t2, ctx);
6     return null;
7 }
```

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.

```
1 Type visitProg(FunParser.ProgContext ctx) {
2     predefine();
3     visitChildren(ctx);
4     Type tmain = retrieve("main", ctx);
5     checkType(MAINTYPE, tmain, ctx);
6     return null;
7 }
```

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 → VOID` (called `MAINTYPE`).

The driver program visits the tree, as given below.

```
1 public static void main(String[] args) {
2     ..
3     // Syntactic analysis
4     ParseTree tree = parser.program();
5     // Contextual analysis
6     FunCheckerVisitor checker = new FunCheckVisitor(tokens);
7     checker.visit(tree);
8 }
```

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 $\text{VOID} \rightarrow T'$. We can represent procedures in a similar way, e.g. $T \rightarrow \text{VOID}$ or $\text{VOID} \rightarrow \text{VOID}$. We also use mapping types for operators, e.g. $+$, $-$, $*$ and \backslash are of type $(\text{INT} \times \text{INT}) \rightarrow \text{INT}$; $=$, $<$ and $>$ are of type $(\text{INT} \times \text{INT}) \rightarrow \text{BOOL}$; and `not` is of type $\text{BOOL} \rightarrow \text{BOOL}$.

The class `Type` is of the following format.

```

1 public abstract class Type {
2     public abstract boolean equiv(Type t);
3
4     public static class Primitive extends Type {
5         ...
6     }
7
8     public static class Pair extends Type {
9         ...
10    }
11
12    public static class Mapping extends Type {
13        ...
14    }
15 }
```

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

```

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

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

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

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

```

1 Type proctype = new Type.Mapping(Type.INT, Type.VOID);
2 Type optype = new Type.Mapping(
3     new Type.Pair(Type.INT, Type.INT), Type.BOOL
4 );
```

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

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 \rightarrow VOID
local	x	INT

At line 9, the type table is the following.

global	x	INT
global	main	VOID \rightarrow VOID
global	p	BOOL \rightarrow 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>();

```



```
3 }
4
5 public void exitLocalScope() {
6     locals = null;
7 }
```

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.

```
1 public void put (String id, A attr) {
2     if (locals != null) {
3         locals.put(id, attr);
4     } else {
5         globals.put(id, attr);
6     }
7 }
8
9 public A get(String id) {
10     if (locals != null && locals.containsKey(id)) {
11         return locals.get(id);
12     } else {
13         return globals.get(id);
14     }
15 }
```

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.

3.3 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.

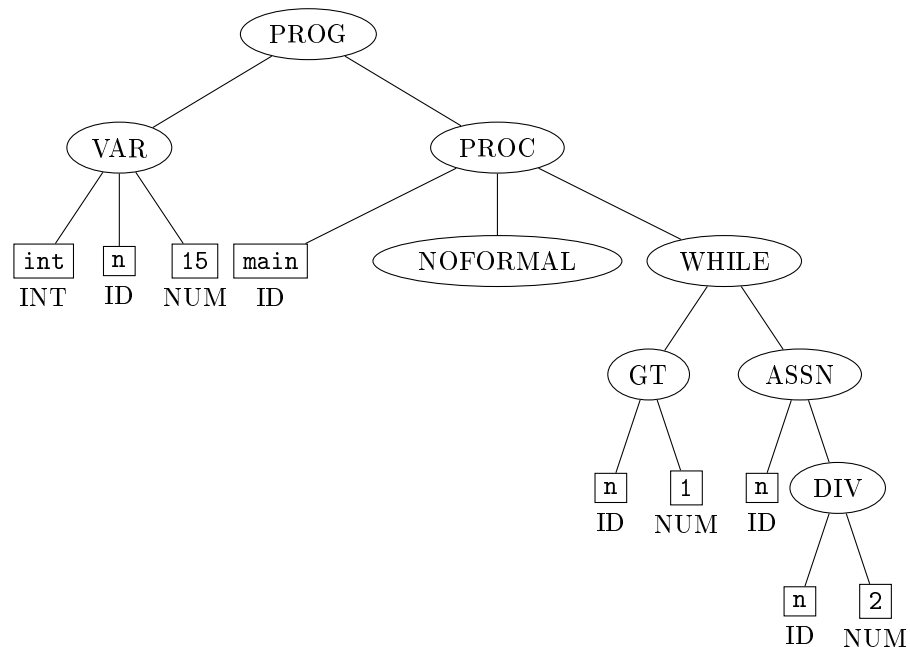
The following is a program in Fun.

```

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 template

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 and 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

$$expr_1 + expr_2.$$

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:

```
1 // load variable m
2 LOADG 3
3 // load constant 7
4 LOADC 7
5 // load variable n
6 LOADG 4
7 // compute 7 * n
8 MUL
9 // compute m + (7 * n)
10 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

$$ID = expr$$

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:

```
1 // load the variable n
2 LOADG 4
3 // load the constant 9
4 LOADC 9
5 // compute n - 9
6 SUB
7 // store the result as variable m
8 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:

```

1 // load variable m
2 LOADG 3
3 // load variable n
4 LOADG 4
5 // compute m > n
6 COMPGT
7 // if false, jump to the end
8 JUMPF *
9 // load variable m
10 LOADG 4
11 // store as variable n
12 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:

```
1 // creates an instance of SVM. The code generator will emit
2 // instructions directly into its code store
3 SVM obj = new SVM();
4 int globalvaraddr = 0;
5 int localvaraddr = 0;
6 int currentLocale = Address.GLOBAL;
7
8 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.

```
1 void visitNum(FunParser.NumContext ctx) {
2     int value = Integer.parseInt(ctx.NUM().getText());
3     obj.emit12(SVM.LOADC, value);
4 }
```

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

The visit identifier method is given below.

```
1 void visitId(FunParser.IDContext ctx) {
2     String id = ctx.ID().getText();
3     Address varaddr = addrTable.get(id);
4     switch (varaddr.locale) {
5         case Address.GLOBAL:
6             obj.emit12(SVM.LOAG, varaddr.offset);
7             break;
8         case Address.LOCAL:
9             obj.emit12(SVM.LOAL, varaddr.offset);
10            break;
11    }
12 }
```

The visit expression method is given below.

```
1 void visitExpr(FunParser.ExprContext ctx) {
2     // generate code to evaluate e1
3     visit(ctx.e1);
4     // if e2 exists, then generate code to evaluate e2
5     if (ctx.e2 != null) {
6         visit(ctx.e2);
7     }
```

```

8      // generate an instruction for each operator
9      switch (ctx.op.getType()) {
10         case FunParser.EQ:
11             obj.emit1(SVM.CMPEQ);
12             break;
13         case FunParser.LT:
14             obj.emit1(SVM.CMPLT);
15             break;
16         case FunParser.GT:
17             obj.emit1(SVM.CMPGT);
18             break;
19     }
20 }
21 }

```

The command `emit1` means just 1 opcode.

The visit assignment method is given below.

```

1 void visitAssn(FunParser.AssnContext ctx) {
2     // generate code to evaluate expr
3     visit(ctx.expr());
4     String id = ctx.ID().getText();
5
6     // find the address of the variable
7     // this always succeeds, because we added the address
8     // during contextual analysis
9     Address varaddr = addrTable.get(id);
10    switch (varaddr.locale) {
11        case Address.GLOBAL:
12            obj.emit12(SVM.STOREG, varaddr.offset);
13            break;
14        case Address.LOCAL:
15            obj.emit12(SVM.STOREL, varaddr.offset);
16            break;
17    }
18 }
19 }

```

The visit if method is given below.

```

1 void visitIf(FunParser.IfContext ctx) {
2     visit(ctx.expr());
3     int condaddr = obj.currentOffset();
4     // This has to be patched later
5     obj.emit12(SVM.JUMPF, 0);
6     // IF without ELSE
7     if (ctx.c2 == null) {
8         visit(ctx.c1);
9         int exitaddr = obj.currentOffset();
10        obj.patch12(condaddr, exitaddr);
11    } else {
12        visit(ctx.c1);
13        int jumpaddr = obj.currentOffset();
14        obj.emit12(SVM.JUMP, 0);
15        int elseaddr = obj.currentOffset();
16        obj.patch12(condaddr, elseaddr);
17        visit(ctx.c2);
18        int exitaddr = obj.currentOffset();
19        obj.patch12(jumpaddr, exitaddr);
20    }
21 }

```

The visit var method is given below.

```

1 void visitVar(FunParser.VarContext ctx) {
2     visit(ctx.expr());
3     String id = ctx.ID().getText();
4     switch (currentLocale) {
5         // adding the variable to the address table always
6         // succeeds because we have done contextual analysis
7         case Address.LOCAL:
8             addrTable.put(id, new Address(localvaraddr++,
9                                     Address.LOCAL));
10            break;
11        case Address.GLOBAL:
12            addrTable.put(id, new Address(globalvaraddr++,
13                                    Address.GLOBAL));
14            break;
15    }
16 }

```

Storage organisation

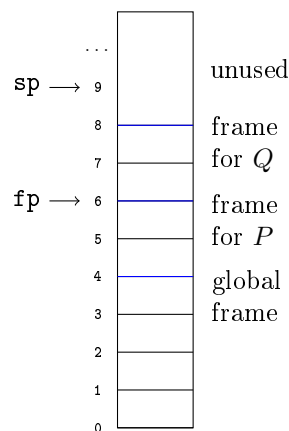
Each variable occupies storage space throughout its lifetime. This storage must be allocated at the start of the variable's lifetime and deallocated at the end of the variable's lifetime. For a statically typed PL, every variable's type is known to the compiler. Moreover, we assume that all variables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

```

1 void visitProc(FunParser.ProcContext ctx) {
2     String id = ctx.ID().getText();
3     Address procaddr = new Address(obj.currentOffset(),
4         Address.CODE);
5     addrTable.put(id, procaddr);
6     addrTable.enterLocalScope();
7     currentLocale = Address.LOCAL;
8     localvaraddr = 2;
9
10    FunParser.Formal_declContext fd = ctx.formal_decl();
11    if (fd != null)
12        visit(fd);
13    List<FunParser.Var_declContext> var_decl = ctx.var_decl();
14    for (FunParser.Var_declContext vd : var_decl) {
15        visit(vd);
16    }
17    visit(ctx.seq_com());
18    // 0 because there is no result
19    obj.emit11(SVM.RETURN, 0);
20    addrTable.exitLocalScope();
21    currentLocale = Address.GLOBAL;
22 }

```

The visit formal method is given below.

```

1 void visitFormal(FunParser.FormalContext ctx) {
2     FunParser.TypeContext tc = ctx.type();
3     if (tc != null) {
4         String id = ctx.ID().getText();
5         // A parameter is like a local variable
6         addrTable.put(id, new Address(localvaraddr++,
7             Address.LOCAL));
8         // copy arguments into the stack frame
9         obj.emit11(SVM.COPYARG, 1);
10    }
11 }

```

Finally, the visit program method is given below.

```

1 void visitProg(FunParser.ProgContext ctx) {
2     // add read and write to the address table
3     predefine();
4     List<FunParser.Var_declContext> var_decl = ctx.var_decl();
5     for (FunParser.Var_declContext vd : var_decl) {
6         visit(vd);
7     }
8     int calladdr = obj.currentOffset();
9     // call the main method (will patch)
10    obj.emit12(SVM.CALL, 0);
11    obj.emit1(SVM.HALT);
12    List<FunParser.Proc_declContext> proc_decl = ctx.proc_decl();
13    for (FunParser.Proc_declContext pd : proc_decl) {
14        visit(pd);
15    }
16    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 }
```