CS 444 - Group 7
Design Documentation For Assignment 2

Bram Cymet, bcymet
Jung Myeng Lee, jm3lee
Thomas Kim, tkim

March 2, 2006

Contents

Chapter 1

Design Changes From Assignment 1

The main change in assignment 2 in terms of design was to move where we enter symbols into our symbol table. Before, we insert the symbols into the symbol table as we encountered them in the scanner. The process is now different: rather than inserting them during scanning, we now enter the symbols into the symbol table during our parse tree traversal.

1.1 Symbol Table

The `SymbolTable` and `SymbolTableEntry` classes were also modified to accommodate other functionality that was required for this assignment. When there are multiple entries with the same name, they are chained together so that we may search through them to resolve overloaded symbols. Figure 1.1 gives a graphical representation of how `SymbolTableEntry` is structured. For the detailed explanation of its components please refer to Section 3.1.1.

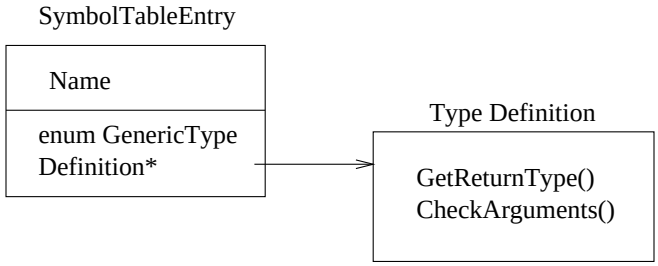

Figure 1.1: Symbol Table Entry Structure

1.1.1 Generic Types

`GenericType` enumeration specifies which category a type belongs to. This helps us to provide a generic interface to various type definitions. (See Section 3.1.1).

Chapter 2

Preamble

2.1 Intermediate Representation

We chose the intermidate representation to be a *parse tree*. Please see the next section on how to produce this output.

2.2 How to Run the Compiler

NOTE: Cross-references are printed automatically, and each cross-references are prefixed with "resolved:".
NOTE: When given the option `-G`, the abstract syntax tree for the standard library is printed as well as the given source. However, `-g` only produces an abstract syntax tree for the input source code.

Please read README file at the top directory. Command-line arguments are shown below.

```
CS 444: Group 7
Usage: adac [options] filename
Options:
 -d [component]: Enable debugging of a given component.
                 p - Parser
                 P - Parse tree
                 s - Scanner
                 v - Visitor traversal
                 t - Type list
                 S - Symbol table
 -p [options]  : Enable debugging of the parse tree.
                 c - Do not collapse parse tree
 -g            : View graphical parse tree (requires dotty)
 -G            : View ascii parse tree
 -h            : Print help (this text)
```

Example invocation:

```
$ adac -G foo.adb
```

to type check `foo.adb` and view the ASCII version of the abstract syntax tree.

2.3 Prerequisites

The program has been compiled and tested on `cpu16.student.cs.uwaterloo.ca`.

1. GCC g++ 3.4.2

2. GNU Make 3.80
3. dot and dotty (version 96c (09-24-96)) to view graphical abstract syntax tree.

The following tools are required for development.

1. Python 2.4.2
2. Bash 2.05b

2.4 How to Compile

Type `make` at the top of the directory structure.

2.5 Assumptions

1. If the input source code is incorrect, the cross-reference output may contain incorrect information.
2. The cross-reference does not have to provide column numbers.

2.6 Features Completed

Analysis component of the compiler including

1. Symbol table and cross-reference construction.
2. Abstract syntax tree construction.
3. Type checking.
4. Error reporting.
5. Primitive symbol resolution and overloading.

2.7 Known Issues and Unsupported Features

Due to the time constraint, a few features that are present in Ada/CS were not included into our program.

1. String and Float data types are not fully supported. – We can easily extend the function prototypes in the standard library to support this. This will be available in the next release.
2. Anonymous types are not supported. – This could be supported if given more time. we would just have to insert it into the type list. It is not necessary to assign an anonymous time a real name. We can query type definitions by providing a `type_t`.
3. Aggregates are not implemented. – Most of the aggregates are implemented as functions. Extension to the standard library will be available in the next release.
4. When subtyping enumerations, the new enumerations in the subtypes are not inserted into the symbol table. Therefore, when you encounter an enumeration that is part of a subtype or a type, it will just be the type. – This can be fixed as we declare the subtype, just insert the enumerations into the symbol table with the new defined type.
5. In a discrete range type declaration, if the range is ambiguous, we do not attempt to resolve the error correctly. – This feature can be implemented if we had more time.
6. The `new` keyword is not supported. – The feature will be available on the next release.

7. Subtypes can be declared, and will be inserted into the type list, but one cannot use subtypes as parameter types in function and procedure declarations. Subtypes symbol resolution is not performed correctly. – This feature will be available on the next release.

8. Functions and procedures can be declared multiple times. – If we had more time, this problem could have been solved easily.

Chapter 3

Design and Implementation

3.1 Overview

The simplified overview of the system is presented in Figure 3.1.

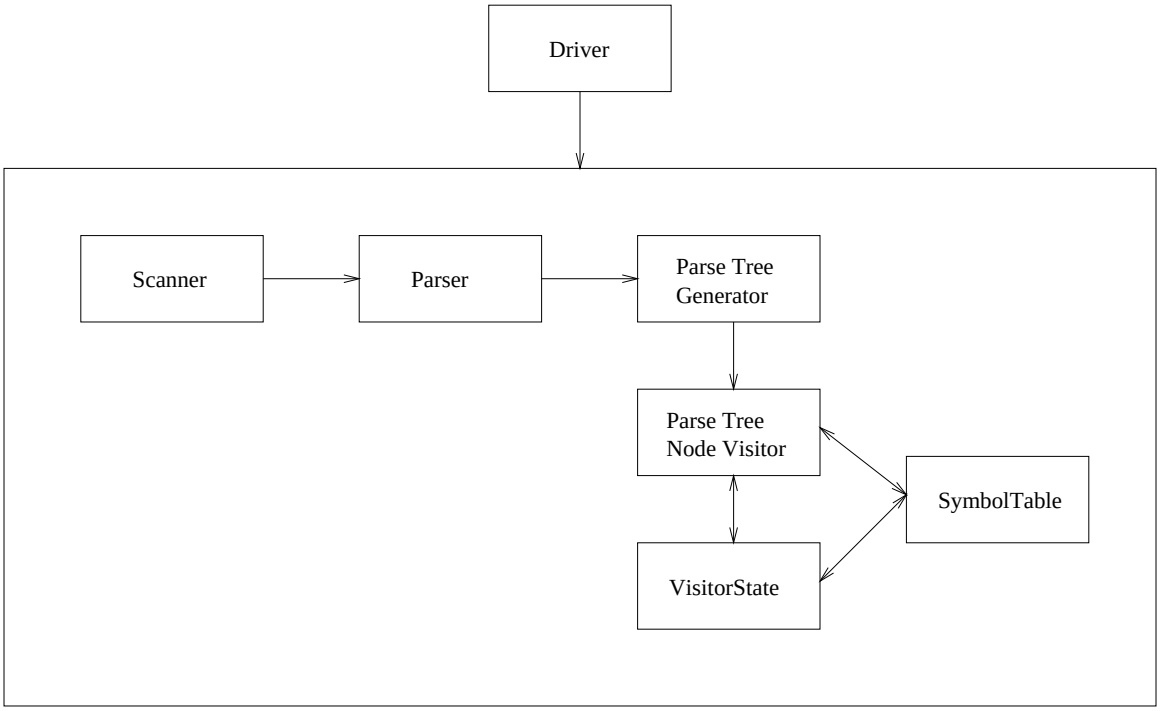

Figure 3.1: General Data Flow

The `Driver` class is responsible for instantiating all the components of the compiler, and start the quest of glory. An arrow represents the flow of the data.

3.1.1 New Classes

With our evolving design, we created more classes. They are described in this section.

1. `ParseTreeNode` An automatically generated classes that define various nodes that form an AST. Each node does not contain much information. However, it defines how a visitor should traverse the AST.

2. `ParseTreeNodeVisitor` An automatically generated visitor class with `Visit()` for each node in the abstract syntax tree.

3. `ParseTreeNodeFactory` An automatically generated factory class for ParseTreeNode.

4. `VisitorState` See Section 3.4 for details.

5. `TypeList` and `TypeDefinition` See Section 3.1.1 for details.

6. `ErrorReporter` This class handles and reports the errors that type analysis phase encounters. It holds all types of errors and appropriate error messages.

3.2 Type List

This class maintains all the types in our system. It is implemented using the singleton design pattern. Since types can be overridden, it also has the functionality to insert new types when you are in a scope where that type was not previously defined. As the type analysis is done, invocations can be made of this class to query the known types in the system. It also assigns a unique identifier to each definition to aid in type checking.

3.2.1 Type Definition

This class is used to define each type in our system. By extending this class, we can specifically define different types with a generalized interface. These definitions holds specific information about the type that is created. See Table 3.1 for the list of derived classes. Also, see Figure 3.2 for the inheritance hierarchy.

Class	Differences from the Base Class
`PackageDefinition`	Holds a symbol table to maintain package specific variables, functions and procedures. This enables us to implement multiple package declarations.
`ProcedureDefinition`	Holds a list of parameters.
`FunctionDefinition`	Holds a list of parameters and the return type.
`ArrayDefinition`	Holds a list of ranges, where each range has a type, lower and upper bound. This enables us to implement multi-dimensional arrays.
`RecordDefinition`	Holds a symbol table to maintain record specific fields.
`VariableDefinition`	None
`EnumerationDefinition`	None
`RangeDefinition`	Holds lower and upper bound of the range.
`SubtypeDefinition`	Holds a base type identifier so that subtypes can be resolved properly at the compile time.

Table 3.1: List of Derived Classes from `TypeDefinition`

3.3 Abstract Syntax Tree

We implemented the algorithm given in the class to generate a "full parse tree." Using this parse tree, we create an abstract syntax tree (AST). The creation of an AST from the full parse tree is done in three passes:

1. Remove nullable nodes.

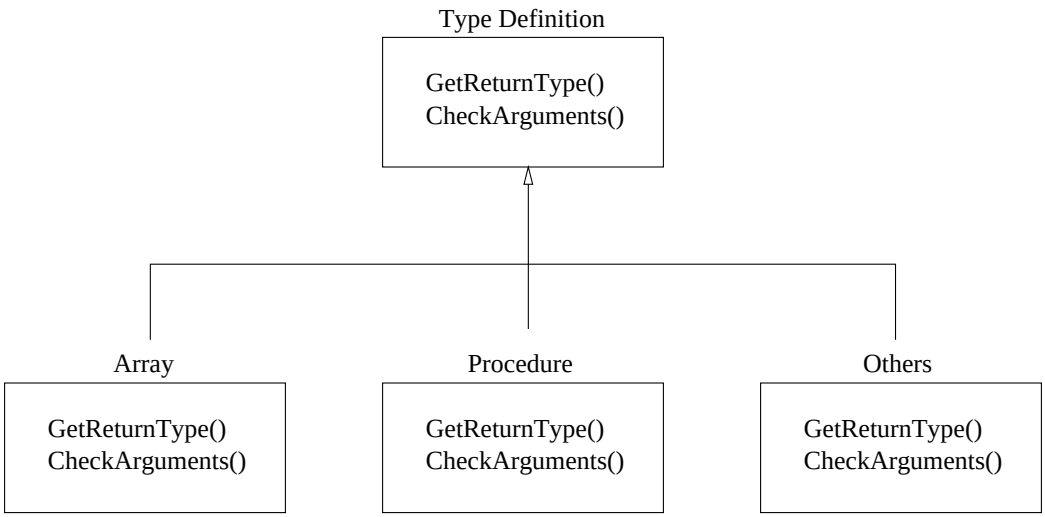

Figure 3.2: Type Definition Inheritance Hierarchy

2. Collapse nodes with only one children with exceptions defined by us. Some exceptions are required to preserve consistent structure of a generated AST. This aids us in simplifying the implementation.

3. Remove useless nodes. In this step, we remove nodes that do not present any useful information in type checking; for example, semicolons.

The three simple phases produces a compact form of the original parse tree. The AST construction has been implemented as a separate class from a parser to keep the parser simple.

3.4 Abstract Syntax Tree Traversal

[1] describes a syntax directed translation using Visitor design pattern. We have decided to use this approach instead of implementing dependency graph generation or topological sort.

3.4.1 Visitor State

At each node of the abstract syntax tree, it is desirable to design a flexible system to allow modification of a visitor's behavior. We implement this with `Visitor` class that has a pointer to an instance of `VisitorState`. `Visitor` also provides accessor methods to change visitor states on the fly.

The advantages of this design are

1. Each token in the abstract syntax tree should be handled differently depending on its siblings and parent nodes. One node does not provide enough information to correctly do syntax directed translation.

2. Inherited and synthetic attributes are propagated throughout the parse tree with a uniform interface. Upon entering a handler for a particular node, the new visitor state can copy information from the previous visitor state. When leaving the handler, information can be copied from the current visitor to the "old" visitor state. This enables us to take advantages of the polymorphism, and provides a high level abstraction.

3. It is possible to turn a visitor state into a DFA. As a visitor goes through the tree, it can dynamically determine what the correct action is depending on the position in the parse tree. Note that these states can be inferred from keywords. For example, private sections in a package can be handled specially after all the children nodes are traversed.

VisitorStateVariableType	VS_Exit	VS_Exception
VisitorStateDeclareVariable	VS_Assignment	VS_Expr
VS_Unary_Expr	VS_CompilerDirectives	VS_BExpr
VS_SubtypeDeclaration	VS_Case_Stmt	VS_Loop
VS_Iteration	VS_Range	VS_Block
VS_CallStatement	VS_Function	VS_ItemSelector
VS_Package	VS_TypeDeclaration	VS_VariableDenotation

Table 3.2: List of Derived Classes from `VisitorState`

The disadvantages of this design are

1. If we have implemented a parser generator, it would have been easier to extend the features. Writing numerous visitor states, and debugging them prove to be difficult.
2. Directly dealing with recursive traversal of the abstract syntax tree can lead to long debugging sessions. Since the code to implement a feature is spread throughout various visitor states, it is troublesome to maintain collective information about the system at a time.

Please refer to Figure 3.3 for the general control flow of this subsystem. In the diagram, every box represents where the control currently is. Arrows represent transition of control flows along with a method that triggers the transition. Circled numbers present the order of the execution. See Figure 3.2 for the complete listing of all the derived classes.

3.5 Error Recovery Semantics

The error recovery semantics in this assignment were much more concrete than in assignment 1. Depending on the error we encountered, we printed the appropriate error message and handled the error. For example, when we encounter an error while declaring a type, we simply mark that the type is invalid and future references to that type will result in another error. See Table 3.3 for the list of error identifiers and corresponding messages.

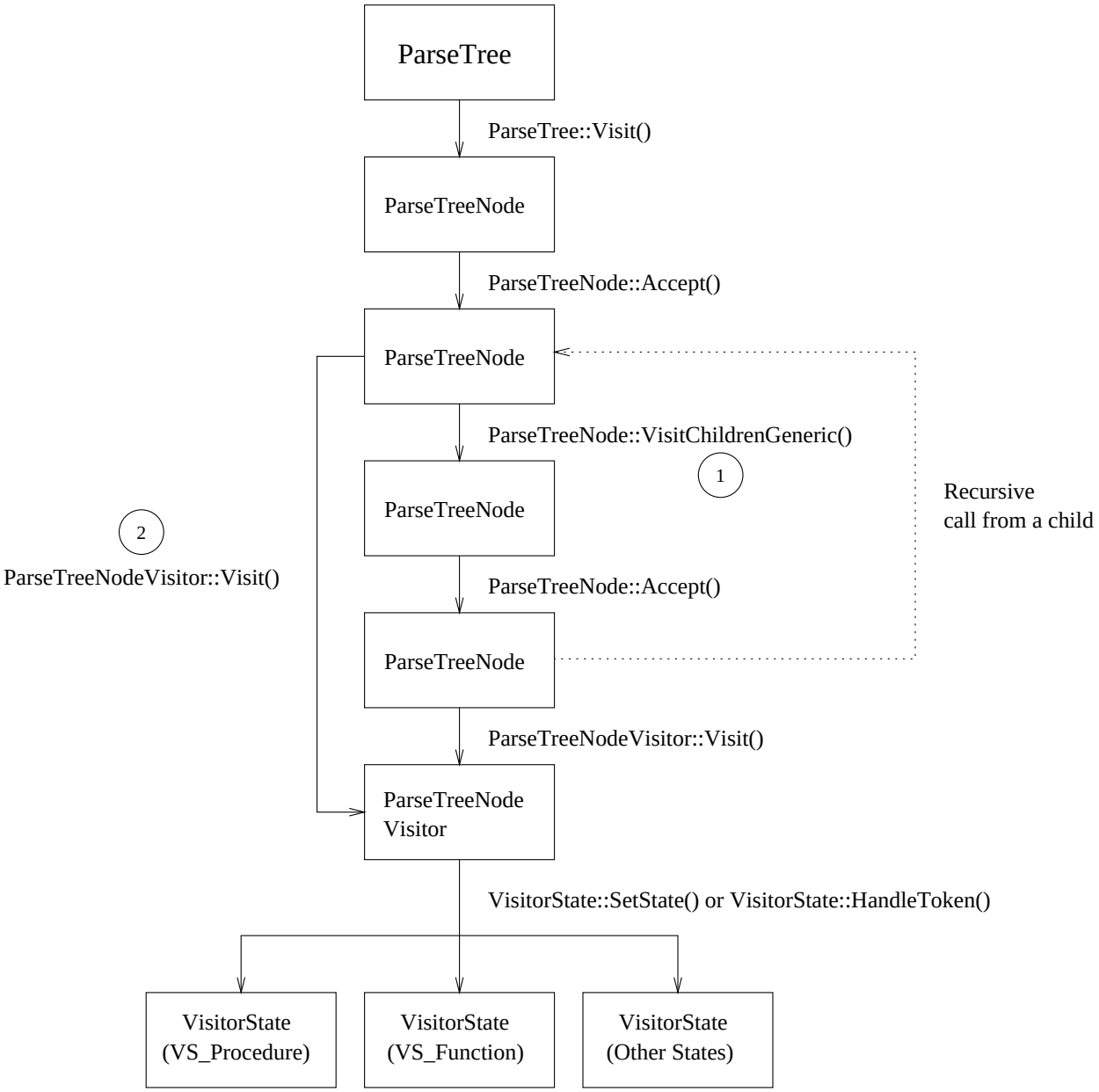

Figure 3.3: `Visitor` and `VisitorState` Control Flow

Error Identifier	Description
UNDECLARED_ID	"Identifier undeclared" Currently this error is reported multiple times if an undeclared variable is referenced multiple times throughout the source code. The ideal behavior would be to report this error only once per variable.
UNDECLARED_TYPE	"Type undeclared"
INVALID_TYPE_ASSIGNMENT	"Left and right values do not match in an assignment statement." When the left and right hand sides of an assignment statement does not match after considering all the overloaded types, this error is issued.
INVALID_PARAMETER	"Invalid parameter" When a function or a procedure is called with incorrect parameter types, this error is issued. When an array is access with incorrect index types, this error is issued.
DECLARED_ALREADY	"Identifier already declared" Multiple declarations of a variable will cause this error.
INCOMPATIBLE_TYPE	"Incompatible type in an expression" This error is issued when the types of operands of either unary or binary operators do not match.
INVALID_FIELD	"Invalid field" When a reference field does not exist in a record type, this error is issued.
NEED_BOOLEAN	"Need a boolean expression" This error is issued when conditional statements do not contain a boolean expression.
CONSTANT_ASSIGNMENT	"Left value in an assignment statement is constant"
CASE_TYPE_WRONG	"Wrong type used in case statement"
CASE_RANGE_TYPE_WRONG	"Wrong type used in case range statement"
NOT_A_PROCEDURE	"Procedure expected" This error is issued when any other type than procedure is reference in a call statement.
ANONYMOUS_ACCESS_NOT_ALLOWED	"Anonymous access type is not allowed here" The type is not allowed to be an anonymous access type in a variable declaration statement,
BAD_RANGE	Problem with range statement
MISSMATCHED_LABEL	Miss matched block labels
FOR_RANGE_BAD	Range type does not match discrete type in for loop
LABEL_ALREADY_DECLARED	"Can not use a previously declared name as a label"
WRONG_LABEL_IN_EXIT	"Wrong label used in exit statement"
EXIT_FROM_NON_LOOP	"Can only exit from loop"
INVALID_TYPE_ASSIGNMENT_DECLARE	"Invalid assignment in declaration"

Table 3.3: List of Error Messages

Chapter 4

Testing

Many of the test cases in assignment 1 were run through our compiler but were not used as main test cases. We had to write a new suite of test cases to specifically test the type analysis. The new test suite tries to test every feature that we have implemented.

New test cases and their descriptions are given in Table 4.1.

Filename	Description
`8Ball.adb`	Magical 8 Ball Program with semantic errors
`Array.adb`	Basic array test
`Constant.adb`	Errors with constant assignment and assignment in declarations
`ErrorTest.adb`	Test with many different types of errors
`PackageTest.adb`	Lots of errors of different kinds
`ambiguious.adb`	Test with good and ambiguious ranges
`aggregate.adb`	Error with aggregates
`casetest.adb`	Tests both good and bad case statements
`constanttest.adb`	Tests constants
`exitTest.adb`	Tests exit statements and labels
`functionOverLoad.adb`	shows correct function overloading
`functionOverLoadBad.adb`	shows function overloading that can not be resolved
`iftest.adb`	test booleans and non boolean expressions in if statements
`labelTest.adb`	tests both good and bad use of labels
`Looptest.adb`	tests proper loops

Table 4.1: List of New Test Cases

Chapter 5

Cross Reference Example

Example Input

```
1 package Arrays is
2 type myarray is array(1..10) of integer;
3 a : myarray;
4 b : integer;
5 body
6 begin
7 declare
8 a:integer;
9 begin
10 b := a(1);
11 end;
12 end;
```

Example Cross Reference with Symbol Table and Type List

Please see Next Page

```
Processing main file
resolved: [ID b] scope 0, GenericType: GT_VARIABLE, Constant: 0, typeID: 1, returnType: 1,
complete: 1
 on line: 10
resolved: [ID a] scope 0, GenericType: GT_VARIABLE, Constant: 0, typeID: 7, returnType: 1,
complete: 1
ARRAY: constrained: 1
Range type: 1
 on line: 10

Symbol Table at 0x108f70
[ID b] scope 0, GenericType: GT_VARIABLE, Constant: 0, typeID: 1, returnType: 1, complete: 1
[ID a] scope 0, GenericType: GT_VARIABLE, Constant: 0, typeID: 7, returnType: 1, complete: 1
ARRAY: constrained: 1
Range type: 1
[ID myarray] scope 0, GenericType: GT_USER_DEFINED_TYPE, Constant: 0, typeID: 7, returnType:
1, complete: 1
ARRAY: constrained: 1
Range type: 1

Type: boolean, typeID: 4, returnType: 4, complete: 1
Type: float, typeID: 6, returnType: 6, complete: 0
Type: integer, typeID: 1, returnType: 1, complete: 1
Type: nonnegativeinteger, typeID: 3, returnType: 3, complete: 1
SUBTYPE: BaseType: 1
Type: positiveinteger, typeID: 2, returnType: 2, complete: 1
SUBTYPE: BaseType: 1
Type: string, typeID: 5, returnType: 5, complete: 0

Compilation Fineshed.
Number of Errors: 0
```

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998.