Universidad
Europea de Madrid

LAUREATE INTERNATIONAL UNIVERSITIES

GENERACIÓN DE CÓDIGO INTERMEDIO

ÁRBOLES DE SINTAXIS ABSTRACTA (ASA)

Índice

Presentación

El objetivo de este tema es comprender la necesidad de la generación de código intermedio, identificar sus posibles modalidades y entender el funcionamiento del Árbol de Sintaxis Abstracta (en adelante, ASA), así como su implementación.

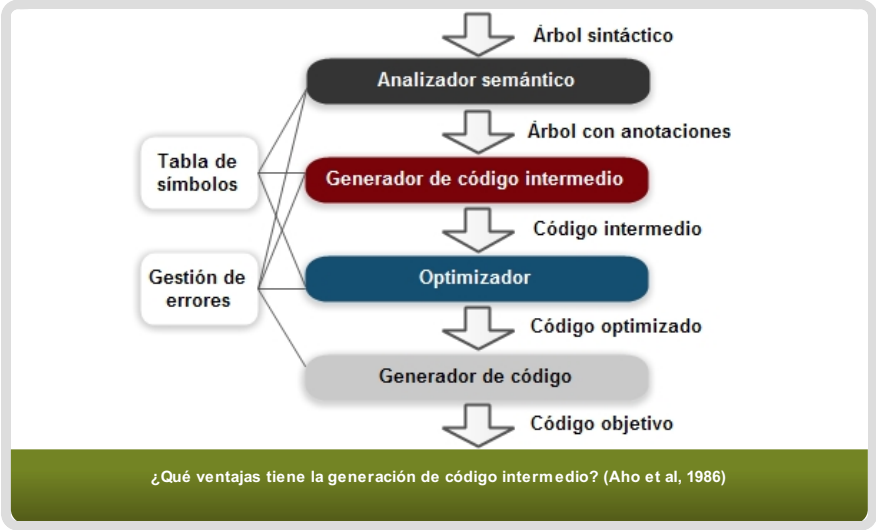
Para todo ello se alcanzarán los siguientes objetivos:

- Comprender la necesidad de generar código intermedio, así como sus ventajas e inconvenientes.
- Conocer qué tipos de estructuras podemos utilizar.
- Entender qué es un ASA y como se construye.
- Entender cómo se construye el ASA para operaciones aritméticas mediante un ejemplo.
- Conocer qué tipos de nodos puede tener un ASA y su complejidad.
- Comprender mediante un ejemplo cuál sería la secuencia de nodos para crear un árbol.
- Conocer otras implementaciones.

Introducción

Una vez se han realizado en el compilador las tres fases de análisis (léxico, sintáctico y semántico) empiezan las fases de sintaxis, que se concretan en la generación de código para una máquina objetivo.

Véase que a partir de realizar el análisis semántico, nos podríamos saltar todos los pasos de la fase de sintaxis, generando código fuente directamente.



¿Qué ventajas tiene la generación de código intermedio? (Aho et al, 1986)

- Se facilita la redestinación, ya que solo habría que cambiar la parte que va asociada a la máquina objetivo.
- Se puede aplicar a la representación intermedia, un optimizador de código independiente de la máquina.

En este tema se definen cuáles son los métodos utilizados y por tanto, las estructuras de datos necesarias para llevar a cabo este proceso de generación de código intermedio.

Como nos indica Louden (2004), 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.

Generación de un código para máquina objetivo

Para llegar a generar el código objetivo, que es el resultado final de todo el proceso de compilación, deberemos haber optimizado ese código a generar y esto se ha realizado a partir de un código abstracto que sirve para cualquier máquina y que se denomina **código intermedio.**

Tipos de estructuras

La estructura de datos que representa el programa fuente durante la traducción se denomina **representación intermedia** (IR, de *Intermediate Representation*) y a la representación intermedia que se parece al código objetivo se le denomina **código intermedio** (Louden, 2004).

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 operando 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 operando fuente y una dirección para el resultado. (a = b + c).

Las tres estructuras que vamos a ver para generar código intermedio son:

1. Árbol de Sintaxis Abstracta (ASA): tiene sentido utilizarlos cuando no se va a realizar una optimización de código y directamente se va a generar código. Tiene una versión mejorada formada por los grafos dirigidos acíclicos (GDA)
2. Notación postfija: no necesita paréntesis, porque la posición y el número de argumentos de los operadores permiten solo una descodificación por ejemplo: (5-3)+8. Esto en notación postfija es: 53-8+.
3. Código de tres direcciones: este código está pensado para las operaciones aritméticas y tiene la siguiente forma: x = y op z.

En este tema veremos el ASA y más adelante el código de tres direcciones. En cualquier caso y se use la notación que se use, para realizar la traducción de código intermedio a código objeto hay que volver a recorrer el árbol para generar el código final.

Código intermedio

Hay autores que dividen los tipos de representaciones intermedias (IR) en dos tipos: IR gráficas ASA y grafos dirigido acíclico GDA e IR Lineales (código de tres direcciones y notación postfija).

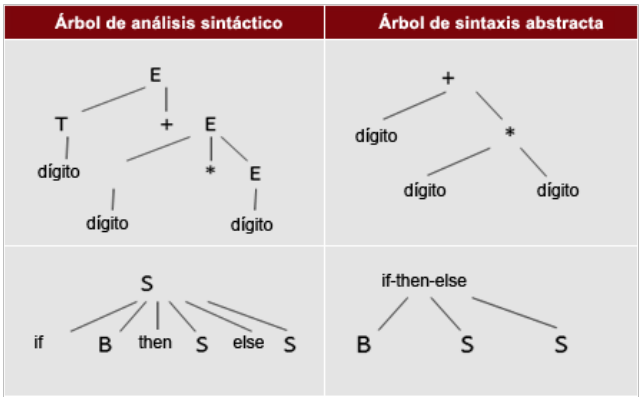
Notación postfija

A la notación postfija también se la denomina notación polaca inversa (RPN).

Árbol de Sintaxis Abstracta (ASA)

Como nos indica Aho el al (1986), 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.

Veamos los siguientes ejemplos:



Como podemos ver, en un ASA nos interesan las operaciones y los operadores y es por eso que eliminamos los símbolos superfluos (terminales de la gramática), generándose un árbol condensado.

En detalle ¿Cómo se construye el ASA?

En detalle

¿Cómo se construye el ASA?

- Se genera un nodo para cada operador y cada operando.
- Los hijos de un nodo operador son las raíces de los nodos que representan las subexpresiones que constituyen los operandos de dicho operador.

Para llevarlo a la práctica necesitamos unas funciones auxiliares:

- hazNodo (operador, izquierda, derecha).
- hazHoja (id, entrada).
- hazHoja (num, val).

Árbol de Sintaxis Abstracta (ASA) II

Vamos a ver con un ejemplo práctico la implementación usando la notación de definición dirigida por la sintaxis para la gramática de operadores aritméticos. Se utiliza como atributo sintetizado un puntero que apunta a las funciones **hazNodo** y **hazHoja** que acabamos de definir.

Partimos de la siguiente gramática, donde op puede representar a cualquier operador aritmético:

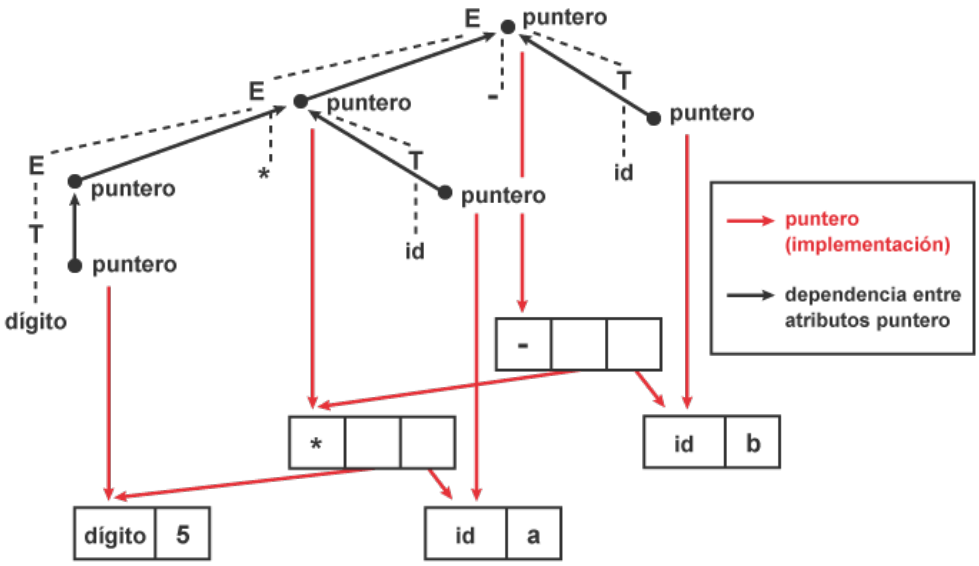
- $E_1 \rightarrow E_2$ op T
- E → T
- T → id
- T → dígito

Generándose las siguientes reglas semánticas:

Producción	Regla semántica
$E_1 \rightarrow E_2$ op T	E_1 puntero = hazNodo (op, E_2.puntero, T.puntero)
E → T	E. puntero = T. puntero
T → id	T.puntero = hazHoja (id, id.nombre)
T → dígito	T.puntero = hazHoja (dígito, dígito.val)

ASA. Ejemplo

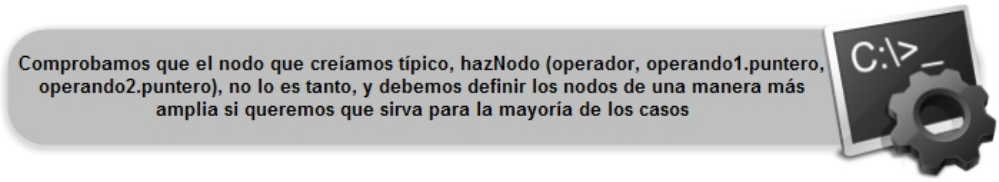
Supongamos la siguiente entrada: 5 * a - b, el ASA correspondiente sería:



Tipos de nodos de un ASA

En el ejemplo que hemos visto el tipo de nodo utilizado es para una operación aritmética, donde lo que se utiliza es una expresión binaria (dos operadores). Se debe tener en cuenta que hay una gran tipología de expresiones a la hora de diseñar el ASA, no solo las binarias.

En detalle Tipos de expresiones



Una estructura genérica en C para un nodo podría ser la siguiente:

```
1.  struct nodo {
2.   int TipoNodo;
3.   struct nodo *campo1;
4.   struct nodo *campo2;
5.   ......
6.  }
```

Donde el TipoNodo contiene un entero para identificar el tipo de nodo, mientras que los otros nodos apuntan a otros nodos del árbol.

En detalle

Tipos de expresiones

En la mayoría de los lenguajes encontramos las siguientes (en este caso utilizamos C como ejemplo):

- Declaraciones: int a.
- Constantes: #define PI 3,141592.
- Asignaciones: a := b.
- Bloques.
- Operaciones unarias: a := op b (donde op, por ejemplo, puede ser +, -, ! -negación-).
- Operaciones binarías: a := b op c (dentro de las operaciones binarias, además de las aritméticas, entran las operaciones lógicas y las relacionales).
- Bifurcaciones: goto A1.
- Bifurcaciones condicionales: if (a == 5) goto A1 (que surgen cuando se desglosan los bucles).
- Bucles. Por ejemplo, while y for.
- while (expr_condicional) {Sentencias }.
- for (a:=0; a < b, a++) {Sentencias }.
- Control de flujo: if condición then S1, if condición then S1 else S2, switch.
- Operaciones con arrays: asignación y carga (a := b[i] o b[i] = a).
- Operaciones con punteros: a := , a := *b, *a:=b.
- Llamadas a funciones: x:= f(arg1, arg2,..argN).

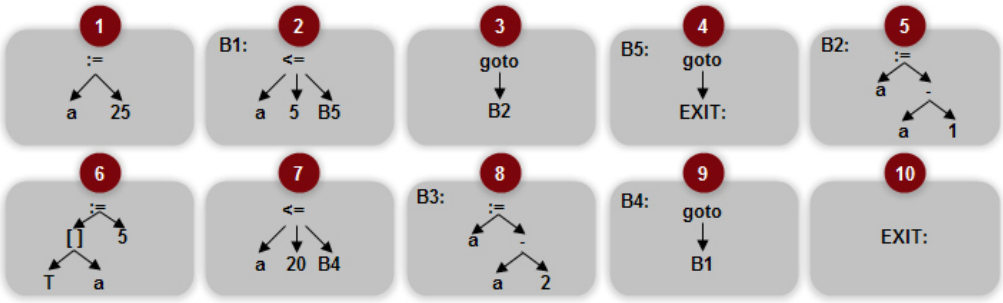
Secuencia de nodos en un ASA

Vamos a suponer el siguiente ejemplo:

```
1.  a := 25;
2.  while a > 5 {
3.  a:= a -1;
4.  T[a] := 5;
5.   if (a > 20) {
6.        a:= a -2;
7.  }
8.  }
```

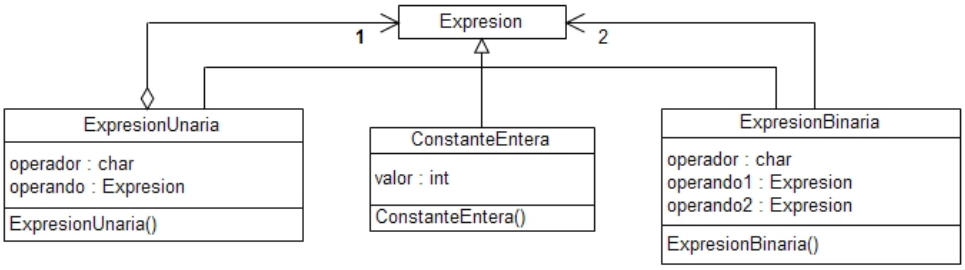
Vemos que el bucle while, se descompone en varios bloques: B1, goto B2 (si se cumple la condición), B5 (si no se cumple la condición). Con la sentencia if, también pasa lo mismo, se descompone en un salto si a<= 20, a B4 en caso de no cumplir la condición y a B3 en caso de cumplir la condición.

Por tanto el AST debe poder unir todos los nodos, para construir el árbol y probablemente deba apoyarse en alguna estructura, tipo vector para ordenar los nodos dentro del struct que forma el árbol, además de apoyarse en la tabla de símbolos para controlar los ámbitos de las variables.



Otras implementaciones de un ASA

Ortin Soler et al, (2004), utilizan el siguiente diagrama de clases, donde todos los nodos son instancias derivadas de la clase abstracta expresión.



Su implementación en Bison sería la siguiente:

```
expresión: expresión '+' termino {$$ = new ExpresionBinaria('+', $1, $3); }
         | expresión '-' termino {$$ = new ExpresionBinaria('-', $1, $3); }
         | termino {$$ = $1; }
         ;
termino: termino '*' factor {$$ = new ExpresionBinaria('*', $1, $3); }
         | termino '/' factor {$$ = new ExpresionBinaria('/', $1, $3); }
         | termino {$$ = $1; }
         ;
factor: '-' factor {$$ = new ExpresionBinaria('-', $2); }
         | '(' expresión ')' { $$ = $2; }
         | CTE_ENTERA  {$$ = new ConstanteEntera($1); }
         ;
```

Clase abstracta expresión

Veremos que todas las expresiones binarias, serán instancias de la clase ExpresionBinaria, y mediante un atributo tipo char se define el operador.

Lo mismo se puede hacer con las expresiones unarias y las declaraciones de constantes.

Implementación

En el documento citado de Ortín Soler et al, se describe como mediante el patrón de diseño Visitor se puede implementar.

Resumen

En este tema hemos comprendido porqué es necesario realizar la generación de código intermedio y qué ventajas tiene, como el facilitar la redestinación o poder aplicar la optimización de forma independiente de la máquina.

También se ha visto que hay distintos tipos de máquinas para las que generar código, en función de las direcciones que podemos utilizar a la vez en una operación (0, 2 o 3 direcciones).

Aunque hay varias modalidades de generación de código intermedio, las más ampliamente usadas son los ASA (cuando no se va a realizar la fase de optimización) o su variante mejorada los grafos dirigidos acíclicos (GDA) y el código de tres direcciones.

Se ha podido ver con varios ejemplos, primero con operaciones aritméticas y posteriormente con uno más completo, como se construye el ASA y que tipos de nodos podemos encontrarnos en un lenguaje genérico. Asimismo se han comentado otras implementaciones que pueden resultar útiles a la hora de diseñar y construir un compilador.