INTRODUCTION

In multi-mode, jjtree will produce one AST class, called ASTproductionname, for each production. All the AST classes extend the SimpleNode class.

jjtree deduces how many subtrees each type of node has from the expansion of the production that generates it: it creates one 1 subtree for each non-terminal and adds them to the production node in the order in which is encounters them.

This will build the outline of a parse tree, since every production has a corresponding node in the parse tree. However, you will want to modify this behaviour for some productions in order to build a syntax tree instead of a parse tree. Here are some of the modifications you will want to make.

REMOVING UNNECESSARY PRODUCTIONS

As mentioned above, jjtree constructs a parse tree by default since every production has a corresponding node in the parse tree. However, ASTs typically do not include all the non-terminals of a grammar because some non-terminals are useful to describe the syntax of a programming language, but serve no further purpose after parsing and therefore do not need to be included in syntax trees.

For example, many grammars will include a statement definition like:

statement → if-statement | while_statement | etc.

However, because the specific statements (e.g. if-statement, while-statement, etc.) are completely self-contained, an AST typically does not include nodes representing generic statement because the nodes for specific types of statements contain all necessary information. As a result, the ASTstatement node should be removed from the AST to simplify the AST and speed up operations performed on it.

You can suppress the production of a AST node by declaring the node to be #void as follows: e.g for the example above.

```
statement() #void :
// the rest is the same
```

Note that for some languages, generic statements or generic expressions might be useful. For example, in languages with many binary operators, it might be useful to construct a generic ASTexpression node with 3 subnodes: the operator and the two operands. In this situation, it might be simpler to declare an ASTexpression interface or class with generic methods for all expressions.

ADDING MEANINGFUL TOKENS TO AST'S

As was stated earlier, by default, jjtree builds outlines of parse trees out of non-terminals in the grammar. Because most tokens in most programming languages are only used to describe the syntax of the language and do not contain otherwise meaningful information, jjtree does not include any tokens in the parse trees that it builds by default. As a result, the parse trees build by jjtree by default are missing all of their token leaves. Since most tokens would be discarded in the process of converting a parse tree into a syntax tree, this is a very reasonable decision.

However, some tokens such as identifiers and literals need to be included in ASTs, and therefore will need to be added to parse tree to convert it into a syntax tree. You can do this in 3 steps:

1) Replace them in the grammar by productions that represent them. For example, a new identifier () production could replace all occurrences of the token <IDENTIFIER> in the grammar and is declared as:

```
void identifier() :
{}
{
   <IDENTIFIER>
}
```

2) Capture the token. To do this, you simply need to declare a token variable in the production's declaration section, and then assign the matched token to that variable.

```
void identifier () :
{Token t;}
{   t=<IDENTIFIER>
}
```

3) Add the value of the token to the node under construction. In the scope of any production, the node under construction is called jjThis and can be accessed directly in Java. Java code can be interspersed in the JavaCC productions by surrounding it with { }. The method used to add the token to the node is jjSetValue().

```
void identifier () :
{Token t;}
{   t=<IDENTIFIER>
    {jjtThis.jjtSetValue(t.getValue());}
}
```

4) Use the SimpleNode.java provided in this course. In that file, toString() has been rewritten to include the token value in the string if there is one.

MAKING A PRODUCTION RETURN THE NODE IT HAS CONSTRUCTED

JJtree constructs ASTs from the bottom up: as nodes are being built, they are pushed onto a node stack. Then when a non-terminal has been fully recognized, all of its subnodes are popped from the stack, combined into a new node, and the result is pushed back onto the stack.

Therefore when a non-terminal is parsed, an entire AST will be constructed for that non-terminal. However, unless specifically instructed to do anything else, the constructed AST is simply pushed back onto the stack to be used by whichever production requested its production.

There are situations when it is desirable to have a production function return the node it has built, either to give it to the main program for further processing, or to return it to an intermediate production to allow that production to manipulate it directly.

To accomplish this, two changes to the structure of production functions are necessary:

1) The production function must return an AST node type instead of void. Typically, the return type is either set to SimpleNode (the general interface for all AST nodes) or the specific type of node for that production.

2) The production function must return the constructed node programmatically.

Here is an example of how the Start() production can be made to return an AST:

```
ASTStart Start() :
{}
{
  Expression() ";"
  { return jjtThis; }
}
```

Note that the Start production must construct a node in order to be able to return it, and therefore cannot be declared to be #void.

However, sometimes you may want to percolate an AST constructed in a subproduction without wrapping in into a new AST node. This is often the case with the starting production of an interpreted language which consists of a choice between expressions and statements. There are two approaches to deal with this issue:

1) The ASTexpression tree can be percolated from the Expression production without building an intermediate ASTstart tree as follows:

```
SimpleNode Start() #void:
{ASTexpression tree;}
{
   tree = Expression() ";"
   { return tree; }
}
// Expression() must now be modified to return an AST
```

2) If modifying the subproduction is difficult, then the AST it has built can be directly popped off the top of the AST stack in Java. The AST stack is called jjtree and the pop method called popNode(). jjtree is defined as a stack of Nodes; Node is the interface for all nodes. However the ASTs on jjtree are all SimpleNodes; SimpleNode is the superclass of all AST nodes and implements Node. SimpleNodes have more methods and are therefore better to work with than Nodes.

```
SimpleNode Start() #void:
{ }
{
   Expression() ";"
      { return (SimpleNode) (jjtree.popNode()); }
}
```

CREATING NEW NODES

Sometimes you will want to create a new type of node from a portion of a production.

For example, look at the following declaration:

declaration → type identifier ("," identifier)* ";"

Since a declaration typically consists of a type followed by a list of identifiers of that type, a good ASTdeclaration tree would have two subtrees, an ASTtype for the type, and an ASTidentlist for the list of identifiers. ASTidentlist could be created with a new identlist production, but this is not necessary. Here is the alternative:

```
void declaration():
{}
{
  type() (identifier() (";" identifier())*) #identlist ";"
}
```

Here, #name acts as a postfix operator. Its scope is the immediately preceding expansion unit. I.e. it was built from all non-terminals in the red parentheses (the identifiers) but the non-terminals before that were not included.

You can also use this technique with productions which regroup different kinds of ASTs. For example, this single loop production

loop → "while" condition() "{" body() "}" ";"
| "repeat" "{" body() "}" "until" condition() ";"

could generate two different ASTs as follows:

```
void loop() #void:
{}
{
  ("while" condition() "{" body() "}" ";")            #while
| ("repeat" "{" body() "}" "until" condition() ";")   #repeat
}
```

When one of the two alternatives has been fully parsed, all of its nodes on the AST stack are popped and combined into a new AST node (either ASTwhile or ASTrepeat) which is pushed onto the stack. Since loop has been declared as #void, this new node remains on the stack to be popped later.

This technique can also be used to "nodify" tokens which don't have a value without storing the token into the AST or to create null ASTs. For example:

```
 void type() #void:
{}
{
    (   <INT>   #typeINT
    |   <CHAR>  #typeCHAR
    |   {}      #notype
    )
}
```

CONDITIONAL NODE CREATION

Some productions will be such that you will only want to generate new nodes under certain conditions.

For example, the grammar for arithmetic expressions usually implements operator precedence as follows:

sum → product() ("+" product)*

In this situation you will only want to generate a ASTsum when at least one "+" sign has been scanned. i.e. when at least 2 product productions have been parsed.
This can be specified with a conditional node:

```
void sum() #void:
{}
{
  (product() ("+" product())*) #sum(>1)
}
```

In this example an ASTsum would be constructed from all the ASTproduct nodes on the AST stack only if there are at least 2 such nodes on the stack. Otherwise, the first ASTproduct node simply remains on the AST stack since the #void declaration inhibits popping the stack to produce a new AST. Therefore the node remaining on the stack after sum() finishes executing is an ASTproduct node.

PASSING NODES FORWARD INTO SUBPRODUCTIONS

Sometimes different productions have similar beginnings and are only differentiated after a few tokens have been parsed. Since LL(1) grammars are desirable, it is a good idea to left-factor such productions to avoid using lookaheads. As a result, the type of AST constructed will often be determined by a sub-production called later on in the expansion of the main productions.

For example, a production for the mysql ALTER statement contains some of the following expansions:

alter → "ALTER" "TABLE" table "ADD" "COLUMN" create_clause
| "ALTER" "TABLE" table "ADD" "INDEX" name (column ("," column)*)
| "ALTER" "TABLE" table "CHANGE" "COLUMN" column create_clause

Left-factoring this grammar to make it LL(1) would yield

alter → "ALTER" "TABLE" table (add | change)
add → "ADD" (add_column | add_index)
add_column → "COLUMN" create_clause
add_index → "INDEX" name (column ("," column)*)
change →"CHANGE" "COLUMN" column create_clause

This grammar contains 5 non-terminals to represent 3 meaningful AST's: ASTalter_add_column, ASTalter_add_index, and ASTalter_change. However, the productions for alter and add contain nodes that the 3 other subproductions need. Fortunately, these nodes are already on the AST stack, so they simply need to be reallocated from the alter and add productions to the other 3 subproductions.

This is accomplished in two steps:

1) Alter and add must be inhibited from using their own subnodes for table, by declaring these productions as #void
2) The other 3 productions must be allowed to use nodes on the AST stack that are not rightly theirs. This is done by increasing the number of nodes that they are allowed to pop:

```
void alter() #void:
{}
{
   "ALTER" "TABLE" table() (add() | change())
}

void add() #void:
{}
{
  "ADD" (add_column() | add_index())
}

void add_column() #void:
{}
{
  "COLUMN" create_clause() #ASTalter_add_column(2)
}

void add_index() #void:
{}
{
   "INDEX" name (column() ("," column())*) #columns
          #ASTalter_add_index(3)
}

void change() #void:
{}
{
  "CHANGE" "COLUMN" column() create_clause()  #ASTalter_change(3)
}
```