3

A software architecture for translators

This chapter covers

- The source, implementation, target, and command languages
- A three-tiered software architecture for programming language translators
- Translator control flow and data flow
- Translator operations in three passes

Translators are large, complex programs! In the end, you will have written thousands of lines of code spread across many interacting classes. Unless you employ good software architecture principles to manage the complexity, a translator project quickly becomes difficult to manage, test, and extend — basically, a nearly impossible task.

The software engineering principles we emphasize include:

- A sustainable software architecture containing components with well-defined roles.
- Well-designed control flow and data flow during the translation process.
- Use of reliable software development tools. A key tool will be ANTLR 4, which will be covered starting in the next chapter.
- Incremental code development where each development cycle produces testable results that the next cycle can build upon.

This chapter emphasizes the first two principles. You'll see evidence of the other two in subsequent chapters.

3.1 The implementation, source, and target languages

We will use the word *language* a lot in this book. Table 3.1 should alleviate any confusion about languages.

Table 3.1: The implementation, source, and target languages

Language	Relationship to translation
Implementation	Translator components are written in the implementation language. You will write component code in Java, the implementation language. The ANTLR 4 tool will generate code in Java for the components that we won't have to write.
Source	Source programs to be translated are written in the source language. This is the language you're designing or modifying, and for which you're developing the translators. Therefore, you must create a grammar for the source language. The example source language in this book is a subset of Pascal. You are encouraged to design and build either a new language or to modify an existing language.
Target	A translator transforms source programs written in the source language to the target language. Examples: A program converter transforms a Pascal source program to an equivalent program in a high-level target language such as Java, C++, or Python. The compiler you'll write in this book will transform a Pascal source program to Jasmin assembly language for the Java Virtual Machine (JVM), a low-level target language that's human-readable. Most commercial compilers directly generate machine code as its target language.
Command	You will develop an interactive debugger for the interpreter. While a source program is executing under the interpreter's control, you can interact with the program using the debugger's command language to set breakpoints, single-step statement execution, examine variable values, etc. Chapters 14 and 15 will cover the debugger and its command language.

3.2 A software architecture

Figure 3.1 shows how to partition a translator's code into three major tiers: the front end, the intermediate tier, and the back end. This is a proven "divide and conquer" strategy to manage complexity. Each tier contains components with well-defined roles.

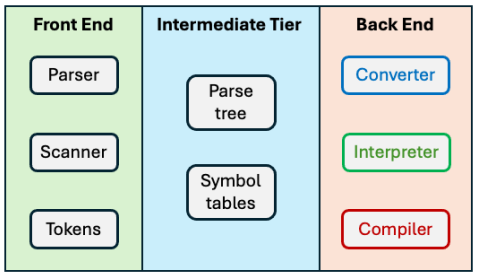

Figure 3.1: The three-tier architecture of a translator and the key components in each tier.

Table 3.2 presents a brief overview of the role of each component. These roles will become clearer and more refined as you develop the translators. The next chapter will describe how the ANTLR 4 tool generates key components of the front end and in the intermediate tiers. You must write the remaining components, but ANTLR 4's generated code provides a framework into which you can fit your components.

Table 3.2: Key translator components and their roles.

Component	Tier	Role
Parser	Front end	The parser directs the frontend operations of a translator. It incorporates the grammar of the source language and performs syntax checking of a source program written in the language. The parser repeatedly requests tokens from the scanner. As it analyzes a source program, it builds a parse tree in the intermediate tier. (See the "Parse tree" entry below.)
Scanner	Front end	The scanner incorporates the token definitions in the source language's grammar. Upon each request by the parser, the scanner recognizes the next token (the token that immediately follows the one that the scanner last recognized) in the source program and returns a token object to the parser.
Tokens	Front end	Token objects represent the tokens recognized by the scanner in the source program. The token definitions in the source language's grammar determine how to create each token object.
Parse tree	Intermediate	The parser builds the parse tree as it analyzes source program. This tree data structure represents the statements and expressions of the source program. Representing the source program as a parse tree makes it much more efficient for the backend components to access information about the statements and expressions without needing to reanalyze the source code.
Symbol tables	Intermediate	Symbol tables contain important information about names in the parse tree. Information about a name includes its datatype and whether it's the name of a variable or a function.
Converter	Back end	A program converter transforms a source program from the source programming language to the target high-level language.
Interpreter	Back end	An interpreter executes a source program under its control, without first transforming the program to a target language. The interpreter you will develop will include an integrated debugger that enables a programmer to interact with a running source program.
Compiler	Back end	A compiler compiles a source program to low-level target code, such as assembly language code or machine language code.

3.2.1 Control flow

The components in table 3.2 have working relationships with each other. Figure 3.2 shows the control flow of the components during the translation of a source program.

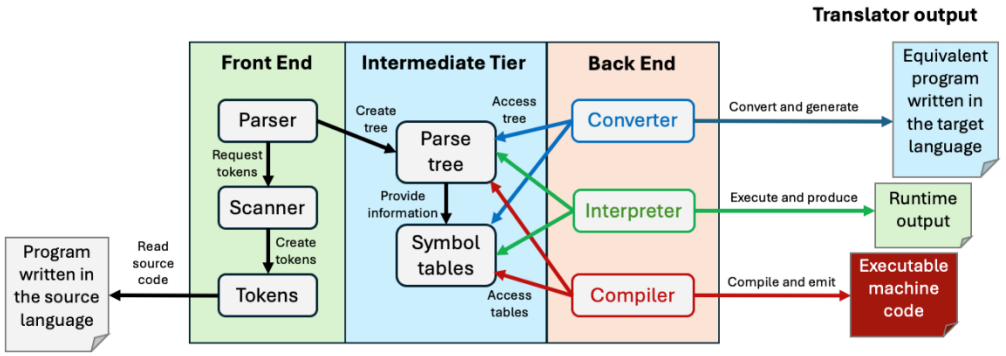

Figure 3.2: Control flow during the translation of a source program. The arrows show how the translator components make requests on each other, from the requestor to the requestee.

The figure illustrates *dependency and invocation* — which component actively calls upon another. The arrows point from the requestor to the requestee. The parser drives the front end and creates the parse tree in the intermediate tier. The backend components only make requests on the intermediate tier components and never on the frontend components. Also, the backend components are independent from each other and never make requests on each other.

We will use some common terms to describe the actions in the front end: the scanner *scans* the source program text by recognizing consecutive tokens (according to the grammar's token definitions) in the source program text and creating token objects for the parser; the parser *parses* a source program by analyzing (according to the grammar rules) the stream of token objects provided by the scanner.

3.2.2 Data flow

We now shift our perspective from control flow to what moves where — the various data flows through the components during a translation. In figure 3.3, the arrows track the *lifecycle of translation data*. The source program's text transforms into token objects, token objects assemble into tree nodes, and tree nodes deposit name information into the symbol tables. Then parse tree data and symbol table data are consumed by each backend component to generate translator output.

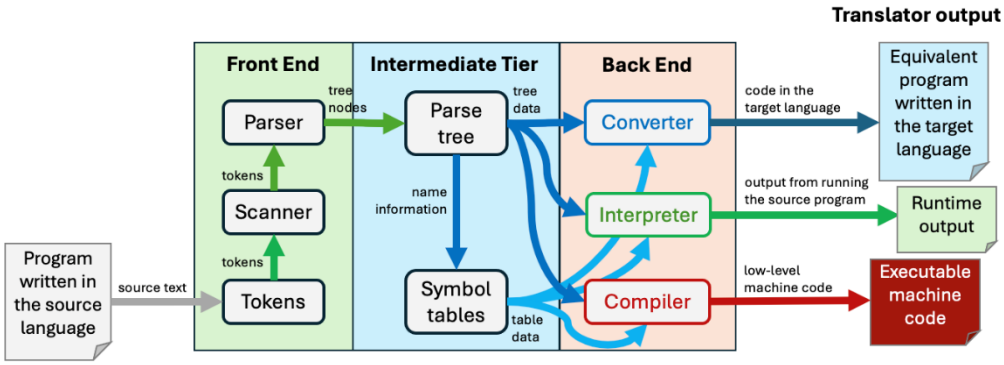

Figure 3.3: Data flow during the translation of a source program.

In addition to the text of the token, a token object also contains important information about the token, including what kind it is (name, number, reserved word, or special symbol) and its location (line number of the source program file and position within the line).

3.2.3 Parse tree

The parse tree is the central representation of the source program that's being translated. Once the parser creates the tree, subsequent translator operations use the tree rather than reanalyze the original source text.

You'll see in the next chapter how the parser incorporates the syntax rules and token definitions of the source language's grammar. Because it "knows" the grammar, the parser performs syntax checking to ensure that the stream of tokens it's receiving from the scanner constitutes valid statements and expressions. If the statements and expressions are syntactically correct, the parser creates a parse tree in the intermediate tier.

The parser analyzes the source program and builds the parse tree once per translation. By isolating this syntax-checking phase at the start, subsequent translator operations can safely and efficiently access the source program's statements and expressions from the tree.

For example, after parsing the following Simple statements, the parser can generate the parse tree represented by figure 3.4.

```
n := i + j - 7;
k := n;
writeln(k);
```

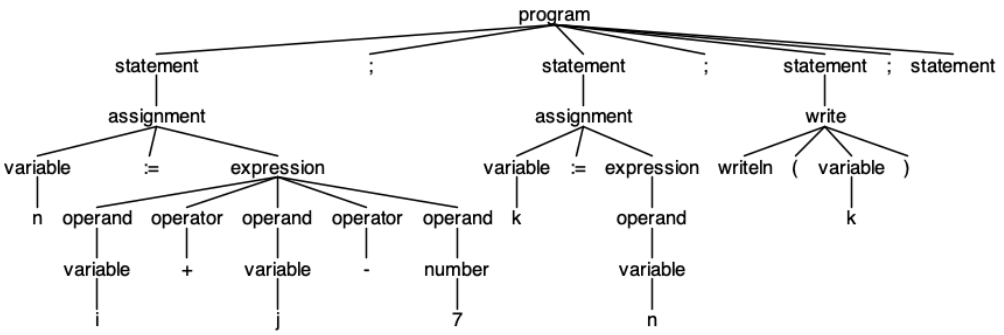

Figure 3.4: A parse tree generated from the three Simple statements.

This parse tree is built from the three Simple source statements, and its structure is based on the Simple grammar. The tree representation is generated by an ANTLR 4 utility program that you'll learn to use in the next chapter. Each token from the source code is represented by a node in the parse tree. The nodes are organized hierarchically according to the grammar. The highlighted line of grammar file `Simple.g4` below states that a program consists of statements separated by semicolons.

Listing 3.1: ANTLR 4 grammar file `Simple.g4`

```
grammar Simple ;

program : statement ( ';' statement )* ;

statement : assignment
          | write
          | /* empty */
          ;

assignment : variable ':=' expression ;
expression : operand ( operator operand )* ;
operand    : variable | number ;
operator   : '+' | '-' ;

write : WRITELN '(' variable ')' ;

variable : LETTER ;
number   : DIGIT ;

WRITELN : 'writeln' ;
LETTER  : [a-z] ;
DIGIT   : [0-9] ;

WHITESPACE : [ \n\r\t]+ -> skip ;
```

Therefore, in the parse tree, the children of the tree's program node are nodes that represent the statements and the semicolons.

3.2.4 Symbol tables

Symbol tables in the intermediate tier collect information about name nodes of the parse tree. In the parse tree in figure 3.4, these are the nodes representing variables. By keeping information about each name in one place rather than repeated among multiple tree nodes, a symbol table further simplifies translator operations.

A symbol table entry for a name includes a text string that is the name itself, its datatype, and whether it's the name of a variable or a function. As your source language becomes more sophisticated, you'll add more information to the symbol tables.

For now, you'll need only one symbol table in the intermediate tier. You'll learn later that a separate symbol table is required for the program as a whole and for each function in the source language.

The next couple of chapters will discuss parse trees and symbol tables and how they're created in greater detail.

3.2.5 Backend components

The program converter, interpreter, and compiler translator components reside in the backend tier. These components work independently from each other, and they interact only with the components of the intermediate tier. Each performs its translation operations by accessing the parse tree and the symbol tables. Because the parser in the front end has already done the analysis, the backend components can assume the source program is syntactically correct. By "walking" the parse tree (accessing its nodes in a specified order) and fetching information from the symbol tables, a backend component obtains the information that it needs about the source program's statements and expressions.

3.2.6 Three-pass translators

A translator's control and data flows are generally considered to happen in multiple phases, called *passes*. The passes represent a high-level, bird's-eye view of a translator's control and data flow. Organizing the operations of the translator's components into passes is yet another way to manage the software's complexity.

The translators that you will develop will each perform its operations in three passes, as shown in table 3.3.

Table 3.3: Translator operations organized into three passes.

Pass	Operations
1	In the front end, the parser requests tokens from the scanner, performs syntax checking, and builds the parse tree in the intermediate tier. This pass is often called *syntax analysis*.
2	Pass 2 performs operations based on the meanings of the parse tree nodes. Therefore, this pass is often called *semantic analysis*. The analysis includes populating symbol tables with information about names contained in the tree nodes and performing type checking to verify datatype compatibilities of the operands in expressions.
3	The backend components independently operate by accessing only the parse tree and symbol tables in the intermediate tier.

By coincidence, you will have a three-tier architecture and three-pass translators — these numbers aren't related. Compilers for sophisticated source languages like Java and C++ can have different numbers of tiers and passes. You can decide later to have more tiers or passes.

3.3 Benefits of this software architecture

This software architecture effectively manages the complexity of developing a translator. The architecture organizes the components into three tiers, with the intermediate tier being the interface between the frontend and backend tiers. The components have well-defined roles and interactions with each other. There is a clear dataflow from the source program through to the translator output.

The intermediate tier components are shared by the backend components. The backend program converter, the interpreter, and the compiler all efficiently access the same parse tree and symbol tables in the intermediate tier. This reduces code duplication and ensures that all three backend components "see" the same source program.

In this architecture, the frontend components and the backend components never interact with each other. Those components only interact with the intermediate tier components. This enables a translator development team to first design the formats of the parse tree and the symbol tables. Then team members can work simultaneously: one group can focus on the frontend components while another group develops the backend components.

Because of the separation between the frontend components and the backend components, the front end is independent of the translator output. By developing backend components for each platform, we can create a multi-platform translator for a programming language that can produce results for different target machines. On the other hand, for a given platform, we can create a multi-language translator by designing a grammar for each language.

The organization of the translator components and the loose coupling among them give the architecture greater maintainability and flexibility for extension.

GCC, the GNU compiler collection

The highly popular GCC compilers are multi-language and multi-platform. It has front ends for many high-level languages, including Ada, C, C++, COBOL, Fortran, Go, and Rust.

These compilers run on and generate code for various platforms, including Linux, MacOS, and Windows. These compilers share common code in their intermediate tiers. GCC's design is a real-world example of the software architecture that this chapter describes: independent frontend and backend tiers connected via a shared intermediate tier.

3.4 Summary

- We need good software engineering principles to manage the complexity of developing translator software.
- The source, implementation, target, and command languages have different purposes during a translation.
- A three-tier software architecture for a programming language translator consists of a front end, an intermediate tier, and a back end.
- Each tier contains components with well-defined roles, control flow, and data flow.
- The parser in the front end builds a parse tree in the intermediate tier as a representation of the source program. A parse tree enables the backend components to access information efficiently about the source program that's being translated.
- The symbol tables in the intermediate tier contain information from parse tree nodes that represent names in the source program.
- The backend components share and access only the parse tree and symbol tables in the intermediate tier.
- We can organize the operations of the translator into three passes.
- The software architecture promoted in this book can support developing multi-language and multi-platform translators.

In the next few chapters, you'll start to implement the software architecture and put it to work. ANTLR 4 will save you time and effort by automatically generating the parser, scanner, token classes, and parse tree infrastructure in the front end and in the intermediate tier.