Clase 1

Introducción y evaluación de lenguajes

Los lenguajes de programación son necesarios para comunicar las máquinas con las personas. *El valor de un lenguaje se debe juzgar según cómo afecta la producción de software y la facilidad para integrarse a otras herramientas.*

Objetivos de diseño de un lenguaje

- Simple y legible: un lenguaje debe permitir hacer programas fáciles de escribir, leer, aprender y enseñar. Algunas cosas que atentan contra esto es tener muchas instrucciones, mismo concepto semántico con distintas sintaxis (algo que se escribe igual pero que hace cosas diferentes), abuso de operadores sobrecargados (para un mismo operador obtenemos diferentes operaciones).

  ```
  #### ejemplo de mismo concepto semántico con distinta sintáxis
  a(x) --> ingresa a un índice del arreglo
  a(x) --> llama a una función pasándole un parámetro
  ```

- Claridad en los bindings (ES: enlace/unión):facilidad de entender cómo se resuelven las referencias a elementos del programa, como variables y funciones. Un lenguaje con bindings claros es más fácil de entender y depurar, mientras que un lenguaje con bindings poco claros puede llevar a errores y comportamientos inesperados.
- Confiabilidad: relacionado con la seguridad.
  - Chequeo de tipos: cuanto antes se encuentren errores menos costoso es hacer arreglos requeridos.
  - Manejo de excepciones: la habilidad para interceptar errores en tiempo de ejecución, tomar medidas correctivas y continuar.
- Soporte: debería ser accesible para cualquiera, implementarlo en distintas plataformas y que exista diferentes medios para poder familiarizarse con el lenguaje (tutoriales, cursos, etc.).
- Abstracción: capacidad de definir y usar estructuras u operaciones complicadas fácilmente.
- Ortogonalidad: característica que permite combinar diferentes aspectos de manera flexible y sin restricciones arbitrarias. Permite usarlas de manera independiente y en combinación entre sí de manera sencilla, lo que puede mejorar la productividad y la facilidad de mantenimiento.
- Eficiencia: relacionado al tiempo, espacio, esfuerzo humano y que sea optimizable.

Sintaxis

Reglas que definen cómo componer letras, dígitos y otros caracteres para formar los programas. Cómo se forma una palabra y cómo esa palabra se combina para formar una sentencia válida.

Sirven para que el programador se comunique con el procesador. Deberá contemplar características como legibilidad, verificabilidad, traducción y evitar ambigüedad.

En resumen, describe la forma legal de escribir cualquier programa.

Elementos de la sintaxis

- Alfabeto o conjunto de caracteres con los que trabaja el lenguaje.
- Identificadores: cada lenguaje define su formar de definir una palabra clave.
- Operadores.
- Palabra clave y palabra reservada.
    - Palabras claves: palabras que tienen un significado dentro de un contexto. Se usan para lo que fueron creadas y además pueden servir como variables.
    - Palabras reservadas: palabras claves que NO pueden ser usadas para otro uso.
- Comentarios y uso de blancos: hacen los programas más legibles.

Estructura sintáctica

- Words: conjunto de caracteres y palabras necesarias para construir expresiones, sentencias y programas.
- Expresiones: funciones que a partir de un conjunto de datos devuelven un resultado.
- Sentencias: se componen de expresiones y words. El conjunto de sentencias arman un programa.

Reglas léxicas y sintácticas

- Léxicas: conjunto de reglas para formar las word.
- Sintácticas: reglas que evalúan si la palabra está bien combinada para formar sentencias válidas.

Tipos de sintaxis

- Abstracta: se refiere a las reglas y estructuras de programación que permiten expresar soluciones para problemas en términos generales, sin atarse a detalles específicos de implementación. Por ejemplo, la sintaxis abstracta se utiliza en la programación orientada a objetos para definir clases, que son estructuras generales que se pueden instanciar para crear objetos específicos con características y comportamientos particulares.
- Concreta: se refiere a la estructura física y gramatical del lenguaje de programación, incluyendo reglas sobre cómo se deben escribir las palabras
- Pragmática: se refiere a cómo se utiliza el lenguaje de programación en un contexto específico y cómo se relaciona con otros componentes del sistema

Definición de sintaxis

Es necesario una definición formal para definir todas las combinaciones de la sintaxis. Hay diferentes formas de definirlas como el lenguaje natural o BNF.

- BNF(Backus Naun Form): notación para describir la estructura sintáctica de los lenguajes. Define reglas de producción que especifican cómo se deben combinar los símbolos para formar las expresiones válidas. Es un metalenguaje porque define sintaxis de otros lenguajes. Es una gramática libre de contexto, lo que se refiere que no importa que no tenga sentido si la estructura está bien definida.

  Una especificación de BNF se escribe de esta forma:

```
<simbolo> ::= <expresión con símbolos>
```

donde <símbolo> es un no terminal, y la expresión consiste en secuencias de símbolos o secuencias separadas por la barra vertical | indicando una opción, el conjunto es una posible sustitución para el símbolo a la izquierda. Los símbolos que nunca aparecen en un lado izquierdo son terminales.

Un ejemplo es un BNF para una dirección postal de EE.UU.

```
<dirección postal> ::= <nombre><dirección><apartado postal>

<nombre>::=<personal><apellido>[<trato>]<EOL> | <personal><nombre>

<personal>::=<primer nombre> | <inicial> "."

<direccion> ::= [<dpto>] <numero de la casa> <nombre de la calle> <EOL>

<apartado postal> ::= <ciudad> "," <código estado> <código postal> <EOL>
```

Esto se traduce como:

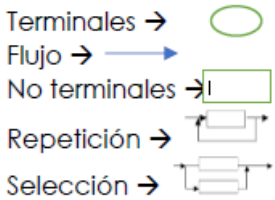
- Una dirección postal consiste en un nombre, seguido por una dirección, seguida por un apartado postal.
- Una parte <personal> consiste en un nombre o una inicial seguido por un punto.
- Un nombre consiste de: una parte personal seguida por un apellido seguido opcionalmente por una jerarquía o el trato que se le da a la persona y un salto de línea (EOL/ end of line) o bien una parte personal seguida por un nombre.
- Una dirección consiste de una especificación opcional del departamento, seguido por un número de casa, seguido por el nombre de la calle, seguido por un EOF.
- Un apartado postal consiste de una ciudad, seguida por una coma, seguido por un código de estado, seguido por un código postal y este seguido por un EOF.

  - < > nombra un símbolo no terminal.
  - ::= significa “se define como”.
  - | para indicar opciones.

- EBNF: es una versión extendida de BNF la cual tiene más meta símbolos y permite hacer una notación más sencilla. Los símbolos agregados son:
  - [ ] indica que un elemento puede ser optativo.
  - (|) elementos optativos.
  - { } repeticiones, que puede acompañarse con * o +.
    - * que se repite 0 o más veces.
    - + que se repite 1 o más veces.

- Describir una gramática de forma gráfica (CONWAY): es un gráfico sintáctico en la que cada diagrama tiene una entrada, una salida y

el camino determina el análisis. Cada diagrama representa una regla o producción.



- Gramática: conjunto de reglas que define un conjunto infinito de posibles sentencias válidas en el lenguaje. Es una 4-tupla, G= (N T S P)
  - N: conjunto de símbolos no terminales, serán las cosas que definiré en la producción.
  - T: conjunto de símbolos terminales, que representan las palabras clave, identificadores, operadores, signos de puntuación, etc., del lenguaje.
  - S: símbolo restringido de la gramática que pertenece a N, que representa la estructura sintáctica básica del lenguaje. Con lo que inicia la producción.
  - P: conjunto de producciones, es un conjunto finito de reglas de producción, que definen cómo se construyen las cadenas de símbolos terminales y no terminales del lenguaje. Las reglas de producción especifican cómo se combinan los símbolos no terminales y terminales para formar cadenas sintácticamente válidas.

```
G = (N, T, S, P)

Palabras del lenguaje:
Juan, Ana, tiene, compra, canta, un, una, canción, manta, perro.


T = {Juana, Ana, tiene, compra, canta, un, una, canción, manta, perro}
N = {<oración>, <sujeto>, <predicado>, <sustantivoPropio>, <articuloIndeterminado>,
<sustantivoComún>, ...}
S = {<oración>}
P = {
      <oración>::= <sujeto><predicado>
      <sujeto>::= <sustantivoPropio> | <articuloIndeterminado><sustantivoComún>
      <sustantivoPropio> ::= Juan | Ana
      <articuloIndeterminado> ::= un | una
      <sustantivoComún> ::= manta | perro | canción
      <predicado> ::= <verbo><objetoDirecto>
      <verbo>::= compra|tiene|canta
      <objetoDirecto>::= <articuloIndeterminado><sustantivoComún>
}
```

Símbolos terminales y no terminales

- Símbolos terminales: elementos del lenguaje que no pueden descomponer más. Representan las palabras reservadas, identificadores, operadores etc. Se escriben en minúsculas y se incluyen en el alfabeto del lenguaje.

  Ejemplos en Java

  - Palabras reservadas: public, class, static, void, if, else, for, while, etc.
  - Identificadores: nombres de variables, métodos, clases, etc. como "numero", "miVariable", "miMetodo", "MiClase", etc.

    - Literales: valores constantes como números (por ejemplo, 42, 3.14), caracteres (por ejemplo, 'a', 'b', '$'), cadenas de texto (por ejemplo, "Hola mundo").

- Símbolos no terminales: aquellos que pueden ser descompuestos en otros símbolos, ya sean terminales o no terminales. Representan las expresiones, las declaraciones, las instrucciones, etc. Estos símbolos se escriben en mayúsculas.

  Ejemplos en Java

    - Expresiones: combinaciones de valores, operadores y/o llamadas a métodos. Ejemplo: "5 + 3", "miVariable * 2", "miMetodo(5, 3)", etc.
    - Declaraciones: instrucciones que definen nuevas variables o métodos. Ejemplo: "int miVariable = 42;", "public void miMetodo() { /* Código */ }", etc.
    - Estructuras de control de flujo: instrucciones que permiten cambiar el orden de ejecución de las instrucciones. Ejemplo: "if (condicion) { /* Código */ } else { / Código */ }", "for (int i = 0; i < 10; i++) { / Código */ }", etc.

Árboles sintácticos

Los árboles sintácticos son una representación visual de una expresión. Cada nodo interno del árbol representa una regla de producción de la gramática, y sus hijos representan los símbolos no terminales y terminales que participan en esa regla. De esta manera, los árboles sintácticos proporcionan una forma clara y visual de entender cómo se estructura una expresión o una frase de acuerdo con las reglas de una gramática determinada.

Es necesario un método de análisis que permita determinar si un string dado es válido o no en un lenguaje, este método es conocido como **Parsing,** el cual es el proceso de analizar una cadena de entrada en un lenguaje formal para determinar si cumple con las reglas de una gramática sintáctica dada. En el contexto de los árboles gramaticales, el parsing implica la construcción del árbol sintáctico correspondiente a una cadena de entrada dada.

Hay dos formas de armar los árboles sintácticos

- Método botton-up: de izquierda a derecha o derecha a izquierda. El método bottom-up es una técnica de análisis sintáctico para construir árboles gramaticales que comienza con los símbolos terminales de una cadena de entrada y los agrupa para formar los símbolos no terminales de la gramática. Este enfoque se llama "bottom-up" porque construye la estructura del árbol desde las hojas (símbolos terminales) hacia la raíz (símbolos no terminales).
- Método top-dow: de izquierda a derecha o derecha a izquierda. El método top-down es una técnica de análisis sintáctico que se basa en el concepto de que una oración o cadena de entrada se puede dividir en varias partes o componentes. La técnica de análisis top-down comienza desde la raíz de un árbol sintáctico y se divide en ramas o subárboles hasta que se llega a las hojas, que son los símbolos terminales de la gramática. En el método top-down, se comienza con el símbolo inicial de la gramática y se trata de expandir la cadena de entrada en términos de los símbolos no terminales de la gramática.

Producciones recursivas

Las producciones recursivas se refiere a que una producción puede expandirse en sí misma de forma.

Por ejemplo la siguiente producción: E -> E + E. Esta producción es recursiva, porque el símbolo no terminal "E" aparece tanto en el lado izquierdo como en el derecho de la producción y significa que una

expresión puede ser la suma de dos expresiones.

- Regla recursiva por izquierda: una regla recursiva por la izquierda es una regla en una gramática que tiene el mismo símbolo no terminal en el lado izquierdo de la regla y en el inicio de una o más producciones en el lado derecho de la regla. Por ejemplo, en la gramática siguiente: A -> A B | C
- Regla recursiva por derecha: una regla recursiva por la derecha es una regla en una gramática que tiene el mismo símbolo no terminal en el lado derecho de la regla y al final de una o más producciones en el lado derecho de la regla. Por ejemplo, en la gramática siguiente: A -> B A | C

Gramática ambigua

Una gramática es ambigua si hay al menos una oración que se puede interpretar de varias maneras. Es problemático en programación porque puede llevar a diferentes interpretaciones para el mismo comando, lo que puede generar errores o confusiones. Por ejemplo, A+B*C se puede resolver como A+(B*C) o como (A+B)*C. Si utilizamos una gramática ambigua para el lenguaje de programación, esta oración podría ser interpretada de diferentes maneras, lo que llevaría a diferentes resultados. Por lo tanto, es importante utilizar gramáticas que no sean ambiguas para evitar posibles problemas en la interpretación del lenguaje.

Subgramática

Una subgramática es una versión simplificada de una gramática, creada eliminando algunas reglas de producción y sus símbolos no terminales asociados. Se utiliza para hacer más fácil el entendimiento de la gramática original, pero hay que tener cuidado de no crear una gramática ambigua o de reducir demasiado el lenguaje generado por la gramática.

Gramáticas libres de contexto y sensibles al contexto

Las gramáticas son reglas que se utilizan para describir la estructura de un lenguaje formal. Las gramáticas libres de contexto son aquellas que no realiza un análisis de contexto. Las gramáticas sensibles al contexto analiza por ejemplo si un identificar está definido dos veces.

Clase 2

Semántica

La semántica describe el significado de los símbolos, palabras y frases de un lenguaje para luego poder darle significado a una construcción del lenguaje.

En la semántica se deben detectar otros errores semánticos que son difíciles de describir con BNF o EBNF. La semántica que analizaremos será la estática (antes de la ejecución) y dinámica (durante la ejecución).

Semántica estática

Relacionado con las formas válidas de la sintaxis. Realizado durante la compilación por eso es estática. BNF/EBNF no sirve en caso de errores de compatibilidad de tipos o de declaración de variables duplicadas por lo tanto se usará la gramática de atributos.

La **gramática de atributos** es una gramática sensible al contexto las cuales resuelven aspectos de la semántica estática y es usada por compiladores.

De esta forma, a las construcciones del lenguaje se les asocia información mediante atributos asociados a los símbolos de la gramática. Los valores de los atributos se obtienen mediante las llamadas *ecuaciones* asociadas a las producciones gramaticales.

Para evaluar reglas semánticas se escribe de forma tabular.

- Las reglas sintácticas (producciones) son similares a BNF.
- Las ecuaciones permiten detectar errores y obtener valores de atributos.
- Los atributos están directamente relacionados a los símbolos gramaticales.

Funcionamiento de la gramática de atributos

- Usa la tabla y machea si encuentra la producción/regla y del otro lado la ecuación que permite llegar a los atributos.
- Mira las reglas (similar a BNF/EBNF) y busca atributos para terminales y no terminales.
- Si encuentra el atributo debe llegar a obtener su valor.
- Lo obtiene generando atributos.

De la ejecución de las ecuaciones se debe tratar de representar todo lo que necesitemos que salte ante la ejecución y no sea detectado sintácticamente.

Semántica dinámica

Aplicado luego de chequear su semántica estática y que la sintaxis sea correcta aparecerá la semántica dinámica en el momento de ejecutar un programa.

No es fácil de escribirla, no hay herramientas estándar como en la sintaxis y es complejo describir la relación entre la entrada y salida del programa y describir cómo se ejecutará en cierta plataforma.

Dentro de la semántica dinámica, las soluciones más usadas son

- Formales y complejas

    - Semántica axiomática: considera al programa como una máquina de estados donde cada instrucción provoca con cambio de estado. Se parte de un axioma (verdad) que sirve para verificar estados y condiciones a probar.

      Se desarrolló para probar la corrección de los programas. La notación empleada es el cálculo de predicados.

      Un estado se describe con un predicado que describe los valores de las variables en ese estado. Existe un estado anterior y un estado posterior a la ejecución del constructor.

      Cada sentencia se precede y se continúa con una expresión lógica que describe las restricciones y relaciones entre los datos

    - Semántica dotacional: basado en la teoría de las funciones recursivas y modelos matemáticos. Se diferencia de la axiomática por la forma que describe los estados. La axiomática lo describe a través de los predicados y la denotación lo describe a través de funciones.

      Define una correspondencia entre los constructores sintácticos y sus significados y describe la dependencia funcional entre el resultado de la ejecución y sus datos iniciales.

- No formal
    - Semántica operacional: el significado de un programa se describe mediante otro lenguaje de bajo nivel implementado sobre una máquina abstracta. Los cambios que se producen en el estado de la máquina abstracta, cuando se ejecuta una sentencia del lenguaje de programación definen su significado. Es un método informal porque se basa en otro lenguaje de bajo nivel y puede llevar a errores. Es el más usado.

Sirven para comprobar la ejecución, la exactitud de un lenguaje y comparar funcionalidades de distintos programas.

Se pueden usar combinados, no sirven todos para todos los tipos de lenguajes de programación.

Procesamiento de un programa

Historia

Anteriormente se programaba con código de máquina pero era complejo y con muchos errores entonces se pensaron soluciones. Reemplazar repeticiones/patrones de bits por código, denominado *código mnemotécnico.* Así fue como surgió el lenguaje ensamblador que usaba estos códigos para programar.

Las computadoras ejecutan un lenguaje de bajo nivel llamado lenguaje de máquina (opera solamente con 0 y 1).

Cada procesador tendrá

- Lenguaje ensamblador.
- Programa ensamblador.
- Código de máquina.
- Código nemotécnico.

Debido a que cada procesador tiene su propio lenguaje de máquina se producirán algunos problemas, los cuales son

- Imposibilidad de intercambiar programas entre distintas máquinas o procesadores.

- Diferentes versiones para una misma CPU pueden tener juegos de instrucciones incompatibles.
- Los modelos evolucionados de una familia de CPU pueden incorporar instrucciones nuevas.

Por eso se buscó una solución, los *lenguajes de alto nivel*, permitiendo una abstracción.

Lenguajes de alto nivel

Para que las máquinas puedan procesar lenguajes de alto nivel existen programas traductores del lenguaje. Algunas alternativas son interpretación, compilación o la combinación de ambos.

La traducción de alto a bajo nivel no es decisión del programador, si no de quien crea el lenguaje.

Interpretación

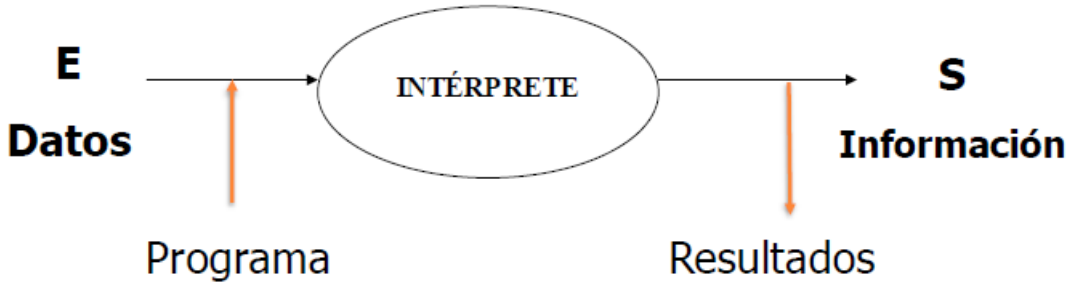
Hay un programa llamado intérprete que realiza la traducción de un lenguaje interpretado en el momento de ejecución. Un lenguaje interpretado es un tipo de lenguaje en el que el código fuente se traduce y ejecuta línea por línea en tiempo de ejecución.

El proceso que realiza cuando se ejecuta sobre cada una de las sentencias del programa es:

1. Leer
2. Analizar
3. Decodificar
4. Ejecutar

Solo pasa por ciertas instrucciones no por todas, según sea la ejecución.

El intérprete cuenta con una serie de herramientas para la traducción a lenguaje de máquina. Por cada posible acción hay un subprograma en lenguaje de máquina que ejecuta esa acción. La interpretación se realiza llamando a estos subprogramas en la secuencia adecuada hasta generar el resultado de la ejecución.



Un intérprete ejecuta repetidamente la siguiente secuencia de acciones:

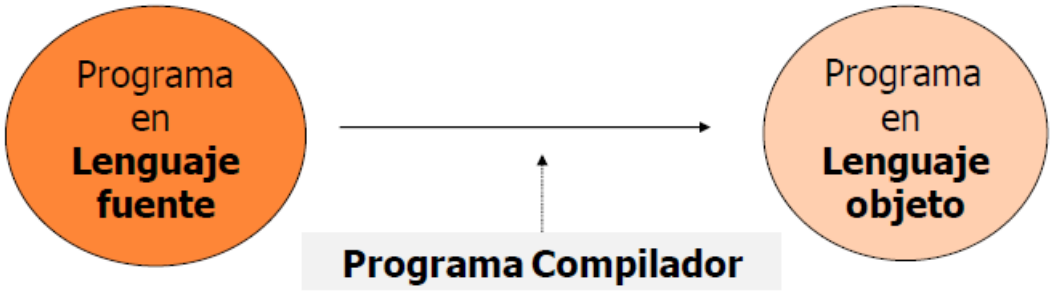
1. Obtiene la próxima sentencia.
2. Determina la acción a ejecutar.
3. Ejecuta la acción.

Compilación

Algunos lenguajes como C++, Fortran o Pascal son lenguajes compilados donde tenemos nuestro programa escrito y usamos un programa llamado compilador que realiza la traducción a un lenguaje de

máquina el cual es compilado antes de ejecutarte. Pasa por todas las instrucciones antes de la ejecución y el código que se genera se guarda y se puede reusar ya compilado.

La compilación implica varias etapas.



El compilador toma todo el programa escrito en un lenguaje de alto nivel que llamamos lenguaje fuente antes de su ejecución. Luego de la compilación va a generar un lenguaje objeto que es generalmente el ejecutable (escrito en un lenguaje de máquina) o un lenguaje de nivel intermedio (o lenguaje ensamblador).

Funcionamiento

Traduce todo el programa y puede generar un código ejecutable o código intermedio. La compilación puede ejecutarse en 1 o 2 etapas.

En ambos casos se cumplen varias sub-etapas, las principales son

1. Etapa de análisis del programa fuente: vinculada al código fuente, incluye la etapa de análisis léxico, análisis sintáctico y análisis semántico.
    - Análisis léxico: etapa que lleva más tiempo donde el código fuente es escaneado y se divide en unidades más pequeñas llamadas "tokens". Estos tokens son los componentes básicos como palabras clave, identificadores, operadores o constantes. El analizador léxico también elimina espacios en blanco, comentarios y cosas no significativas y analizará el tipo de cada token para ver si son válidos.

      El resultado de este paso será el descubrimiento de los *tokens*.
    - Análisis sintáctico: en esta etapa, se analiza la estructura del código fuente y se verifica si cumple con la sintaxis del lenguaje de programación usado. Usa la secuencia de tokens generada en la etapa anterior y la compara con las reglas gramaticales del lenguaje de programación. Si la estructura cumple con las reglas gramaticales, se genera un árbol de sintaxis que representa la estructura del programa en términos de su sintaxis.

      El árbol de sintaxis abstracta se usa en etapas posteriores del proceso de compilación, como la generación de código y la optimización, para producir el código ejecutable final. Si se encuentra un error sintáctico, se genera un mensaje de error que indica el problema y la ubicación en el código fuente donde se produjo el error.
    - Análisis semántico: se verifica que el código fuente cumpla con las reglas semánticas del lenguaje de programación. Usa la información de la tabla de símbolos generada en el análisis léxico y sintáctico para verificar que las operaciones y expresiones en el programa sean válidas y coherentes en términos de su significado. Incluye la verificación de tipos de datos, comprobación de la asignación y uso adecuado de variables y la verificación de la coherencia en las llamadas a funciones.

Si se encuentra un error semántico, se genera un mensaje de error que indica el problema y la ubicación en el código fuente donde se produjo el error. Si no se encuentran errores, el código fuente se considera válido y se pasa a la siguiente etapa del proceso de compilación, la generación de código.

2. Etapa de síntesis: se uso el código intermedio generado en la etapa anterior para producir el código objeto o ejecutable final del programa. Durante la síntesis, se realiza la optimización del código intermedio, la asignación de registros y se genera el código objeto o ejecutable. También puede incluir la resolución de referencias a variables y funciones, o la inclusión de bibliotecas externas. El compilador realiza una serie de transformaciones y optimizaciones en el código intermedio para generar un código objeto lo más eficiente posible.

   Una vez que se ha completado la etapa de síntesis, se genera el código objeto.

Entre las dos etapas se puede generar **código intermedio**. El código intermedio es un código generado después de la primera etapa (etapa de análisis del programa fuente). Sirve como una representación abstracta del código fuente que facilita la optimización del código y generación del código objeto.

Optimización

No se hace siempre y no lo hacen todos los compiladores y es optativo. Los optimizadores de código pueden ser herramientas independientes, o estar incluidas en los compiladores e invocarse por medio de opciones de compilación.

Hay diversas formas y cosas a optimizar, generalmente

- Elegir la velocidad de ejecución y tamaño del código ejecutable.
- Generar código para un microprocesador dentro de una familia de microprocesadores.
- Eliminar la comprobación de rangos o desbordamientos de pila.
- Evaluación para expresiones booleanas.
- Eliminación de funciones no utilizadas.

Compilador vs. intérprete

	Intérprete	Compilador
Cómo se ejecuta	Ejecuta el programa de entrada directamente. Dependerá de la acción del usuario o de la entrada de datos y de alguna decisión del programa. Siempre se debe tener el programa intérprete. El programa fuente será público.	Se usa el compilador antes de la ejecución, produce un programa equivalente en lenguaje objeto. El programa fuente no será público.
Orden de ejecución	Sigue el orden lógico de ejecución	Sigue el orden físico de las sentencias.
Tiempo consumido de ejecución	Por cada sentencia que pasa realiza el proceso de decodificación (lee, analiza y ejecuta) para determinar las operaciones y sus operandos y es repetitivo. Si la sentencia está en un proceso iterativo (for/ while), se realizará la tarea de decodificación tantas veces como sea requerido. La velocidad de proceso se puede ver afectada.	Pasa por todas las sentencias, no repite lazos, traduce todo una sola vez, genera código objeto ya compilado y la velocidad de compilar dependerá del tamaño del código.
Eficiencia	Más lento en ejecución. Se repite el proceso	Más rápido ejecutar desde el punto de vista del

posterior	cada vez que se ejecuta el programa. Para ser ejecutado en otra máquina se necesita tener si o si el intérprete instalado y el programa fuente será público.	hardware porque ya está en un lenguaje de más bajo nivel. Detectó más errores al pasar por todas las sentencias. Está listo para ser ejecutado. Ya compilado es más eficiente. Por ahí tarda más en compilar porque se verifica todo previamente y el programa fuente no es público.
Espacio ocupado	No pasa por todas las sentencias entonces ocupa menos espacio de memoria. Cada sentencia se deja en la forma original y las instrucciones interpretadas necesarias para ejecutarlas se almacenan en los subprogramas del interprete en memoria. Cosas como tablas de símbolos, variables y otros se generan cuando se usan en forma más dinámica.	Pasa por todas las sentencias. Una sentencia puede ocupar decenas o centenas de sentencias de máquina al pasar a código objeto. Las tablas de símbolos, variables, etc se generan siempre se usen o no. El compilador hace ocupar más espacio
Detección de errores	Las sentencias del código fuente pueden ser relacionadas directamente con la sentencia en ejecución entonces se puede ubicar el error. Es más fácil detectarlos por donde pasa la ejecución. Es más fácil de corregirlos.	Se pierde la referencia entre el código fuente y el código objeto y es casi imposible ubicar el error. Se deben usar otras técnicas como la semántica dinámica.

Traducción

La interpretación pura y compilación pura son dos extremos. En la práctica muchos lenguajes combinan ambas técnicas para sacar provecho a cada una.

Los compiladores y los intérpretes se diferencian en cómo reportan los errores de ejecución. También hay otras diferencias vistas en la comparación. Ciertos entornos de programación contienen las dos versiones: interpretación y compilación.

Combinación de técnicas de traducción

- Primero interpreto y después compilo.

  Se usa el intérprete en la etapa de desarrollo para facilitar el diagnóstico de errores. Con el programa validado se compila para generar un código objeto más eficiente.

- Primero compilo y luego interpreto.

  Se hace traducción a un código intermedio a bajo nivel que luego se interpretará. Sirve para generar código portable o sea código fácil de transferir a diferentes máquinas y con diferentes arquitecturas.

Clase 3

Entidad	Atributo
Variable	Nombre, tipo, área de memoria, etc.
Rutina	Nombre, parámetros formales, parámetros reales, etc.
Sentencia	Acción asociada.

El descriptor es un lugar donde se almacenan los atributos.

Binding

Los programas trabajan con entidades las cuales tienen atributos y estos atributos tienen que establecerse antes de poder usar la entidad. El binding será la asociación entre la entidad y el atributo.

Diferencias entre los lenguajes de programación

- Número de entidades.
- Número de atributos que se les pueden ligar.
- Momento en el que se hacen las ligaduras o binding time.
- La estabilidad de la ligadura, o sea si una vez establecida podré modificarla.

Los binding podrán ser estáticos o dinámicos

- Ligadura estática: se establece antes de la ejecución y no se puede cambiar. El término estático referencia al momento del binding y a su estabilidad.
  - Definición del lenguaje.
  - Implementación del lenguaje.
  - Compilación.
- Ligadura dinámica: si se establece en el momento de la ejecución y puede cambiarse de acuerdo a alguna regla específica del lenguaje, excepto por las constantes.
  - Ejecución.

Variable

Las variables son un atributo conformado por **<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>**

- Nombre: string de caracteres que se usa para referenciar a la variable o sea el identificador.

  Con respecto al nombre, hay ciertos aspectos de diseño como

Longitud máxima	Se refiere al número máximo de caracteres que se pueden utilizar para nombrar una variable. Algunos lenguajes como Python, Pascal o Java no tiene límite y otros por ejemplo C suele tener un máximo de 32
Caracteres aceptados	Referido a los caracteres que se pueden utilizar para nombrar una variable. Python, C o Pascal aceptan “_” en cambio Ruby solo permite letras en minúsculas para variables locales y “$” para comenzar los nombres de variables globales.

Sensitivos	Se refiere a si un lenguaje distingue entre mayúsculas y minúsculas en los nombres de las variables. Por ejemplo C y Python sí es sensible en cambio Pascal no.

- Alcance: conjunto de instrucciones del código en el que una variable es visible y accesible.

  Las instrucciones del programa pueden manipular una variable a través de su nombre dentro de su alcance. Los lenguajes adoptan diferentes reglas para ligar un nombre a su alcance.

Alcance estático/ alcance léxico	En el alcance estático, el alcance se define en función de la estructura léxica del programa y se vincula estáticamente a una declaración sin necesidad de ejecutar el programa.
Alcance dinámico	A diferencia del alcance estático, este alcance se define en la ejecución del programa y no de la estructura léxica.

  Otros conceptos asociados al alcance serán

  - Local: son todas las referencias que se han creado dentro del programa o del subprograma.
  - No local: son todas las referencias que se usan dentro del subprograma pero que no han sido creadas en él.
  - Global: son todas las referencias creadas en el programa principal.

- Tipo: conjunto de valores y operaciones que están permitidas en esa variable. Antes de que una variable pueda ser usada, se le debe asignar un tipo, lo que permite verificar las operaciones realizadas y asegurar que sean válidas al tipo correspondiente.

  La asignación de tipos protege a las variables de operaciones no permitidas y ayuda a prevenir errores. Además permite que el compilador o intérprete haga optimizaciones. Tenemos diferentes tipos para las variables

Predefinidos	Son los tipos base que están descriptos en la definición por ejemplo boolean, acepta valores true o false y solo puedo hacer operaciones lógicas como and, or, not, etc. Los valores se ligan en la implementación a representación de máquina
Definidos por el usuario	Los lenguajes permiten definir nuevos tipos a partir de los predefinidos.
Tipos de datos abstractos (TADs)	Son una forma de definir nuevos tipos que no están en el lenguaje. Para definir un TDA hay que especificar tanto la representación como las operaciones que pueden realizar. Suelen representar una estructura compleja que puede estar compuesta por tipos básicos u otros TADs. Por ejemplo una clase Libro.

No es lo mismo los tipos definidos por el usuario que los tipos de datos abstractos ya que los definidos por el usuario se definen en términos de datos básicos que ya existen en el lenguaje mientras que los TADs permiten definir tanto la estructura de datos como las operaciones que pueden hacer.

MOMENTO ESTÁTICO

La ligadura de tipo se realiza durante la compilación del programa y no puede cambiarse en la ejecución del programa, también verificará que todas las operaciones hechas con la variable sean válidas.

La ligadura puede hacerse de forma explícita, inferida o implícita

- Explícita: se establece mediante una declaración.
- Inferido: el tipo de una variable se deduce a partir de los tipos de su componente. Por ejemplo, una suma entre un entero y un flotante, el resultado se inferirá como flotante para lograr mayor precisión.
- Implícito: el tipo de una variable se deduce mediante reglas establecidas. Un ejemplo de ligadura implícita podría ser en el lenguaje C, donde las constantes literales son ligadas a un tipo automáticamente según su formato. Por ejemplo, la constante "123" será ligada como un entero (int), mientras que "123.45" será ligada como un número de punto flotante.

MOMENTO DINÁMICO

El tipo se liga en ejecución y puede cambiarse. Es más flexible, más costoso en ejecución, realiza un chequeo dinámico y tiene menor legibilidad.

- L-value: área de memoria ligada a una variable. Surgen dos conceptos, el de tiempo de vida (periodo de tiempo que existe la ligadura) y el de alocación (momento que se reserva la memoria). Entonces, el tiempo de vida es el tiempo en que la variable está alocada en memoria.

  La alocación puede ser

  - Estática: antes de que se ejecute el programa.
  - Dinámica: durante la ejecución del programa
  - Persistente: referido a la reserva de memoria para objetos que no se liberan hasta que se termina la ejecución del programa. Por ejemplo, en una base de datos, los registros pueden residir en la memoria persistente y no se liberan hasta que se elimina el registro o se cierra la BBDD.
- R-value: valor codificado almacenado en la ubicación de la variable. Por ejemplo, en la instrucción "x = 5", "x" es un "l-value" ya que es el objeto que recibe el valor 5. El "r-value", por otro lado, se refiere al valor que se encuentra en el lado derecho de la asignación. Por ejemplo, en la instrucción "x = 5", el "r-value" es 5. Se accederá a la variable a través del l-valor y se puede modificar el r-value.
  - R-value dinámico: valores que pueden cambiar durante la ejecución del programa
  - R-value constante: el valor está fijo y no puede cambiar durante la ejecución

  INICIALIZACIÓN

  Es la declaración para especificar el valor inicial del r-valor. Esta estrategia por defecto establece un valor predefinido para el r-valor de una variable si no se especifica uno. Por ejemplo los enteros no inicializados empiezan con 0, los string en blanco, etc.

VARIABLES ANÓNIMAS Y REFERENCIAS

Algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable

ALIAS

Dos nombres que denotan la misma entidad en el mismo punto de un programa. Las variables tendrán distintos nombres pero tendrán ligada la misma entidad.

La desventaja es que las alias pueden hacer programas difíciles de leer.

SOBRECARGA

En el alias dos nombres apuntan a la misma entidad. En el caso contrario de la sobre carga, un único nombre apunta a distinta entidades.

Se considerará un nombre sobrecargado si referencia a más de una entidad y hay suficiente información para permitir ligaduras unívocamente.

Espacio de nombres

Un espacio de nombre es una zona separada donde se pueden declarar y definir objetos, funciones o cualquier identificador de tipo, clase, estructura, etc. al que se le asigna un nombre propio. Ayudan a evitar problemas con identificadores con el mismo nombre en proyectos grandes o con bibliotecas externas.