Universidad del Valle de Guatemala
Diseño de Lenguajes de Programación
Catedrático: Tomás Gálvez
Semestre 2, 2015

Introducción a la compilación

Un lenguaje de programación es una forma de comunicar instrucciones a una computadora. Lo que nosotros escribimos y vemos en un programa es llamado el **código fuente**, y es la parte entendible para un humano de un programa.

La computadora no entiende código fuente, sino **código objeto**. Como la computadora es, al final, hardware, la forma física en la que realiza cálculos y procesos es con señales eléctricas, y la forma en la que nosotros más nos podemos acercar a esto sin entrar directamente a la electricidad es por medio de la especificación de dichas señales: código binario. Como sabrán, el código binario tiene todavía una capa encima que lo hace más inteligible para los humanos: el código ensamblador, o *assembler*. Puesto que este es casi únicamente una "verbalización" del código binario, y puesto que el código binario es específico para cada pieza de hardware, el *assembler* del que hablamos es un lenguaje diseñado específicamente para dicha pieza (un **lenguaje de bajo nivel**).

Nosotros podemos entonces escribir un programa en lenguaje ensamblador, pero escribir en *assembler* es casi como que si quisiéramos comunicarnos con alguien que habla otro idioma usando *Yahoo! Babel Fish Translator* (ni siquiera *Google Translate*) en el sentido de que a menos que sepamos exactamente cómo comunicar lo que necesitamos decir, y cómo el traductor funciona, el resultado puede ser una idea distinta o incluso inentendible. Por ello se crearon **lenguajes de alto nivel**, y éstos se acercan más a los lenguajes naturales hablados por nosotros. Sin embargo, hablar a la computadora con un lenguaje de alto nivel requiere una traducción más sofisticada pues se deben traducir las ideas a lenguajes de bajo nivel de forma tal que la computadora realice lo que nosotros deseamos. El traductor encargado de esto es el **compilador**.

Como ejemplo, pensemos en un típico ciclo de Python:

```
for i in range(1,10)
```

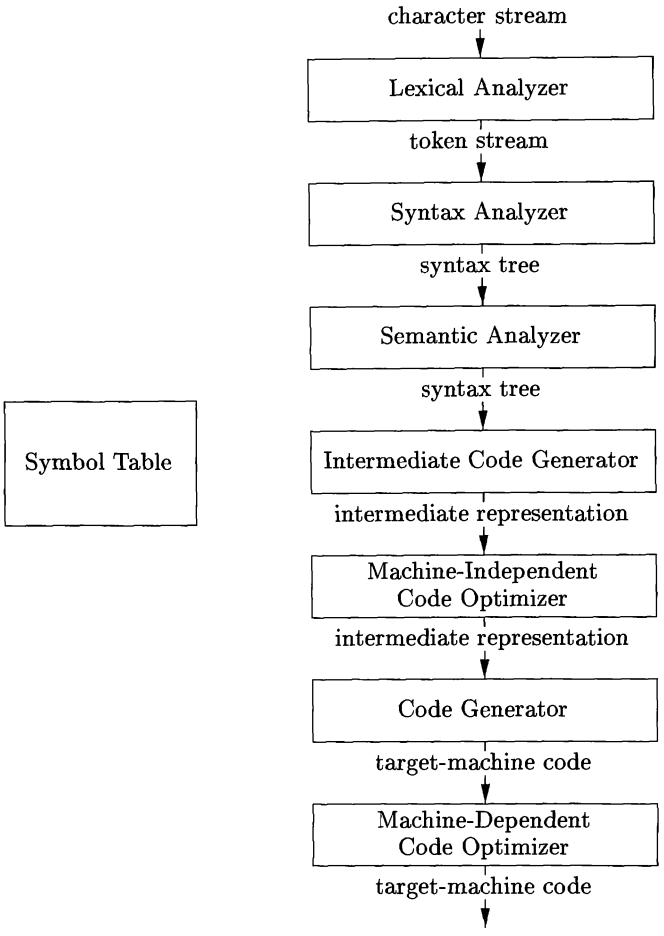
Este ciclo podemos casi traducirlo a español como "para cada elemento en el rango de 1 a 10 haga lo siguiente". Siendo Python un lenguaje de alto nivel podemos ver la diferencia con *assembler* donde dicho ciclo debería ser expresado como lo haríamos en un diagrama de flujo o un algoritmo narrativo: "haga lo siguiente, y verifique que el valor del registro x sea tal; si no es así, vuelva a la línea tal", y eso sin contar cómo manejamos el almacenamiento del valor de verificación, cómo marcamos la línea a la que se debe regresar, etc.

Debemos observar que aunque los compiladores normalmente traducen lenguajes de alto nivel a lenguaje de máquina, un compilador puede traducir a cualquier lenguaje. Es por eso que pertenece, junto a los **intérpretes**, a la categoría de **procesadores de lenguajes**. La diferencia entre intérpretes y compiladores es que un intérprete ejecuta las instrucciones *on the go*, una por una, y a partir de una traducción intermedia en lugar de traducir todo el programa primero a código ejecutable para después ser ejecutado por la máquina, como hace un compilador.

Ventajas de los intérpretes son que como el programa se ejecuta conforme se lee es más fácil encontrar errores, y por lo tanto desarrollar en un lenguaje interpretado. También el procesamiento del lenguaje fuente es más rápido con un intérprete. La desventaja general es que un programa interpretado corre bastante más lento que un programa primero compilado.

El funcionamiento de un compilador se divide en dos: **análisis** y **síntesis**. El análisis es el reconocimiento de la estructura del programa fuente con todo y recolección de información (cosas como nombres de variables, si son locales o globales, tipos, etc.), y la síntesis es la construcción de la traducción a partir de la estructura y la información recolectada.

La construcción y operación de un compilador se separa en las siguientes fases:



La **Tabla de Símbolos** es una estructura de datos donde se almacena la información recolectada durante el análisis. Dado que esta información es manejada en todo momento mientras opera el compilador, la tabla de símbolos está presente en todas las fases.

Universidad del Valle de Guatemala
Diseño de Lenguajes de Programación
Catedrático: Tomás Gálvez
Semestre 2, 2015

Fase 1: análisis léxico

Llamado también el *scanning*, esta fase se encarga de identificar los elementos del programa fuente de acuerdo a la **gramática** del lenguaje analizado. Esta fase almacena información en la tabla de símbolos sobre cada uno de estos elementos, y produce ***tokens*** que presentan la identificación de qué es el elemento dado, y cuál es su posición asociada en la tabla de símbolos. Un *scanneo* transforma el programa en una representación de elementos gramaticales en la tabla de símbolos. Por ejemplo, asignar una expresión a una variable como a continuación:

```
Position = initial + rate * 60
```

Se transforma, luego del *scanneo*, en:

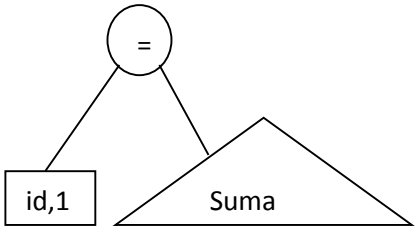
<**id,** 1><**=**><**id,** 2><**+**><**id,** 3><*****><**60**>

Significando que `Position`, `initial` y `rate` son tres identificadores de variables separados, y lo demás son operadores, o valores inmediatamente reconocibles.

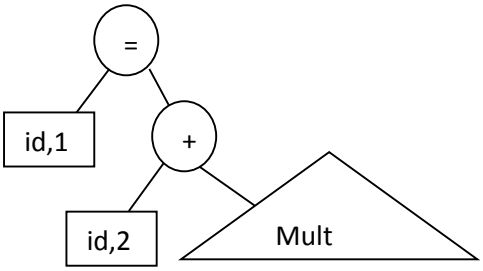
Fase 2: análisis sintáctico

También llamado *parsing*, es la parte que impone la estructura gramatical al programa fuente. Usa los *tokens* producidos por el *scanner* para armar una representación llamada **árbol sintáctico**, la cual presenta el orden en el que las operaciones deben ser realizadas. Para la línea producida por el *scanner* en el ejemplo, el árbol sintáctico se arma así:

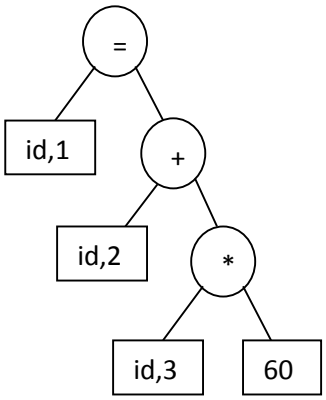
1. ¿Qué operación se realiza en toda la línea? Una asignación. ¿Cuáles son sus argumentos? <**id,** 1> y una "expresión de suma":



2. La suma es otra operación. ¿Cuáles son sus argumentos? <**id,** 2> y "una operación de multiplicación":



3. Finalmente, ¿cuáles son los argumentos de la multiplicación? <**id,** 3> y <**60**>



Es fácil observar en el ejemplo que las hojas del árbol son siempre argumentos a operaciones, las cuales a su vez son nodos internos. En este caso la raíz determina la operación final del programa. Estas observaciones se aplican a árboles sintácticos de programas más grandes, donde la raíz es puede ser la declaración de un programa con nombre (`Program HelloWorld`) como se haría en Pascal, por ejemplo.

Es importante observar que durante el *scanneo* se pudo haber encontrado un elemento irreconocible (por ejemplo, un símbolo '/' si la operación de división no está definida); y durante el *parseo* se pudo haber detectado un orden incorrecto de *tokens* (por ejemplo, una operación binaria como la asignación sin uno de sus argumentos, equivalente a que hubiéramos escrito sólo un signo = seguido de la suma, o sólo el <**id,** 1> y el <**=**> sin una expresión que asignar). Estos son los errores de sintaxis que conforman la primera capa de procesamiento de un compilador (y normalmente un intérprete también), y es tarea del compilador detectar y notificar estos errores antes de continuar con el procesamiento del lenguaje.

Fase 3: análisis semántico

El análisis semántico usa el producto del *parser* (que usualmente es el árbol sintáctico), y la tabla de símbolos para verificar la semántica (es decir, el significado) de las instrucciones en el programa fuente. Gran parte de este análisis es revisión de tipos, donde se chequea que las operaciones realizadas usen los tipos que les son adecuados. Otra parte del análisis semántico es ver las *variable bindings* a través de ámbitos o ***scopes***, que es determinar la relación entre instrucciones de un programa. Esta parte del análisis es la que permite diferenciar entre dos variables que se llamen igual cuando una es global y la otra es local.

Fase 4: generación de código intermedio

Aunque parezca un paso extra, la generación de código intermedio presenta varios beneficios potenciales. Primero, hace portables las primeras fases del compilador. Este código intermedio podría pasar por diferentes implementaciones de la fase de generación de código objeto para traducirse tanto a diferentes lenguajes de alto nivel como a diferentes lenguajes de máquina o, incluso, a diferentes versiones de lenguaje de máquina que tomen en cuenta recursos de hardware específicos. Un excelente ejemplo de la portabilidad aquí mencionada es la plataforma .NET de Microsoft. Otro ejemplo es la generación de *bytecode* de Java. Esta última es especial porque este código intermedio es resultado de una compilación, y está destinado a ser interpretado por la JVM (*Java Virtual Machine*).

Otro beneficio potencial del código intermedio es la introducción de optimizaciones. El código intermedio es una representación del programa que se parece al árbol sintáctico y al código *assembler* que se necesitará, pero no es ninguno de los dos. Este código debe ser fácil de producir y fácil de traducir. Por ello, código intermedio como el ***three-address code*** propuesto en el libro del dragón (Aho) limita las instrucciones a un máximo de tres operandos. Además, introduce variables de almacenamiento que representan los registros del procesador. Una característica ejemplar que facilita la producción de este código es permitir la introducción de tantos registros como se necesite, y dejar el manejo de estos recursos a la fase de generación de código. Un código intermedio puede también hacer más directa la traducción del programa a lenguaje de máquina por medio del ordenamiento de operaciones (recordemos que *assembler* comúnmente no permite expresiones matemáticas largas, entonces no hay precedencia de operadores). El siguiente ejemplo muestra el *three-address code* para el árbol sintáctico presentado anteriormente:

```
t1 = inttofloat(60)

t2 = id3 * t1

t3 = id2 + t2

id1 = t3
```

Fase 5: generación de código objeto

Esta fase comprende simplemente la producción o generación del código objeto deseado. Uno de los problemas más importantes en esta fase es el correcto manejo de recursos como los registros del procesador (en el caso de generación de *assembler*). Recordando que en la generación de código intermedio nos despreocupamos de este asunto, en esta fase se implementan algoritmos (por ejemplo, el algoritmo de coloración) para manipular y llevar registro de dónde están los valores de nuestras variables en todo momento, si es necesario guardar información en la pila o no, las instrucciones que se utilizarán de acuerdo al código objeto, etc.

Los compiladores regularmente poseen unas fases extra llamadas las **fases de optimización**. Estas fases rodean la generación de código (o sea que pueden ocurrir luego de la generación de código intermedio y/o luego de la generación de código objeto). En algunos textos estas fases se descomponen en más fases. Sin embargo, se pueden resumir en optimización independiente de máquina y optimización dependiente de máquina, donde la independiente de máquina se aplica al código intermedio para producir un mejor código que todavía es portable para generación de código objeto en diferentes máquinas; y la dependiente de máquina se aplica directamente al código objeto y aprovecha las características de la máquina que ejecutará dicho código.

La optimización de código se encarga de tareas específicas para el lenguaje y la máquina con las que se trabaja. Optimizaciones que se toman en cuenta pueden ser la reducción de una asignación como `Y = X * 0` a `Y = 0`, o la eliminación de una condición o un ciclo del código a producir si se puede determinar de antemano que su condición no se cumplirá nunca.

La optimización de código es generalmente opcional, aunque un compilador sin optimización de código difícilmente tendría lugar en la industria. Lo que sí es común es que se elija sólo un tipo de optimización (dependiente o independiente de máquina).

Las pasadas

Una pasada es una lectura y procesamiento del programa fuente (o una representación del mismo) que agrupa varias de las fases mencionadas. Lo normal es que un compilador haga más de una pasada, comprendiendo en la primera casi siempre sólo las fases sintácticas (*scanning* y *parsing*), y a veces la fase semántica. Es por esta razón que a veces al compilar nuestro programa se detectan primero unos errores y, luego de ser corregidos, se detectan otros más profundos. En ocasiones se interrumpe el procesamiento del programa si se encuentran errores en las fases iniciales, pues errores en las siguientes fases es muy probable que sean consecuencia de ello. Podría decirse que los intérpretes realizan una pasada inicial que llega hasta la generación de código intermedio, y parten desde allí a la ejecución línea por línea que les caracteriza.

Características de un lenguaje de programación

¿Para qué sirve la palabra reservada `static` en Java? Sabemos que cuando declaramos una variable de clase como estática todas las instancias de esa clase compartirán esa variable. Si profundizamos más, sabemos que lo que sucede es que al declararla estática se le está asignando a esa variable un espacio de memoria fijo de modo que, para todas las instancias, ese nombre de variable apunta al mismo lugar. En referencia al proceso de compilación, la variable es estática porque al momento de compilar el programa estamos provocando que la variable quede asignada a un espacio de memoria, mientras que cuando una variable no es estática se le asigna un espacio de memoria en cada instancia durante la ejecución.

El significado de una declaración estática va más allá. Muchos sabrán que el contrario de estático (en el contexto de lenguajes de programación, al menos) es dinámico. Pues hablamos de políticas y ámbitos estáticos o dinámicos cuando un compilador toma decisiones durante tiempo de compilación (o antes) o tiempo de ejecución, respectivamente.

No hay que confundir ámbitos con **ambientes**. Un ámbito es una porción de programa donde aplican determinadas asociaciones entre nombres y entidades (variables, métodos, etc.). El ambiente se refiere a la relación entre variables (específicamente, sus nombres) y ubicaciones en el almacenamiento. Estas ubicaciones contienen un valor; y dicha relación de contención entre ubicaciones y valores es lo que conocemos como estado. Si tomamos un *snapshot* de un programa en ejecución, el ambiente es el *mappeo* de los nombres de variables en ese programa a ubicaciones en la memoria donde se almacenan sus valores. Nótese que dependiendo del ámbito de una variable el ambiente puede cambiar en cualquier momento. Por otro lado, el estado es un concepto más fácil de visualizar porque lo hemos visto antes: hablamos del estado de un programa cuando hablamos de "el valor de sus variables en un momento dado". Específicamente, este estado es una referencia al *mappeo* entre las ubicaciones de memoria usadas por el programa, y los valores almacenados en ellas.

Muchos lenguajes usan una estructura por bloques con **ámbito estático**. Las cosas que permiten determinar dicho ámbito en el programa fuente son los delimitadores como las llaves, y las palabras reservadas como `begin`, `end`, `public`, `private` o `protected` (estas últimas proveen lo que se conoce como **control de acceso explícito**, promoviendo la encapsulación). En una estructura por bloques, los bloques son una secuencia de declaraciones seguida por una secuencia de *statements* delimitados con llaves. Los bloques en sí son *statements* y, por el punto anterior, pueden estar anidados. Los bloques denotan la pertenencia y visibilidad de una variable, donde una variable tiene como ámbito todo el bloque donde fue declarada a excepción de los bloques internos donde hay una variable declarada con el mismo nombre.

La estructura por bloques y el ámbito estático relacionan el uso de un nombre de variable con la declaración más cercana en cuanto al bloque donde se hace este uso. Existe una relación de nombres con las declaraciones más recientes en cuanto al procedimiento o rutina en ejecución. Esta relación es la llamada de **ámbito dinámico**, y la vemos comúnmente en el polimorfismo. El ámbito es determinado por la última rutina llamada a ejecución, y se maneja una pila de asociaciones del nombre en cuestión con el objeto o valor que determina el ámbito. En polimorfismo lo observamos cuando tenemos clases que implementan una misma interfaz. Llamar a dicho método sobre una variable del tipo de la interfaz hace imposible saber, al compilar, la implementación de cuál de las clases se debe ejecutar. Cuando el compilador evalúa la llamada al método con el formado <objeto>.<método>, al evaluar <objeto> empuja un *binding* en la pila de asociaciones con el nombre de <objeto>, lo cual determina el ámbito (clase) de la cual se obtendrá el método a llamar.

Universidad del Valle de Guatemala
Diseño de Lenguajes de Programación
Catedrático: Tomás Gálvez
Semestre 2, 2015

Sobre la clasificación de lenguajes

Las **generaciones** comprenden acercamientos supuestamente cada vez más intuitivos a la programación. La primera generación es la de lenguaje de máquina puro y duro. La segunda generación consiste en el *assembler*, una versión más humana del lenguaje de máquina. La tercera generación posee la mayoría de lenguajes que conocemos, los lenguajes de alto nivel como C, Java, Fortran, etc. La cuarta generación es una especialización de los lenguajes: SQL, MATLAB, R y, de acuerdo a Wikipedia, Unix Shell. Lenguajes considerados de la quinta generación son los que usan relaciones y reglas para solucionar problemas en lugar de algoritmos. Ejemplos de este tipo son Haskell y Prolog, que casualmente vienen a ser lenguajes de tipo **declarativo**, uno de los tipos de lenguaje. Tenemos los lenguajes **imperativos**, que son los que manejamos comúnmente, y se llaman así por indicar a la computadora de forma imperativa cómo hacer las cosas. Los lenguajes que no son imperativos se consideran regularmente declarativos, cuya característica es que especifican simplemente qué hacer, sin detallar cómo. Aunque varios lenguajes declarativos se asocian con la quinta generación, hay lenguajes de otras generaciones que también son declarativos como el SQL.

Podría decirse que estos "tipos" de lenguajes son en realidad paradigmas de programación, entre los que podemos contar la orientación a objetos que ya conocemos bien, y el *scripting*. Los *scripting languages* son interpretados. Además, los *scripting languages* son comúnmente fáciles de usar en el sentido de que se pueden simplemente escribir en un archivo cualquiera e importar o ejecutar, como módulos escritos en Python y Javascript, o rutinas de ejecución para el *shell* de Linux o Windows (.bat's). Muy frecuentemente se hace equivalencia o uso indiferente de los términos de lenguaje interpretado y *scripting language*, pero diferencias que se han traído a observación son que un lenguaje de *scripting* se enfoca en determinar una secuencia de ejecuciones como en un *bash* más que especificar el desarrollo de un programa. También se ha notado que los lenguajes interpretados abarcan más que el *scripting*, debido a que no todos los lenguajes interpretados son usados para *scripting* (ej.: *bytecode*). Además, en teoría se pueden usar lenguajes compilados para hacer *scripting*. Por ello podríamos decir que el *scripting* es un acercamiento.

Fuentes:

- http://en.wikipedia.org/wiki/Programming_language
- https://en.wikipedia.org/wiki/Variable_(computer_science)
- http://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_6/declarative/miniweb/index.htm
- Aho, A. V., Lam, M. S., Ravi, S., & Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools.* Pearson.
- Appel, A. (2004). *Modern Compiler Implementation in (Java|C|ML).* Cambridge.
- Stanford University. (2012). *Video Lectures*. Retrieved from Compilers: https://class.coursera.org/compilers-selfservice/lecture/index