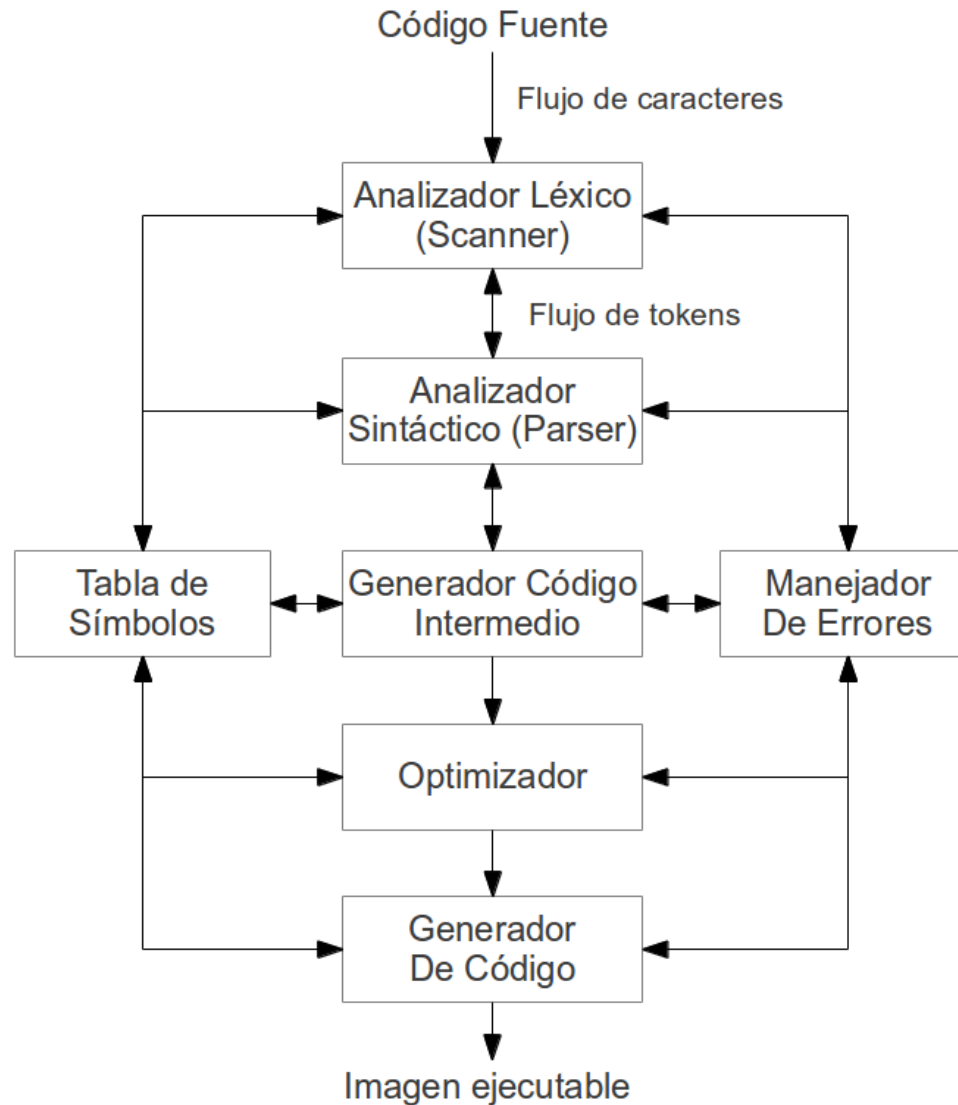


# Introducción

- Análisis (Front end)
  - Scanner: es el analizador léxico, recibe una secuencia de caracteres (el código fuente), reconoce los lexemas de los distintas categoría léxicas (LR) y genera como salida una secuencia de tokens
  - Parser: es el analizador sintáctico, recibe los tokens generados por el scanner y verifica que cumplan la GIC del lenguaje
  - Análisis semántico: realiza comprobaciones que incluyen al contexto, como el tipo de las variables, que no hay definiciones duplicadas y otras. A su vez generan un primer código intermedio
- Síntesis (Back end)
  - Optimización del código intermedio
  - Generación del código para la máquina concreta donde debe ejecutar

# Etapas



# Lenguaje Micro

- El único tipo de dato es entero.
- Todos los identificadores son declarados implícitamente y con una longitud máxima de 32 caracteres.
- Los identificadores deben comenzar con una letra y están compuestos de letras y dígitos.
- Las constantes son secuencias de dígitos (números enteros).
- Hay dos tipos de sentencias:
  - Asignación
    - `ID := Expresión;`  
Expresión es infija y se construye con identificadores, constantes y los operadores + y –; los paréntesis están permitidos.
  - Entrada/Salida
    - `leer (lista de IDs);`
    - `escribir (lista de Expresiones);`
- Cada sentencia termina con un "punto y coma" (;). El cuerpo de un programa está delimitado por **inicio** y **fin**.
- **inicio**, **fin**, **leer** y **escribir** son palabras reservadas y deben escribirse en minúscula.

# Gramática Léxica

<token> → *uno de* <identificador> <constante>  
<palabraReservada> <operadorAditivo>  
<asignación> <carácterPuntuación>  
<identificador> → <letra> {<letra o dígito>}  
<constante> → <dígito> {<dígito>}  
<letra o dígito> → uno de <letra> <dígito>  
<letra> → *una de* **a-z A-Z**  
<dígito> → *uno de* **0-9**  
<palabraReservada> → *una de* **inicio fin leer escribir**  
<operadorAditivo> → *uno de* **+ -**  
<asignación> → **:=**  
<carácterPuntuación> → *uno de* **( ) , ;**

# Tabla de Símbolos

- Estructura de datos compleja que sirve durante toda la fase de análisis para guardar y procesar información
- El diseño depende de cada implementación
- Contiene (o puede contener)
  - Identificadores (guardado por todas las implementaciones)
    - Nombre
    - Tipo
    - Lugar de almacenamiento
    - Ámbito
  - Palabras reservadas
  - Constantes literales
  - Constantes numéricas
    - Si se guardan en la TS se guarda la cadena que lo representa.
- Auxiliares: se pueden utilizar auxiliariamente: buffers, tablas de hashing, Pilas de TS

# Tabla de Símbolos

- Interactúa principalmente con las fases de análisis
  - léxico
  - sintáctico
  - semántico
- Mínimo: guardar identificadores y sus atributos (siempre enumeraciones)
- Variables
  - tipo
  - ámbito
- Funciones
  - cantidad y tipo de parámetros
  - modo de transferencia
  - tipo que retorna
- Palabras reservadas con atributo reservada (Si se guardan en la TS).

# Tabla de Símbolos

- Ejemplo función `int fx(int x, double y)`
  - (fx, funcion, int, 2, int, double)
- Ejemplo `int vdatos[20][40];`
  - (vdatos, arreglo, int, 2, 20, 40)
- TS en pila para bloques (ámbito) o variante de hash más ámbito. Suele resolver una rutina semántica
- Devoluciones (caso constante numérica)
  - Token, índice
  - Token, valor

# Tabla de Símbolos

- Caso typedef
  - typedef double** velocidad;
  - Inicialmente el escáner ingresa en la TS a velocidad como un identificador
  - Debe cambiar el atributo identificador a algo que indique “tipo de dato” y su valor sea double
  - La modificación la realiza una rutina semántica
- Buffer global para los strings
  - En la TS se guarda solo un puntero a un buffer donde se van colocando los lexemas uno tras otro en la medida que se van ingresando



# Centinelas

- En centinela es un carácter “espurio” que indica el final del lexema que se está reconociendo. Es decir, es un carácter que NO puede formar parte de ese lexema.
  - Por ejemplo, si está reconociendo un identificador y encuentra un espacio en blanco
  - En general: si encuentra un carácter que no es parte de token que se está reconociendo
- Los LR infinitos **SIEMPRE** necesitan de un centinela.
- Los LR finitos pueden necesitar o no
  - No necesitan si los lexemas del LR no comparten prefijos
  - Si hay prefijos iguales el lexema más corto necesita de centinela: P.ej: + puede seguir como ++ o encontrar una letra que forma parte de un identificador, esa letra sería el centinela
  - Si bien las palabras reservadas son finitas igual necesitan centinela porque pueden ser prefijo de un identificador. P.ej: for es prefijo de forense

# Scanner, implementación directa

- Implementaciones directas
  - Un proceso por cada LR
    - La primer parte detecta la categoría léxica, luego reconoce el lexema en particular.
    - Típicamente un switch sobre el primer carácter leído y subsecuentes aperturas por if o switch
  - Tabla de transición por switch
    - Un loop mientras haya caracteres y un switch en base al estado. En cada uno discrimina según el carácter (o clase de caracteres) leído.
    - En alguna época se usaban goto para programarlos, en lugar de variable estado y switch, se saltaba directo desde el estado anterior.

# Scanner, implementación directa

- Ventajas
  - El código sigue casi “directamente” la definición del AF
  - Fácil de escribir
  - Rápido al ejecutar
- Contras
  - Mucho código similar
  - Aburrido de escribir si no es un ejemplo simple
  - Casi nada es reutilizable para otro Scanner
  - Se pueden escapar diferencias sutiles con respecto a la definición y se hacen difíciles de detectar y corregir

# Scanner, implementación con tabla de transición

- Es como hicimos en el TP para reconocer los identificadores de un lenguaje hipotético.
- Puede ser
  - A mano como hicimos en el ejemplo
  - Con una herramienta como lex
- Filas para los estados y columnas para los caracteres o grupo de caracteres
- Al final dependiendo de si acepta (y cuál fue el estado aceptor) o no ejecutará código que maneje la situación

# Scanner, implementación con tabla de transición

- Ventajas
  - Más reutilizable
  - Menos trabajo para implementarlo
  - Si lo hicimos mediante herramientas estamos seguros de cumplir con la definición
- Contras
  - Semi mágico (documentar bien la tabla)
  - Algo más lento que la implementación directa

# Seudocódigo para TT

```
estado := INICIAL;
while not debo_parar(estado)
    Leer caracter
    estado := T[estado][caracter];
    /* otras acciones como contar líneas
       Armar lexema, etc */
end while;
if acceptor(estado) then
    if centinela(estado) then
        unput(caracter);
        accept(state); /* retornar token */
    else
        error; /* devolver token de error
                o invocar rutina de manejo de error */
    end if ;
end if ;
```

# Scanner, otras consideraciones

- Los espacios en blanco (espacio, tabulador, nueva línea) suelen ser simplemente ignorados (no siempre, por ejemplo: python)
- En los casos que es necesario un centinela, luego de leerlo se lo devuelve al flujo de entrada.
- Los LR infinitos ~~requieren~~ de un centinela. Los finitos pueden requerirlo o no.
- Es común que al detectar el final de archivo se envíe un Token que lo represente.

# Tabla de Transición

TT	L	D	+	-	(	)	,	;	:	=	fdt	esp	otro
0-	1	3	5	6	7	8	9	10	11	14	13	0	14
1	1	1	2	2	2	2	2	2	2	2	2	2	2
2+	99	99	99	99	99	99	99	99	99	99	99	99	99
3	4	3	4	4	4	4	4	4	4	4	4	4	4
4+	99	99	99	99	99	99	99	99	99	99	99	99	99
5+	99	99	99	99	99	99	99	99	99	99	99	99	99
6+	99	99	99	99	99	99	99	99	99	99	99	99	99
7+	99	99	99	99	99	99	99	99	99	99	99	99	99
8+	99	99	99	99	99	99	99	99	99	99	99	99	99
9+	99	99	99	99	99	99	99	99	99	99	99	99	99
10+	99	99	99	99	99	99	99	99	99	99	99	99	99
11	14	14	14	14	14	14	14	14	14	12	14	14	14
12+	99	99	99	99	99	99	99	99	99	99	99	99	99
13+	99	99	99	99	99	99	99	99	99	99	99	99	99
14	99	99	99	99	99	99	99	99	99	99	99	99	99



# Comentarios de la TT

Estado	Comentarios
0-	Inicial
1	Reconociendo Identificador, va almacenando cadena
2+	Identificador reconocido. Debe hacer ungetc, comprobar si el identificador es Palabra Reservada (PR), si no es, verificar largo (a lo sumo 32) y almacenar en TS
3	Reconociendo Constante (numérica), va almacenando cadena
4+	Constante reconocida. Debe hacer ungetc y en la implementación del libro agregarla a la TS
5+ al 10+	Son casos donde el lexema es un único carácter y no necesita centinela. Por tanto simplemente reconocieron cada uno el token que les corresponde
11	Reconociendo asignación
12+	Asignación reconocida
13+	Fdt reconocido (similar en cierto modo a los casos 5 al 10)
14	Error Léxico

# Gramática Sintáctica

$\langle \text{programa} \rangle \rightarrow \mathbf{inicio} \langle \text{listaSentencias} \rangle \mathbf{fin}$

$\langle \text{listaSentencias} \rangle \rightarrow \langle \text{sentencia} \rangle \{ \langle \text{sentencia} \rangle \}$

$\langle \text{sentencia} \rangle \rightarrow \langle \text{identificador} \rangle := \langle \text{expresión} \rangle ; \mid$

$\mathbf{leer} ( \langle \text{listaIdentificadores} \rangle ) ; \mid$

$\mathbf{escribir} ( \langle \text{listaExpresiones} \rangle ) ;$

$\langle \text{listaIdentificadores} \rangle \rightarrow \langle \text{identificador} \rangle \{ , \langle \text{identificador} \rangle \}$

$\langle \text{listaExpresiones} \rangle \rightarrow \langle \text{expresión} \rangle \{ , \langle \text{expresión} \rangle \}$

$\langle \text{expresión} \rangle \rightarrow \langle \text{primaria} \rangle \{ \langle \text{operadorAditivo} \rangle \langle \text{primaria} \rangle \}$

$\langle \text{primaria} \rangle \rightarrow \langle \text{identificador} \rangle \mid \langle \text{constante} \rangle \mid ( \langle \text{expresión} \rangle )$

# Parser

- Hay varios tipos de parsers. Para este ejemplo usaremos el “Análisis Sintáctico Descendente Recursivo” (ASDR)
- Un ASDR es un parser del tipo top-down, es decir, parte del axioma y va armando el Árbol de Análisis Sintáctico (AAS) con derivación a izquierda.
- Otros parsers son bottom-up es decir, a partir de los terminales (hojas del árbol) van reduciendo hacia el axioma.
- El AAS queda de modo tal que en las hojas están los terminales y en los nodos interiores los no terminales.
- Cada nodo tiene como hijos la secuencia de terminales y no terminales que representa una posible producción de ese no terminal.
- El AAS no siempre es una estructura de datos. En el caso del ASDR el árbol es virtualmente armando por los llamados recursivos entre funciones

# ASDR

- La idea es que por cada No Terminal haya un Procedimiento de Análisis Sintáctico (PAS) que lo implemente, usando el mismo nombre.
- Si el No terminal tiene una única producción la implementación es muy simple
  - Por cada No Terminal llamamos a las PAS correspondiente (y acá está la parte recursiva)
  - Por cada terminal llamamos una función Match(t) que compruebe que el terminal que provee el scanner coincida con el terminal que la producción prevee.
- Si el No Terminal tiene más de una producción se agregará código adicional para manejar la situación (veremos con el ejemplo del lenguaje Micro)

# ASDR

- Primero agregaremos un nuevo axioma que incluya en consideración el fdt, esto es para simplificar el ASDR
- También reemplazamos los Terminales por las enumeraciones que los representan. Esto NO es necesario, es solamente una cuestión didáctica, para hacerla más coincidente con los PAS que se verán luego
- **IMPORTANTE:** Match pertenece al scanner y es la **única** que hace avanzar mismo
  - Compara el token a coincidir con el próximo siguiente a entregar por el scanner.
  - Si coinciden se sigue el análisis sin problemas
  - Si no coinciden implica un error sintáctico
  - En ambos casos habilita al scanner a avanzar leyendo un nuevo token

# Gramática Sintáctica modificada

<objetivo> → <programa> FDT

<programa> → INICIO <listaSentencias> FIN

<listaSentencias> → <sentencia> {<sentencia>}

<sentencia> → ID ASIGNACION <expresión> PUNTOYCOMA |

LEER PARENIZQUIERDO <listaIdentificadores> PARENDERECHO PUNTOYCOMA |

ESCRIBIR PARENIZQUIERDO <listaExpresiones> PARENDERECHO PUNTOYCOMA

<listaIdentificadores> → ID {COMA ID}

<listaExpresiones> → <expresión> {COMA <expresión>}

<expresión> → <primaria> {<operadorAditivo> <primaria>}

<primaria> → ID |

CONSTANTE |

PARENIZQUIERDO <expresión> PARENDERECHO

<operadorAditivo> → *uno de* SUMA RESTA

# PAS de producciones simples

Por cada No Terminal llamamos al PAS correspondiente y por cada Terminal llamamos Match(Terminal)

```
void Objetivo (void) {  
    /* <objetivo> -> <programa> FDT */  
    Programa();  
    Match(FDT);  
}
```

```
void Programa (void) {  
    /* <programa> → INICIO <listaSentencias> FIN */  
    Match(INICIO);  
    ListaSentencias();  
    Match(FIN);  
}
```

# Producciones con repeticiones

- En los casos como <listaSentencias> o como <listaIdentificadores> tenemos un comienzo simple, pero luego se puede repetir opcionalmente una cantidad desconocida de veces
- Para saber si debemos repetir o no usaremos la función ProximoToken() que devuelve cuál es el token siguiente (sin avanzar el scanner, que **solo avanza** con la función **Match**)
- El ejemplo que estamos viendo tiene una particularidad en su GIC y es que es de tipo LL(1) lo que permite que viendo solamente el próximo token se pueda tomar la decisión



# PAS ListaSentencias

```
void ListaSentencias (void) {  
    /* <listaSentencias> -> <sentencia> {<sentencia>} */  
    Sentencia(); /* la primera de la lista */  
    while (1) { /* un ciclo indefinido */  
        switch (ProximoToken()) {  
            case ID:  
            case LEER:  
            case ESCRIBIR: /* detectó token correcto */  
                Sentencia(); /* procesa la secuencia  
                             opcional */  
                break;  
            default:  
                return;  
        } /* fin switch */  
    }  
}
```

# PAS producciones con opciones

- Si el No Terminal Tiene varias producciones hay que verificar si lo que sigue coincide con una de ellas (y determinar cuál de todas).
- Dado que la gramática que usamos es LL(1) podemos saberlo leyendo un token por adelantado con ProximoToken()
- Si el token leído no es ninguno de los esperados, entonces se produce un error sintáctico

# PAS Sentencia

```
void Sentencia(void) {  
    TOKEN tok = ProximoToken();  
    switch (tok) {  
        case ID: /* <sentencia> -> ID := <expresion>; */  
            Match(ID); Match(ASIGNACION);  
            Expresión(); Match(PUNTOYCOMA);  
            break;  
        case LEER: /* <sentencia> -> LEER ( <listaIdentificadores> ); */  
            Match(LEER); Match(PARENIZQUIERDO);  
            ListaIdentificadores();  
            Match(PARENDERECHO); Match(PUNTOYCOMA);  
            break;  
        case ESCRIBIR: /* <sentencia> -> ESCRIBIR (<listaExpresiones>); */  
            Match(ESCRIBIR); Match(PARENIZQUIERDO);  
            ListaExpresiones();  
            Match(PARENDERECHO); Match(PUNTOYCOMA);  
            break;  
        default:  
            ErrorSintactico(tok); break;  
    }  
}
```

# Más ejemplos de PAS

## Otro ejemplo de PAS con repeticiones

```
void listaExpresiones (void) {  
    /* <listaExpresiones> → <expresión> {COMA  
    <expresión>} */  
    Expresion(); /* la primera de la lista de  
                  expresiones */  
    while (ProximoToken() == COMA) { /* El resto de  
                                      las opcionales  
    */  
        Match(COMA); Expresion();  
    }  
}
```

# Más ejemplos de PAS

## Otro ejemplo de PAS con opciones

```
void Primaria(void) {  
    /* <primaria> → ID | CONSTANTE  
        | PARENIZQUIERDO <expresión> PARENDERECHO */  
    TOKEN tok = ProximoToken();  
    switch (tok) {  
    case ID:  
        Match(ID); break;  
    case CONSTANTE:  
        Match(CONSTANTE); break;  
    case PARENIZQUIERDO:  
        Match(PARENIZQUIERDO); Expresion();  
        Match(PARENDERECHO); break;  
    default:  
        ErrorSintactico(tok); break;  
    }  
}
```

# Análisis Semántico

- El analizador semántico, o generador de código intermedio, se encarga de:
  - Hacer los controles que son propios de una gramática sensible al contexto:
    - Comprobar si las variables están definidas
    - Comprobar el tipo de las variables
    - Comprobar tipo y cantidad de parámetros de una función
    - etc
  - Ir generando código de una máquina virtual (MV) que el compilador usa como intermedio antes de generar código para una arquitectura de procesador específica

# Análisis Semántico

- El parser es quien va llamando al analizador semántico en momentos determinados, de modo similar a como hace con el escáner
- Para ello primero se modifica la gramática y se agregan **símbolos de acción**. Para cada símbolo de acción se programará su correspondiente **rutina semántica**.
  - La gramática con los símbolos de acción agregados se conoce como gramática con anotaciones
- Los símbolos se agregan en el punto en el cuál debe llamarse a la rutina semántica. Los indicamos con un # adelante
- Para que las rutinas semánticas puedan trabajar necesitan datos.
  - Desde el punto de vista formal se conocen como atributos
  - Desde el punto de vista práctico se conocen como registros semánticos

# Rutinas semánticas

- Donde ubicar las rutinas semánticas
  - Donde haya que hacer chequeos de semántica estática (en nuestro caso solo declarar las variables cuando aparecen por primera vez)
  - Donde haya que generar código
  - Donde haya que transmitir / recibir / procesar datos (registros semánticos)



# Registros semánticos

- Dada la simpleza de micro solo hay dos tipos de registros semánticos
  - REG\_OPERACION, que solo contendrá el valor del token SUMA o RESTA.
  - REG\_EXPRESION, que contendrá el tipo de expresión y el valor que le corresponde. Este valor puede ser:
    - una cadena (para el caso de un identificador)
    - un número entero (para el caso de una constante)

# Máquina Virtual de micro

- Para Micro usaremos una MV con instrucciones del tipo:  
**OP A,B,C**
- **OP**: Código de Operación
- **A,B**: operandos
- **C**: donde almacenar el resultado
- Dependiendo de OP todos los demás operandos pueden estar o no
- Ejemplos
  - Declara dato,Entero
  - Resta dato,27,resultado (resultado := dato - 27)

# Gramática con Anotaciones

<objetivo> → <programa> FDT **#terminar**

<programa> → **#comenzar** inicio <listaSentencias> fin

<listaSentencias> → <sentencia> {<sentencia>}

<sentencia> → <identificador> := <expresión> **#asignar** ; |

leer ( <listaIdentificadores> ); |

escribir ( <listaExpresiones> );

<listaIdentificadores> → <identificador> **#leer\_id** {, <identificador> **#leer\_id**}

<listaExpresiones> → <expresión> **#escribir\_exp** {, <expresión> **#escribir\_exp**}

<expresión> → <primaria> {<operadorAditivo> <primaria> **#gen\_infijo**}

<primaria> → <identificador> |

CONSTANTE **#procesar\_cte** |

( <expresión> )

<operadorAditivo> → SUMA **#procesar\_op** | RESTA **#procesar\_op**

<identificador> → ID **#procesar\_id**

# Algunas rutinas semánticas

- Se agrega la producción <identificador> de modo tal de llamar a la rutina ProcesarID cada vez que se haga Match(ID)
- ProcesarID:
  - chequea el identificador, es decir, si no está en la Tabla de símbolos lo agrega
  - Genera el código de declaración de la variable.
  - Arma y devuelve el registro semántico del identificador
- ProximoToken: cuando pide un token al scanner lo busca en Tabla de símbolos. Esto permite corregir el token cuando es una palabra reservada (recordar que el scanner devuelve ID).

# Ejemplo PAS modificada

- PAS original

```
void Expresion (void) {  
    /* <expresion> -> <primaria> {<operadorAditivo> <primaria>} */  
    token t;  
    Primaria();  
  
    for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken()) {  
        OperadorAditivo();  
        Primaria();  
    }  
}
```

- Producción con anotaciones

<expresión> → <primaria> {<operadorAditivo> <primaria> #gen\_infijo}

- Además de generar la operación debe manejar los registros semánticos

# Ejemplo PAS modificada

- PAS con rutinas semánticas

```
void Expresion (REG_EXPRESION *resultado)
{
    /* <expresión> -> <primaria> {<operadorAditivo> <primaria> #gen_infijo}*/

    REG_EXPRESION operandoIzq, operandoDer;
    REG_OPERACION op;
    token t;

    Primaria(&operandoIzq);

    for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken()) {
        OperadorAditivo(&op);
        Primaria(&operandoDer);

        /* Probable generación de variables temporales */
        operandoIzq = GenInfijo(operandoIzq, op, operandoDer);
    }

    *resultado = operandoIzq;
}
```

# Generar infijo

- GenInfijo: Genera la instrucción para una operación infija y construye un registro semántico con el resultado
  - Va generando variables temporales según haga falta (es donde va a almacenar el resultado de la operación)
  - Según el registro de operación genera el código de operación del MV que corresponde
  - Si los registros semánticos son de un ID los chequea (incorpora a TS y genera instrucción de declaración)
  - Chequea la variable temporal (para declararla) y genera instrucción MV
  - Arma registro semántico con la variable temporal y lo devuelve como resultado

# Ejemplo de compilación

inicio

leer(a, b);

Declare a,Integer,  
Read a,Integer,  
Declare b,Integer,  
Read b,Integer,

a := 3 + b - 5;

Declare Temp&1,Integer,  
ADD 3,b,Temp&1  
Declare Temp&2,Integer,  
SUBS Temp&1,5,Temp&2  
Store Temp&2,a,

escribir (a);

Write a,Integer,

fin

Stop , ,



# Ejemplo con paréntesis

inicio

leer(a, b);

Declare a,Integer,  
Read a,Integer,  
Declare b,Integer,  
Read b,Integer,

a := 3 + (b - 5);

Declare Temp&1,Integer,  
SUBS b,5,Temp&1  
Declare Temp&2,Integer,  
ADD 3,Temp&1,Temp&2  
Store Temp&2,a,

escribir (a);

Write a,Integer,

fin

Stop , ,

# Licencia

*Esta obra, © de Eduardo Zúñiga, está protegida legalmente bajo una licencia Creative Commons, **Atribución-CompartirDerivadasIgual 4.0 Internacional**.*

*<http://creativecommons.org/licenses/by-sa/4.0/>*

*Se permite: copiar, distribuir y comunicar públicamente la obra; hacer obras derivadas y hacer un uso comercial de la misma.  
Siempre que se cite al autor y se herede la licencia.*

