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 488 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 class is currently empty. This, and the other placeholder classes, make it easier to overload methods based on the most important groups of AST subclasses. You may find it a useful place to add fields or methods that are common to all nodes in your design. It has six direct subclasses:

ASTList Holds a list of AST nodes, for keeping track of lists of things like expressions or statements. It is implemented using a LinkedList Object.

Expn All expressions are subclasses of this class. It is an empty placeholder.

Indentable This class holds the methods used for indenting the dump of an AST. Anything which needs to be indented when printed is a subclass of this class.

Type The subclasses of this placeholder class are the types of the language.

Readable The class representing readable things.

Printable The class representing printable things.

Printing the AST

Considerable effort in the design and implementation of the AST data structures was expended in making it easy to produce a human readable version of the data structure. Printing of the AST can be invoked using the dump flag (`-D`) provided by the compiler driver program. The human readable version of the AST is intended to make it easier to develop and debug the AST data structures. You are strongly encouraged to maintain this mechanism 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.

Expn

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

NotExpn	Class for the Boolean **not** operator.
SubsExpn	This AST treats array subscripting as a unary operator. The variable being subscripted is handled directly.
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).

Indentable

Some parts of a computer program (*e.g.*, if-then-else statements), are typically printed on several lines, or alone on a single line (*e.g.*, return statements). The AST classes that correspond to constructs that are indented when printed are all subclasses of Indentable. This class is used to make it easier to produce a human readable dump of the AST. It has 3 direct subclasses:

Declaration The superclass of all declarations.

RoutineBody This class is used to represent the body of a function or a procedure.

Stmt The superclass of all statements.

The Indentable class provides static methods for printing indented code fragments. It also provides a default `printOn` method that uses the `toString` method. Many of the subclasses of Indentable rely on the default `printOn` method. This Class manages the translation of the indentation depth into the correct spacing to document the hierarchal structure of the AST.

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., **var** x , y , z : **Integer**)

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	All 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 subclasses of AST **Readable** and **Printable** 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 data structure 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 `printOn` 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

  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.