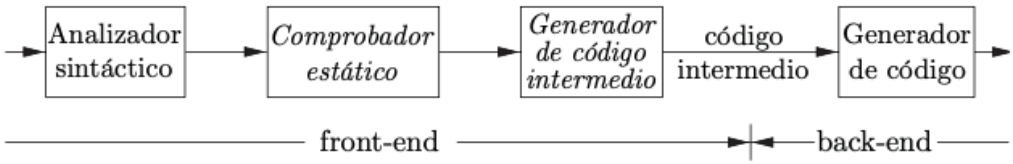
Generación de Código Intermedio

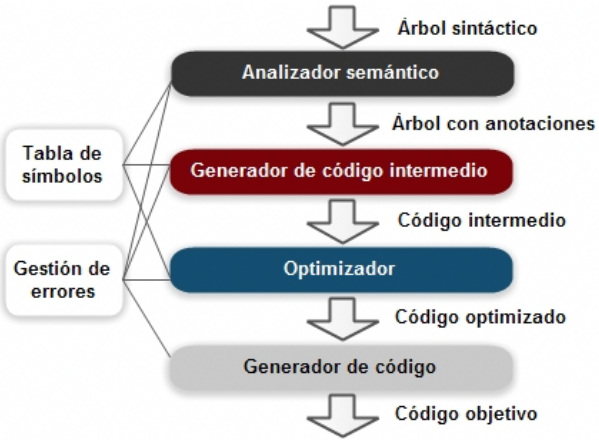
En el *modelo de análisis y síntesis de un compilador*, el *front-end* analiza un programa fuente y crea una ***representación intermedia***, a partir de la cual el *back-end* genera el código destino. Lo apropiado es que los detalles del lenguaje fuente se confinen al *front-end*, y los detalles de la máquina de destino al *back-end*.



El motivo por el que se realiza este paso intermedio es una razón de simplicidad, se **genera código intermedio** es evitar tener que construir un compilador desde cero para todas y cada una de las **diferentes arquitecturas**. Además, ello va a reportar otros beneficios, ya que nos permitirá optimizar mejor el código, ya que el **proceso de optimización** trabajará sobre este mismo código intermedio. Por último, también favorece la reutilización de código.

Tome en cuanta que **a partir de la realización el análisis semántico**, nos podríamos saltar el resto de los pasos de la fase de síntesis, **generando código fuente directamente**, como lo hacen algunos lenguajes que realizan **la ejecución sobre el árbol con anotaciones**.

La tarea de análisis suele terminar generando un código intermedio. El **código intermedio no es el lenguaje de programación de ninguna máquina real**, sino que corresponde a una máquina abstracta, que se debe de definir lo más general que se pueda, de forma que sea posible traducir este código intermedio a cualquier máquina real.



En el análisis semántico, **la comprobación estática y la generación de código intermedio se realizan en forma secuencial**; algunas veces pueden combinarse y mezclarse en el análisis semántico. La comprobación estática incluye la comprobación de tipos, la cual asegura que los operadores se apliquen a los operandos compatibles.

La generación de código es la tarea más complicada de un compilador. Las ventajas de utilizar esta **representación intermedia, independiente de la máquina** en la que se va a ejecutar el programa, son:

- Se puede **crear un compilador para una nueva máquina distinta** uniendo la etapa final de la nueva máquina a una etapa inicial ya existente. Lo que facilita cambiar el destino del código objeto.

- Se puede aplicar, a la representación intermedia, un **optimizador de código independiente de la máquina**.

El **código intermedio puede tener muchas formas**, existiendo casi tantos estilos de código intermedio como compiladores. Algunos de estos métodos tratan de ***linealizar* el árbol sintáctico**, es decir, realizan una representación secuencial del árbol sintáctico. Esto es así cuando después se va a aplicar un optimizador y se quiere obtener código muy eficiente.

Hay que indicar que la **representación del código intermedio depende de la máquina para la que se está generando el código** (máquina objeto) y esta puede ser de la siguiente forma:

- ***Máquinas de pila (0 direcciones)*** Se refiere a una máquina (física o virtual) con un conjunto de instrucciones de 0 direcciones (todas las direcciones son implícitas), llamada así porque opera con valores que se leen o insertan en el tope de la pila (operaciones de push y pop).

- ***Máquinas de registro (2 direcciones)*** Cuando se utilizan máquinas de registro una de las direcciones se usa para especificar uno de los operandos fuente el resultado y la otra dirección para el otro operando fuente (a = a + b).

- ***Máquinas RISC (3 direcciones)*** Cuando se cuenta con instrucciones de tres direcciones (lo habitual en el entorno PC) se utilizan dos direcciones para los operandos fuente y una dirección para el resultado. (a = b + c).

Tipos de Código Intermedio

No existe un lenguaje intermedio universal, de manera que revisaremos las estructuras mas conocidas para generar código intermedio, se tienen *3 tipos de implementaciones*:

1. ***Árbol de Sintaxis Abstracta (ASA)***: tiene sentido utilizarlos cuando no se va a realizar no una optimización de código y directamente se va a generar código. Tiene una versión mejorada formada por los grafos dirigidos acíclicos (GDA)

2. ***Código P***: comenzó como un código ensamblador objetivo estándar producido por varios compiladores de Pascal en la década de 1970 y principios de la de 1980. Fue diseñado par el código real de una máquina de pila hipotética.

3. ***Código de tres direcciones***: este código está pensado para las operaciones aritméticas y tiene la siguiente forma: $x = y\ op\ z$.

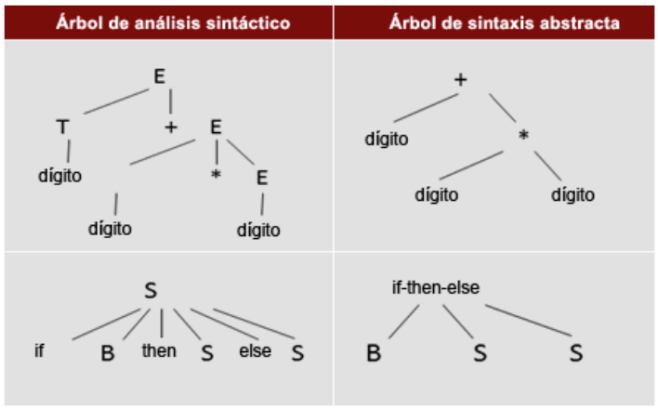
Veremos la forma de implementar la representación con ASA, código de tres direcciones y código P. En cualquier caso y se use la notación que se use, para realizar la traducción de código intermedio a código objeto no hay que volver a recorrer el árbol para generar el código final.

Algunos autores de compiladores suelen dividir los tipos de representaciones intermedias (IR) en dos tipos:

- IR gráficas ASA y grafos dirigido acíclico GDA
- IR Lineales (código de tres direcciones y código P).

Árbol de Sintaxis Abstracta (ASA)

El uso de árboles sintácticos como representación intermedia permite que la traducción se separe del análisis sintáctico. Es importante indicar que los tipos de árboles que nos interesan son los árboles de sintaxis abstracta (ASA) que son árboles de análisis sintácticos a los que se les han eliminado los símbolos superfluos y es muy útil para representar las construcciones del lenguaje.



En un ASA nos interesan las operaciones y los operadores y es por eso por lo que eliminamos los símbolos innecesarios (terminales de la gramática), generándose un árbol condensado.

Sabemos que un ASA se construye:

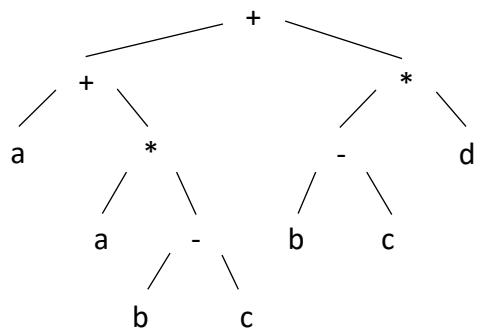
- Generando un nodo para cada operador y cada operando.
- Los hijos de un nodo operador son las raíces de los nodos que representan las sub-expresiones que constituyen los operandos de dicho operador.

Para llevarlo a la práctica necesitamos unas funciones auxiliares que como vimos en el análisis sintáctico se encargan de la creación de los nodos dentro del árbol de sintaxis abstracta (ASA).

Los nodos en un árbol sintáctico representan construcciones en el programa fuente; los hijos de un nodo representan los componentes significativos de una construcción. Un grafo acíclico dirigido (**GDA**) para una expresión identifica a las subexpresiones comunes (que ocurren más de una vez) de la expresión. Como veremos, pueden construirse **GDAs** mediante el uso de las mismas técnicas que construyen los árboles sintácticos.

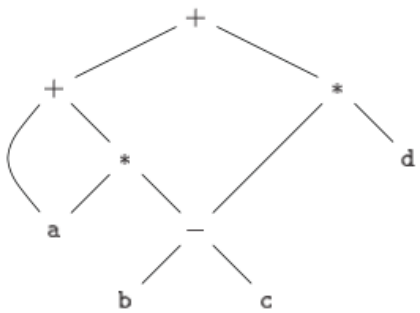
GDA para expresiones

Al igual que el árbol sintáctico para una expresión, un GDA tiene hojas que corresponden a los operandos atómicos, y códigos interiores que corresponden a los operadores. La diferencia es que un nodo N en un GDA tiene más de un padre si N representa a una subexpresión común; en un árbol sintáctico, el árbol para la subexpresión común se replica tantas veces como aparezca la subexpresión en la expresión original. Por ende, un **GDA** no solo representa a las expresiones en forma más breve, sino que también proporciona pistas importantes al compilador, en la generación de código eficiente para evaluar las expresiones:




```
a + a * (b - c) + (b - c) * d
```

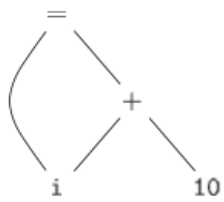
La hoja para *a* tiene dos padres, ya que a aparece dos veces en la expresión. Lo más interesante es que dos ocurrencias de la subexpresión común *b−c* se representan mediante un nodo, etiquetado como −. Ese nodo tiene dos padres, los cuales representan sus dos usos en las subexpresiones *a*(b−c) y (b−c)*d*. Aun cuando *b* y *c* aparecen dos veces en la expresión completa, cada uno de sus nodos tiene un padre, ya que ambos usos se encuentran en la subexpresión común *b−c*.



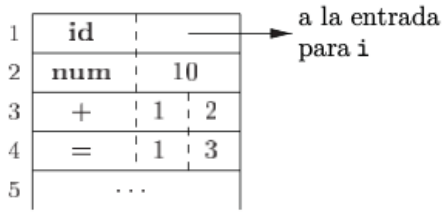
Antes de crear un nuevo nodo, estas funciones primero comprueban que ya existe un nodo idéntico. En caso de ser así, se devuelve el nodo existente. Asumimos que entrada-*a* apunta a la entrada en la tabla de símbolos para *a*, y de manera similar para los demás identificadores.

El método para construir **GDAs** requiere que los nodos de un árbol sintáctico o **GDA** se almacenan en un arreglo de registros. Cada fila del arreglo representa a un registro y, por lo tanto, a un nodo. En cada registro, el primer campo es un código de operación, el cual indica la etiqueta del nodo. En (b) las hojas tienen un campo adicional, el cual contiene el valor léxico (ya sea un apuntador a una tabla de símbolos o una constante, en este caso), y los nodos interiores tienen dos campos adicionales que indican a los hijos izquierdo y derecho.

Podríamos utilizar apuntadores en vez de índices enteros, pero de todas formas nos referiremos a la referencia de un nodo como su "*número de valor*". Si se almacenan en una estructura de datos apropiada, los números de valor nos ayudan a construir los GDAs de expresiones con eficiencia.

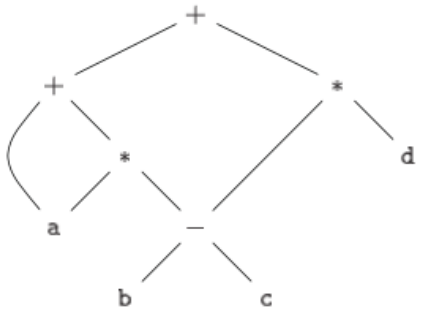


(a) DGA



(b) Arreglo

Los nodos están almacenados en un arreglo, y se hace referencia a cada nodo por su número de valor. Un nodo interior es el triple **⟨op, i, d⟩**, en donde op es la etiqueta, i el número de valor de su hijo izquierdo y d el número de valor de su hijo derecho.



1	id	a	
2	id	b	
3	id	c	
4	-	2	3
5	*	1	4
6	+	1	5
8	id	d	
9	*	4	8
10	+	6	9

Buscar en el arreglo un nodo M con la etiqueta *op*, el hijo izquierdo i y el hijo derecho r. Si hay un nodo así, devolver el número de valor M. Si no, crear un nuevo nodo N en el arreglo, con la etiqueta *op*, el hijo izquierdo i y el hijo derecho d, y devolver su número de valor.

Aunque el método produce la salida deseada, se requiere mucho tiempo para localizar un nodo, en especial si el arreglo contiene expresiones de todo un programa. Un método más eficiente podría ser usar una tabla hash.

Código de tres direcciones

Como ya se menciono de manera general existen dos tipos principales de generación de código intermedio, los gráficos representados por el ASA y GDA y los lineales representados por el código de tres direcciones.

La instrucción básica del código de tres direcciones está diseñada para representar la evaluación de **expresiones aritméticas**, aunque no de forma exclusiva, y tiene la siguiente forma general:

```
x = y op z
```

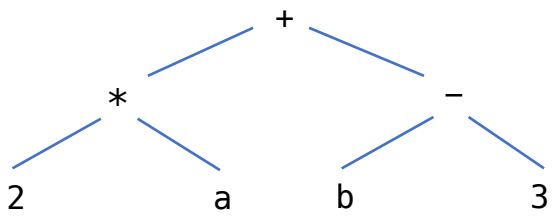
Esta instrucción expresa la aplicación de operador **op** a los valores **y** y **z**, y la asignación de este valor para que sea el nuevo valor del identificador **x**. Aquí **op** puede ser un operador aritmético como + o — o algún otro operador que pueda actuar sobre los valores de los identificadores **y** y **z**.

El nombre "código de tres direcciones" viene de esta forma de instrucción, ya que por lo general cada uno de los nombres **x**, **y** y **z** representan una dirección de la memoria. Sin embargo, observe que el uso de la dirección de **x** difiere del uso de las direcciones de **y** y **z**, y que tanto **y** como **z** (pero no **x**) pueden representar constantes o valores de literales sin direcciones de ejecución.

Para ver cómo las secuencias de código de tres direcciones de esta forma pueden representar el cálculo de una expresión, considere la expresión aritmética

```
2 * a + (b - 3)
```

con árbol abstracto



El código de tres direcciones correspondiente es

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

El código de tres direcciones requiere que el compilador **genere** nombres para **elementos temporales**, para los que hemos llamado t1, t2 y t3. Estos elementos temporales corresponden a los **nodos interiores del árbol sintáctico** y representan sus valores calculados, con el último elemento temporal (t3, en este ejemplo) representando el valor de la raíz.

La manera en que estos elementos temporales finalmente son asignados en la memoria no es especificada por este código; por lo regular serán asignados a registros.

El código de tres direcciones que se acaba de dar representa una ***linealización*** de **izquierda a derecha del árbol sintáctico**, debido a que el código correspondiente a la evaluación del subárbol izquierdo de la raíz se lista primero.

Si se desea tener *capacidad para todas las construcciones de un lenguaje de programación estándar*, será *necesario variar la forma del código de tres direcciones* en cada construcción. Si un lenguaje contiene características poco habituales, puede ser necesario incluso *inventar nuevas formas del código de tres direcciones* para expresarlas. Esta es una de las razones por las que no existe una forma estándar para el código de tres direcciones.

A continuación, se presenta un programa de muestra que calcula la función factorial de un número. El código contiene varias formas diferentes de código de tres direcciones:

1. Las operaciones integradas de entrada y salida **read** y **write** se tradujeron directamente en instrucciones de una dirección.

2. Existe una instrucción de salto condicional **if_false** que se utiliza para traducir tanto sentencias **if** como sentencias **repeat** y que contiene dos direcciones: el valor condicional que se probará y la dirección de código a la que se saltará mediante instrucciones **label**.

3. Una instrucción **halt** (sin dirección) sirve para marcar el final del código.

```
{ programa en TINY
calcula el factorial }

read x;
if 0 < x then
     fact := 1;
     repeat
          fact := fact * x;
          x := x-1
     until x = 0;
     write fact
end
```

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

La sentencia de programa de muestra `fact := fact * x;` se traduce a dos instrucciones de 3 direcciones

```
t2 = fact * x
fact = t2
```

El código de tres direcciones se basa en dos conceptos: **direcciones** e **instrucciones**. De manera alternativa, podemos implementar el código de tres direcciones usando registros con campos para las direcciones; mas adelante hablaremos sobre los registros, conocidos como cuádruplos y tripletas.

Una **dirección** puede ser una de las siguientes opciones:

- *Un nombre*. Por conveniencia, permitimos que los nombres de los programas fuente aparezcan como direcciones en código de tres direcciones. En una implementación, un nombre de origen se sustituye por un apuntador a su entrada en la tabla de símbolos, en donde se mantiene toda la información acerca del nombre.

- *Una constante*. En la práctica, un compilador debe tratar con muchos tipos distintos de constantes y variables.

- *Un valor temporal generado por el compilador*. Es conveniente, en especial con los compiladores de optimización, crear un nombre distinto cada vez que se necesita un valor temporal. Estos valores temporales pueden combinarse, si es posible, cuando se asignan los registros a las variables.

Las etiquetas simbólicas son para uso de las **instrucciones** que alteran el flujo de control. Una etiqueta simbólica representa el índice de una instrucción de tres direcciones en la secuencia de instrucciones. Los índices reales pueden sustituirse por las etiquetas. He aquí una lista de las formas comunes de instrucciones de tres direcciones:

- Instrucciones de asignación de la forma *x = y op z*, en donde *op* es una operación aritmética o lógica binaria, y *x, y* y *z* son direcciones.

- Asignaciones de la forma *x = op y*, en donde *op* es una operación unaria.

- Instrucciones de copia de la forma x = y, en donde a *x* se le asigna el valor de *y*.

- Salto incondicional *goto L*. La instrucción de tres direcciones con la etiqueta *L* es la siguiente que se va a ejecutar.

- Saltos condicionales de la forma *if x goto L* e *if_false x goto L*. Estas instrucciones ejecutan a continuación la instrucción con la etiqueta L si x es verdadera y falsa, respectivamente. En cualquier otro caso, la siguiente instrucción en ejecutarse es la instrucción de tres direcciones que siga en la secuencia, como siempre.

- Las llamadas a los procedimientos y los retornos se implementan mediante el uso de las siguientes instrucciones: *param x* para los parámetros; *call p, n* y *y = call p, n* para las llamadas a procedimientos y funciones, respectivamente; y *return y*, en donde *y*, que representa a un valor de retorno, es opcional.

- Instrucciones de copia indexadas, de la forma *x = y[i]* y *x[i] = y*. La instrucción *x = y[i]* establece *x* al valor en la ubicación que se encuentra a *i* unidades de memoria más allá de *y*.

- Asignaciones de direcciones y apuntadores de la forma *x = &y*, *x = * y* y *** x = y*. La instrucción *x = &y* establece el **r-value** de *x* para que sea la ubicación (**l-value**) de *y*.

Se supone que *y* es un nombre, tal vez temporal, que denota a una expresión con un **l-value** tal como *A[i][j]*, y que *x* es el nombre de un apuntador o valor temporal.

La elección de operadores permitidos es una cuestión importante en el diseño de una forma intermedia. Es evidente que el conjunto de operadores debe ser lo bastante amplio como para implementar las operaciones en el lenguaje fuente. Los operadores cercanos a las instrucciones de máquina facilitan la implementación de la forma intermedia en una máquina de destino.

Tipos de códigos de tres direcciones

La descripción de las instrucciones de tres direcciones especifica los componentes de cada tipo de instrucción, pero no especifica la representación de estas instrucciones en una estructura de datos. En un compilador, estas instrucciones pueden implementarse como objetos o como registros, con campos para el operador y los operandos. Tres de estas representaciones se conocen como "cuádruplos", "tripletas" y "tripletas indirectas".

La implementación mas común, conocida como **cuádruplos** consiste en emplear cuatro campos: uno para la operación y tres para las direcciones. Para las instrucciones que necesitan menos de tres direcciones, uno o más de los campos de dirección proporcionan un valor nulo.

Un cuádruplo tiene cuatro campos, a los cuales llamamos ***op***, ***arg1***, ***arg2*** y ***resultado***. El campo *op* contiene un código interno para el operador. Por ejemplo, la instrucción de tres direcciones *x = y + z* se representa colocando a *+* en *op*, *y* en *arg1*, *z* en *arg2* y *x* en *resultado*. A continuación, se muestran dos excepciones a esta regla:

- Las instrucciones con operadores unarios no utilizan arg2. Observe que para una instrucción de copia como *x = y*, *op* es *=*, mientras que, para la mayoría de las otras operaciones, el operador de asignación es implícito.
- Los operadores como param no utilizan arg2 ni resultado.

El código anterior puede ser representado mediante cuádruplos:

`{ programa en TINY`
`calcula el factorial }`
 	
`read x;`
`if 0 < x then`
`  fact := 1;`
`  repeat`
`  fact := fact * x;`
`  x := x-1`
`  until x = 0;`
`  write fact`
`end`
	`(rd,x,_,_)`
	`(gt,x, 0,t1)`
	`(if_f,t1,L1,_)`
	`(asn,1,fact,_)`
	`(lab,L2,_,_)`
	`(mul,fact,x,t2)`
	`(asn,t2,fact,_)`
	`(sub,x,1,t3)`
	`(asn,t3,x,_)`
	`(eq,x,0,t4)`
	`(if_f,t4,L2,_)`
	`(wri,fact,_,_)`
	`(lab,L1,_,_)`
	`(halt,_,_,_)`

Una implementación diferente del código de tres direcciones consiste en utilizar las instrucciones mismas para representar los elementos temporales. Esto reduce la necesidad de campos de dirección de tres a dos, puesto que en una instrucción de tres direcciones que contiene la totalidad de las tres direcciones, la dirección objetivo es siempre un elemento temporal. Una implementación del código de tres direcciones de esta clase se denomina **triple** y requiere que cada instrucción de tres direcciones sea referenciable, ya sea como un índice en un arreglo.

Un **triple** sólo tiene tres campos, a los cuales llamamos ***op***, ***arg1*** y ***arg2***. Observe que el campo *resultado* se utiliza principalmente para los nombres temporales. Al usar tripletas, nos referimos al resultado de una operación *x op y* por su posición, en vez de usar un nombre temporal explícito. Por ende, en vez del valor temporal t1, una representación en tripletas se referiría a ***la posición (0)***. Los números entre paréntesis representan apuntadores a la misma estructura de las tripletas. A las posiciones o apuntadores a las posiciones se les llamó números de valor.

Las referencias de triple también se distinguen de las constantes poniéndolas entre paréntesis en los mismos triples. Adicionalmente, eliminamos las instrucciones **label** y las reemplazamos con referencias a los índices de triple mismos.

```
{ programa en TINY
calcula el factorial }

read x;
if 0 < x then
     fact := 1;
     repeat
          fact := fact * x;
          x := x-1
     until x = 0;
     write fact
end
```

```
(0)   (rd,x,_)
(1)   (gt,x, 0)
(2)   (if_f,(1),(11))
(3)   (asn,1,fact)
(4)   (mul,fact,x)
(5)   (asn,(4),fact)
(6)   (sub,x,1)
(7)   (asn,(6),x)
(8)   (eq,x,0)
(9)   (if_f,(8),(4))
(10)  (wri,fact,_)
(11)  (halt,_,_)
```

Código P

El código P comenzó como un código ensamblador objetivo estándar producido por varios compiladores Pascal en la década de 1970 y principios de la de 1980. Fue diseñado para ser el código real de una máquina de pila hipotética, denominada máquina P, para la que fue escrito un intérprete en varias máquinas reales.

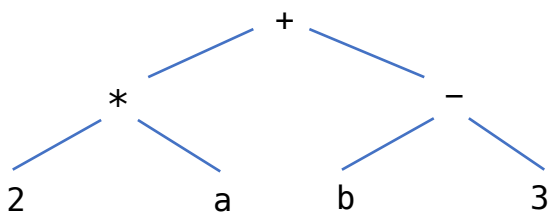
La idea era hacer que los compiladores de Pascal se transportaran fácilmente requiriendo sólo que se volviera a escribir el intérprete de la máquina P para una nueva plataforma. El código P también ha probado ser útil como código intermedio, y se han utilizado varias extensiones.

Como el código P fue diseñado para ser directamente ejecutable, contiene una descripción implícita de un ambiente de ejecución particular que incluye tamaños de datos. además de mucha información específica para la máquina P.

Para nuestros propósitos la máquina P está compuesta por una memoria de código, una memoria de datos no especificada para variables nombradas y una pila para datos temporales, junto con cualquier registro que sea necesario para mantener la pila y apoyar la ejecución.

Como un primer ejemplo de código P, considere la expresión

```
2 * a + (b – 3)
```



Nuestra versión de código P para esta expresión es la que se muestra en seguida:

```
ldc 2      ;carga la constante 2
lod a      ;carga el valor de la variable a
mpi        ;multiplicación entera
lod b      ;carga el valor de la variable b
ldc 3      ;carga la constante 3
sbi        ;sustracción o resta entera
adi        ;adición de enteros
```

Estas instrucciones se ven como si representaran las siguientes operaciones en una máquina P. En primer lugar, ldc 2 inserta el valor 2 en la pila temporal. Luego, lod a inserta el valor de la variable a en la pila. La instrucción mpi extrae estos dos valores de la pila, los multiplica (en orden inverso) e inserta el resultado en la pila. Las siguientes dos instrucciones (lod b y ldc 3)insertan el valor de b y la constante 3 en la pila (ahora tenemos tres valores en la pila). Posteriormente, la instrucción sbi extrae los dos valores superiores de la pila, resta el primero del segundo, e inserta el resultado. Finalmente, la instrucción adi extrae los dos valores restantes de la pila, los suma e inserta el resultado. El código finaliza con un solo valor en la pila, que representa el resultado del cálculo.

Retomando el ejemplo de código Tiny que genera la factorial de un numero veamos ahora su representación en código P.

```
lda x      ; carga dirección de x
rdi        ; lee un entero, almacena a la
           ; dirección en el tope de la pila ( y la extrae)
lod x      ; carga el valor de x
ldc 0      ; carga la constante O
grt        ; extrae y compara dos valores del tope
           ; inserta resultado booleano
fjp L1     ; extrae resultado booleano, salta a L1 si es falso
lda fact   ; carga dirección de fact
ldc 1      ; carga la constante 1
sto        ; extrae dos valores, almacenando el primero en la
```

```
          ; dirección representada por el segundo
lab L2    ; definición de etiqueta L2
lda fact  ; carga dirección de fact
lod fact  ; carga valor de fact
lod  x    ; carga valor de x
mpi       ; multiplica
sto       ; almacena el tope a dirección del segundo y extrae
lda  x    ; carga dirección de x
lod x     ; carga valor de x
ldc 1     ; carga constante 1
sbi       ; resta
sto       ; almacena
lod x     ; carga valor de x
ldc  0    ; carga constante O
equ       ; prueba de igualdad
fjp L2    ; salto a L2 si es falso
lod fact  ; carga valor de fact
wri       ; escribe tope de pila y extrae
lab L1    ; definición de etiqueta L1
stp
```

Comparación del código P con el código de tres direcciones

El código P en muchos aspectos está más cercano al código de máquina real que al código de tres direcciones. Las instrucciones en código P también requieren menos direcciones, "una dirección" o "cero direcciones".

Por otra parte, el código P es menos compacto que el código de tres direcciones en términos de números de instrucciones y el código P no está "auto contenido" en el sentido que las instrucciones funcionen implícitamente en una pila. La ventaja respecto a la pila es que contiene todos los valores temporales necesarios en cada punto del código y el compilador no necesita asignar nombres temporales a ninguno de ellos, como en el código de tres direcciones.

Técnicas básicas de generación de código

El código intermedio se puede procesar como un atributo sintetizado

La generación de código intermedio se puede ver como un cálculo de atributo similar a muchos de los problemas de atributo estudiados previamente. En realidad, si el código generado se ve como un atributo de cadena, entonces este código se convierte en un atributo sintetizado que se puede definir utilizando una gramática con atributos, y generado directamente durante el análisis sintáctico o mediante un recorrido postorden del árbol sintáctico.

Para ver cómo el código de tres direcciones, o bien, el código P se pueden definir como un atributo sintetizado, considere la siguiente gramática que representa un pequeño subconjunto de expresiones en C:

```
exp → id = exp | aexp
aexp → aexp + factor | factor
factor → ( exp ) | num | id
```

Esta gramática sólo contiene dos operaciones, la asignación y la suma. El *token* ***id*** representa un identificador simple, y el *token* ***num*** simboliza una secuencia simple de dígitos que representa un entero. Se supone que ambos *tokens* tienen un atributo ***strval*** previamente calculado, que es el valor de la cadena, o lexema, del *token* (por ejemplo, "42" para un num o "xtemp" para un id).

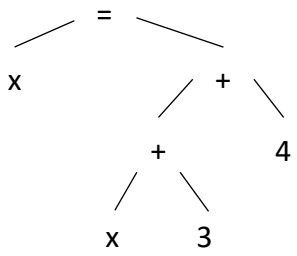
Veamos el caso de la generación del código P en primer lugar, ya que la gramática con atributos es más simple debido a que no es necesario generar nombres para elementos temporales. Sin embargo, la existencia de asignaciones incrustadas es un factor que implica complicaciones.

En esta situación deseamos mantener el valor almacenado como el valor resultante de una expresión de asignación, ya que la instrucción de código P estándar **sto** es destructiva, pues el valor asignado se pierde. Resolvemos este problema introduciendo una instrucción de almacenamiento no destructiva **stn** en nuestro código P, ya que como la instrucción **sto**, supone que se encuentra un valor en la parte superior o tope de la pila y una dirección debajo de la misma; **stn** almacena el valor en la dirección, pero deja el valor en el tope de la pila, mientras que descarta la dirección. Con esta nueva instrucción, una gramática con atributos para un atributo de cadena de código P se proporciona en la siguiente tabla.

En esa gramática utilizamos el nombre de atributo ***pcode*** para la cadena de código P. También empleamos dos notaciones, diferentes para la concatenación de cadena: ++ cuando las instrucciones van a ser concatenadas *con retornos de línea* insertados entre ellas y || cuando se está construyendo una *instrucción simple* y se va a insertar un espacio.

En base a las reglas semánticas el cálculo del atributo **pcode** para la expresión *x=x+3+4* genera la linealización del código:

```
lda x
lod x
ldc 3
adi
ldc 4
adi
stn
```



Gramática de código P como un atributo de cadena sintetizado

Regla gramatical	**Reglas semánticas**
exp1 → id = exp2	exp1.pcode="**lda**"||id.strval++ exp2.pcode ++ "**stn**"
exp → aexp	exp.pcode = aexp.pcode
aexp1 → aexp2 + factor	aexp1.pcode= aexp2.pcode ++ factor.pcode ++ "**adi**"
aexp → factor	aexp.pcode = factor.pcode
factor →(exp)	factor.pcode = exp.pcode
factor → num	factor.pcode = "**ldc**" || num.strval
factor → id	factor.pcode = "**lod**" || id.strval

Una gramática de expresión simple con atributos para código de tres direcciones se proporciona en la tabla, utilizamos el atributo de código ***tacode*** (para código de tres direcciones), y como en el ejemplo anterior, ++ se emplea para concatenación de cadena con un retorno de línea y || para concatenación de cadena con un espacio.

A diferencia del código P, el código de tres direcciones requiere que se generen nombres temporales para resultados intermedios en las expresiones, y esto requiere que la gramática con atributos incluya un nuevo atributo ***name*** para cada ***n***. Este atributo también es sintetizado, para asignar nombres temporales recién generados a nodos interiores con la función ***newtemp()*** que genera secuencias *t1*, *t2*, *t3*, ... de nombres. Sólo los nodos interiores necesitan nombres temporales; la operación de asignación utiliza el nombre de la expresión en el lado derecho.

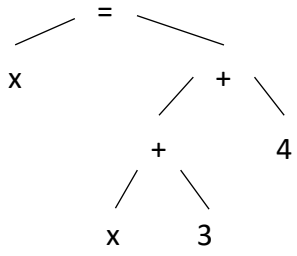
Advierta en la tabla que en el caso de las producciones unitarias exp→aexp y aexp→factor, el atributo ***name***, además del atributo ***tacode***, se transporta de hijos a padres y que, en el caso de los nodos interiores del operador, se generan nuevos atributos ***name*** antes del ***tacode*** asociados. observe también que, en las producciones de hoja factor → num y factor → id, el valor de cadena del *token* se utiliza factor.name, que (a diferencia del código P) no se genera ningún código de tres direcciones en tales nodos (utilizamos " " para representar la cadena vacía).

Gramática de código de tres direcciones como un atributo de cadena sintetizado

Regla gramatical	**Reglas semánticas**
exp1 → id = exp2	exp1.name = exp2.name exp1.code = exp2.code ++ id.strval ||"="||exp2.name
exp → aexp	exp.name = aexp.name exp.code = aexp.code
aexp1 → aexp2 + factor	aexp1.name = **newtemp**() aexp1.code = aexp2.code ++ factor.code ++ aexp1.name || "=" || aexp2.name ||"+"||factor.name
aexp1 → - aexp2	aexp1.name = **newtemp**() aexp1.code = aexp2.code ++ aexp1.name = "=" || "-"|| aexp2.name
aexp → factor	aexp.name = factor.name aexp.code = factor.code
factor →(exp)	factor.name = exp.name factor.code = exp.code
factor → num	factor.name = num.strval factor.code = ""
factor → id	factor.name = id.strval factor.code = ""

Para la expresión *x=x+3+4* genera la linealización del código de tres direcciones:

```
t1 = x + 3
t2 = t1 + 4
x = t2
```



La función ***newtemp()*** es llamado en postorden y genera nombres temporales comenzando con *t1*. Note cómo la asignación ***t1 = x + 3*** genera dos instrucciones de tres direcciones utilizando un elemento temporal. Esto es una consecuencia del hecho de que la evaluación de atributos siempre crea un elemento temporal para cada sub-expresión, incluyendo los lados derechos de las asignaciones.

Visualizar la generación de código como el cálculo de un ***atributo de cadena sintetizado*** es útil para mostrar claramente las relaciones entre las secuencias de código de las diferentes partes del árbol sintáctico y para comparar los diferentes métodos de generación de código, pero es poco practico como técnica para generación de código real, por varias razones.

En primer lugar, el uso de la concatenación de cadena causa que se desperdicie una cantidad desmesurada de copiado de cadenas y memoria a menos que los operadores de la concatenación sean muy complejos. En segundo, por lo regular es mucho más deseable generar pequeños segmentos de código a medida que continúa la generación de código y escribir estos segmentos en un archivo o bien insertarlos en una estructura de datos (tal como un arreglo de cuádruples).

Finalmente, aunque es útil visualizar código como puramente sintetizado, la generación de código en general depende mucho de atributos heredados, y esto complica en gran medida las gramáticas con atributos. En vez de eso, emplearemos técnicas de generación código más directas.

Generación de código en la practica

Las técnicas estándar de generación de código involucran modificaciones de los recorridos postorden del árbol sintáctico. El algoritmo básico puede ser descrito mediante un procedimiento recursivo

```
procedure genCode ( T: treenode )
{
    if T no es null then
         genere código para preparar en el caso del código del hijo
            izquierdo de T;
         genCode(hijo izquierdo de T);
         genere código para preparar en el caso del código del hijo
            derecho de T;
         genCode(hijo derecho de T);
         genere código para implementar la acción de T;
}
```

Observe que este procedimiento recursivo no sólo tiene un componente postorden (que genera código para implementar la acción de T) sino también un componente de *preorden* y un componente *enorden* (que genera el código de preparación para los hijos izquierdo derecho de T). En general, cada acción que representa T requerirá una versión un poco diferente del código de preparación *preorden* y *enorden*.

Basados en la estructura para un árbol abstracto podemos escribir un procedimiento ***genCode*** para generar código. En esa función haremos los comentarios siguientes acerca del código. En primer lugar, el código utiliza la función estándar de impresión para concatenar las cadenas en el elemento temporal local *codestr*. En segundo lugar, se llama el procedimiento ***emitCode*** para generar una línea simple de código, ya sea en una estructura de datos o en un archivo de salida.

```
void gencode( SyntaxTree t)
{
   char codestr[CODESIZE];
   /* CODESIZE = longitud máxima de 1 línea de código P */
   if (t != NULL) {
      switch(t->kind){
         case OpKind:
            switch (t->op) {
               case Plus:
                  genCode(t->lchild);
                  genCode(t->rchild);
                  emitcode( "adi");
                  break;
               case Assign:
                  sprintf(codestr,"%s %s","lda",t->strval);
                  emitCode(codestr);
                  genCode(t->lchild);
                  emitCode("stn");
                  break;
               default:
                  emitCode("Error");
                  break;
            }
            break;
         case ConstKind:
            sprintf(codestr,"%s %s","ldc",t->strval);
            emitCode(codestr);
            break;
         case IdKind:
            sprintf(codestr,"%a %a","lod",t->strval);
            emitCode(codestr);
            break;
         default :
            emitCode("Error");
            break;
      }
   }
}
```

Generación de código en sentencias de control y expresiones lógicas

La generación de código intermedio para ***sentencias de control***, tanto en *código de tres direcciones* como en *código P*, involucra la generación de etiquetas de una manera similar a la generación de nombres temporales en el código de tres direcciones, pero en este caso representan direcciones en el código objetivo a las que se harán los saltos.

Las ***expresiones lógicas***, o booleanas, que se utilizan como pruebas de control, y que también se pueden emplear de manera independiente como datos, se analizan a continuación, particularmente respecto a la *evaluación de cortocircuito*, en la cual difieren de las expresiones aritméticas.

Las expresiones booleanas están compuestas de los operadores booleanos (AND, OR y NOT) que se aplican a elementos que son variables booleanas o expresiones relacionales. Estas expresiones relacionales son de la forma E1 rel E2, en donde E1 y E2 son expresiones aritméticas. En esta sección, consideraremos las expresiones booleanas generadas por la siguiente gramática:

```
B → B or B | B and B | not B
          | (B) | exp rel exp | true | false
```

En el ***código de corto circuito*** (o de salto), los operadores booleanos *and*, *or* y *not* se traducen en saltos. Los mismos operadores no aparecen en el código; en vez de ello, el valor de una expresión booleana se representa mediante una posición en la secuencia de código.

La instrucción:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

podría traducirse en el código de tres direcciones:

```
if x < 100 goto L2
if_false x > 200 goto L1
if_false x != y goto L1
lable L2
x = 0
lable L1
```

En esta traducción, la expresión booleana es verdadera si el control llega a la etiqueta L2. Si la expresión es falsa, el control pasa de inmediato a L1, ignorando a L2 y la asignación x = 0.

La generación de código para sentencias de control

Consideremos primero las siguientes dos formas de sentencias ***if*** y ***while***, son similares en muchos lenguajes, en general las gramáticas se comportan como sigue:

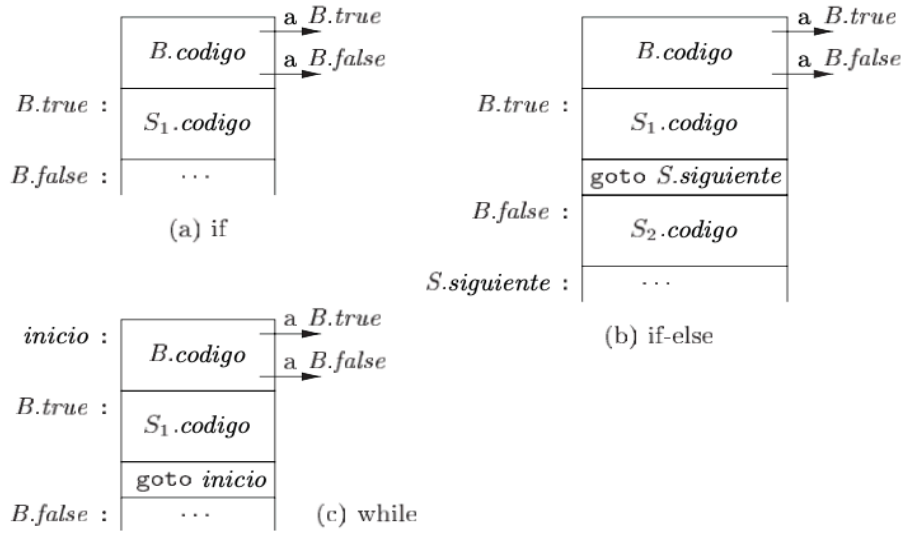
```
sent-if → if (B) S |
          if (B) S else S

sent-while → while (B) S
```

En estas producciones, el no terminal ***B*** representa a una ***expresión booleana*** y en el no terminal S que representa a una instrucción o secuencias de intrucciones. El problema principal en la generación de código para tales sentencias es traducir las características de control estructurado en un equivalente "no estructurado" que involucre saltos.

Tanto ***B*** como ***S*** tienen un *atributo sintetizado* llamado ***codigo***, el cual proporciona la traducción en *instrucciones de tres direcciones*. Por simplicidad, generamos las traducciones *B.codigo* y *S.codigo* como cadenas, usando ***definiciones dirigidas por la sintaxis***. Las reglas semánticas que definen los atributos ***codigo*** podrían implementarse y emitir código durante el recorrido de un árbol.

La traducción de ***if (B) S_1*** consiste en *B.codigo* seguida de *S_1.codigo*, como se muestra en la figura (a). Dentro de *B.codigo* hay saltos con base en el valor de *B*. Si *B* es verdadera, el control fluye hacia la primera instrucción de *S_1.code*, y si *B* es falsa, el control fluye a la instrucción que sigue justo después de *S_1.codigo*.



Las etiquetas para saltos en *B.codigo* y *S.codigo* se pueden administrar usando ***atributos heredados***. Con una *expresión booleana B*, asociamos dos etiquetas: *B.true*, que fluye el control si *B* es verdadera, y *B.false*, que fluye el control si *B* es falsa. Con una instrucción S, asociamos un *atributo heredado S.siguiente* que denota a una etiqueta para la instrucción que sigue justo después del código para *S*. En algunos casos, la instrucción que va justo después de *S.codigo* es un salto hacia alguna etiqueta ***L***. Un salto hacia ***L*** desde el interior de *S.codigo* se evita mediante el uso de *S.siguiente*.

Regla gramatical	**Reglas semánticas**
`P → S`	`S.next = newLabel()` `P.code = S.code || lable(S.next)`
`S → S = E`	`S.code = E.code`

`S → if (B) S1`	`B.true = newLabel()`
	`B.false = S1.next = S.next`
	`S.code = B.code || label(B.true) || S1.code`

`S → if(B) S1 else S2`	`B.true = newLabel()`
	`B.false = newLabel()`
	`S1.next = S2.next = S.next`
	`S.code = B.code`
	`|| label(B.true) || S1.code`
	`|| 'goto' S.next`
	`|| label(B.false) || S2.code`

`S → while ( B ) S1`	`begin = newLabel()`
	`B.true = newLabel()`
	`B.false = S.next`
	`S1.next = begin`
	`S.code = label(begin) || B.code`
	`|| label(B.true) || S1.code`
	`|| 'goto' || begin`

Un programa consiste en una instrucción generada por P → S. Las reglas semánticas asociadas con esta producción inicializan S.siguiente con una nueva etiqueta. P.codigo consiste en S.codigo seguido de la nueva etiqueta S.siguiente.

Al traducir S → if (B) S1, las reglas semánticas crean una nueva etiqueta B.true y la adjuntan a la primera instrucción de tres direcciones generada para la instrucción S1. Por ende, los saltos a B.true dentro del código para B irán al código para S1. Además, al establecer B.false a S.siguiente, aseguramos que el control ignore el código para S1, si B se evalúa como falsa.

Advierta que todas estas secuencias de código finalizan en una declaración de etiqueta, a la que podríamos llamar etiqueta de salida de la sentencia de control. Muchos lenguajes proporcionan una construcción de lenguaje que permite salir de los ciclos o bucles desde ubicaciones arbitrarias dentro del cuerpo del ciclo.

Una característica de la generación de código para sentencias de control que puede provocar problemas durante la generación de código objetivo es el hecho de que, en algunos casos, deben generarse saltos a una etiqueta antes de que ésta se haya definido. Durante la generación de código intermedio, esto presenta pocos problemas, puesto que una rutina de generación de código simplemente puede llamar al procedimiento de generación de etiqueta cuando se necesita una para generar un salto hacia delante y grabar el nombre de la etiqueta (de manera local o en una pila) hasta que se conozca la ubicación de la etiqueta.

Durante la generación del código objetivo, las etiquetas simplemente se pueden pasar a un ensamblador si se está generando código ensamblador, pero si se va a generar código ejecutable real, estas etiquetas deben transformarse en ubicaciones de código absolutas o relativas.

La generación de código para expresiones lógicas
Las expresiones booleanas tienen dos propósitos principales. Se usan para calcular valores lógicos, pero más a menudo se usan como expresiones condicionales en declaraciones que alteran el flujo de control, como declaraciones if-then-else, while-do.

Las expresiones booleanas se componen de operadores booleanos (and, or y not) aplicados a elementos que son variables booleanas o expresiones relacionales. Las expresiones relacionales son de la forma E1 relop E2, donde E1 y E2 son expresiones aritméticas.

Aquí consideramos expresiones booleanas generadas por la siguiente gramática:

```
E → E or E | E and E | not A | (E)
     | id oprel id | true | false
```

Métodos de traducción de expresiones booleanas:
Existen dos métodos principales para representar el valor de una expresión booleana:

- Codificar numéricamente true y false y evaluar una expresión booleana de forma análoga a una expresión aritmética. A menudo, 1 se usa para denotar verdadero y 0 para denotar falso.

- Implementar expresiones booleanas por flujo de control, es decir, representar el valor de una expresión booleana por una posición alcanzada en un programa. Este método es particularmente conveniente para implementar las expresiones booleanas en sentencias de flujo de control, como las sentencias if-then y while-do.

Representación numérica
Aquí, 1 denota verdadero y 0 denota falso. Las expresiones se evaluarán completamente de izquierda a derecha, de manera similar a las expresiones aritméticas.

Por ejemplo , la traducción de **`a or b y not c`** es la secuencia de tres direcciones

```
t1 = not c
t2 = b and t1
t3 = a or t2
```

Una expresión relacional como **`a < b`** es equivalente al enunciado condicional

```
if a < b goto label1
t1 = 0
goto lable2
lable1
t1 = 1
lablel2
```

La expresion **x > y and z < 6** se traduce:

```
if x > y goto L1
t1 = 0
goto L2
label L1
t1 = 1
label L2
if z < 6 goto L3
t2 = 0
goto L4
label L3
t2 = 1
label L4
t3 = t1 and t2
```

Código de cortocircuito:
También podemos traducir una expresión booleana en un código de tres direcciones sin generar código para ninguno de los operadores booleanos y sin que el código evalúe necesariamente toda la expresión.

Implementar las expresiones booleanas por control de flujo: permite optimizar la evaluación de expresiones booleanas, calculando solamente lo necesario para determinar su valor. Esto depende de la semántica del lenguaje.

Regla gramatical	Reglas semánticas
B → B1 or B2	B1.true = B.true B1.false = newLabel() B2.true = B.true B2.false = B.false B.code = B1.code || label(B1.false) || B2.code
B → B1 and B2	B1.true = newLabel() B1.false = B.false B2.true = B.true B2.false = B.false B.code = B1.code || label(B1.true) || B2.code
B → not B1	B1.true = B.false B1.false = B.true B.code = B1.code
B → E1 oprel E2	B.code = E1.code || E2.code || 'if' || E1.name || oprel || E2.name || 'goto' || label(B.true) || 'goto' || label(B.false)
B → true	B.code = 'goto' || label(B.true)
B → false	B.code = 'goto' || label(B.false)

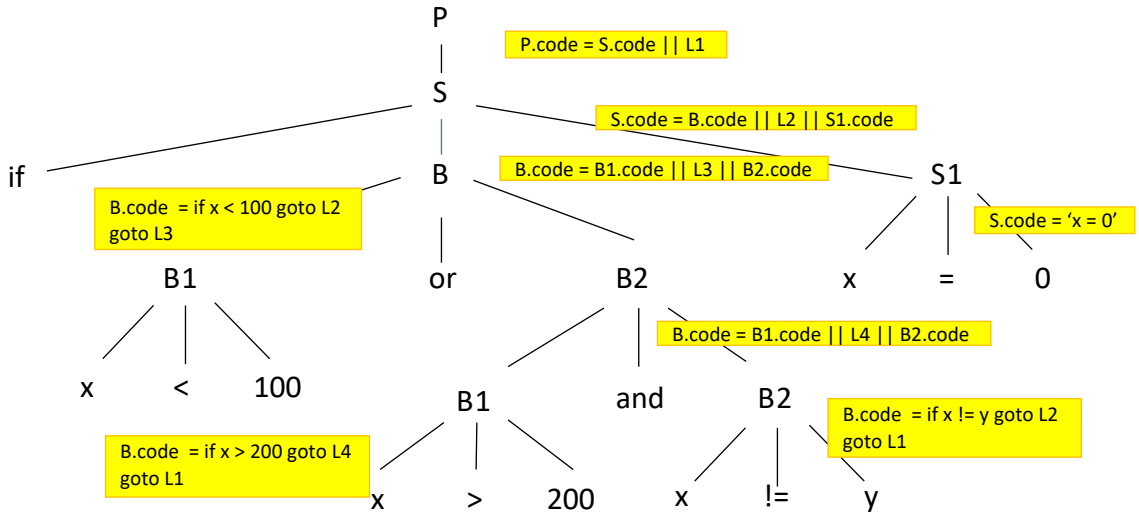
La producción, B → E1 rel E2, se traduce directamente en una instrucción de tres direcciones de comparación, con saltos hacia los lugares apropiados. Por ejemplo, B de la forma a < b se traduce en:

```
if a < b goto B.true
goto B.false
```

Para la producción de la forma B1 || B2. Si B1 es verdadera, entonces sabemos de inmediato que B en sí es verdadera, por lo que B1.true es igual que B.true. Si B1 es falsa, entonces hay que evaluar B2, para hacer que B1.false sea la etiqueta de la primera instrucción en el código para B2. Las salidas verdadera y falsa de B2 son iguales que las salidas verdadera y falsa de B, respectivamente.

Considere de nuevo la siguiente instrucción:

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```



Si utilizamos las definiciones dirigidas por la sintaxis obtendremos el código:

```
if x < 100 goto L2
goto L3
label L3
if x > 200 goto L4
goto L1
lable L4
if x != y goto L2
goto L1
lable L2
x = 0
label L1
```

Si las operaciones lógicas son de corto circuito, como en C, es necesario un uso adicional de los saltos. Por ejemplo, si a es una expresión booleana que se calcula como falsa, entonces la expresión booleana a and b se puede determirtar inmediatamente como falsa sin evaluar b. De manera similar, si se conoce que a es verdadera, entonces a or b se puede determinar como verdadera sin evaluar b. Las operaciones de cortocircuito son muy útiles para el codificador, ya que la evaluación de la segunda expresión ocasionaría un error si un operador no fuera de corto circuito.

Optimización de código intermedio

La optimización de código intermedio se puede realizar:

- a nivel local: sólo utilizan la información de un bloque básico para realizar la optimización.
- a nivel global: que usan información de varios bloques básicos.

El término optimización de código es inadecuado ya que no se garantiza el obtener, en el sentido matemático, el mejor código posible atendiendo a maximizar o minimizar una función objetivo (tiempo de ejecución y espacio). El término de mejora de código sería más apropiado que el de optimización.

Nos concentraremos básicamente en la optimización de código de tres direcciones, puesto que son siempre transportables a cualquier etapa final, son optimizaciones independientes de la máquina. Las optimizaciones a nivel de la máquina, como la asignación de registros y la utilización de instrucciones específicas de la máquina, se salen del contexto de esta materia.

La mayoría de los programas emplean el 90% del tiempo de ejecución en el 10% de su código. Lo más adecuado es identificar las partes del programa que se ejecutan más frecuentemente y tratar de que se ejecuten lo más eficientemente posible. En la práctica, son los lazos internos los mejores candidatos para realizar las transformaciones.

Además de la optimización a nivel de código intermedio, se puede reducir el tiempo de ejecución de un programa actuando a otros niveles: a nivel de código fuente y a nivel de código objeto.

La fase de optimización de código debe seguir las tres normas:

- El código de salida no debe, de ninguna manera, cambiar el sentido del programa.

- Optimización debe aumentar la velocidad del programa y si es posible, el programa debe exigir menos cantidad de recursos.

- Optimización debe ser rápido y no debe retrasar el proceso de compilación general.

En terminos generales la optimización puede clasificarse en dos grandes categorías: ***independiente de la máquina***, en esta optimización, el compilador toma en el código intermedio y transforma una parte del código que no implique un registros de la CPU y/o ubicaciones de memoria absoluta. Mientras que si existe ***dependencia de la máquina*** el código se transforman de acuerdo a la arquitectura del equipo de destino.

Bloques Básicos

Por lo general, los códigos fuente tienen una serie de instrucciones que se ejecutan siempre en orden y están consideradas como los bloques básicos del código. Estos dos bloques básicos no tienen instrucciones de salto entre ellos, es decir, cuando la primera se ejecuta la instrucción, todas las instrucciones en el mismo bloque básico será ejecutado en su secuencia de aparición sin perder el control de flujo del programa.

Un programa puede tener diversas construcciones como bloques básicos, `if-else`, `switch-case`, las sentencias condicionales, bucles, como `while`, `for` y `do-while`.

Identificación del bloque básico

Podemos utilizar el siguiente algoritmo para encontrar los bloques básicos en un programa:

- Declaraciones del inicio, Búsqueda de todos los bloques básicos desde donde se inicia un bloque básico:

    - Primera declaración de un programa.
    - Las declaraciones que son objetivo de cualquier rama (condicional o incondicional).
    - Las declaraciones que siguen cualquier rama.

- Las declaraciones del inicio y las declaraciones siguientes forman un bloque básico.

- Un bloque básico no incluye cualquier inicio declaración de cualquier otro bloque básico.

Bloques básicos son conceptos importantes de generación de código y optimización.

w= 0;
x = x + y;
y = 0;
if (x > y)
{
y = x;
x++;
}
else
{
y = z;
z++;
}
w = x + z;
	w= 0;
	x = x + y;
	y = 0;
	if (x > y)
	
	y = x;
	x++;
	
	y = z;
	z++;
	
	w = x + z;

Bloques básicos desempeñan un papel importante para identificar las variables, que se están utilizando más de una vez en un único bloque básico. Si cualquier variable se utiliza más de una vez, el registro memoria asignada a la variable no es necesario vaciar el bloque a menos que termine la ejecución.

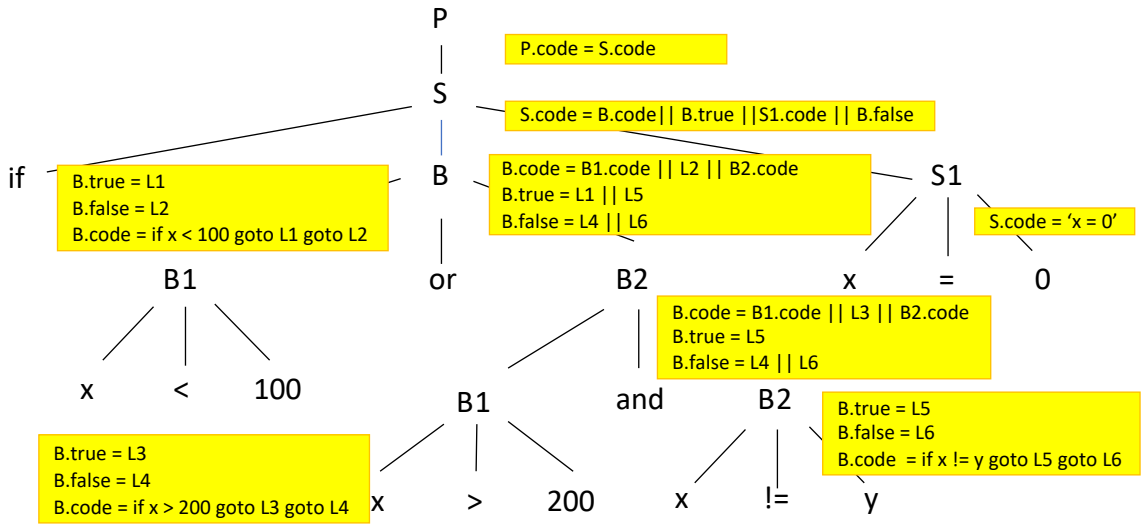
Implementación con atributos sintetizados

Regla gramatical	**Reglas semánticas**
P → S	P.code = S.code
S → N = E	N.name = E.name S.code = E.code || id || '=' || E.name ++
S → if (B) S1	S.code = B.code|| B.true ||S1.code || B.false
S → if(B) S1 else S2	end = newLabel() S.code = B.code || B.true || S1.code ||'goto' || end ++ B.false || S2.code || end ++
S → while (B) S1	begin = newlabel() S.code = begin ++ B.code || B.true || S1.code || 'goto' || begin || B.false
S → S1 S2	S.code = S1.code||S2.code

Regla gramatical	**Reglas semánticas**
B → B1 or B2	B.code = B1.code ++ B1.false ++ B2.code B.true = B1.true || B2.true B.false = B2.false
B → B1 and B2	B.code = B1.code ++ B1.true ++ B2.code

	B.true = B2.true B.false = B1.false || B2.false
B → not B1	B.code = B1.code B.true = B1.false B.false = B1.true
B → E1 oprel E2	B.true = **newtemp**() B.false = **newtemp**() B.code = E1.code || E2.code || 'if' || E1.name || oprel || E2.name || 'goto' || B.true ++ 'goto' || B.false ++
factor → true	B.true = **newtemp**() B.code = 'goto' || B.true ++
factor → false	B.false = **newtemp**() B.code = 'goto' || B.false ++

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```



Si utilizamos las definiciones dirigidas por la sintaxis obtendremos el código:

```
if x < 100 goto L1
goto L2
label L2
if x > 200 goto L3
goto L4
lable L3
if x != y goto L5
goto L6
lable L1
label L5
x = 0
label L4
label L6
```