SEFASGEN

Manual de usuario

Alumno: Ferrer González, Sergio
Tutor: Jorge Revelles Moreno

Índice

1. Introducción

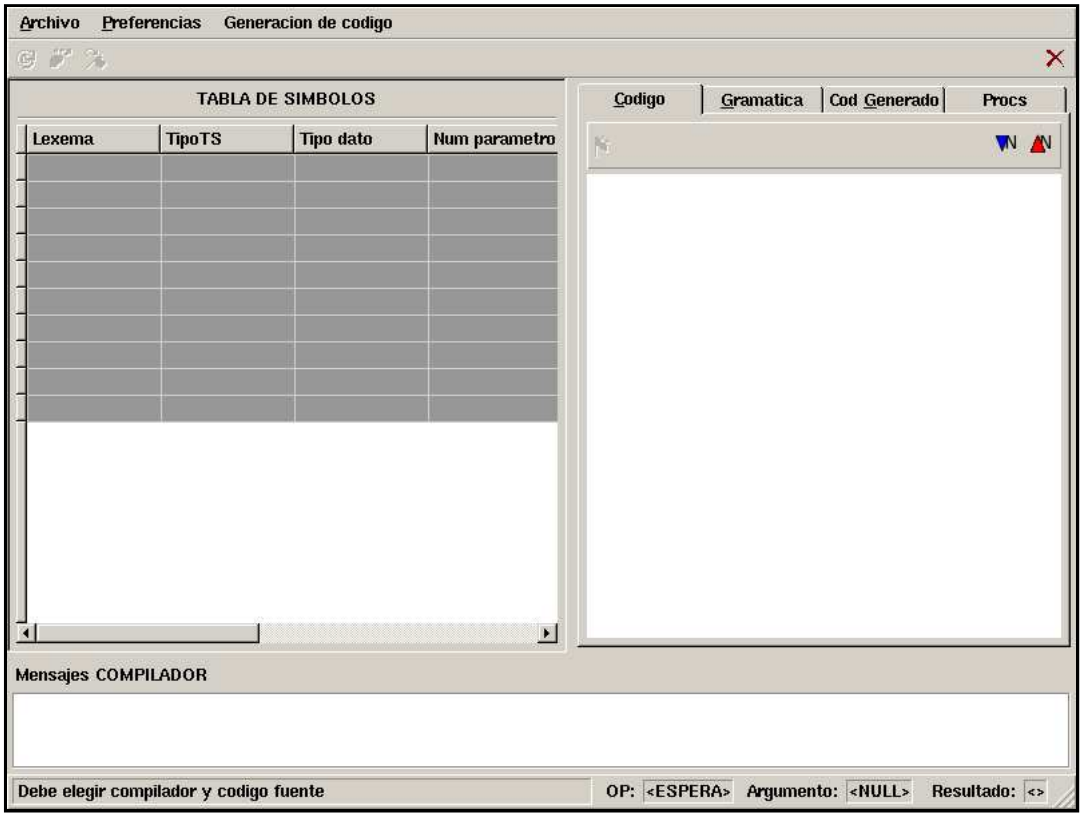
Sefasgen es un proyecto informático desarrollado como apoyo a las asignaturas de Procesadores de Lenguajes y Traductores. Su propósito es mostrar el desarrollo de las fases de Análisis Semántico y Generación de código.

1.1. Requerimientos

- Sistema operativo Linux
- Compilador C y C++
- Flex y Yacc
- sed
- Biblioteca FOX http://www.fox-toolkit.org
- Bibliotecas para desarrollo X instaladas

1.2. Un primer vistazo

Al ejecutar la aplicación se muestra la ventana principal:



Como se puede apreciar el interfaz es sencillo y cualquier usuario familiarizado con aplicaciones Windows debería hacerse con él en pocos minutos.

La ventana se divide en 3 partes principales:

- Tabla de Símbolos. A la izquierda. Se podrá ver cómo se añaden y eliminan símbolos a medida que progresa la compilación.
- Código. A la derecha. Mediante las pestañas se puede acceder a:
  - Código fuente.
  - Gramática.
  - Código generado.
  - Código generado para procedimientos.
- Mensajes del compilador. Campo de texto de la parte inferior.

Otro elemento importante de la interfaz es la Barra de Estado que se sitúa en la parte inferior de la ventana. Aquí aparecerán distintos campos que muestran información sobre la operación en ejecución.

2. Creación de un compilador para Sefasgen

Antes de utilizar el programa se debe crear un compilador para el lenguaje que se desee utilizar. Lo más sencillo sería utilizar Flex y Yacc para dicha tarea. Ésta es la opción recomendada por el autor del software.

La base de funcionamiento de Sefasgen es proporcionar al usuario una serie de funciones en C que puede utilizar en su gramática para realizar operaciones con la Tabla de Símbolos y generar código.

El usuario incorpora llamadas a estas funciones en la especificación de una gramática en las acciones semánticas de las reglas para crear un compilador

Este compilador se cargará en el entorno **Sefasgen** para seguir el proceso de compilación. Resulta necesario comentar que dicho compilador **no** podrá ser utilizado de modo independiente. Sólo funcionará en combinación con el entorno gráfico.

Ejemplo:

```
DECLARACION : TIPO ID { pet_introTS($2); }
```

En este ejemplo se introduce en la tabla de simbolos el simbolo ID

Muy importante No utilizar la entrada ni la salida estándar.

Ambas están reservadas para la comunicación con el entorno gráfico. Si desea que el compilador muestre algún mensaje al usuario (errores por ejemplo) hágalo a través
de la salida estándar de error *stderr* mediante la función fprintf o similar. Debe asegurarse de terminar cada mensaje con un carácter de nueva línea.

2.1. Tipo de los elementos de la pila

El tipo a utilizar para YACC, equivalente a una entrada de la tabla de símbolos es el siguiente:

```
/** @enum
 * Tipo de entrada en la TS
 */

typedef enum { MARCA = 0 /**< Inicio/Fin de bloque */
, PROC /**< Procedimiento   */
, VARIABLE
, PAR_FORMAL /**< Parametro Formal */
, DESC_CTRL /**< Descriptor de Control para Saltos */
```

```
, FUNCION /**< Funcion */
, RANGO /**< Rangos para indices ARRAYS */
} tEntrada;

/** @enum
 * Tipos de dato admitidos por el lenguaje
 */

typedef enum { NO_ASIG = 0  /**< No Asignado */
, DESC /**< Desconocido / Erroneo */
, ENTERO, REAL, LISTA_REAL, LISTA_ENTERO
, BOOLEANO, LISTA_BOOLEANO, CARACTER, LISTA_CARACTER
, STRING /**< Cadena */
, ARRAY
, CJTO
, PILA
} tDato;

/**
 Valor asociado a entrada de la TS
 */


typedef union valor {
int entero;
float real;
unsigned char booleano;
char caracter;
} tValor;

/**
 Entrada de la TS.
 */

typedef struct entrada_ts {
char *lexema; /**< Lexema del token */
tEntrada tipoTs; /**< Tipo de entrada */
tDato tipoDato; /**< Tipo del dato */
int nParam; /**< Num parametros formales, si es proc */
tValor valor; /**< Valor del token */

/*** Localizacion en Cod Fuente **/
int linIni; /**< Localizacion: linea Inicio */
```

```
int linFin; /**< Localizacion: linea Fin */
int colIni; /**< Localizacion: columna Inicio */
int colFin; /**< Localizacion: columna Fin */

/** NUEVOS CAMPOS */
int nDim; /**< Num de dimensiones (ARRAY) */
tValor minRango; /**< Valor minimo (CJTO) */
tValor maxRango; /**< Valor maximo (CJTO) */
int minIndx; /**< Indice minimo (ARRAY) */
int maxIndx; /**< Indice maximo (ARRAY) */

/***  Campos para la GENERACION de CODIGO   *****/

char *nomTemp; /**< Nombre de la variable temporal */
char *etqtEnt; /**< Nombre etiqueta de entrada */
char *etqtSal; /**< Nombre etiqueta de salida */
char *etqtElse; /**< Nombre etiqueta else */
char *nomVarCtrl; /**< Variable de control en EXP Condicion */
char *formato; /**< Cadena de formato para E/S */
char *variables; /**< Cadena de variables usadas en E/S */
} stEntrada;


#define YYSTYPE stEntrada
```

Muy importante No se debe redefinir YYSTYPE.

2.2. Preparando el analizador de léxico

El primer paso para la creación del compilador es modificar el analizador de léxico. El compilador que se va a crear espera que los tokens reconocidos tengan asociada información sobre su posición en el código fuente.

Esta información se mide en caracteres y es responsabilidad del analizador de léxico establecer el valor de los campos correspondientes de la estructura. En concreto estos campos son

linIni Linea Inicio

linFin Linea Fin

colIni Columna Inicio

colFin Columna Fin

2.3. Acciones semánticas

Una vez modificado el analizador de léxico se pueden añadir las funciones proporcionadas a la gramática. Es responsabilidad del usuario elegir qué funciones y parámetros ha de utilizar.

Para poder utilizar las funciones se debe incluir el archivo *sefasgen.h* proporcionado. Este archivo contiene los prototipos y definición del tipo de dato de los elementos de la pila ya visto.

La totalidad de las funciones disponibles se describe en la siguiente sección.

2.4. Documentación de las funciones

2.4.1. char* pet_BuscarEtq (unsigned char *tipoEtq*)

Peticion busqueda Etiqueta en ultimo DESCRIPTOR de CONTROL en la TS.

Parámetros:

tipoEtq Tipo de etiqueta necesaria

2.4.2. unsigned char pet_BuscarPARAM (stEntrada *reg*)

Peticion para comprobar tipo de Dato de un argumento en una llamada a PROCEDIMIENTO.

Devuelve:

1 si el tipo es correcto 0 en otro caso

2.4.3. int pet_BuscarPROC (stEntrada *reg*)

Peticion de busqueda PROCEDIMIENTO para comprobar numero y tipo de argumentos correcto en llamada.

Devuelve:

Numero de parametros declarados -1 si hubo algun error

2.4.4. tDato pet_BuscarTS (stEntrada *reg*)

Busqueda del tipo de un simbolo en el ambito actual.

Parámetros:

reg Se debe haber establecido en la estructura el lexema del simbolo y las variables de localizacion

Devuelve:

Tipo de dato del simbolo DESC si no se encuentra

2.4.5. void pet_EscEtq (char * *etq*)

Peticion escribir etiqueta.

Parámetros:

etq Cadena con la etiqueta a escribir

2.4.6. void pet_GenAsig (int *numOp*, char * *op1*, char * *op2*, char * *op3*, char * *op4*)

Peticion generacion de codigo para ASIGNACION.

Se pueden utilizar un minimo de 2 operandos y un maximo de 4.

Parámetros:

numOp Numero de operadores/operandos

op1,op2,op3,op4 operandos/operadores

2.4.7. void pet_GenCadFormato (int *colIni*, int *colFin*, char ** *formato*, char ** *vars*, tDato *tipo*, char * *nuevoArg*, unsigned char *tipoSent*)

Peticion generacion cadena formato para Entrada/Salida.

Post:

- Formato contiene la nueva cadena Ej: "La suma 2 + 4 = %d\n"
- Vars contiene los nombres de las nuevas variables ", temp"

Parámetros:

colIni columna de inicio de la expresion

colFin columna final de la expresion

tipo Tipo de dato del argumento a añadir

nuevoArg nombre del nuevo argumento a añadir

tipoSent Tipo de Sentencia 0 - SALIDA 1 - ENTRADA

2.4.8. void pet_GenDecVar (tDato *tipo*, char * *nom*)

Peticion generacion Codigo para DECLARACION de VARIABLE.

Parámetros:

tipo Tipo de dato de la variable

nom Nombre de la variable

2.4.9. void pet_GenENTRADA (char * *fmt*, char * *vars*)

Peticion generar codigo para Entrada/Salida.

Parámetros:

fmt Cadena que describe el formato

vars Cadena que describe las variables usadas

tipoSent Tipo de sentencia ENTRADA o SALIDA

2.4.10. char* pet_GenEtiq (void)

Peticion de un nuevo Nombre de ETIQUETA.

2.4.11. void pet_GenFinBlq (void)

Peticion generacion de codigo para FIN BLOQUE.
Disminuir nivel Tabulacion y Escribir "}"

2.4.12. void pet_GenIni (void)

Peticion generar Inicio de Programa.
Inclusion de archivos de cabecera y MAIN

2.4.13. void pet_GenIniBlq (char * *comentario*, int *ini*, int *fin*)

Peticion generacion de codigo para INICIO BLOQUE.

Escribir "{" y Aumentar nivel de Tabulacion. Se puede incluir de manera *opcional* un comentario para facilitar trazabilidad. Si no desea incorporar el comentario utilice `NULL` como argumento.

2.4.14. void pet_GenSALIDA (char * *fmt*, char * *vars*)

Peticion generar codigo para Entrada/Salida.

Parámetros:

fmt Cadena que describe el formato

vars Cadena que describe las variables usadas

tipoSent Tipo de sentencia ENTRADA o SALIDA

2.4.15. void pet_GenSaltoCond (char * *nomVar*, char * *etq*)

Peticion generar sentencia de salto CONDICIONAL.

Parámetros:

nomVar Variable de la expresion en salto condicional

etq Etiqueta destino del salto

2.4.16. void pet_GenSaltoInCond (char * *etq*)

Peticion generar sentencia de salto INCONDICIONAL.

Parámetros:

etq Etiqueta destino del salto

2.4.17. int pet_introTS (stEntrada *reg*, tEntrada *tipoTS*)

Peticion Insertar nueva entrada en TS.

Parámetros:

reg Nueva entrada a insertar

tipoTS Tipo de entrada (MARCAR, PROC, ...)

Devuelve:

1 si la insercion fue correcta, 0 en otro caso.

2.4.18. int pet_VerifTIPO (stEntrada *reg1*, stEntrada *reg2*)

Peticion de comprobacion de tipos.

Aunque podría realizarse en el propio compilador está disponible para que sea reflejada en Sefasgen.

Advertencia Los campos de localización de las entradas (*linIni, linFin, colIni y colFin*) son utilizados para marcar la porción de código que representa a las entradas en el interfaz gráfico. Téngalo en cuenta.

Parámetros:

reg1 1era entrada a comparar

reg2 2a entrada

Devuelve:

1 OK

0 Los tipos no coinciden.

2.5. La función main del compilador

Como último paso antes de la compilación se debe incluir en la especificación de la gramática en la zona de Procedimientos de usuario el archivo *main.c*. Tal y como su nombre indica, contiene la función main necesaria para que el compilador funcione de modo correcto en combinación con la interfaz gráfica de usuario. Una simple directiva include bastará.

2.6. Creación del ejecutable compilador

Como se verá en secciones posteriores, el entorno Sefasgen permite seguir un proceso de compilación en la gramática que define el lenguaje. En todo momento se señala en este fichero cuál es la operación de la Tabla de Símbolos , Comprobación semántica o generación de código que se está ejecutando.

Para conseguirlo se debe insertar una llamada a la función *informar* antes de cada llamada a una petición. Para evitarle trabajo innecesario y propenso a errores al usuario se ha creado un escáner que se encarga del trabajo. La definición de este escáner flex se encuentra en el archivo marcar.lex

Para crear el escáner y otras tareas necesarias en la creación de un compilador compatible con Sefasgen se proporciona el siguiente fichero *makefile*

```
#
# SEFASGEN
```

```
#
# Makefile para contruir compilador
#
# Autor: Ferrer Gonzalez, Sergio
#

# Poner aqui el nombre del fichero
# con la especificacion YACC
GRAM = gram.y

all : compilador

compilador : $(GRAM) genmens peticiones.c marcar
./marcar < $(GRAM) > FINAL.y
yacc -d -t -v FINAL.y
flex scanner.lex
mv y.tab.c y.tab.copia
sed -nf script.sed y.tab.copia
rm -f y.tab.copia
cat y.output | ./genmens | cat > msj.err
gcc -o sinsem y.tab.c peticiones.c lex.yy.c

genmens : mes.l
flex mes.l
gcc -o genmens lex.yy.c


# Scanner que inserta llamadas a informar()
# Esta funcion nos servira para localizar
# la funcion en uso en la gramatica

marcar : marcar.lex
flex -omarcar.c marcar.lex
gcc -o marcar marcar.c
```

MUY IMPORTANTE Durante la ejecución del makefile anterior se creará una nueva gramática **FINAL.y** Ésta es la gramática que debe cargarse en el entorno y no la creada por el usuario.

Advertencia El nombre del fichero ejecutable del compilador a cargar en el entorno Sefasgen será *sinsem*.

3. Uso del entorno gráfico

3.1. Cómo compilar paso a paso

Una vez creado un compilador siguiendo los pasos definidos en la sección anterior, el usuario se encuentra en disposición de utilizar la aplicación gráfica que supone el núcleo de Sefasgen.

Para iniciar la compilación de un código fuente escrito en el lenguaje definido por el usuario es imprescindible cargar 2 ficheros:

- Compilador. Creado según se describe en este manual. El nombre del ejecutable será *sinsem* si se ha utilizado el makefile proporcionado.
- Código fuente. Escrito en el lenguaje que traduce el compilador anterior.

De forma opcional, aunque recomendable, se puede cargar la gramática que define el lenguaje. Se recuerda al lector que la gramática a cargar debe ser la generada durante la creación del compilador **NO** la original escrita por el usuario. El nombre del fichero será *FINAL.y* si se usó el makefile proporcionado.

Cargar un fichero de gramática hace más sencillo comprender la compilación ya que se señala la regla y acción en ejecución.

Una vez cargados los ficheros necesarios puede comenzar el proceso de compilación al habilitarse el botón de inicio (1er botón a la izquierda en la barra de herramientas)

Al pulsar el botón comienza la compilación y el usuario puede observar en la barra de estado cuál es la primera operación solicitada por el compilador

A partir de este momento se habilita el botón Paso (Segundo en la barra de herramientas, con forma de huella). El usuario no tiene más que pulsar este botón cada vez que quiera que se ejecute una nueva acción y consultar el estado de la tabla y pestañas de Código para seguir la compilación. Cuando ésta termina aparece un mensaje indicándolo al usuario.

En un instante dato de tiempo se puede guardar en disco el código generado hasta el momento mediante la entrada *Guardar cod generado* del menú *Archivo*.

3.2. Uso de breakpoints

En ocasiones no nos interesará realizar el seguimiento completo de cada una de las operaciones que se ejecutan. Tal vez sólo estemos interesados en la generación de código o simplemente se desea que se compile hasta un determinado punto en el código fuente.

Para evitar tener que pulsar continuamente el botón de Paso se puede establecer un punto de ruptura o breakpoint. El botón para ello está a la izquierda en la barra de herramientas del panel de Código Fuente.

Al pulsar el botón se añadirá un nuevo breakpoint que aparece señalado en el código fuente como **[B]** en verde. Para lanzar la compilación hasta el nuevo break, basta pulsar el botón correspondiente: el tercero a la izquierda en la barra de herramientas principal, con forma de bandera.

Algunas advertencias sobre el uso de breakpoints:

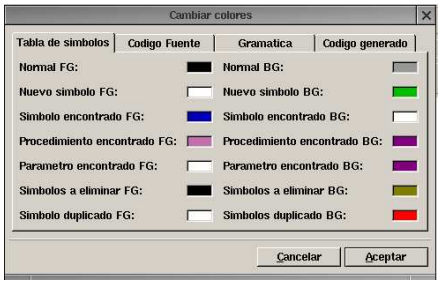
- Si se sitúa un breakpoint más allá del final del archivo, se realizará la compilación completa.
- **No** situar un nuevo breakpoint en una posición anterior a uno ya existente. El comportamiento de la aplicación en este caso está indefinido y seguramente será inestable.
- Situar un breakpoint de tal modo que se tengan que ejecutar un gran número de acciones puede provocar que la aplicación se bloquee.

4. Preferencias de usuario

Sefasgen permite al usuario personalizar la apariencia del interfaz. Los distintos elementos que pueden ser personalizados son:

- Colores frente y fondo para la Tabla de Simbolos y pestañas de código. Así, por ejemplo, se puede establecer un color para los símbolos buscado y encontrados en la Tabla y otra combinación diferente para los nuevos símbolos.
- La fuente para los elementos anteriores también puede ser establecida por el usuario.

- Se permite definir reglas para colorear los mensajes del compilador que aparecen en la parte inferior de la ventana principal. Un posible uso de esta capacidad sería utilizar una combinación de colores para los errores de léxico y otra distinta para los mensajes informativos.



El control de las preferencias se realiza desde el menú del mismo nombre, en la parte superior de la ventana principal.

Otra característica del programa es que las preferencias de usuario pueden ser almacenadas en el Registro para no tener que re-introducirlas cada vez que se ejecute el programa. Para ello se dispone de una entrada en el menú Preferencias.

Nota: La localización del archivo de Registro es:

```
<directorio_home>/.foxrc/SFGSoft/SEFASGEN
```

Advertencia: Si se modifica alguna de las preferencias y no se utiliza la entrada de menú para guardar antes de terminar el programa las preferencias **no** se almacenarán.

4.1. Coloreado de mensajes del compilador

Las reglas de coloreado de Sefasgen constan de 4 elementos:

- Nombre. Una cadena descriptiva para el usuario
- Patrón. Una expresión regular que define las porciones de los mensajes que se verán afectados por la regla.
- Color de frente del texto si encaja la regla
- Color de fondo del texto del mensaje si encaja la regla

La sintaxis de las expresiones regulares es similar a la utilizada por Perl ó Flex. Se recomienda no utilizar las comillas dobles. En vez de estas, utilizar la barra invertida para escapar caracteres con significado especial.

Hay que señalar que un mismo mensaje puede encajar con varias reglas al mismo tiempo. Esto permite asignar un color al número de línea en un mensaje y otro al tipo de error, por ejemplo.

Se intentan emparejar todas las reglas con cada mensaje recibido del compilador por lo que se aconseja no utilizar expresiones complejas.

Las reglas se gestionan desde su propia entrada del menú Preferencias y también son almacenadas en el Registro.

5. Esquema de traducción

Como última característica destacable de Sefasgen se hablará sobre el esquema de traducción.

En la generación de código se permite al usuario especificar el lenguaje intermedio que desea utilizar especificándolo en un archivo.

Junto al software se proporciona el archivo *esquema.ets* que define como esquema de traducción una versión simplificada del lenguaje C. Si desea utilizar otro archivo puede especificarlo mediante la entrada *Archivos...* del menú *Generación de código*.

Advertencia: El nombre del archivo de esquema a utilizar se almacena en el Registro y se guarda al utilizar la entrada *Guardar Preferencias*. Si desea utilizar otro archivo que no sea el guardado en el Registro debe cambiarlo, guardar las preferencias, terminar la aplicación y volver a ejecutarla.

El formato del archivo con el esquema de traducción debe ajustarse al lenguaje definido por la siguiente gramática YACC:

```
%token ID STRING VAR CAD TIPO CAD_OPC
%token REGLA FIN_REGLA

%start ESQUEMA

%%

ESQUEMA : ESQUEMA DEC_REGLA
| DEC_REGLA
;

DEC_REGLA : REGLA ID LISTA_CAMPOS FIN_REGLA
;

LISTA_CAMPOS : LISTA_CAMPOS TIPO_CAMPO
```

```
| TIPO_CAMPO
;

TIPO_CAMPO : CAD
| TIPO
| VAR
| CAD_OPC
| STRING
|
;
```

Los distintos tipos de campo son:

CAD Una cadena de tamaño variable. El compilador debería enviar el tamaño de la cadena antes de enviar la cadena en sí.

CAD_OPC Igual que la anterior pero puede aparecer o no.

TIPO Constante numérica que identifica a uno de los tipos definidos en el lenguaje.

STRING Una cadena fija que se define en la propia regla

El uso de un esquema de traducción permite utilizar, por ejemplo, Perl o Tripletas como lenguaje intermedio.

A continuación se muestra un ejemplo para clarificar el uso del archivo *esquema.ets*:

Sea la definición de una regla que genera código intermedio para una sentencia de declaración de variables:

```
# Regla 2
# Declaracion de variables

REGLA decVar
TIPO
" "
CAD
";\n"
FIN_REGLA
```

Veamos cómo se genera código usando esta regla:

1. El compilador al reconocer una sentencia de declaración de variables puede invocar una petición de generación de código usando esta regla.

2. La aplicación, al recibir la petición, consulta el esquema de traducción y determina que debe esperar que el compilador le envíe una variable tipo de las definidas en el lenguaje y una cadena que en esta regla representa el nombre de la variable a declarar.

3. El compilador envía los datos: tipo "ENTEROz cadena "num".

4. Al recibir los datos la aplicación realiza la traducción al lenguaje intermedio y genera el código. Supongamos que el tipo ENTERO en el lenguaje definido por el usuario se traduce como "int". Se genera el siguiente código:

   ```
   int num;\n
   ```

Advertencia Aunque lo deseable sería poder identificar las reglas por su nombre (ID) en realidad se hace por su puesto en el archivo de traducción, empezando la numeración con 0. Si se desea modificar el esquema o las funciones de petición de generación de código debería tenerse muy en cuenta esto.

Apéndices

A. Lista de archivos

A continuación se presenta una lista de los archivos más importantes proporcionados con el software y una pequeña descripción de los mismos

A.1. Ficheros para la creación del compilador

def_msjs.h Constantes que definen los tipos de mensaje. No deberían ser utilizadas por el usuario final.

error.y Gramática para la generación de mensajes de error.

gram.y Gramática de ejemplo.

makefile Makefile para la construcción del compilador.

mes.l Scanner para la generación de mensajes de error.

peticiones.c Implementación de las operaciones disponibles

plantilla.c Función main a usar en la construcción del compilador

scanner.lex Scanner para la gramática de ejemplo.

script.sed Script sed para realizar sustituciones de texto necesarias en la construcción del compilador.

sefasgen.h Fichero de cabecera a usar en la gramática creada por el usuario. Debería incluirse en la sección inicial de la especificación YACC

marcar.lex Scanner para añadir llamadas a informar() en la gramática creada por el usuario

B. Referencia del Archivo sefasgen.h

B.1. Descripción detallada

Definicion de la estructura que representa una entrada de la Tabla de Simbolos, constantes y prototipos de funciones.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Clases

- union **valor**

  Valor asociado a entrada de la TS.

- struct **entrada_ts**

  Entrada de la Tabla de Simbolos.

Tipos de etiqueta

- #define **ETQ_ENT** 0
- #define **ETQ_SAL** 1
- #define **ETQ_ELSE** 2

Operaciones de la Tabla de Simbolos

- int **pet_introTS** (**stEntrada** reg, **tEntrada** tipoTS)
- **tDato pet_BuscarTS** (**stEntrada** reg)
- void **pet_SacarTS** (void)
- int **pet_BuscarPROC** (**stEntrada** reg)
- unsigned char **pet_BuscarPARAM** (**stEntrada** reg)
- int **pet_VerifTIPO** (**stEntrada** reg1, **stEntrada** reg2)

Generacion de codigo

- void **pet_GenIni** (void)
- void **pet_GenDecVar** (**tDato** tipo, char *nom)
- void **pet_GenIniBlq** (char *comentario, int ini, int fin)
- void **pet_GenFinBlq** (void)
- void **pet_GenAsig** (int nOp, char *op1, char *op2, char *, char *)
- char * **pet_GenTemp** (void)
- void **pet_GenCadFormato** (int ini, int fin, char **formato, char **vars, **tDato** tipo, char *nuevoArg, unsigned char tipoSent)
- void **pet_GenENTRADA** (char *fmt, char *vars)
- void **pet_GenSALIDA** (char *fmt, char *vars)
- void **pet_GenSaltoCond** (char *nomVar, char *etq)
- void **pet_GenSaltoInCond** (char *etq)
- char * **pet_GenEtiq** (void)
- void **pet_EscEtq** (char *etq)
- char * **pet_BuscarEtq** (unsigned char tipoEtq)
- void **pet_SacarCTRL** (void)
- void **pet_Rango** (char *lex, int *min, int *max)
- void **pet_GenDecProc** (char *lex, **tDato** tipo)
- void **pet_GenParam** (**tDato** tipo, char *lex)
- void **pet_GenFinCabecera** (void)
- void **pet_GenFinDecProc** (void)
- void **pet_GenLlamada** (char *proc, char *args)

Funciones auxiliares.

- void **check** (void)
- void **reiniciar** (void)
- void **informar** (int pos, int tam)
- char * **convCadena** (**tDato** tipo, **tValor valor**)
- unsigned char **flagREI**

Variables definidas por FLEX

- int **yylineno**
- FILE * **yyin**
- char * **yytext**

Definiciones

- #define **YYSTYPE stEntrada**
- #define **PRINT** 0
- #define **SCAN** 1

Tipos definidos

- typedef **valor tValor**
- typedef **entrada_ts stEntrada**

Enumeraciones

- enum **tEntrada** {
  MARCA = 0, **PROC**, **VARIABLE**, **PAR_FORMAL**,
  DESC_CTRL, **FUNCION**, **RANGO** }
- enum **tDato** {
  NO_ASIG = 0, **DESC**, **ENTERO**, **REAL**,
  LISTA_REAL, **LISTA_ENTERO**, **BOOLEANO**, **LISTA_BOOLEANO**,
  CARACTER, **LISTA_CARACTER**, **STRING**, **ARRAY**,
  CJTO, **PILA** }

B.2. Documentación de los tipos definidos

B.2.1. typedef union valor tValor

Valor asociado a entrada de la TS.

B.2.2. typedef struct entrada_ts stEntrada

Entrada de la Tabla de Simbolos.

Tipo de dato de los elementos de la pila en YACC

B.3. Documentación de las enumeraciones

B.3.1. enum tEntrada

Tipo de entrada en la TS.

Valores de la enumeración:

MARCA Inicio de bloque.

PROC Procedimiento.

VARIABLE Variable.

PAR_FORMAL Parametro Formal.

DESC_CTRL Descriptor de Control para Saltos.

FUNCION Funcion.

RANGO Rangos para indices de ARRAYS.

B.3.2. enum tDato

Tipos de dato admitidos por el lenguaje.

Valores de la enumeración:

NO_ASIG No Asignado.

DESC Desconocido / Erroneo.

ENTERO

REAL

LISTA_REAL

LISTA_ENTERO

BOOLEANO

LISTA_BOOLEANO

CARACTER

LISTA_CARACTER

STRING Cadena.

ARRAY

CJTO

PILA

C. Referencia de la Unión valor

`#include <sefasgen.h>`

Atributos públicos

- int **entero**
- float **real**
- unsigned char **booleano**
- char **caracter**

D. Referencia de la Estructura entrada_ts

`#include <sefasgen.h>`

D.1. Descripción detallada

Tipo de dato de los elementos de la pila en YACC

Atributos públicos

- char * **lexema**
- **tEntrada tipoTs**
- **tDato tipoDato**
- int **nParam**
- **tValor valor**
- int **nDim**
- **tValor minRango**
- **tValor maxRango**
- int **minIndx**
- int **maxIndx**

Localizacion en Cod Fuente

- int **linIni**
- int **linFin**
- int **colIni**
- int **colFin**

Campos para la GENERACION de CODIGO

- char * **nomTemp**
- char * **etqtEnt**
- char * **etqtSal**
- char * **etqtElse**
- char * **nomVarCtrl**
- char * **formato**
- char * **variables**

D.2. Documentación de los datos miembro

D.2.1. char* entrada_ts::lexema

Lexema del token.

D.2.2. tEntrada entrada_ts::tipoTs

Tipo de entrada.

D.2.3. tDato entrada_ts::tipoDato

Tipo del dato.

D.2.4. int entrada_ts::nParam

Num parametros formales, si es proc.

D.2.5. tValor entrada_ts::valor

Valor del token.

D.2.6. int entrada_ts::linIni

Localizacion: linea Inicio.

D.2.7. int entrada_ts::linFin

Localizacion: linea Fin.

D.2.8. int entrada_ts::colIni

Localizacion: columna Inicio.

D.2.9. int entrada_ts::colFin

Localizacion: columna Fin.

D.2.10. int entrada_ts::nDim

Num de dimensiones (ARRAY).

D.2.11. tValor entrada_ts::minRango

Valor minimo (CJTO).

D.2.12. tValor entrada_ts::maxRango

Valor maximo (CJTO).

D.2.13. int entrada_ts::minIndx

Indice minimo (ARRAY).

D.2.14. int entrada_ts::maxIndx

Indice maximo (ARRAY).

D.2.15. char* entrada_ts::nomTemp

Nombre de la variable temporal.

D.2.16. char* entrada_ts::etqtEnt

Nombre etiqueta de entrada.

D.2.17. char* entrada_ts::etqtSal

Nombre etiqueta de salida.

D.2.18. char* entrada_ts::etqtElse

Nombre etiqueta else.

D.2.19. char* entrada_ts::nomVarCtrl

Variable de control en EXP Condicion.

D.2.20. char* entrada_ts::formato

Cadena de formato para E/S.

D.2.21. char* entrada_ts::variables

Cadena de variables usadas en E/S.