Análisis sintáctico

Compiladores e intérpretes

Daniel Murrillo Porras 200854763
Jeffry Torres Alvarez 200422424

11/04/2014

Introducción

El proceso de reconocer la estructura del lenguaje fuente se conoce con el nombre de análisis sintáctico (parsing). Para nuestro proyecto se hará de forma descendente LL(k) con ayuda de la herramienta JavaCC.

La principal tarea del analizador sintáctico no es comprobar que la sintaxis del programa fuente sea correcta, sino construir una representación interna de ese programa y en el caso en que sea un programa incorrecto, dar un mensaje de error.

El analizador sintáctico comprueba que el orden en que el analizador léxico le va entregando los tokens es válido. Si esto es así significará que la sucesión de símbolos que representan dichos tokens puede ser generada por la gramática correspondiente al lenguaje del código fuente.

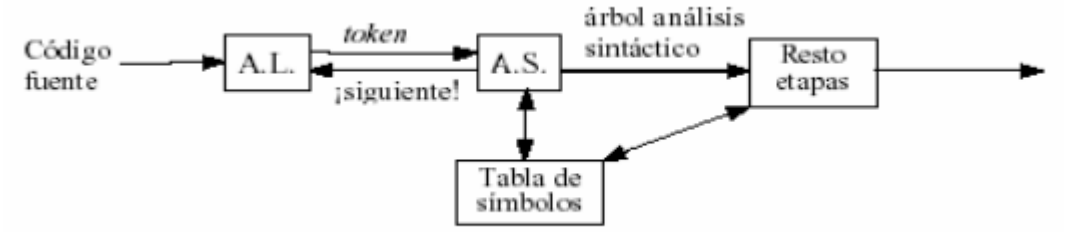


Fig. 1

JavaCC es una herramienta que generalmente se utiliza para generar analizadores léxicos y sintácticos (parsers) de una forma muy cómoda, actualmente se encuentra mantenida por Sun Microsystems y es muy robusta, eficiente y fiable. Que facilitará el proceso de elaboración del *parser*.

Soluciones de implementación

Para el desarrollo del scanner y parser, se configura el archivo .jj de la herramienta *javaCC* donde se establecen los *tokens* que se van a utilizar y las reglas de la gramática que se va a utilizar.

PARSER_BEGIN: aquí se agrega nuestro código que se quiere adicionar al archivo .jj. se llama a *parser.goal();* quien inicializa el scanner y parseo y retorna un árbol heterogéneo de las clases de nuestro AST del recorrido de un código fuente específico.

SKIP: Son todos los token que se quieren ignorar, ej. Los espacios en blancos.

SPECIAL_TOKEN: Aquí se configura para ignorar los código o bien para reconocer los comentarios (incluye el comentario anidado) y no afecten en el recorrido y elaboración del parser.

```
45  SPECIAL_TOKEN : /* COMMENTS */
46  {
47    <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
48  | <FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
49  | <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
50  }
```

TOKEN: Se agregan todas las palabras reservadas de las gramática que se están utilizando.

```
68  | < BOOLEAN: "boolean" >
69  | < CLASS: "class" >
70  | < INTERFACE: "interface" >
71  | < ELSE: "else" >
72  | < EXTENDS: "extends" >
73  | < FALSE: "false" >
74  | < IF: "if" >
75  | < WHILE: "while" >
76  | < WHITCH: "switch" >
77  | < INTEGER: "int" >
78  | < LENGTH: "length" >
79  | < MAIN: "main" >
```

GoalAST Goal(): Es el primer método llamado para inicializar el recorrido y elaboración del árbol AST, en este caso se llama a cada uno de los no-terminales de acuerdo a *token* que lee.

Para la solución de la recursividad se establecieron *ArrayList* donde es almacenado cada uno de los hijos que tiene la repetición.

Por cada vez que entra a la repetición crea el hijo y al final lo agrega a la lista.

JJ solución de las repeticiones	**gramática**

```
249 ClassDeclarationAST ClassDeclaration() :
250 {
251     IdentifierAST id=null;
252     VarDeclarationAST vd=null;
253     ArrayList<VarDeclarationAST> vdlist = new ArrayList<VarDeclarationAST>();
254     MethodDeclarationAST md=null;
255     ArrayList<MethodDeclarationAST> mdlist = new ArrayList<MethodDeclarationAST>();
256 }
257 {
258   "class" id=Identifier()
259     "{" (
260           vd=VarDeclaration()
261           {vdlist.add(vd);}
262         )*
263         (
264           md=MethodDeclaration()
265           {mdlist.add(md);}
266         )*
267     "}"
268     {return new ClassDeclarationAST(id,vdlist,mdlist);}
269 }
```

ClassDeclaration ::= “class” Identifier “{”

(VarDeclaration)* (MethodDeclaration)* “}”

Retorna la estructura elaborada de acuerdo a sus hijos e identificadores que posee su constructor.

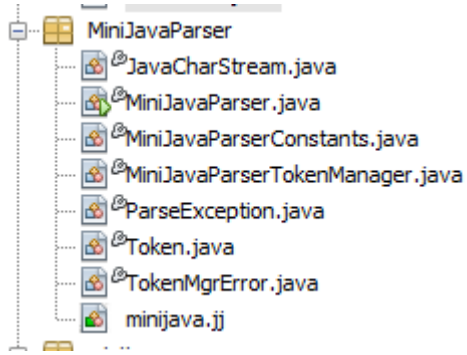
Lockahead: Facilitó a eliminar errores de ambigüedad en el parser. Observa *k* tokens adelante para decidir a qué hijo recorrer. Conforme incrementa *k*, es mayor la complejidad, por eso es necesario establecer cuando es necesario observar hacia adelante y mejorar la eficiencia.

```
477     LOOKAHEAD("{" "}")
478     a=Block()
479     {return a;}
480   |
481     LOOKAHEAD(Identifier() "=")
482     b=AssignmentStatement()
483     {return b;}
484   |
485     LOOKAHEAD(Identifier() "[" Expression() "]" "=" )
486     c=ArrayAssignmentStatement()
487     {return c;}
488   |
489     LOOKAHEAD("if" "(" Expression() ")" Statement() "else" Statement() )
490     d=IfElseStatement()
491     {return d;}
492   |
493     LOOKAHEAD("if")
494     e=IfStatement()
495     {return e;}
```

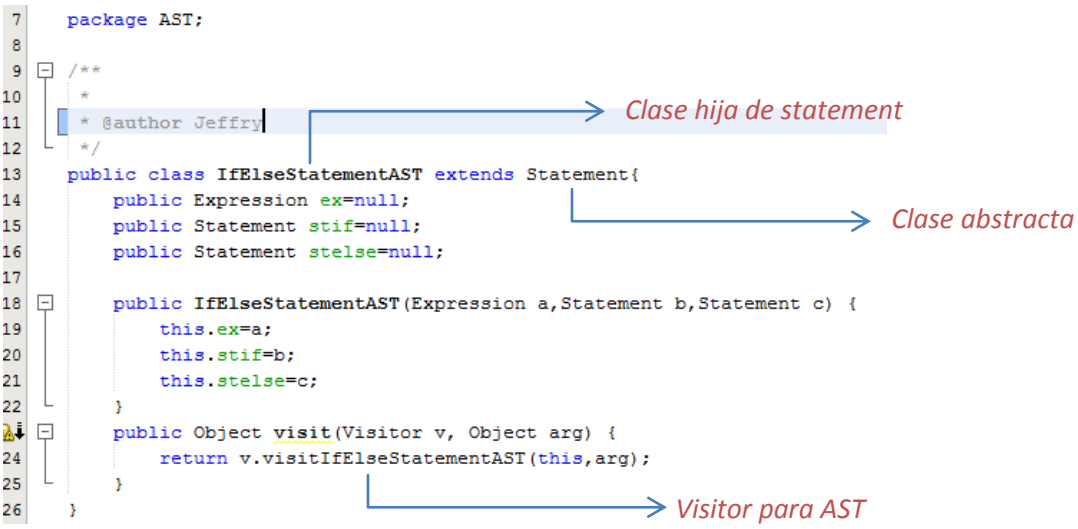
Token.image se obtiene el valor de un identificador, Integer_literal, y los demás.

```
879  IntegerLiteralAST IntegerLiteral() :
880  {}
881  {
882    <INTEGER_LITERAL>
883      {return new IntegerLiteralAST(token.image);}
884  }
```

Una vez compilado y corrido el archivo .jj genera automáticamente las clases necesarias para el recorrido (parser, tokens, errores, scanner, etc...).

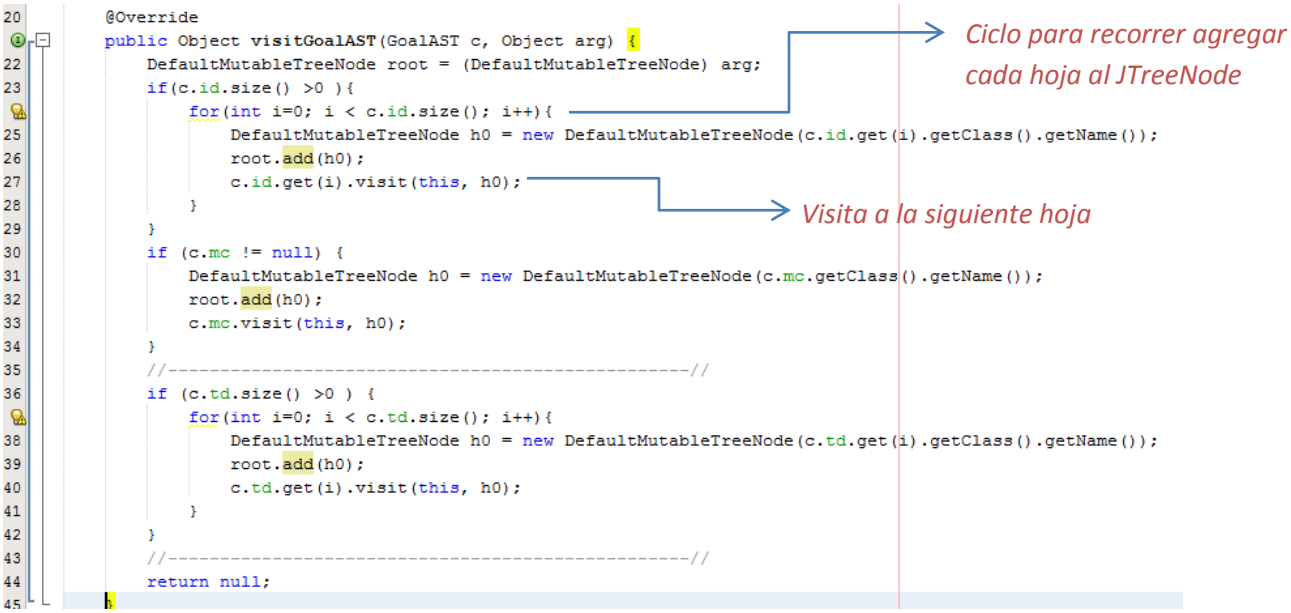


Dentro del package *AST* se encuentran todas las clases necesarias para la elaboración del árbol de tipo AST. Incluye clases abstractas como padres, que dan herencia a otras.



Cada una incluye un *visitor*, que es una interface genérica para recorrer el AST ya que con una sola corrida es imposible elaborarlo, se necesita retroceder y para recorrer cada uno de los hijos, y los hijos de los hijos.

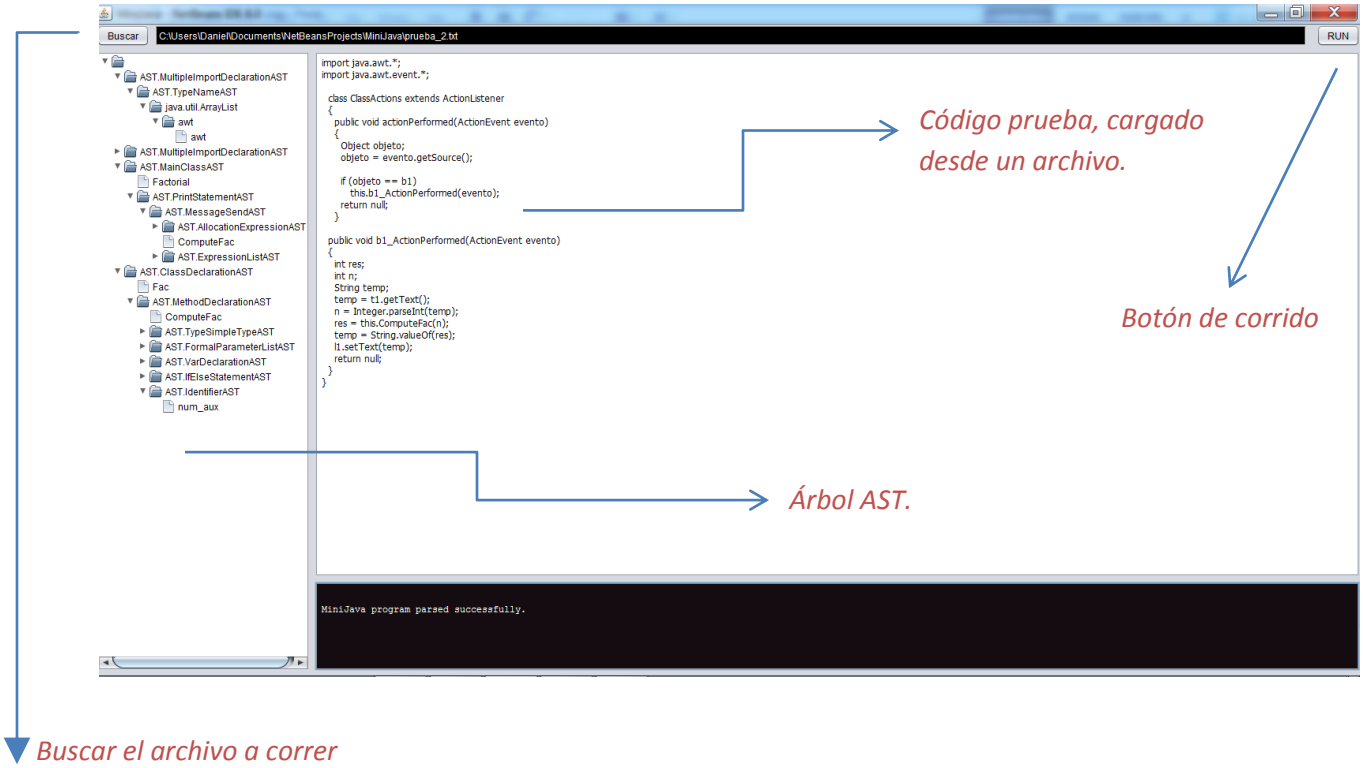
Para imprimir el árbol se requiere de un método llamado *ASTPRINT*, que implementar el *visitor* para recorrer cada uno de los hijos e irlos agregando a al componente *DefaultMutableTreeNode* y poder ser visto en la interfaz gráfica.



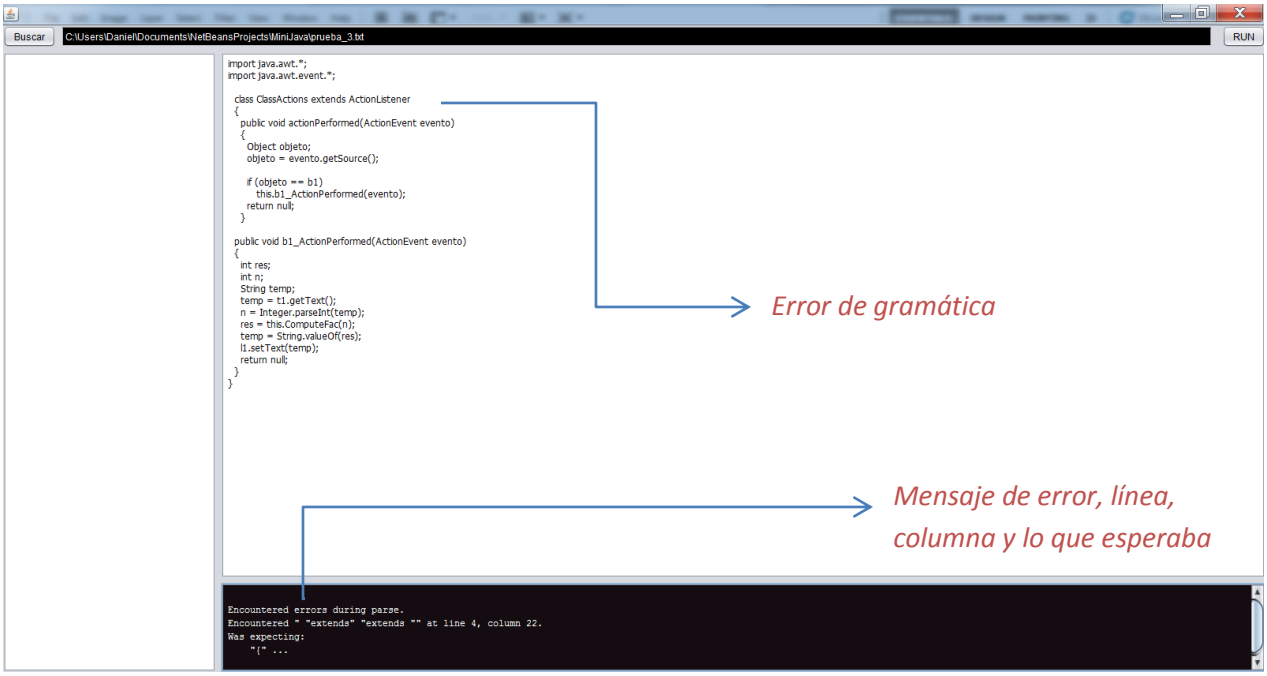
Resultados obtenidos

Objetivos	Resultados
Elaborar correctamente la gramática a través de la herramienta JavaCC.	Completado.
Mostrar los errores de gramática en la interfaz del usuario, en la línea y columna	Completado.
Elaborar las clases del AST para la creación del árbol.	Completado.
Imprimir el árbol en un componente para el usuario utilizando el visitor.	Completado.
Crear una intefaz agradable	Completado.
Agregar a la gramática la sentencia del *switch*	Completado.

Códigos de prueba



Buscar el archivo a correr



Bibliografía

Java Compiler Compiler. Java Generator Parser. https://javacc.java.net/doc/javaccgrm.html (visto la última vez 11/04/2014.

Ejemplo de JavaCC http://cs.lmu.edu/~ray/notes/javacc/