Herramientas para el Desarrollo de Software Embebido para Aplicaciones Aeronáuticas.

Resumen:

El desarrollo y mantenimiento de software aeronáutico, y especialmente el software utilizado a bordo de aeronaves, constituye una tarea de enorme complejidad. Debido a los prolongados ciclos de vida que caracterizan a estas aplicaciones, muchas veces se encuentran desarrolladas en lenguajes de programación para los cuales no existen las sofisticadas herramientas disponibles para lenguajes más actuales. Ello provoca que los desarrolladores deban utilizar a menudo herramientas ad-hoc, de baja reusabilidad.
El objetivo es mejorar el proceso de desarrollo de aplicaciones aeronáuticas mediante la implementación de herramientas reusables, que faciliten la producción de software de alta calidad.

Palabras Claves: Jovial, Lenguajes de Programación, Análisis de Código, Parser, Eclipse

Tools for Development of Embedded Software for Aeronautic Applications

Abstract:

Developing and mantaining aeronautic software, such as on board software, is a very hard and complex task. Because of the long lifecycles of these applications, they are often developed in programming languages lacking the sophisticated tools that are available in today languages. Therefore, the software developers are forced to use ad-hoc, non reusable tools.
The goal of this project is to improve de development process of aeronautic applications through the implementation of reusable tools, aimed to facilitate the production of high quality software.

Key words: Jovial, Programming Languages, Code Analisis, Parser, Eclipse.

Arias. Silvia E.[13], Montes. A. Miguel [12], Banchio. Enrique G. [12], Gonzalez Kriegel. Bernardo J.[14], Muñoz Gabriel J. [12]

1. Departamento Tecnologías de Información. Instituto Universitario Aeronáutico.
2. Prosecretaría de Informática. Universidad Nacional de Córdoba.
3. Facultad de Ciencias Exactas, Físicas y Naturales. Universidad Nacional de Córdoba.
4. Facultad de Matemática Astronomía y Física. Universidad Nacional de Córdoba.

sarias@iua.edu.ar, mmontes@iua.edu.ar, ebanchio@iua.edu.ar, bgk@iua.edu.ar, gmunoz@iua.edu.ar

Introducción

Desde sus orígenes la programación se ha revelado como una actividad de enorme complejidad. Esto es particularmente cierto en el caso de aplicaciones críticas, en las que las eventuales fallas implican altísimos costos, tanto humanos como económicos.

Las aplicaciones aeronáuticas constituyen un caso paradigmático. En tales aplicaciones, además de la complejidad del desarrollo de software se combinan las dificultades propias de las aplicaciones de tiempo real y de las de sistemas embebidos. Por ello, no sólo este tipo de aplicaciones es una en la que las fallas pueden tener un costo humano y económico elevado, pero además se presenta como un área particularmente proclive a las fallas de software.

En el desarrollo de aplicaciones aeronáuticas se utilizan compiladores que generan software que serán utilizadas en arquitecturas diferentes a las que son desarrollados.

En la especificación de los lenguajes de programación existen características de éste lenguaje de programación que delegan decisiones de implementación al compilador que se utilice. Si los programadores no tienen en cuenta estas características, programas bien escritos (o legales) pueden tener comportamiento diferente al esperado de acuerdo al compilador que se use.

Para el seguimiento de tales procesos, sin embargo, es necesario contar con herramientas que asistan al programador. La carencia de las mismas suele conducir a los desarrolladores a elaborar herramientas ad-hoc que permiten salir circunstancialmente del paso hasta descubrir que una nueva herramienta ad-hoc debe ser elaborada desde cero. En la práctica esto implica desviar recursos humanos capacitados en el desarrollo de software aeronáutico para el desarrollo de herramientas que requiere otro tipo de capacitación. El resultado es mayor demora en el desarrollo del software aeronáutico y obtención de herramientas de baja confiabilidad y casi nula reusabilidad (herramientas ad-hoc).

Con el paso del tiempo los lenguajes de programación han evolucionado, tienen nuevos ambientes de desarrollo que ayudan a disminuir su tasa de errores, el tiempo de desarrollo y se crean nuevos compiladores que generan código de mejor rendimiento. Por ello, las organizaciones dedicadas al mantenimiento y desarrollo de software para tales aplicaciones utilizan procesos de desarrollo destinados a evitar tales fallas.

En el caso particular de las aplicaciones aeronáuticas las herramientas y los lenguajes que se utilizan son implementaciones que tienen ciclos de vida muy prolongados. Aunque en muchos otros sectores se hayan dejado de usar estos lenguajes, en el sector aeronáutico están actualmente en vigencia como es el caso de JOVIAL, lenguaje creado en el año 1959.

En definitiva, se hace necesario desarrollar herramientas más genéricas, que faciliten la tarea de los desarrolladores, y que permitan la generación de código de calidad y libre de errores. Ese es exactamente el problema que tratamos de resolver, apuntando a generar herramientas de este tipo, de aplicación en el desarrollo de software aeronáutico. Concretamente con este trabajo se genera una herramienta que ayuda a detectar los lugares del código fuente cuya interpretación pueda depender del compilador.

Estado del arte:

En los lenguajes de programación actuales, estas herramientas se encuentran muy difundidas. Mediante su uso, el desarrollador puede concentrarse en el problema a resolver, en vez de luchar con particularidades del lenguaje utilizado. Un ejemplo de estas herramientas es Eclipse. Eclipse tiene una estructura basada en plugins, lo cual le permite ser adaptable a distintos lenguajes, pero su uso es predominante en el desarrollo

con Java. En este caso, brinda capacidades tales como detección de errores de sintaxis, e incluso posibles errores lógicos (uso de variables no inicializadas, declaración de variables que luego no se utilizan, código inalcanzable, etc.). También facilita la tarea de edición de los programas mediante el uso de sugerencias, y generación de código para casos típicos o para corrección de errores. Sin embargo, su uso en el tipo de lenguajes utilizados en plataformas aeronáuticas es mucho más acotado. Esto se debe a los largos ciclos de vida que caracterizan a estas aplicaciones, que provoca que las herramientas disponibles se encuentren muy por detrás del estado del arte en lenguajes más modernos.

Breve descripción y reseña del lenguaje JOVIAL:

JOVIAL es una sigla de Jule's Own Version of International Algebraic Languaje, su primera versión fue en el año **1959** en **USA**, es un lenguaje de programación de segunda generación. Esta basado en el lenguaje **IAL** (**ALGO 58**), con extensiones para la programación en tiempo real a gran escala. Tiene repositorios comunes compartidos a tiempo de ejecución, llamados **COMPOOL**, que utilizan todos los programas que se están ejecutando en el sistema para hacer manejo centralizado de las comunicaciones entre ellos. Las estructuras de datos que pueden compartirse son variables y tablas.

JOVIAL se usa extensivamente en la fuerza aérea de Estado Unidos, la mayoría del software para **AWACS**[1] esta escrito en **JOVIAL**, como los **AOCP**[2] que se ejecutan en equipos compatibles con **IBM 360**[3].

Influenciado por el lenguaje **IAL**, **JOVIAL** ha evolucionado a partir del lenguaje **CLIP** incorporando características de **COMPOOL** y **MAD**. A su vez **JOVIAL** influyó en los lenguajes **CMS-2**, **CORAL**, **MPL**, **VS BASIC**. **BASIC** es un subset de **JOVIAL**. **IPLT-1** se escribió utilizando **JOVIAL** y **JTS** implementa el dialecto de **JOVIAL**.

Los lenguajes **JOVIAL J2**, **SPL** y **SYMPL** son evoluciones de **JOVIAL**.

En el año 1958 se crea el lenguaje **JOVIAL** y sus principales características son:

Basado en el lenguaje **IAL** se incluyen características para realizar programación en tiempo real a gran escala. **Y** un repositorio común de para compartir información entre todos los programas en tiempo de ejecución llamado *COMMunications POOL (COMPOOL).* Basado en el lenguaje **MAD** se incluyen características para soportar modos, variables, registros y tablas.

Con el paso del tiempo **JOVIAL** fue evolucionando, en el año 1960 se le incorporó un intérprete del lenguaje donde sus principales características fueron: Convertir programas escritos en **JOVIAL** a un lenguaje intermedio. Mecanismos para generar ambientes de pruebas. Ejecutar programas interpretados en una **IBM 209**. Imprimir en papel o en pantalla el estado del programa en un momento dado

El propósito de este intérprete era monitorear la ejecución del programa mostrando la instantánea actual o su traza. Con estos cambios se lo considera como un lenguaje de programación de alto nivel y aspira ser un lenguaje independiente de una arquitectura específica.

En el año 1961 el lenguaje es independiente de la arquitectura, es un lenguaje orientado a procedimientos y las características que el lenguaje posee son: Delimitadores: como por ejemplo las comillas simples, paréntesis, corchetes. Identificadores: nombres de variables, procedimientos, ítems, tablas y archivos. Fórmulas: clasificadas al tipo de valor que retornan (numéricas, literales, de estado, binarias y de indicadores). Tipos de datos soportados: numérico, literal, información de estado y binario.

[1] AWACS: Airborne Warning and Control System.
[2] Airborne Operational Computer Program.
[3] Sitio oficial de las especificaciones y arquitectura: http://www.research.ibm.com/journal/rd/441/amdahl.pdf

Constantes: toman valores en cualquiera de los tipos de datos soportado. Sentencias: ordenamiento de delimitadores y cláusulas. Algunos ejemplos son las declaraciones básicas, las sentencias básicas, las sentencias for, entre otras. Interruptores: la sentencia switch entre otras. Procedimientos. Mecanismos de Entrada / Salida y manejo de Archivos.
Acompañando esta evolución **JOVIAL** posee implementaciones en docenas de arquitecturas de hardware diferentes.
En el año **1962** el lenguaje evoluciona al **Jovial 2** y luego en **1967** a **JOVIAL 3**, en **1973** se crea el estandar **J73** que es el que actualmente sigue en vigencia. Este trabajo fue desarrollado con la especificación del stándar **J73**.

Selección de herramienta

Existen diversas herramientas de metacompilación, en el presente trabajo se ha centrado la atención en las que generan parsers escritos en lenguaje Java dado que permite generar una herramienta multiplataforma. Se focalizó en dos de ellas, una que genera parsers descendentes, y otra que genera parsers ascendentes. Para cada una de ellas se han estudiado su característica y comportamiento y se detallan a continuación:
Se pueden realizar dos tipos de análisis, el descendente y el ascendente. Los métodos ascendentes se caracterizan porque analizan la cadena de componentes léxicos de izquierda a derecha, obtienen la derivación más a la derecha y el árbol de derivación se construye desde la raíz hasta las hojas. Éste análisis es utilizado por Java-Cup, mediante métodos de recursión a derecha (LR[1]).El análisis descendente permite la utilización de gramáticas más generales, tiene facilidad para depurar el código generado, habilidad de parsear cualquier no Terminal de la gramática, y la capacidad de pasar atributos en el árbol sintáctico durante el parsing. Este análisis es utilizado por JavaCC.Los métodos de análisis sintáctico LR permiten reconocer la mayoría de las construcciones de los lenguajes de programación. Son métodos muy potentes, con mayor poder de reconocimiento que los métodos LL, las gramáticas LL son un subconjunto de las gramáticas LR.
El análisis léxico es la primera fase de un compilador, dicha fase es responsabilidad del analizador léxico, cuya función principal es leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos llamada secuencia de tokens, que utiliza el parser para llevar a cabo el análisis sintáctico. En el caso de JavaCup es posible incorporar analizadores léxicos generados automáticamente por JLex u otros generadores de analizadores léxicos que generen código Java. Esto permite mayor modularidad. Cuando se utiliza JavaCC, las tareas involucradas en la etapa de análisis léxico son asignadas al Token Manager que se genera a partir de la especificación léxica que forma parte del archivo de configuración de JavaCC.
Para generar un parser para un lenguaje determinado, se necesita tener como entrada la especificación de dicho lenguaje. La especificación se define por medio de una serie de cláusulas que inicializan algunos parámetros del parser, los tokens del lenguaje, declaración de terminales y no terminales, y por último la sintaxis del lenguaje, siempre siguiendo una convención sintáctica que a su vez se define por medio de una gramática.
Ambas herramientas tienen incorporadas las funcionalidades necesarias para realizar estas tareas.
JavaCup no incorpora nativamente un analizador léxico y utiliza analizadores externos que generan automáticamente código Java.

[1]L: Procesamos la cadena de entrada de izquierda a derecha. R: Proporcionan la derivación más a la derecha de la cadena de entrada en orden inverso

En éste trabajo se utilizará JLex como analizador léxico, al ser estas herramientas diferentes, cada una tiene su propio archivo de configuración.
En JavaCC tanto las especificaciones léxicas como las gramaticales se escriben en el mismo archivo, de esta manera la gramática puede ser leída y mantenida fácilmente gracias al uso de las expresiones regulares dentro de la gramática. JavaCup tiene una parte en su archivo de configuración para definir la precedencia de los Tokens.
JavaCC permite extender especificaciones BNF en las especificaciones léxicas y gramaticales, mediante la utilización de expresiones regulares, tales como (A)*, (A)+ para la implementación de precedencia. En JLex cada regla léxica, está precedida por una lista de estados léxicos y las transiciones de estados se realizan por la invocación de la función yybegin(state). JavaCC también ofrece estados léxicos y la capacidad de agregar acciones léxicas. Estos estados permiten trabajar con especificaciones más claras, a la vez que permiten mejor manejo de mensajes de error y advertencias.
En JavaCC se necesitan gramáticas no ambiguas y sin recursividad por la derecha, restricción que no posee JavaCup por utilizar analizadores sintácticos ascendente.
En el archivo de configuración de JavaCup llamado Main.java se pude habilitar la opción "-dump" que produce un informe de la gramática que es capaz de ser leído por un humano. En éste informe se listan los estados construidos necesarios para resolver conflictos de parsing y las tablas de parsing. JavaCC incluye una herramienta llamada JJDoc que convierte los archivos de la gramática en archivos de documentación.
En Java-CUP todos los tokens son definidos y tratados de la misma forma. En JavaCC los tokens, que en la especificación léxica se definen como tokens especiales, durante el parsing son ignorados, aunque se hallan disponibles para ser procesados por la herramienta. El procesamiento de comentarios, es una de las aplicaciones más comunes que posee la definición de tokens especiales.
El alfabeto de JLex es el conjunto de caracteres ASCII, entre los códigos 0 y 127 inclusive. El analizador léxico de JavaCC puede manejar entradas Unicode, y las especificaciones léxicas también pueden incluir cualquier carácter Unicode.
En JavaCC las especificaciones léxicas pueden definir tokens de manera tal que a nivel global no se diferencien las mayúsculas de las minúsculas en la especificación léxica completa, o en una especificación léxica individual. En JLex se tiene que especificar en cada token.
En JavaCup el usuario define su estructura de árbol ad-hoc. JavaCC incluye JJTree que es un preprocesador para el desarrollo de árboles.
Resultado de la elección de la herramienta: Las dos herramientas presentan las carácterísticas que se necesitan para el desarrollo de este trabajo y su posible reutilización. Se utilizará JavaCup porque tiene análisis ascendente y precedencia.

Análisis de aspectos léxicos del lenguaje y su aplicación en la herramienta seleccionada

En este trabajo se utilizará la herramienta generadora de lexer llamada **Jlex. Jlex** para la construcción de los token utiliza un archivo de configuración que tiene tiene tres secciones separadas por los caracteres **$% %$**. Las secciones son:
Código de usuario: El código que se escribe debe ser código java correcto y se copia directamente sin modificaciones al principio del fichero analizador Java.
Directivas JLex: En éste lugar se pueden declarar muchas cosas, por ejemplo macros.
Reglas para las expresiones regulares: Esta sección consiste en reconocer la secuencia de caracteres de entrada y asociarla a los tokens correspondientes. Estas reglas especifican expresiones regulares y les asocian acciones de código Java.

Como ejemplo transcribimos la definición de un nombre de variable y un número flotante en la especificación MIL-STD-1589C del lenguaje Jovial y su correspondencia en Jlex.

Definición de un nombre de variable

Definición en el lenguaje Jovial. MIL-STD-1589C 8.2.1

```
<name> ::= <letter-or-$><letter-digit-$-or-prime>...
<letter-or-$> ::= <letter>
                | $
<letter-digit-$-or-prime> ::= <letter>
                | <digit>
                | $
                | '
<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

Correspondencia en JLEX

```
[$A-Z][$'A-Z0-9]* { return new Symbol(sym.VAR, new String(yytext())); }
```

Definición de un número flotante en forma fraccional

Definición en el lenguaje Jovial. MIL-STD-1589C 8.1

```
<floating-literal> ::= <digit> ...
        | <fractional-form>
<fractional-form> ::= <digit>... .
        | [<digit>...] . <digit>...
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

En esta

Correspondencia de <fractional-form> en JLEX

```
[0-9]*[.][0-9]* { return new Symbol(sym.NUM, new FlotanteJovial(yytext())); }
```

Para determinar cuales son los tokens del lenguaje y su implementación con la herramienta seleccionada, se ha tomado como definición del lenguaje JOVIAL el estándar del Ministerio de Defensa de los Estado Unidos: MIL-STD-1589C USAF del 6 de Julio de 1984, JOVIAL (J73), en este documento las secciones están organizadas con una metodología top-down [2]. La primera sección describe los conceptos globales y luego en el siguiente orden: las declaraciones, procedimientos y funciones, sentencias, formulas, referencias de datos, correspondencia entre tipos y conversiones, elementos básicos y directivas.

La mayoría de las secciones están divididas en tres partes: “sintaxis”, “semántica” y “restricciones”. En la parte de sintaxis se define la gramática del lenguaje con notación BNF modificada. En la parte de semántica se define el significado de construcciones que satisfacen la sintaxis y las restricciones. Las restricciones enumeran los requerimientos no sintácticos que deben ser cumplidos para dar construcciones correctas. Los símbolos terminales son escritos en mayúsculas y los no-terminales son encerrados entre <y>. Se dispone la definición de los elementos básicos entre ellos encontramos: caracteres / variables / literales / comentarios / blancos.

Análisis de los aspectos sintáctico del lenguaje y su aplicación en la herramienta seleccionada

Para implementar las instrucciones del lenguaje JOVIAL en la herramienta seleccionada, el generador de parser utilizado es JavaCup, se caracteriza, como su nombre lo indica, por generar parsers escritos en lenguaje Java. Para generar un parser para un lenguaje determinado, CUP necesita tener como entrada la especificación de dicho lenguaje.

La sintaxis de las especificaciones de gramáticas CUP puede considerarse dividida en cuatro partes diferentes:

[2]Top-down: de lo general a lo particular

Las declaraciones preliminares permiten especificar cómo debe generarse el parser y también proveer código para ser ejecutado en tiempo de ejecución (todas estas opciones pueden estar definidas o no). Las opciones package e import, se utilizan de la misma manera que en un programa Java 1.5 normal.
La primera sección de declaración action code permite incluir el código de una clase no pública separada que contiene todas las rutinas y variables que serán necesarias para el código asociado a la gramática. La declaración tiene la forma: *action code* {: ... :}; donde {: ... :} es un texto de código cuyo contenido será ubicado directamente en la declaración de la clase *action class*.
La opción *parser code* permite especificar métodos y variables directamente en la clase parser generada. La declaración tiene la forma: *parser code* {: ... :}; cuyo código se incluye directamente en el código de la clase parser generada. Esta opción será utilizada para inicializar las variables y definir el método main.
La opción *init with* es un código que inicializa el estado del parser antes de comenzar con el escaneo de los tokens. Normalmente la inicialización se refiere al scanner, las tablas y demás estructuras de datos que podrían ser necesarias para la ejecución de las acciones semánticas. La declaración tiene la forma: *init with* {: ... :}; donde el código pasa a conformar el cuerpo de un método void miembro de la clase parser.
La opción *scan with* indica la manera en la que el parser debe obtener el siguiente token del scanner. La declaración tiene la forma: *scan with* {: ... :}; donde el código pasa a conformar el cuerpo de un método de la clase parser, que retorna un objeto de tipo java_cup.runtime.Symbol, por lo cual el código incluído debe retornar un valor de este tipo.
En la parte de declaración de terminales y no terminales se define la lista de símbolos que se declaran en la especificación, sirve para nombrar y definir el tipo de cada uno de los símbolos terminales y no terminales que aparecen en la gramática. El nombre de cada uno de los símbolos debe ser diferente de las palabras reservadas de JavaCUP. Cada símbolo se representa en tiempo de ejecución como un objeto de tipo Symbol.
En la definición del terminal se debe especificar la clase del valor contenido en cada Symbol devuelto por el JLex. En la definición del no terminal se debe especificar la clase del valor que se utilizará en el parser. En caso de no tener especificada la clase del valor, tanto para los terminales como para los no terminales, implica que no tiene un valor.
La parte de especificación de la precedencia y asociatividad de los terminales: se utiliza cuando se trabaja con una gramática ambigua, justamente para desambiguar las expresiones, por medio de la precedencia y la asociatividad, es decir que se utiliza para resolver los conflictos de ambigüedad donde por medio de la definición gramatical se podrían seguir dos cursos de análisis, este conflicto se lo llama conflicto desplaza/reduce.
Hay tres clases de asociatividad, y pueden ser “a izquierda”, “a derecha” o “no asociativo”.
Los terminales que no aparecen en las declaraciones de precedencia y asociatividad se tratan con la más baja precedencia.
La parte de especificación de la sintaxis del lenguaje es la funcionalidad del parser y se escriben con sintaxis BNF. Esto es un no “terminal” en el lado izquierdo seguido por “::=”, el lado derecho compuesto por cero o más ``terminales" o ``no terminales", acciones asociadas, terminando con punto y coma (“;”).
Las acciones asociadas a las producciones se escriben en código Java como texto de código entre los delimitadores {: ... :}. Estas acciones son ejecutadas por el parser cuando la porción de la producción a la izquierda de la acción ha sido reconocida.

Si un no terminal tiene varias expansiones diferentes, pueden ser declaradas juntas con una barra vertical (“|”) que las separa y el conjunto total de expansiones debe finalizar con el punto y coma (“;”).

Como se describió en el punto anterior, en la definición del lenguaje JOVIAL MIL-STD-1589C 2.1.1, la sintaxis esta descripta con notación BNF modificada, esto ayuda mucho dado que en la especificación de sintaxis en JavaCup también se utiliza notación BNF.

Declaraciones preliminares: permiten especificar cómo debe generarse el parser, y también proveer trozos de código para que sean ejecutados en tiempo de ejecución.

Declaración de Terminales y No Terminales, con su respectiva clase de objeto asociada: en este caso, los terminales y no terminales pueden o no tener un tipo asociado, y en caso de no poseer un tipo especificado, significa que no tienen ningún valor asociado. Por el contrario, si se especifica un tipo determinado, el terminal o no terminal debe tomar un valor dentro del tipo declarado.

Especificación de la precedencia y asociatividad de los terminales, en la que la última declaración de precedencia proporciana a sus terminales la más alta precedencia.

Especificación de la sintaxis del lenguaje, con las producciones escritas en modo BNF. Las acciones asociadas a las producciones aparecen en el lado derecho como trozos de código Java entre los delimitadores {: ... :}, estas acciones son ejecutadas por el parser cuando la porción de la producción de la izquierda de la acción ha sido reconocida.

A modo de ejemplo transcribiremos parte de la definición de una declaración de variable y la sentencia IF del lenguaje Jovial y su correspondencia con las producciones escritas en modo BNF.

Definición de una declaración de variable

Definición en el lenguaje Jovial. MIL-STD-1589C 2.1.1

```
<item-declaration> ::= ITEM <item-name>
                       <item-type-description>
                       [<item-preset>];
<item-name> ::= <name>
<item-preset> ::= = <item-preset-value>
<item-preset-value> ::= <compile-time-formula>
```

Correspondencia en las producciones escritas en modo BNF de JavaCup

```
declaracion ::= ITEM VAR:v VAR:t SEMI
{: RESULT=parser.factoria.decCrearVariable(v,t); :}
        | ITEM VAR:v VAR:t EQ expr_num:e SEMI
{: RESULT=parser.factoria.decCrearyAsignarVariable(v, t, e); :}
```

Definición de una sentencia IF

Definición en el lenguaje Jovial. MIL-STD-1589C 4.3

```
<if-statement> ::= IF <boolean-formula> ;
            <conditional-statement>
            [<else-clause>]
<boolean-formula> ::= <bit-formula>
<conditional-statement> ::= <statement>
<else-clause> ::= ELSE <statement>
<statement> ::= <simple-statement>
            | <compound-statement>
<compound-statement> ::= BEGIN <statement>... END
```

Correspondencia en las producciones escritas en modo BNF de JavaCup

```
sen_if          ::= IF expr_bin_o_rel:e SEMI sentencia:t
                    {: RESULT=parser.factoria.senIf(e, t); :}
        | IF expr_bin_o_rel:e SEMI sentencia:t ELSE sentencia:f
        {: RESULT=parser.factoria.senIf(e, t, f); :}
        ;
expr_bin_o_rel          ::= expr_bin:e
        {: RESULT=e; :}
        | expr_rel:e
        {: RESULT=e; :};
sentencia ::= sen_simple:s
        {: RESULT=s; :}
        | sen_compuesta:s
        {: RESULT=s; :}
```

```
        ;
sen_compuesta ::= BEGIN secuencia:s END
        {: RESULT= parser.factoria.senCompuesta(s); :}
        ;
secuencia ::= |sentencia:s secuencia:q
        {: RESULT= parser.factoria.senSecuencia(s,q); :}
        ;
```

Desarrollo del analizador sintáctico

Para realizar el analizador sintáctico, se siguieron los siguientes pasos:
Se desarrollaron tantas clases Java como tipos de terminales se definieron en JLex y tantas clases Java como tipos de no terminales se definieron en JavaCup.
Se realizó una interface Java para independizar las clases JAVA del analizador sintáctico y una interface Java para soportar múltiples estilos de imprimir el árbol sintáctico sin necesidad de recompilar el programa, además una aplicación simple que realiza el método toString() del árbol sintáctico con un estilo aplicado.
En la figura 1 se muestra la relación entre las Clases.

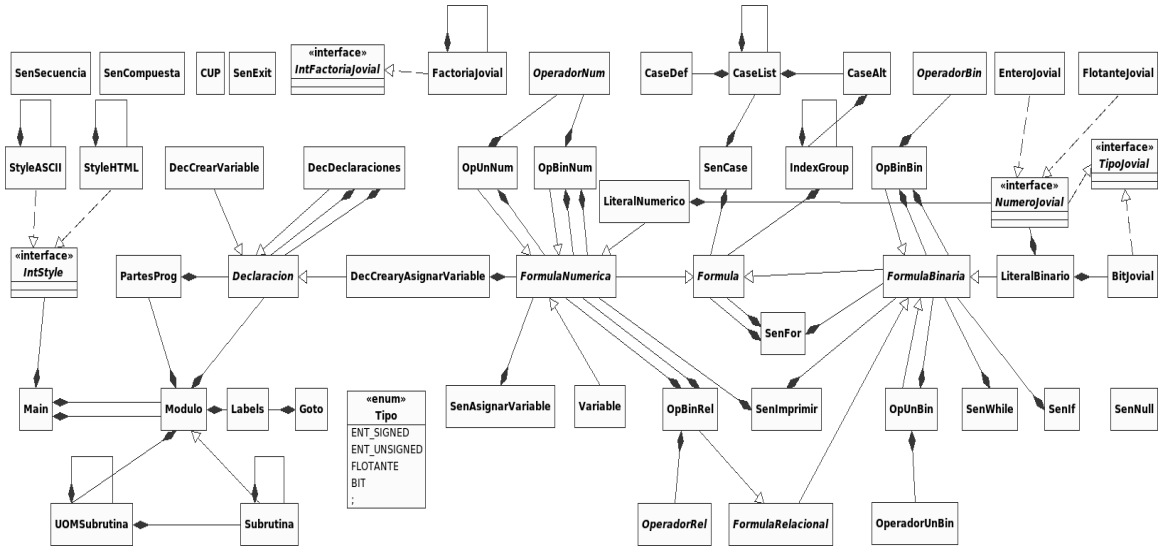


Fig 1. Diagrama de las Clases Generadas

A modo de ejemplo vamos a explicar como se representó la sentencia CASE y la sentencia IF.

Sentencia Case

En la figura 2 es un subdiagrama que muestra la relación de las clases intervinientes. En donde la clase SenCase tiene una instancia de la clase Formula que representa la fórmula a evaluar y tiene una instancia de la clase CaseList que representa la lista de alternativas de la sentencia Case.
La clase CaseList tiene una instancia de la clase CaseAlt que represente una alternativa de la sentencia Case, instancia de la clase CaseDef que representa la alternativa por defecto y una instancia de la clase CaseList para representar la lista de alternativas.
La clase CaseAlt tiene una instancia de la clase IndexGroup que representa una suseción de alternativas que serán evaluados por la fórmula de la clase SenCase y una instancia de alguna clase que extienda la clase Sentencia que representa la sentencia que se ejecutará.
La clase IndexGroup representa una suseción de índices que serán evaluados en la fórmula de la clase SenCase, cada índice es representado por una instancia de alguna clase que implemente la clase Formula.

La clase CaseDef representa la alternativa por defecto de un sentencia Case, ésta tiene una intancia de alguna clase que implemente la clase Sentencia.

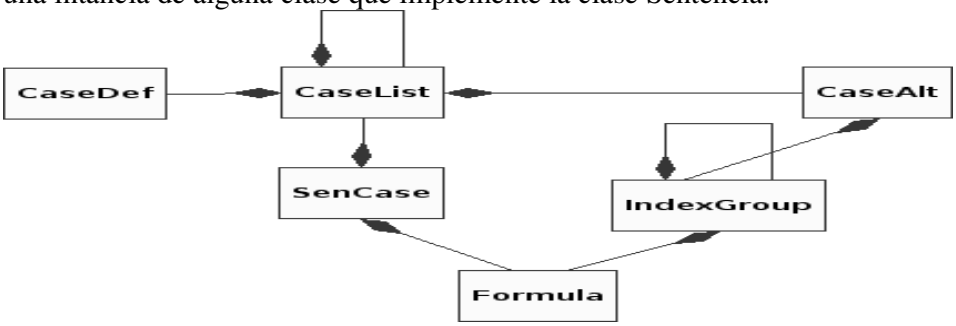


Figura 2. Subdiagrama de las clases que representan la sentencia Case

Un código fuente válido en lenguaje Jovial que tiene una sentencia CASE es el siguiente:

```
START
PROGRAM
        BEGIN
                CASE  2*3;
                  BEGIN
                                (2+2) :  1;
                                (2+3, 3+3) :  2;
                                (DEFAULT) :  3;
                  END
        END
TERM
```

Una posible visualizacion del árbol sintático de este código fuente se puede obtener utilizado la clase StyleTEX desarrollada en este trabajo y su resultado se muestra en la figura 3.

```
                                PROGRAM
                                  SC
                                  CASE
                                    CL
                                *
                                  CL              CD
                                2 3
                                CA          CL
                                                SI
                                            CA
                                +   SI                  3
                                2 2  1    +    +    SI
                                          3 3  2 3   2
```

Referencias: SC:Sentencia Compuesta. SI:Sentencia Imprimir. CL:CaseList. CA:CaseAlt. CD: CaseDef

Figura 3: Representación del árbol sintáctico de la sentencia case

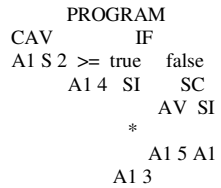
Sentencia IF

La clase SenIf representa a la sentencia IF, para ello tiene una instancia de la clase FormulaBinaria y dos instancias de clases que extiendan a la clase Sentencia, una de estas clases es para representar la secuencia de sentencias a ejecutar por verdadero y la otra para representar la secuencia de sentencia a ejecutar por falso.

Un código fuente válido en lenguaje Jovial que tiene una sentencia IF es el siguiente:

```
START
        ITEM A1 S = 2;
PROGRAM
        IF   A1>=4;
                A1*3;
        ELSE
        BEGIN
                A1=5;
                A1;
END
```

TERM
Una visualizacion del arbol sintáctico se muestra en la figura 4



Referencias: SC: Sentencia Compuesta. SI: Sentencia Imprimir. CAV:CrearyAsignarVariable. AV: AsignarVariable

Figura 4: Representación del árbol sintáctico de la sentencia IF

Conclusión y futuras líneas de investigación

Síntesis de diseño e implementación:
La figura 5 representa las diferentes etapas por las que pasa el código fuente hasta conseguir el reporte que se genera con este trabajo.

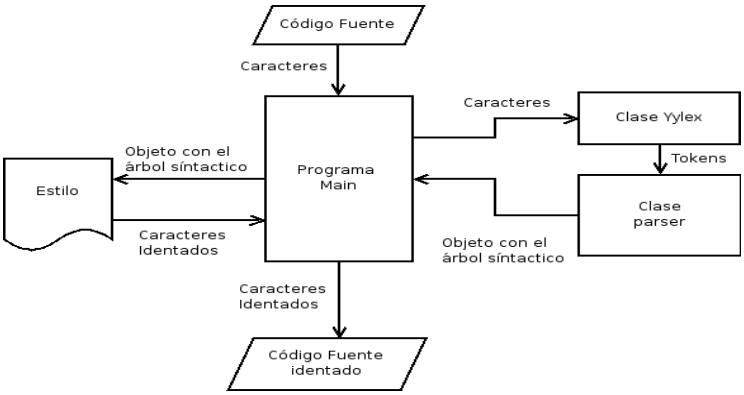


Figura 5. Etapas por las que pasa el código fuente

Analizador Léxico: El análisis léxico es la primera fase de un compilador. Dicha fase se halla bajo la responsabilidad del analizador léxico, cuya función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos llamada secuencia de tokens, que utiliza el parser para llevar a cabo el análisis sintáctico. En esta etapa se realiza el archivo de configuración correspondiente a la herramienta seleccionada.

Analizador Sintáctico: El analizador sintáctico consume los tokens generados por el lexer, genera el árbol sintáctico y comprueba si la secuencia de tokens recibida es válida en la gramática del lenguaje detectando errores sintácticos. En esta etapa se realiza el archivo de configuración correspondiente a la herramienta seleccionada.

Interfaz para analizadores semánticos:
Este trabajo puede ser utilizado para desarrollar un analizador semántico, el mismo brinda una interfaz que se llama IntFactoriaJovial.
Para que no tenga que ser modificado el parser, éste debe proveer un método para indicar qué factoría utilizar cuando se ha reconocido una acción. Esta factoria debe implementar la interfaz IntFactoriaJovial. La interfaz debe proveer de todos los métodos necesarios por el parser. Este trabajo debe proveer también todas las clases necesarias para utilizar la interfaz, con sus correspondientes constructores y que el método

toString() genere salidas con distintos estilos aplicados. Se propone que estas clases sean extendidas y se les agreguen los métodos que sean necesarios como por ejemplo el método eval para realizar el análisis semántico.

Trabajo Futuro:
Este trabajo es continuado actualmente para realizar un analizador semántico del lenguaje Jovial. Además brinda las bases necesarias para desarrollar un plugin para el ambiente de desarrollo Eclipse del lenguaje Jovial.
Este trabajo se desarrolló como una etapa inicial, debido al interés de la Fuerza Aérea Argentina, en desarrollar una herramienta que ayude al desarrollo y mantenimiento del software aeronáutico.

Referencias

- El producto "Understand for JOVIAL" de la empresa Scientific Toolworks, Inc. http://scitools.com/prod
- El producto"JOVIAL Programming Language Development Tools" de la empresaSemanticDesigns,Incorporated:http://www.semdesigns.com/Products/LanguageTools/JOVIALTools.html?Hom
- El producto "System Cross-Reference Tool" de la empresa Software Engineering Associates, Inc http:// www.seadeo.com/ development-tools.htm

Bibliografía

1. MILITARY STANDARD JOVIAL (J73), MIL-STD-1589C (USAF) 6 de Julio de 1984.
2. The Not So Short Introduction to LATEX, autor: Tobias Oetiker, Hubert Partl, Irene Hyna and Elisabeth Schlegl. Version 4.20, 31 de Mayo de 2006.
3. Compiladores. Principios, técnicas y herramientas, autores: Alfred V.Aho, Ravi Sethi y Je_rey D.Ullman. 1990 Addison-Wesley Iberoamericana, SA.
4. Schwartz, Jules I. _The Development of JOVIAL_ SIGPLAN Notices 13.8 (1978): 203-214.