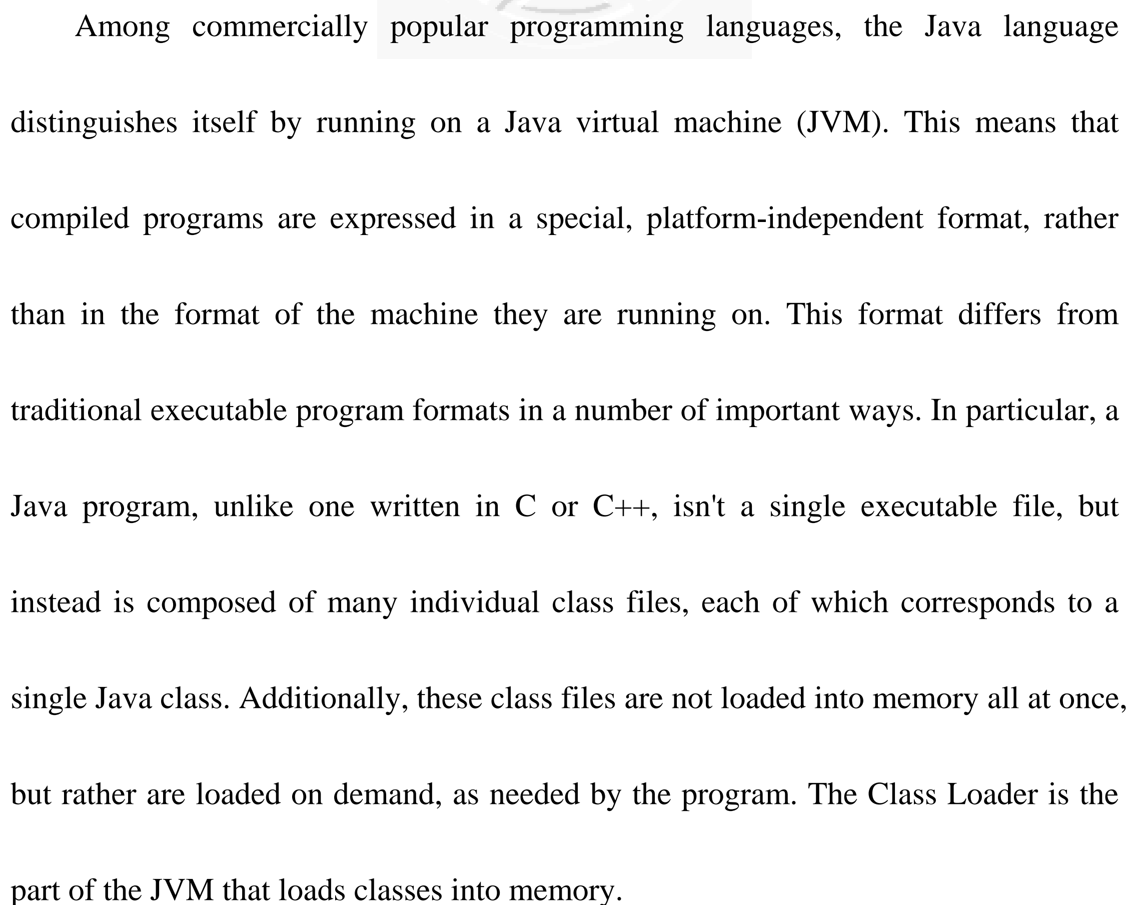
Chapter 2 Background

2.1 Java basics

2.1.1 Class loader



Among commercially popular programming languages, the Java language distinguishes itself by running on a Java virtual machine (JVM). This means that compiled programs are expressed in a special, platform-independent format, rather than in the format of the machine they are running on. This format differs from traditional executable program formats in a number of important ways. In particular, a Java program, unlike one written in C or C++, isn't a single executable file, but instead is composed of many individual class files, each of which corresponds to a single Java class. Additionally, these class files are not loaded into memory all at once, but rather are loaded on demand, as needed by the program. The Class Loader is the part of the JVM that loads classes into memory.

The Java Class Loader, furthermore, is written in the Java language itself. This means that it's easy to create your own Class Loader without having to understand the finer details of the JVM.

A class loader is an object that is responsible for loading classes. The class ClassLoader is an abstract class. Given the binary name of a class, a class loader should attempt to locate or generate data that constitutes a definition for the class. A typical strategy is to transform the name into a file name and then read a "class file" of that name from a file system.

Every class object contains a reference to the ClassLoader that defined it. Take array for example. Class objects for array classes are not created by class loaders, but are created automatically as required by the Java runtime. The class loader for an array class, as returned by Class.getClassLoader () is the same as the class loader for its element type; if the element type is a primitive type, then the array class has no class loader. Applications should implement subclasses of ClassLoader in order to extend the manner in which the Java virtual machine dynamically loads classes.

2.1.2 Handle error with exception

What Is an Exception?

The term exception is shorthand for the phrase "exceptional event. "

Definition: An exception is an event that occurs during the execution of a

program that disrupts the normal flow of instructions during the execution of a program.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack (see the Figure 2-1).

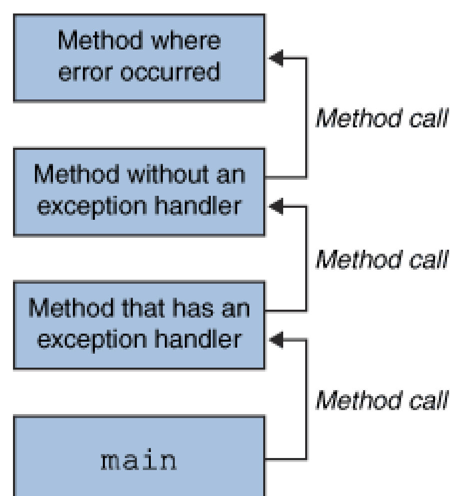


Figure 2-1 : Searching the error handling the method where the error occurred in list of method

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order that the methods were called. When an appropriate handler is found, the run-time system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler. The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

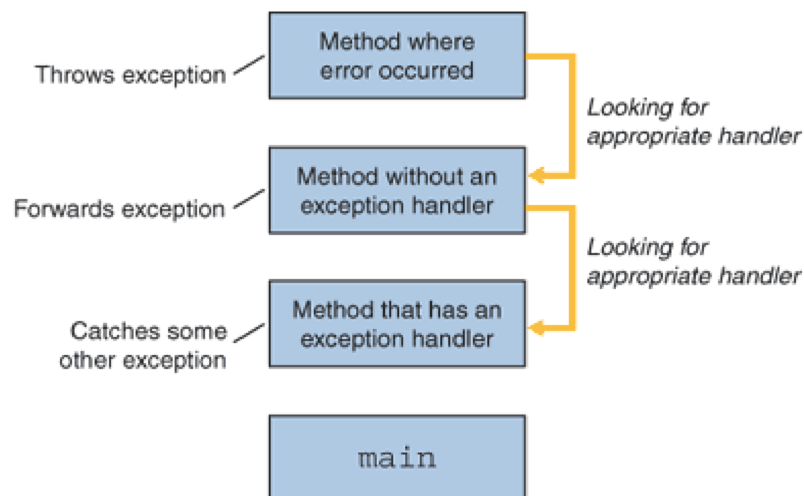


Figure 2-2 : Find appropriate handler and pass the exception to the handle.

2.1.3 Java Access Bridge

Our system is based on Windows platform. However, part of our system is Java application, so our system needs to use Java Access Bridge to communicate with the part.

Java Access Bridge for the Microsoft Windows Operating System makes it possible for a Windows based Assistive Technology to get at and interact with the Java Accessibility API. The Java Accessibility API is implemented in the Java Foundation Classes (JFC) Project Swing user interface components.

The Java Access Bridge is a class which contains "native methods." Part of the code for the class is actually supplied by a dynamically linked library (DLL) on the Microsoft Windows system. The assistive technology running on the host platform (for example, a screen reader running in Windows) communicates with the Microsoft Windows native DLL portion of the bridge class. In turn, the native code of the Access Bridge Class communicates through the Java Virtual Machine with the Java Accessibility utility support and the Java Accessibility API on the individual user interface objects of the Java-based application.

2.2 Program slicing

Program slicing is a decomposition technique that extracts from program statements relevant to a particular computation. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest. Slicing has applications in testing and debugging, re-engineering, program comprehension and software measurement. Informally, a slice provides the answer to the question "What program statements potentially affect the computation of variable v at statement s?"

2.2.1 Static program slicing

There are many forms of slice, so it will be helpful to start off with a simple form of slice; the static slice. The other forms of slice can be thought of as augmentations of this static form.

A slice is constructed by deleting those parts of the program that are irrelevant to the values stored in the chosen set of variables at the chosen point. The point of interest is usually identified by annotating the program with line numbers which identify each primitive statement and each branch node. For Example, the example program which ask for two parameters. Show the slice of this program with respect to criterion <9, total>.

```
Original program:
1 int test(int x,int y)
{
2     int total=0;
3     int sum=0;
4     if(x<=1)
5        sum=y;
6     else
      {
7     sum=x+y;
8     total=x*y;
      }
9     return (total+sum);
}
```

```
Criterion: <9, total>
1 int test(int x,int y)
{
2     int total=0;

4     if(x<=1)

6     else
      {

8     total=x*y;
      }
9     return (total+sum);
}
```

Figure 2-3 : The slice of this program with respect to criterion <9, total>

Having picked a slicing criterion, we can construct one of two forms of slice: a backward slice or a forward slice. A backward slice contains the statements of the program which can have some *effect on* the slicing criterion, whereas a forward slice contains those statements of the program which are *affected by* the slicing criterion.

A *backward slice* consists of all program points that affect a given point in the program. For example, the backward slice shown in Figure 2-4 (A) indicates exactly which statements influence the output of variable *i*. A *forward slice* consists of all program points that are affected by a given point in the program. For example, the forward slice shown in Figure 2-4 (B) with under line indicates exactly which statements are influenced by the initialization of variable *sum*.

```
void main()
{
int i = 1;
int sum = 0;
while (i<11) {
sum = add(sum, i);
i = add(i, 1);
}
printf("sum = %d\n", sum);
printf("i = %d\n", i);
}
static int add(int a, int b)
{
return(a+b);
}
```

(A). Backward *slice from* `printf("i = %d\n", i);`

```
void main()
{
int i = 1;
int sum = 0;
while (i<11) {
sum = add(sum, i);
i = add(i, 1);
}
printf("sum = %d\n", sum);
printf("i = %d\n", i);
}
static int add(int a, int b)
{
return(a+b);
}
```

(B). ***Forward*** *slice from* `sum = 0;`

Figure 2-4 : (A) Backward slice (B) Forward slice

2.2.2 Dynamic program slicing

When we debug a program, we will usually have executed it, and presumably found that it produced an unexpected value. For example, the program in Figure 2-5 might have been executed with the input value 0 for the variable x. Now, when we execute the program in Figure 2-5 with the value 0 we will find that the variable p contains the wrong value. Instead of consulting the static slice to locate the cause of this bug, it would make more sense to construct a slice which *exploited* the information available about the input which caused the program to go wrong. Such a slice is called a dynamic slice, because it is constructed with respect to the traditional

static slicing criterion *together* with dynamic information (the input sequence supplied to the program, during some specific execution).

```
scanf("%d",&n);
s=0;
p=0;
while (n>0)
{
s=s+n;
p=p*n;
n=n-1;
}
printf("%d%d",p,s); }
/* the slice point is the end of the program */
```

Figure 2-5 : A program fragment to be sliced

We construct a dynamic slice with respect to three pieces of information. Two of these - the variable whose value appears to be wrong and the point of interest within the program - are just the same as we find in static slicing. The third is the sequence of input values for which the program was executed. Collectively, this information is called the `dynamic slicing criterion'. We shall say that we construct a dynamic slice *for* a variable *v*, *at* a point *n*, *on* an input *i*. To describe the input sequence, *i*, we shall enclose the sequence of values in angled brackets. Thus <1, 4, 6> represents a sequence of three input values the first of which is 1, the second of which is 4 and the last of which is 6. In the case of the example in Figure 2-5, we shall construct a

dynamic slice for the variable p at the end of the program on the input sequence <0>. The dynamic slice constructed with respect to this criterion is shown in Figure 2-6 (A). It is far simpler than the corresponding static slice Figure 2-6 (B) and it clearly highlights the bug in the original program.

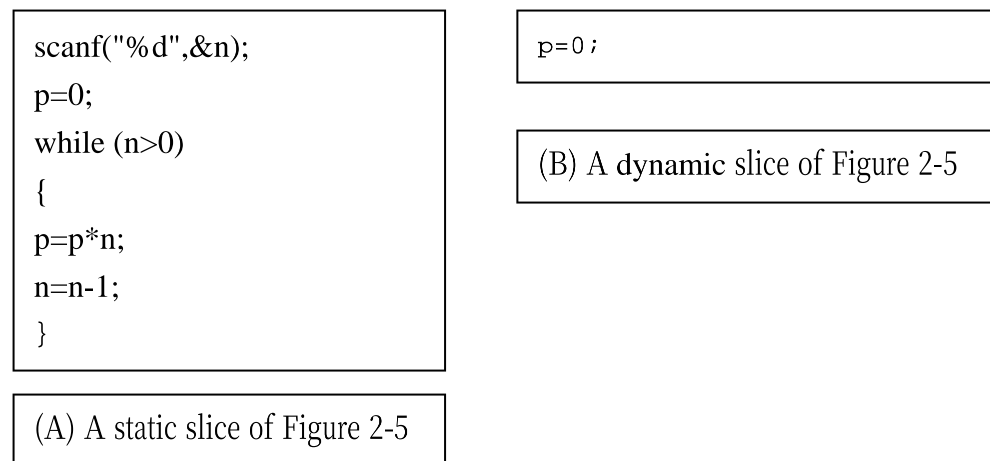


Figure 2-6 : (A) Static slice of Figure 2-5 (B) Dynamic slice of Figure 2-5

Obviously, we were rather lucky when we constructed the dynamic slice for the example program in Figure 2-5; we achieved the maximum possible level of simplification. The example was chosen to highlight the advantage of dynamic slicing. In practice we shall probably be less lucky. Of course, if we have a number of test cases, all of which cause some failure, then there's nothing to stop us having the slicing tool choose which case we should deal with by constructing a dynamic slice for each and selecting the smallest.

2.3 Software component: JavaBean

One thing can be stated with certainty: components are for composition. *Nomen est omen*. Composition prefabricated “things” to be reused by rearranging them in ever new composites. Beyond that trivial observation, much is unclear.

JavaBeans technology is the component architecture for the Java 2 Platform, Standard Edition (J2SE). Components (JavaBeans) are reusable software programs that you can develop and assemble easily to create sophisticated applications. JavaBeans technology is based on the JavaBeans specification.

“A Java Bean is a reusable software component that can be manipulated visually in a builder tool.”

This covers a wide range of different possibilities.

The builder tools may include web page builders, visual application builders, GUI layout builders, or even server application builders. Sometimes the “builder tool” may simply be a document editor that is including some beans as part of a compound document.

Some Java Beans may be simple GUI elements such as buttons and sliders. Other Java Beans may be sophisticated visual software components such as database viewers, or data feeds. Some Java Beans may have no GUI appearance of their own, but may still be composed together visually using an application builder.

Some builder tools may operate entirely visually, allowing the direct plugging together of JavaBeans. Other builders may enable users to conveniently write Java classes that interact with and control a set of beans. Other builders may provide a simple scripting language to allow easy high-level scripting of a set of beans.

Individual Java Beans will vary in the functionality they support, but the typical unifying features that distinguish a Java Bean are:

- Support for "introspection" so that a builder tool can analyze how a bean works
- Support for "customization" so that when using an application builder a user can customize the appearance and behaviour of a bean.
- Support for "events" as a simple communication metaphor than can be used to connect up beans.
- Support for "properties", both for customization and for programmatic use.
- Support for persistence, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

A bean is not required to inherit from any particular base class or interface.

Visible beans must inherit from java.awt.Component so that they can be added to visual containers.

Note that while beans are primarily targeted at builder tools they are also entirely usable by human programmers. All the key APIs such as events, properties, and persistence, have been designed to work well both for human programmers and for builder tools.

Many beans will have a strong visual aspect, in both the application builder and in the final constructed application, but while this is common it is not required.

2.4 ProGrammar

2.4.1 Introduction to ProGrammar

ProGrammar is a visual development environment that simplifies the process of building parsers. It's designed for programmers who need to start building parsers right away. ProGrammar consists of the following components:

- Interactive Development Environment (IDE) - Visual environment for building, testing and debugging parsers.
- Grammar Definition Language (GDL) - High-level language used to express data syntax.

- Parse Engine - Runtime component that parses input data. The parse engine is available as a Windows DLL, static library, and as an ActiveX control.
- Application Programmers Interface (API) - Programming interface for calling the parse engine at runtime.

2.4.2 Build a Java parser with Programmar

ProGrammar divides the overall task of building a parser into two distinct phases. The first phase is defining the grammar, which is done in the visual IDE. The second phase involves calling the parse engine from your application, in order to parse data and process the results. This separation of tasks is important because it allows you to focus entirely on the process of defining a grammar--until you get it right--before writing application-specific code. In fact, by the time your application calls the parser, most of the work will already have been accomplished in the visual environment. Typically, your program needs only to pull the data that it requires from a parse tree.

The following steps are involved in building and using a parser:

1 Define the grammar. The first step is to write a grammar that describes the syntax of the data that will be parsed. The ProGrammar already provide example that

include Java GDL. So, we don't need create a new grammar for Java.

2 A binary grammar file is generated by the IDE whenever a project is built. This file contains instructions for parsing a particular type of data, and is used directly by the parse engine at runtime. We could use the Java GDL that ProGrammar provide to build the binary grammar file.

3 Call the parser from our system. Our system interacts with parse engine by calling the ProGrammar API. The API provides methods to load grammars, parse data, process errors, and retrieve data elements from the parse tree. The API can be called as an ActiveX control or a native C++ library. Our system is build use by MFC. So, we call the API use the C++ library. Figure 2-7 illustrates the architecture.

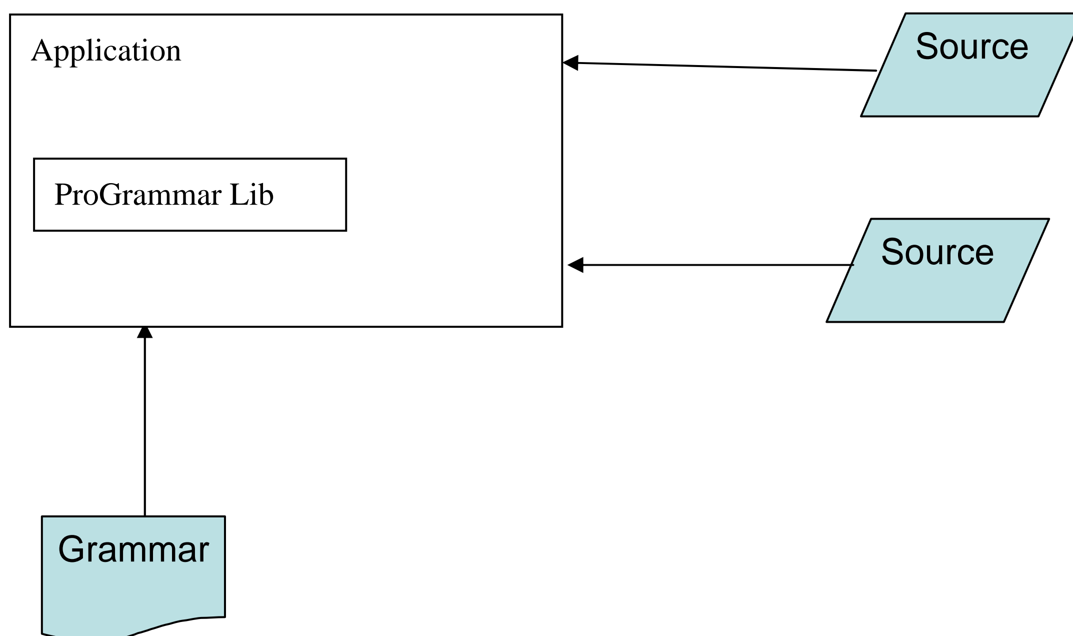


Figure 2-7 : The relation ship with grammar and programmer lib

4 The parse engine is the runtime component that reads input data, parses it

according to the rules of the grammar, and generates a parse tree. The input data can be read from a disk file or memory buffer. We just pass the path of Java source to ProGrammar.

5 Retrieve data from the parse tree. After each successful parse, the parse engine generates a parse tree, which is a data structure that maps grammar symbols to elements in the input stream. According to the grammar symbol, we could get any element of the tree that we want correctly. We could call API methods to navigate, search, and retrieve data elements from the parse tree.

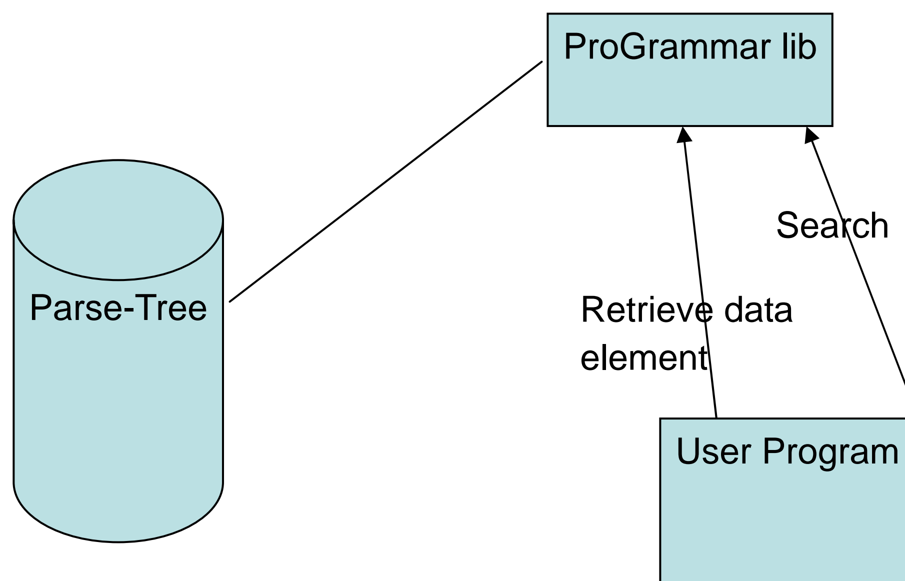


Figure 2-8 : Use programmer lib to retrieve data from the parse tree

2.4.3 The parser tree in Programmar

A parse tree is a tree data structure that is built by the parse engine to represent the hierarchical structural of the input data. As the parser consumes data from the input file, it adds nodes to the parse tree. Each node associates a symbol in the grammar to a subset of the input file. Each node label contains the name of a grammar symbol, while its value is the range of characters from the sample file that corresponds to the value parsed by that symbol.For example,

Figure 2-9 is a java source file that we input. Figure 2-10 is the parse tree that is build by Programmer

```
(1)public class test
{
(2)     int f1;
(3)     public static String getS(String s2)
        {
(4)             int a=0;
(5)             return "test";
        }
}
```

Figure 2-9 : An example of source file

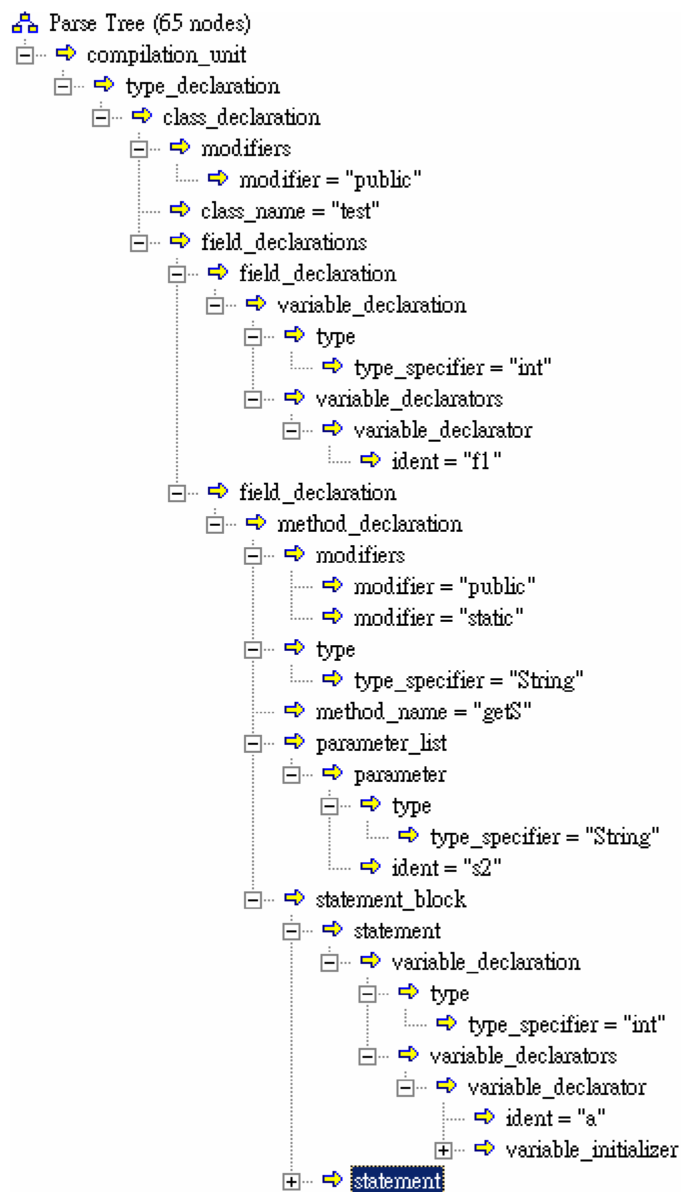


Figure 2-10 : The parse tree of the Figure 2-9

We use the Programmar's component and grammar file to do the parser program. The component will build a tree for each time the java file it load. Know the structure of the tree is very important for building a parser program by using Programmar's component.

We could query any node that we need though Programmar lib for implementing

the parser. If we need find out all the value that declare in the source that just query all the node that symbol is “declaration.” The Programmar lib would return all nodes that we need.

2.5 Dot

dot draws a graph in four main phases. Knowing this helps you to understand what kind of layouts *dot* makes and how you can control them. The layout procedure used by *dot* relies on the graph being acyclic. Thus, the first step is to break any cycles which occur in the input graph by reversing the internal direction of certain cyclic edges. The next step assigns nodes to discrete ranks or levels. In a top-to-bottom drawing, ranks determine Y coordinates. Edges that span more than one rank are broken into chains of “virtual” nodes and unit-length edges. The third step orders nodes within ranks to avoid crossings. The fourth step sets X coordinates of nodes to keep edges short, and the final step routes edge splines.

dot accepts input in the *DOT* language. This language describes three kinds of objects: graphs, nodes, and edges. The main (outermost) graph can be directed (digraph) or undirected graph. Because *dot* makes layouts of directed graphs, all the following examples use digraph. Within a main graph, a subgraph defines a subset of nodes and edges.

Figure 2-11 is an example graph in the *DOT* language. Line 1 gives the graph name and type. The lines that follow create nodes, edges, or subgraphs, and set attributes. Names of all these objects may be C identifiers, numbers, or quoted C strings. Quotes protect punctuation and white space. Figure 2-12 is the drawing result of Figure 2-11.

```
1:  digraph G {
2:      main -> parse -> execute;
3:      main -> init;
4:      main -> cleanup;
5:      execute -> make_string;
6:      execute -> printf
7:      init -> make_string;
8:      main -> printf;
9:      execute -> compare;
10: }
```

Figure 2-11 : An example graphic in the DOT language

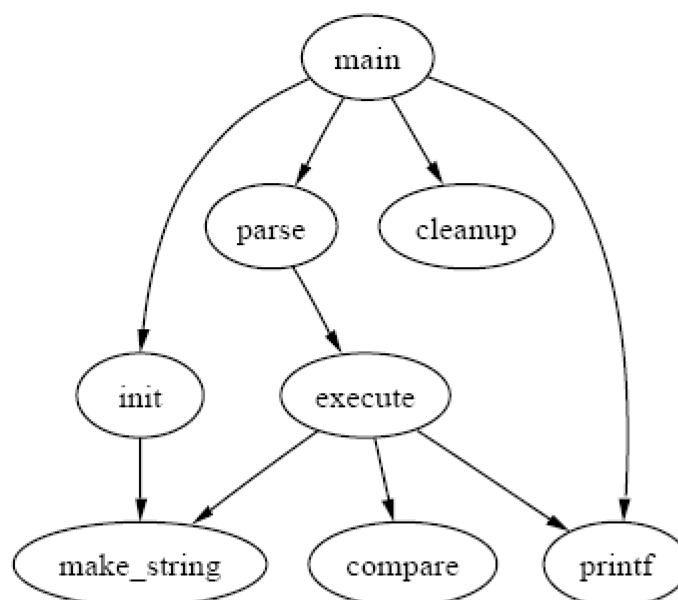


Figure 2-12 : Draw of Figure 2-11