Asignatura 240702

Séptimo semestre del Grado en Ingeniería Informática

Procesadores de Lenguajes

Laboratorio

Federico Fariña Figueredo

23 de septiembre de 2019

1. Estructura de las practicas

Las practicas de la asignatura 240702 deben verse como un único proyecto en el que el alumno debe realizar parte de un compilador para un lenguaje de programación utilizando flex y bison. En este documento se describe el lenguaje, aunque no de modo completo. Algunas de las características deben ser definidas por el alumno.

Aunque el proyecto se divide en dos *practicas*, una primera que le indica que cree con flex un scanner y otra que le pide que cree un parser/generador de código intermedio con bison, no debe olvidar en ningún caso que ambos elementos deben cooperar para compilar un lenguaje. No tome ninguna decisión en la primera parte sin considerar sus efectos en la segunda.

2. Enunciado y calendario de las prácticas

2.1. Práctica 1

Utilice la utilidad flex para crear un scanner que sea capaz de reconocer todos los tokens que considere necesarios para un implementar un compilador del lenguaje que se describe a continuación. El scanner debe ser capaz de recibir un archivo con texto que contenga un programa correcto e informar por pantalla de cuáles son los tokens vistos. En el caso de tokens multivaluados debe indicar tanto la identificación del token leído como su valor. Para ahorrar trabajo posterior, se requiere que los literales numéricos se escriban como números, no como cadenas de texto[1].

Entrega: Uno de los alumnos de cada grupo de laboratorio debe entregar en la tarea que se abrirá a tal efecto en miAulario una carpeta comprimida que contenga:

- Un fichero de texto que contenga los nombres de los componentes del grupo y cualquier otra cuestión que considere que debe poner en conocimiento del profesor.
- Un fichero llamado scanner.l con el fichero de flex que se haya desarrollado
- Uno o varios ficheros con nombre programaX.alg (donde X es un número) con ejemplos que se le pueden pasar a su scanner para comprobar que funciona correctamente.
- Cualquier otro fichero necesario para la compilación/ejecución de su scanner. Por ejemplo, si utiliza make para construir/ejecutar su scanner (deseable), el makefile correspondiente.

Su profesor le avisará de cuándo se cerrará la tarea, pero no olvide que será antes de que comience la cuarta sesión de prácticas. Para que pueda organizar su tiempo tenga en cuenta que debe:

1. Obtener la documentación: Le recomiendo que como manual para flex utilice el ms cercano pero posterior a la versión instalada en el laboratorio. Búsquelo en internet.
2. Leer este manual probando algunos de los ejemplos que describe.
3. Definir los tokens que considere necesarios para el lenguaje (lo cual exige que haya leído la descripción del lenguaje).
4. Crear el fichero para flex
5. Crear y probar algunos ejemplos
6. Arreglar los posibles problemas que encuentre

Tenga en cuenta que está en cuarto del grado en Ingeniería del software, por lo que se le exigen criterios de calidad tanto en cuanto a la usabilidad de su programa como en lo referente a la legibilidad de su código o en lo referente a documentación de su programa.

[1] Es decir, debe emplear %d o %f en lugar de %s a la hora de imprimir estos valores

2.2. Práctica 2

Utilice la utilidad bison para crear un parser que chequee la sintaxis de un programa escrito en el lenguaje que se describe a continuación y genere el código intermedio correspondiente. El parser debe ser capaz de recibir como entrada un archivo con texto que contenga un programa. El parser debe informar en otro fichero de cuáles son los tokens a los que hace shift y cuáles son las reducciones que se producen. En otro fichero debe escribir información que permita reconstruir la tabla de símbolos que se ha creado durante la compilación. En un tercer fichero debe escribir la tabla de cuadruplas que se genera.

Ante un fichero que contenga errores, su parser debe construir los ficheros indicados sólo hasta el primer punto de error (no vamos a entrar a programar la parte de resolución de errores).

Entrega: Uno de los alumnos de cada grupo de laboratorio debe entregar en la tarea que se abrirá a tal efecto en miAulario una carpeta comprimida que contenga:

- Un fichero llamado parser.y con el fichero de bison que se haya desarrollado.
- Un fichero llamado scanner.l con el fichero de flex que se esté utilizando (puede que no sea exactamente el mismo que ya entregó).
- Uno o varios ficheros con nombre programaX.alg (donde X es un número) con ejemplos que se le pueden pasar a su compilador para comprobar que funciona correctamente.
- Cualquier otro fichero necesario para la compilación/ejecución de su scanner. Por ejemplo, si utiliza make para construir/ejecutar su scanner (deseable), el makefile correspondiente; el resto de ficheros .c que se hayan desarrollado para ayudar al parser ....
- Un breve fichero de texto que indique cómo crear su compilador desde sus ficheros (esencial si no hay makefile), cuáles son las características de este que se hayan acordado durante el desarrollo de las prácticas y una autovaloración del trabajo realizado por cada alumno del griupo. El archivo también contendrá cualquier explicación adicional que el alumno considere necesario poner el conocimiento el profesor

Su profesor le avisará de cuándo se cerrará la tarea, pero no olvide que será antes del examen ordinario de la asignatura. Para que pueda organizar su tiempo tenga en cuenta que debe:

1. Obtener la documentación: Le recomiendo que nuevamente que la versión de manual sea la más cercana a la versión de bison que tenga instalada..
2. Leer este manual probando (y entendiendo) algunos de los ejemplos que describe.
3. Definir la gramática para el lenguaje (lo cual exige que haya leído su descripción).
4. Crear el fichero para bison
5. Comprobar los conflictos y el modo en que bison los resuelve. Definir prioridades y asociatividades para que se resuelvan del modo que se desea (%left, %right, %nonasoc)
6. Asignar tipos a los tokens y variables de la gramática para que se pueda almacenar la información necesaria para generar código (YYSTYPE, yylval, %union, %type).
7. Crear una estructura para la tabla de símbolos (tanto en ejecución como en almacenamiento).
8. Crear una estructura para la tabla de cuadruplas (tanto en ejecución como en almacenamiento)
9. Crear rutinas semánticas.
10. Crear y probar algunos ejemplos
11. Arreglar los posibles problemas que encuentre

Tenga en cuenta que está en cuarto del grado en Ingeniería del software, por lo que se le exigen criterios de calidad tanto en cuanto a la usabilidad de su programa como en lo referente a la legibilidad de su código o en lo referente a documentación de su programa

3. Un lenguaje de programación algorítmica: ProAlg

3.1. Introducción

En algunas asignaturas de los primeros semestres del Grado en Ingeniería Informática en la UPNa, se ha empleando un lenguaje de programación algorítmico que permite enseñar metodologías de programación evitando los problemas inherentes a los lenguajes concretos. Obviamente, no existe, hasta hoy, ningún compilador para este lenguaje. En este laboratorio se pretende desarrollarlo.

3.2. Presentación del lenguaje

Un programa ProAlg contiene una descripción de sentencias a realizar y una descripción de los datos que manipulan dichas sentencias. Las sentencias se describen mediante instrucciones y los datos por medio de declaraciones.

Los datos se representarán por valores de variables. El lenguaje es fuertemente tipado, por lo que las variables deben declararse antes de poder ser usadas. Una declaración asocia a la variable un tipo que define el dominio de dicha variable. Los tipos predefinidos son: entero, carácter, real, cadena y booleano. Se pueden definir subrangos de enteros, booleanos y carácteres.

El lenguaje ProAlg permite definir tipos compuestos. Se admiten dos tipos de composiciones: tablas en los que todos los elementos son del mismo tipo (básico o de otro tipo creado por el programador) y se accede a un elemento por medio de un índice; y tuplas a cuyos campos se accede a través de un nombre. También permite la creación de variables usando memoria dinámica.

El conjunto de instrucciones de ProAlg es reducido. Permite asignaciones de expresiones a variables y composiciones secuenciales, alternativas e iterativas. Un programa ProAlg está compuesto por una cabecera, un bloque principal y un conjunto de acciones y funciones. Los parámetros de las funciones son siempre de entrada.

A continuación se va a describir tanto el léxico como la sintaxis del lenguaje. Se incluyen comentarios diversos sobre a semántica.

3.3. El Léxico

3.3.1. El alfabeto

Los tokens del lenguaje se construyen combinando elementos del siguiente conjunto de caracteres[2]:

- Letras mayúsculas y minúsculas A..Za..z .
- Cifras 0..9
- Caracteres especiales " / ( ) = [ ] + * { } , ; . : - _ > <
- Caracteres espacio en blanco, tabulador y salto de línea

El resto de caracteres pueden aparecer en entornos concretos (como literales de texto o comentarios) sin significado semántico especial.

Hay que destacar que el lenguaje NO diferencia entre mayúsculas y minúsculas. Es decir, aunque la variable *pepe* es la misma que la variable *PePe* De igual modo, la palabra reservada *mientras* puede escribirse también de la forma *Mientras* o incluso *miENtras*.

En lo que sigue, para evitar confusiones, se indicarán subrayados los símbolos del alfabeto cuando aparezcan en una expresión regular.

[2]Lo que viene a continuación es una descripción del alfabeto Σ que permite definir formalmente los AFNs de los patrones. No quiere decir que deba escribir una regla para cada uno de los elementos que siguen

3.3.2. Convenciones sobre espacios

Un programa es una secuencia de tokens. Los saltos de línea, tabuladores y espacios en blanco se usan sólo para separar los tokens de modo que el lector pueda entender el programa. No tienen ningún valor semántico adicional.

3.3.3. Tokens

El lenguaje utiliza un conjunto de tokens que incluye al menos los que se indican a continuación. Cabe destacar que lo que sigue no es un listado completo ni detallado. Se indican algunos que podrían unirse por comodidad de la gramática en tokens multivaluados e incluso es posible que las características semánticas del lenguaje obliguen a separar algún token aquí descrito en varios. También cabe desatacar que, aunque se emplea un lenguaje similar al de expresiones regulares para describir los tokens, se han empleado ciertos *atajos* para ahorrar notación. Pregunte calquier cosas que le resulte extraña.

Los identificadores

Se utilizan para nombres de variables, tipos, campos de registros, acciones, funciones y palabras reservadas. Su forma es:

```
identificador → letra letra_o_cifra*
letra_o_cifra → letra | cifra
letra → mayuscula | minuscula
```

Literales

Se utilizan para definir valores constantes. Se pueden definir para todos los tipos básicos del lenguage. Sus formas son:

```
literal_entero → (+ | −)?cifra cifra*((e|E)cifra cifra*)?
literal_real → (+ | −)?cifra cifra*(.cifra cifra*)?((e|E)cifra cifra*)?
literal_booleano → verdadero | falso
literal_caracter → cualquier caracter encerrado entre comillas dobles
literal_cadena → cualquier cadena de caracteres sin comilla simple (o la comilla simple
                 con / por delante), encerrado entre comillas simples
```

Comentarios

Los comentarios son una parte fundamental del lenguaje. Los hay de dos tipos. Por un lado hay comentarios informátivos, que pueden aparecer en cualquier parte del programa, y no tienen valor semántico ni sintáctico. Por otro lado, la sintáxis del lenguaje impone la existencia de al menos dos comentarios: la precondición y la postcondición.

En cualquier caso, los comentarios comienzan con una llave de apertura y finalizan por una llave de cierre. En medio de esas llaves puede ir cualquier cadena de texto con una excepción: el carácter } sólo puede aparecer si está precedido por /.

Palabras reservadas

Para facilitar la labor, vamos a utilizar un conjunto de palabras reservadas que el usuario no puede utilizar como identificadores (recuerde que lo son tanto en la forma que se indica abajo como cambiando cualquier minúscula por la misma letra en mayúscula). Son las palabras: accion, algoritmo, booleano, cadena, caracter, const, continuar, de, dev, div, ent, entero, e/s, faccion, falgoritmo, falso, fconst, ffuncion, fmientras, fpara, fsi, ftipo, ftupla, funcion, fvar, hacer, hasta, mientras, mod, no, o, para, real, ref, sal, si, tabla, tipo, tupla, var, verdadero, y.

Le aviso de que no puedo asegurar que el listado sea completo. Tampoco que todo lo anterior vaya a ser realmente necesario. Recuerde que parte de su trabajo va a consistir en terminar de definir el lenguaje llegando a acuerdos con su profesore sobr qué partes va a implementar y qué partes no.

Otros posibles tokens

Obviamente, pueden existir otros tokens. Se trata, de agrupaciones de caracteres que tienen una sintáxis y/o semántica especial. Recuerde que no existe una frontera clara entre lo que es un token y lo que no. Lo que se indica a continuación (y lo anteriormente explicado) son algunos ejemplos de cosas que podrían ser tokens. La decisión de si lo son o no se deja en su mano.

La gramática va a emplear diversos operadores construidos con caracteres especiales. Algunos son: asignacion (:=), composición secuencial (;), separador (,), subrango (..), definición de tipo de una variable(:), entonces (->), si no si ([]), cración de tipo (=), inicio de array([), fin de array (]).

Por último, el lenguaje utiliza los símbolos habituales para expresar los operadores relacionales y aritméticos.

4. La Sintaxis

A continuación se explica la sintaxis del lenguaje. Se van a dar algunas ideas de cómo sería una gramática inicial, pero debe tenerse en cuenta que no es completa. En cualquier caso, para evitar errores, siempre se indicarán subrayados los *tokens* (nótese que no se está dando una gramática de contexto libre, de hecho en muchos casos se han utilizado expresiones regulares al explicar las reglas para ahorrar notación). Le recuerdo que no tiene que usar exactamente los tokens que le propongo (en algunos casos, es mejor no hacerlo) puede que su decisión y la mía difieran.

4.1. Un Programa en ProAlg

El lenguaje ProAlg permite especificar algoritmos. Un algoritmo debe contener en su definición la declaración de todas las acciones y funciones de los que vaya a hacer uso. Se presupone que el algoritmo no es interactivo en el sentido de que la interacción con el usuario es implícita a través de las variables declaradas como de entrada y salida. Se presupone que el algoritmo debe incluir su especificación, por lo que se fuerza a que existan dos comentarios que se suponen contendrán la precondición (siempre comienza por Prec) y la postcondición (siempre comienza por Post) del algoritmo. La gramática del algoritmo es:

```
desc_algoritmo → algoritmo identificador ; cabecera_alg bloque_alg falgoritmo .
cabecera_alg → decl_globales decl_a_f decl_ent_sal comentario
bloque_alg → bloque comentario
decl_globales → (declaracion_tipo | declaracion_const) decl_globales | ε
decl_a_f → (accion_d | funcion_d) decl_a_f | ε
bloque → declaraciones instrucciones
declaraciones → (declaracion_tipo | declaracion_const | declaracion_var ) declaraciones | ε
```

Como le decía anteriormente, he abusado de la notación de expresiones regulares. Los paréntesis que acaba de leer (al igua que sucede de aquí en adelante), indican que se ha tratado de escribir menos. Debe *distribuir* las concatenaciones. Es decir, por ejemplo, lo que acaba de leer para decl_globales realmente significa

```
decl_globales → declaracion_tipo decl_globales | declaracion_const decl_globales | ε
```

Las declaraciones de un bloque son sólo válidas y visibles dentro de ese bloque mientras que las globales son visibles en todo el algoritmo. Puede verse que esta estructura obliga a que todas las acciones y funciones se declaren en una única zona de programa: después de las declaraciones y antes de las declaraciones de las variables que permiten establecer la precondición y postcondición del algoritmo (es decir, las que entenderemos que se emplean para comunicar con el usuario datos y resultados)

4.1.1. Declaraciones

El lenguaje define varias formas de entidades con nombre: constantes, variables, tipos, campos de tuplas, acciones o funciones. Las declaraciones asignan un nombre a dichas entidades.

Las zonas en las que se declaran variables, constantes y tipos están marcadas por tokens que indican el principio y el final de la zona de declaraciones. Así

```
declaracion_tipo → tipo lista_d_tipo ftipo;
declaracion_cte → const lista_d_cte fconst;
declaracion_var → var lista_d_var fvar;
```

El ámbito de validez de estas declaraciones difiere. Los tipos y constantes no pueden usarse hasta haber sido declarados y su nombre no puede coincidir con el de ningún otro objeto de un programa. Las variables no pueden ser usadas antes de declararse pero su rango de validez es única y exclusivamente el bloque donde son definidas.

4.1.2. Declaraciones de tipos

Podemos declarar tipos nuevos o asignarle un nombre nuevo a un tipo conocido:

```
lista_d_tipo → identificador = d_tipo; lista_d_tipo | ε
d_tipo → tupla lista_campos ftupla | tabla[ expresion_t .. expresion_t ] de d_tipo
d_tipo → identificador | expresion_t .. expresion_t | ref d_tipo | tipo_base
expresion_t → expresion | literal_caracter
lista_campos → identificador: d_tipo; lista_campos | ε
```

Los tipos base son los conocidos:

- Entero: la implementación deberá especificar el dominio aunque debe permitir valores positivos y negativos.
- Booleano: Admite los valores verdadero y falso (terminales de la gramática). El modo de almacenamiento de estos valores depende de las implementaciones.
- Carácter: Admiten valores enteros entre 0 y 255. Suponemos que existe una tabla ASCII por detrás, pero no entramos en cual es su forma.
- Real: La implementación debería aclarar el tamaño de alamacenamiento y la precisión.
- Cadena: Se implementarán como una tabla de caracteres. La primera posición indicará el tamaño

4.1.3. Declaración de constantes y variables

El lenguaje permite declarar únicamente constantes de alguno de los tipos escalares básicos. La declaración se efectúa siguiendo el siguiente esquema:

```
lista_d_cte → identificador = literal; lista_d_cte | ε
```

Donde literal representa cualquiera de los literales que se han comentado con anterioridad.

Las variables se declaran indicando identificadores y tipos. Los tipos pueden ser un identificador o una d_tipo. Un tipo que aparezca en una declaración de variables no puede usarse más que por las variables que aparezcan en la misma declaración:

```
lista_d_var → lista_id : (identificador | d_tipo); lista_d_var | ε
lista_id → identificador, lista_id | identificador
```

Un caso particular de declaración de variables es el de las variables de entrada salida del algoritmo. Estas se declaran por medio de la estructura sintáctica:

```
decl_ent_sal → decl_ent | decl_ent decl_salida | decl_salida
decl_ent → ent lista_d_var
decl_sal → sal lista_d_var
```

Nótese que no existe el concepto de variables de entrada/salida a nivel de algoritmo.

4.1.4. Expresiones

El lenguaje trabaja con expresiones aritméticas sencillas y expresiones lógicas. Se supone que, en las expresiones aritméticas, el compilador debe diferenciar el tipo del resultado en función de los tipos de los operandos. Los operandos pueden ser constantes, variables o partes de variables estructuradas.

```
expresion → exp_a | exp_b | funcion_ll
exp_a → exp_a + exp_a | exp_a - exp_a | exp_a * exp_a | exp_a / exp_a | exp_a mod exp_a
exp_a → exp_a div exp_a | (exp_a) | operando | literal_numerico | - exp_a
exp_b → exp_b y exp_b | exp_b o exp_b | no exp_b | operando | verdadero | falso
exp_b → expresion oprel expresion | (exp_b)
operando → identificador | operando.operando | operando [expresion] | operando ref
```

Obviamente, oprel son los operadores relacionales típicos ($<, >, =, <>, >=, <=$). Las normas sobre conmutatividad y asociatividad de los operadores indicados son las habituales.

4.1.5. Instrucciones

Se definen como instrucciones la asignación la composición secuencial, la alternativa y la composición iterativa.

```
instrucciones → instruccion; instrucciones | instruccion
instruccion → continuar | asignacion | alternativa | iteracion | accion_ll
asignacion → operando := expresion
alternativa → si expresion -> instrucciones lista_opciones fsi
lista_opciones →[] expresion -> instrucciones lista_opciones | ε
iteracion → it_cota_fija | it_cota_exp
it_cota_exp → mientras expresion hacer instrucciones fmientras
it_cota_fija → para identificador := expresion hasta expresion hacer instrucciones fpara
```

Nótese que el terminal ; se emplea como separador de instrucciones, no como finalizador.

El compilador va a presuponer que la alternativa es exclusiva. Es decir, que la conjunción de todas las opciones es verdadero y que nunca son verdaderas dos alternativas a la vez. No se va a comprobar este hecho pues comprobar la veracidad de la conjunción es un problema NP-completo. Sin embargo, se pide que el compilador tenga en cuenta la exclusividad de la alternativa a la hora de generar código.

En la iteración acotada por expresión se evaluará la expresión y si es verdadera, se ejecutarán las instrucciones. Si es falsa se pasará a la siguiente expresión. En la iteración de cota fija la asignación debe ser simple y hacia una variable de tipo entero. Si el valor de la variable es menor o igual que el resultado de la expresión, se realizarán las i nstrucciones. Tras realizarse las instrucciones se asignará a la variable su sucesor.

4.1.6. Acciones y funciones

El lenguaje admite acciones como instrucciones y funciones como expresiones. No se puede emplear una acción ni una función no declarada previamente. La declaración consiste simplemente en el propio código de la función. Es decir, en un programa el código de acciones y funciones debe ir por delante que el programa principal. Siguen las gramáticas:

```
accion_d → accion a_cabecera bloque faccion
funcion_d → funcion f_cabecera bloque dev expresion ffuncion
a_cabecera → identificador ( d_par_form );
f_cabecera → identificador ( lista_d_var ) dev d_tipo;
d_par_form → d_p_form; d_par_form | ε
d_p_form → ent lista_id : d_tipo | sal lista_id : d_tipo | e/s lista_id : d_tipo
```

La expresión de funcion_d debe ser de un tipo compatible con el indicado en f_cabecera. Sólo se pueden devolver tipos básicos. Los parámetros indicados como ent son parámetros de entrada. Es decir, la acción usa los valores ue se le pasen a través de ellos pero no puede modificarlos. Los de salida (sal) son parámetros a los que la acción debe asignar un valor sin utilizar su contenido en la entrada. En los de entrada salida (e/s) la acción puede usar sus valores y modificarlos. En las funciones sólo podemos tener parámetros de entrada. De hecho, el lenguaje pide que ni se indique que el parámetro es de entrada. Las llamadas a acciones y funciones siguen el esquema:

```
accion_ll → identificador ( l_ll )
funcion_ll → identificador ( l_ll )
l_ll → expresion, l_ll | expresion
```

La lista l_ll será una lista con el mismo número de parámetros que los que se dieron cuando se definió la acción/ función. Los tipos deben coincidir uno a uno. A un parámetro formal de salida o de entrada salida debe corresponderle en l_ll una variable del tipo adecuado. A un parámetro de entrada puede corresponderle una variable o una expresión. En ambos casos, el tipo debe ser el adecuado.

5. El lenguaje intermedio

Como lenguaje intermedio vamos a emplear un código de tres direcciones similar al que se explica en clase pero ampliado con dos instrucciones. Son:

- Input variable
- Output variable

Las instrucciones input variable serán el código que deba generarse cuando se lea la definición de una variable de entrada del algoritmo. Las acciones output variable serán las que se produzcan, cuando finalice la ejecución del algoritmo para todas las variables de salida del algoritmo.

6. Tratamiento de errores

El compilador avisa al usuario de los errores cometidos ofreciéndole la máxima información posible. Si es posible, no detiene su ejecución: trata de resolver el error y seguir compilando.

7. Criterios de corrección y ejemplos

Para superar la práctica debe entregar en tiempo y forma toda la información que se ha indicado. Obviamente se trata de una práctica abierta en la que parte de los objetivos los fija usted mismo. Ahora bien, debe tener en cuenta que hay unos mínimos que debe cumplir para superar la calificación de cinco:

1. Análisis de las aplicaciones flex y bison
2. Análisis de la gramática
3. Construcción de una pareja scanner-parser que se comuniquen correctamente.
4. Debe generar código en tres direcciones para, al menos, programas escritos para una versión reducida de ProAlg que no contenga apuntadores, tablas, tuplas, acciones ni funciones

A continuación puede encontrar ejemplos de algoritmos que su compilador deberían compilar. Tenga en cuenta que si ha realizado alguna modificación sobre la gramática que le he suministrado (cosa harto probable y 100 % correcta) debe modificar en el mismo sentido los ejemplos:

Ejemplo1: Algoritmo para intercambiar dos valores

```
algoritmo intercambio;
    ent a, b: entero;
    sal a, b: entero;
{Prec: a= A AND b = B}
    var
        aux: entero;
    fvar
    aux:= a;
    a:= b;
    b:= aux
{Post: b=A AND a = B}
falgoritmo
```

El código de tres direcciones que debería generarse debería ser similar a:

```
1 input a
2 input b
3 aux := a
4 a:= b
5 b := aux
6 output a
7 output b
```

Ejemplo 2: Algoritmo para ordenar tres valores

```
algoritmo orden3;
    ent a, b, c: entero; sal max, min, med: entero; {Prec: a = A AND b = B AND c = C}
    si a >= b y a >= c ->
        max := a;
        si b >= c ->med := b; min := c
        [] b <c ->med := c; min := b
        fsi
    [] b >= a y b >= c ->
        max := b;
```

```
        si a >= c ->med := a; min := c
        [] a <c ->med := c; min := a
        fsi
    [] c >= b y c >= a ->
        max := c;
        si b >= a ->med := b; min := a
        [] b <a ->med := a; min := b
        fsi
    fsi
{Post: a=A AND b = B AND c= C ...}
falgoritmo
```

El código de tres direcciones que debería generarse debería ser similar a:

```
1 input a
2 input b
3 input c
4 if a >= b goto 6
5 goto 17
6 if a >= c goto 8
7 goto 17
8 max := a
9 if b >= c goto 11
10 goto 14
11 med := b
12 min := c
13 goto 16
14 med := c
15 min := b
16 goto
17 ...
```

Ejemplo 3: Algoritmo de suma rápida

```
algoritmo sumacomb;
    ent n, m: entero;
    sal suma: entero;
{Prec: n= N AND m = M AND M >= N >0}
    var
        i, comb: entero;
    fvar
    i:= 0;
    suma:= 0;
    comb:= m;
    mientras i <n hacer
        suma := suma + comb;
        comb := comb * (m - i -1) div (i + 2);
        i := i+1
    fmientras;
{Post: n =N AND suma = SUMATORIO(i=1..N)(M sobre i)}
falgoritmo
```

El código de tres direcciones que debería generarse debería ser similar a:

```
1 input m
```

```
2 input n
3 i := 0
4 suma:= 0
5 comb:= m
6 if i <n goto 8
7 goto 19
8 t1 := suma + comp
9 suma := t1;
10 t2 := m  i
11 t3 := t2  1
12 t4 := comb * t3
13 t5 := i + 2
14 t6 := t4 / t5
15 comb := t6
16 t7 := i +1
17 i := t7
18 goto 8
19 output suma
```