Project 2: Semantic Checker

E-JUST
Egypt-Japan University of Science and Technology

Group members:

Mennatullah Abdelrahman “Developing”.

Fadi Alahmad “testing”.

Martin Ihab “documentation “.

Contents:

---

1. Introduction:

1.1 Semantic analysis

is the process of ensuring that a program's declarations and statements are semantically correct, that is, that their meaning is clear and consistent with how control structures and data types.

1.2 Semantic analysis typically entails:

– Type checking: Data types are used in ways that are consistent with their definition (i.e., only with compatible data types, only with operations that are defined for them, and so on).

– Label Checking: Labels that are mentioned in a program must exist.

– Flow control checks: control structures must be used correctly (no GOTOs into a FORTRAN DO statement, no breaks outside of a loop or switch statement, and so on).

Project 2: Semantic Checker

E-JUST
Egypt-Japan University of Science and Technology

1.3 Semantic Analysis Performed in a Compiler

– Semantic analysis does not exist as a separate module within a compiler. It is typically a collection of procedures called by the parser at appropriate times as the grammar requires.

– Implementing semantic actions in recursive descent parsing is conceptually simpler because they are simply added to the recursive procedures.

– Including semantic actions in a table action-driven LL (1) parser necessitates the addition of a third type of variable to the productions as well as the software routines required to process it.

---

2. problem definition

2.1 The semantic checks implemented

- No identifier is used before it is declared.
- The program contains a definition for a method called main that has no parameters and returns void.
- The hint literal in an array declaration must be greater than 0.

---

3. Design

By extend last program we have done in the lab,

3.1 lexical analysis

First, and the initial phase of a compiler is lexical analysis. It uses modified source code written in the form of sentences from language preprocessors. By removing any whitespace or comments from the source code, the lexical analyzer breaks these syntaxes down into a series of tokens, the lexical analyzer generates an error if a token is found to be incorrect. The lexical analyzer and the syntax analyzer function in tandem. When the syntax analyzer requests it, it pulls character streams from the source code, verifies for legal tokens, and provides the data to it.

3.2 Parse-Tree Listeners

ANTLR's runtime provides the Parse Tree Walker class to walk a tree and trigger calls into a listener. To create a language application, we create a Parse Tree Listener implementation that contains application-specific code that usually calls into a larger surrounding application.

ANTLR creates a grammar-specific Parse Tree Listener subclass with enter and exit methods for each rule. When the walker comes across the rule assign node, for example, it calls enter Assign and passes the Assign Context parse-tree node to it. The walker calls exit Assign after visiting all the children of the assign node. The thick dashed line in the tree diagram above depicts Parse Tree Walker performing a depth-first walk.

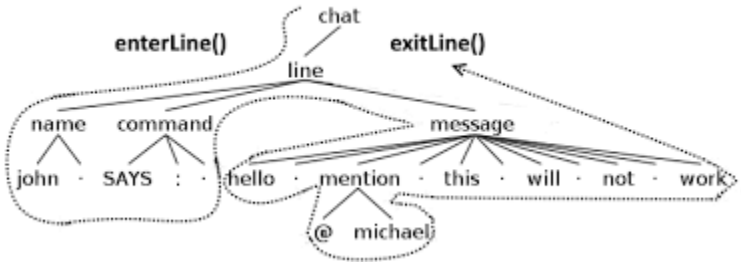

Figure 1

It specifies where in the walk the enter and exit methods for rule assignment are called by Parse Tree Walker. (The other listener calls are not displayed.) and the diagram in Figure 1, Parse Tree Walker call sequence, depicts the entire sequence of calls made by Parse Tree Walker to the listener for our statement tree.

Figure 2

3.3 Parse-Tree Visitors

There are times when we want to have complete control over the walk, specifically requesting techniques to see children. Option -visitor instructs ANTLR to create a visitor interface from a grammar that contains a visit method for each rule. On our parse tree, we have the following familiar visitor pattern, A depth-first traverse of the parse tree is shown by the thick dashed line. The method call sequence among the visitor methods is indicated by the thin dashed lines. Our application-specific code would generate a visitor implementation and call visit to start a tree walk.

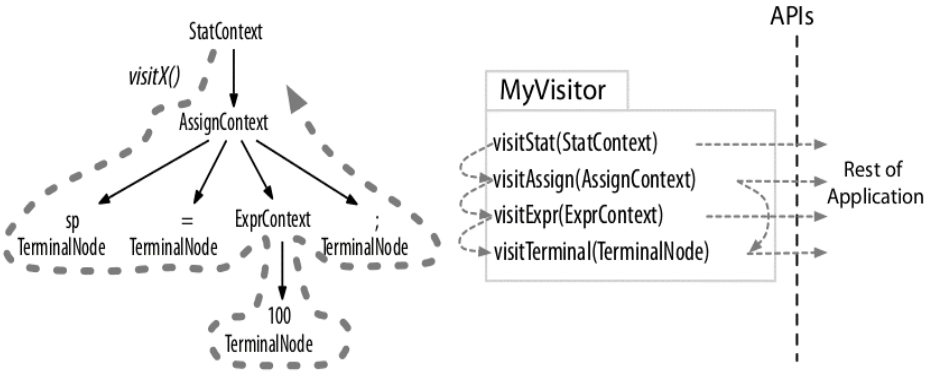

Figure 3

4. How each part "checker" work.

4.1 No identifier is used before it is declared

list is created to store all the non-null "IDs" appeared in the var_dec section. Then, by using the Visitor we visited all the nodes in the section. We made the same with field declaration and statement section. In the Location section we checked if the variable used has been added to the list before o not.

4.2 The program contains a definition for a method called main that has no parameters and returns void.

we used Listener technique; we created a flag "main found" with false default value. Then, we checked if the value of the flag changed. As we change the value once we found main. Then, check if the value still false, if yes them the program doesn't contain main.

4.3 The hint literal in an array declaration must be greater than 0

we also managed using Visitor as we checked the field declaration, we checked the Antiliteral and check its length. Ch part "checker" work.

Project 2: Semantic Checker

E-JUST
Egypt-Japan University of Science and Technology

5. conclusion

5.1 Test cases

a)
```
class Program {
  void main() {
    int x;
    y = 1;
  }
}
```

b)
```
class Program {
  int main(int x) {
    int x;
    y = 1;
  }
}
```

c)
```
class Program {
  int A[0xA];

  void bar() {
    for i = 0, 0xA {
      A[i] = i;
    }
  }

  void main() {
    bar();
  }
}
```

5.2 The Output

```
class Program {void main() {int x; y = 1;}}
(program class Program { (method_decl void main (
Undeclared Symbol: y
```

```
class Program {int main(int x) {int x; y = 1;}}
(program class Program { (method_decl (type int) main ( (
There is no main in the program
Undeclared Symbol: y

class Program {  int A[0];  void bar() {    for i = 0, 0xA {      A[i] = i;
(program class Program { (field_decl (type int) A [ 0 ] ;) (method_decl void b
Error, Arrays length should be greater than 0
```

6. summary of work

6.1 understanding java

Java is quite difficult to learn and can be understood in a short span of time as it has a lot of syntax and specific semantic.

6.2 Dealing with all programs in same

time working on parse-tree visitors and parse-tree listeners and choose best program of ours.