University of Toronto

CSC 488S Compilers & Interpreters **Winter 2015/2016**

The Abstract Syntax Tree

Introduction

This document describes the Abstract Syntax Tree (AST) data structure that has been designed[1] as an intermediate representation for the CSC 488S Course Project compiler. The AST has been designed with (just enough) information to be able to represent all possible programs in the source language.
In order to keep this document down to a manageable length, the actual definitions of the AST data structures are not included here, so this document should be read in conjunction with the AST source files and JavaDoc web pages.

The AST Classes

This document describes the overall structure and design rationale of the Abstract Syntax Tree (AST). For details of the fields and member functions provided by each class, see the online JavaDoc documentation.

AST

The AST interface defines the core operations that all AST nodes must include support for. Currently this includes pretty-printing the AST. you may find it useful to add new operations to support other features. There are several top-level classes used to define the structure of the tree:

BaseAST	The BaseAST class implements the AST interface and provides a default implementation.
ASTList	Holds a list of AST nodes, for keeping track of lists of things like expressions or statements. It is implemented by subclassing Java's built-in LinkedList class, and is itself an AST node.
Declaration	All declarations are subclasses of this class. It contains data common to all declarations.
Stmt	All statements are subclasses of this class. It is an empty placeholder.
Expn	All expressions are subclasses of this class. It is an empty placeholder.
Type	The subclasses of this placeholder class are the types of the language.
Readable	An interface for representing readable things.
Printable	An interface for representing printable things.

Printing the AST

Considerable effort was expended in the design and implementation of the AST data structures to facilitate printing the AST in a human readable form. The intention was to make it easier to develop and debug the AST data structures. You are strongly encouraged to maintain the printing mechanisms as you develop your version of the AST.

[1]The original AST design was done in C by Profs. Marsha Chechik and Dave Wortman. Converted to Java and significantly reorganized by Danny House. Final Java design by Profs. Dave Wortman and Marsha Chechik. Further enhancements in 2015 by Peter McCormick.

Expn

The Expn class is currently a placeholder. Its subclasses are the expressions in the language.
It has seven direct subclasses:

IdentExpn	A leaf expression consisting of an identifier, which is a reference to a scalar variable.
BinaryExpn	Abstracts the common features of binary expressions.
ConditionalExpn	Expressions of the form *expression* **?** *expression* **:** *expression*.
ConstExpn	All literal constant expressions.
FunctionCallExpn	A call to a function with or without arguments.
SubsExpn	All instances of array subscript references. The variable being subscripted is handled directly.
UnaryExpn	Abstracts the common features of unary expressions.

Subclasses of UnaryExpn

The UnaryExpn class is used for all expressions involving one operator and one operand. It has two subclasses:

NotExpn	Class for the Boolean **not** operator.
UnaryMinusExpn	Class for the negation (unary minus) operator.

Subclasses of BinaryExpn

BinaryExpn has four subclasses that distinguish various forms of expressions involving an operator and two operands:

ArithExpn	Arithmetic expressions where both operands must be integer values.
BoolExpn	Boolean expressions where both operands must be Boolean values.
EqualsExpn	Equality and inequality comparisons where both operands must be of the same type, but that type could be either integer or Boolean.
CompareExpn	Ordered comparisons (i.e., less than, greater than) where both operands must be integer values.

Subclasses of ConstExpn

This placeholder class is the superclass of the classes that represent the literal constants.
It has four subclasses:

BoolConstExpn	The Boolean constants **true** and **false**
IntConstExpn	All integer constants
SkipConstExpn	The pseudo-constant **newline** used in the **write** statement.
TextConstExpn	All text constants (strings).

Pretty Printing

Each node of the AST is capable of producing a textual representation according to the grammar of the source language. This is accomplished by each node implementing the `PrettyPrintable` interface (which the `AST` interface itself requires) and providing a definition of the `prettyPrint` method.

PrettyPrintable	Implemented by anything that can pretty-print itself.
PrettyPrinter	The operations which any pretty-printing output must support.
BasePrettyPrinter	A provided implementation of `PrettyPrinter` that can output to any Java `PrintStream`, or the system output.

Subclasses of Type

This placeholder class is the superclass of all data types in the language. It has 2 subclasses:

BooleanType	The type for Boolean expressions
IntegerType	The type for integer expressions.

Subclasses of Declaration

Declaration is the superclass for all declarations in the language. It has 3 subclasses:

RoutineDecl	Class for function and procedure declarations.
MultiDeclarations	Class for declaring multiple elements of the same type in one declaration (e.g., **integer** x , y , z)
ScalarDecl	Class for the declaration of a simple scalar variable

DeclarationPart

The class MultiDeclarations makes use of DeclarationPart, which holds just the name of an element declared, not its type. It has 2 subclasses:

ScalarDeclPart	Class for scalar variable declaration part
ArrayDeclPart	Class for array variable declaration part

Stmt

This is the superclass for all types of statements in the language. There is a subclass for each statement type.

AssignStmt	Assignment statements.
ExitStmt	Both forms of the **exit** statement
IfStmt	Both forms of the **if** statement.
LoopingStmt	The superclass for all loop building statements in the language.
ProcedureCallStmt	Class for procedure calls

ReadStmt	The **read** statement.
ReturnStmt	Both forms of the **return** statement.
Scope	The class for representing scopes.
WriteStmt	The **write** statement.

There are two subinterfaces of AST **Readable** and **Printable** that have been defined to assist in building the lists of printable things for the **write** statement and readable things for the **read** statement

Subclass of Scope

The Scope class has one subclass **Program** that is used to represent the entire program being compiled.

Program	The class representing the main program.

Subclasses of LoopingStmt

LoopingStmt is the superclass for all loop building statements in the language. It has 2 subclasses:

RepeatUntilStmt	The **repeat . . . until** statement.
WhileDoStmt	The **while . . . do** statement.

What the AST Doesn't Contain

There are several other fields that logically belong in the AST but because these fields depend on the design and implementation decisions made by each team, they are not included in the initial AST design. *Each team is free, and even encouraged, to modify the AST class hierarchy as required for their implementation*. Changes to the AST should be carefully documented as part of the documentation submitted with assignments. It is *strongly recommended* that you keep the `prettyPrint` and `toString` methods updated to correspond to changes that you've made in the AST.

Source Coordinate Field There is no mechanism in the AST as defined to keep track of source program coordinates for error messages during semantic analysis and code generation. Since the project compiler only deals with individual files, a line number field in the AST class might be sufficient depending on the granularity of error message reporting that the team implements.

Symbol Table Links The AST as defined keeps a `String` for each occurrence of an identifier. At some point during semantic analysis, symbol table entries will be built for most of these identifiers, so keeping direct references to the symbol table entries in the AST would make a lot of processing more efficient.

Type Tracking The AST classes describe the structure of expressions in the language, but they don't provide fields for keeping track of the type of expressions because this ultimately may depend on the declared type of identifiers used in expressions. You may find it convenient to add *isInteger* and *isBool* member functions or public variables to the subclasses derived from Expn.

Building the AST

The bottom up parsing scheme used for the course project is ideally suited to building the AST as the program is parsed. The overall strategy goes like this:

- A stack that runs parallel to the parse stack (accessed via RESULT, and colon tags in JavaCUP) is used to hold references to parts of the AST as it is being built.
- The leaves of the AST are constants and identifiers. The grammar rules that process these constructs should create and return instances of the appropriate AST classes.
- The bottom up parsing order guarantees that when the parser recognizes a construct, for example

  ifStatement = **if** *expression*:expn **then** *statement*:strue **else** *statement*:sfalse **end**

  the embedded expressions and statements have already been processed into AST subtrees. The colon tags *:expn* , *:strue* and *:sfalse* can be used to access these subtrees. The processing for this statement node simply requires building an AST *ifStmt* node with links to the appropriate subtrees. The node built for the *ifStatement* is returned by assigning it to the Java CUP pseudo variable *RESULT*.

Implementing Semantic Analysis and Code Generation

As forthcoming lectures will make clear, semantic analysis and code generation can both be performed by doing a depth first walk of the AST. This is one reason that abstract syntax trees are a popular choice for the compilers intermediate representation. Given the AST data structures that have been provided, there are several choices for implementing semantic analysis and code generation.

1. Implement a *Visitor pattern* with double dispatch as described in the course text book.
2. Build semantic analysis and code generation *into* the AST classes. For example, add member functions *doSemantics* and *doCodeGen* to each of the AST classes. Most of these member functions will be simple and easy to implement. If you are comfortable with recursive tree processing, this is probably the least effort approach.
3. Build a separate class that does the tree walk starting from the root of the AST. This approach might require changing some of the AST fields from private to public.
   Skeleton classes for this approach are provided as a part of the software packages for Assignments 3 .. 5, but it is *your choice* whether you use them.

Each team can choose the method used to implement semantic analysis and code generation.