INTRODUCCIÓN A LOS COMPILADORES

José Sánchez Juárez

Septiembre, 2018

II

Índice general

Índice de figuras

Índice de cuadros

Introducción a los compiladores

ARTICULO 1.- El Instituto Politécnico Nacional es la institución educativa del Estado creada para consolidar, a través de la educación, la Independencia Económica, Científica, Tecnológica, Cultural y Política para alcanzar el progreso social de la Nación, de acuerdo con los objetivos Históricos de la Revolución Mexicana, contenidos en la Constitución Política de los Estados Unidos Mexicanos.

Ley orgánica.
Instituto Politécnico Nacional

El rol de la computadora en la vida diaria crece cada año. Con el aumento de Internet, las computadoras y el software que corre sobre ellas proveen aplicaciones, noticias, entretenimiento y seguridad. Las computadoras embebidas han cambiado las formas en que construimos automóviles, aviones, teléfonos, televisores y radios. La computación ha creado categorías de actividad completamente nuevas, desde videojuegos para redes sociales. Las supercomputadoras predicen el clima a diario y el curso de las tormentas violentas. Las computadoras integradas sincronizan los semáforos y entregan un correo electrónico a un usuario.

Todas estas aplicaciones informáticas se basan en programas informáticos de software que crean herramientas virtuales además de las abstracciones de bajo nivel proporcionadas por el hardware subyacente. Casi todo este software se traduce por una herramienta llamada compilador. Un compilador es simplemente un programa de computadora que traslada otros programas de computadora para prepararlos para su ejecución, gráficamente mostrado en la figura 2 [5] .

Arquitectura de los compiladores e intérpretes

Para determinar la estructura general de los compiladores e intérpretes con base en la funcionalidad de sus etapas y fases. Podemos modelar el proceso

Figura 3: Diagrama del concepto del programa objeto.

de traducción entre dos lenguajes como el resultado de dos etapas, tal como se presenta en la figura 1 . En la primera etapa se analiza la entrada para averiguar qué es lo que se intenta comunicar. Esto es lo que se conoce como análisis. El fruto de esta etapa es una representación de la entrada que permite que la siguiente etapa se desarrolle con facilidad. La segunda etapa, es la síntesis, toma la representación obtenida en el análisis y la transforma en su equivalente en el lenguaje destino. En el caso de la interpretación, se utiliza la representación intermedia para obtener los resultados deseados.

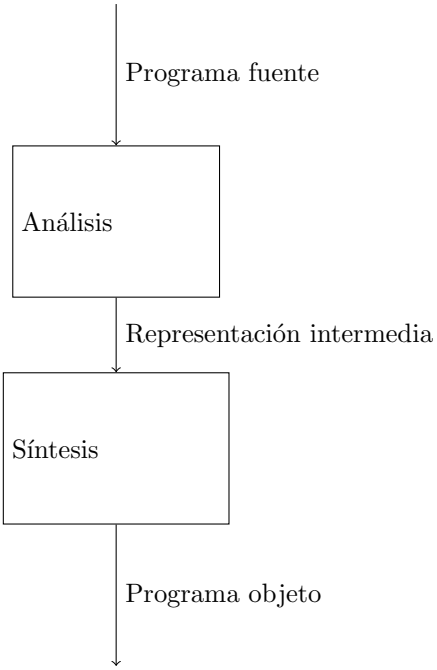

Figura 1: La arquitectura de un compilador.

Figura 2: Diagrama del concepto de compilador.

Si el programa objeto es un ejecutable en lenguaje de máquina, este entonces puede ser llamado por el usuario para procesar entradas y producir salidas gráficamente mostrado en la figura 3 [3] .

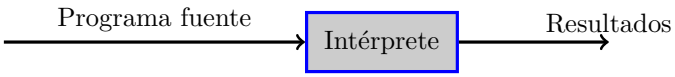

Figura 4: Diagrama del concepto del intérprete.

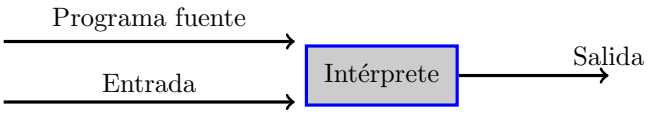

Figura 5: Diagrama del intérprete para ejemplificar con el compilador.

Un intérprete es otra clase común de lenguaje de procesador. En lugar de producir un lenguaje objeto como una traducción, un intérprete aparece para directamente ejecutar las operaciones de especificadas en el programa fuente sobre las entradas alimentadas por el usuario.

Definición de compiladores e intérpretes

Mientras que el objetivo de los compiladores mostrado en la figura 2 , es obtener una traducción del programa fuente a otro lenguaje, los interprete tienen como objeto la obtención de los resultados del programa mostrado en la figura 4 . Para ello deben realizar dos tareas: analizar su entrada y llevar a cabo las acciones especificadas por ella. La parte de análisis puede realizarse de manera idéntica a como se lleva a cabo en los compiladores. Es la parte de síntesis la que se diferencia sustancialmente. En el caso de la interpretación, se parte del árbol de sintaxis abstracta y se recorre, junto con los datos de entrada, para obtener los resultados.

DEFINICIÓN 1 (Compilador.) *Es un programa de computadora que traduce a otro programa de computadora [5] .*

DEFINICIÓN 2 (Intérprete.) *Otro tipo de traductor, llamado intérprete, procesa en forma interna el programa fuente y los datos al mismo tiempo. Es decir, la interpretación del formulario de origen interno se produce en tiempo de ejecución y no se genera ningún programa objeto.*
[7] .

Un intérprete es simplemente un dispositivo que toma alguna representación de un programa y lleva a cabo las operaciones que el programa especifica, es decir, imita o simula las operaciones que una máquina llevaría a cabo si fuera directamente capaz de procesar programas escritos en ese idioma. Un compilador toma una representación similar de un programa y produce instrucciones que, cuando es medido por una máquina, llevará a cabo las operaciones que el programa especifique. La diferencia entre un intérprete y una máquina bajo

esta definición no es muy bueno: el microprograma de una computadora es un intérprete que dice que el programa de código de máquina imita el comportamiento que una máquina "real" que haría expresar ese programa dado [**?**] .

Un intérprete es otra forma de implementar un lenguaje de programación. La interpretación comparte muchos aspectos con la compilación. análisis léxico y verificación de tipos están en un intérprete como en un compilador. Pero en lugar de generar el código del árbol de sintaxis, el árbol de sintaxis se procesa directamente para evaluar expresiones y sentencias de ejecución, etc. Un intérprete puede necesitar el mismo fragmento del árbol de sintaxis (por ejemplo, el cuerpo de un bucle), la interpretación es típicamente más lenta que la ejecución de un programa de compilador. Pero escribir un intérprete es a menudo más simple que escribir un compilador y el intérprete es más fácil de emigrar a una máquina diferente, por lo que para aplicaciones donde la velocidad no es esencial, a menudo se usan intérpretes [4] .

Arquitectura de los compiladores e intérpretes

La arquitectura de un compilador tiene las dos etapas: análisis y síntesis. Mientras que un intérprete solamente tiene la etapa de análisis.

Máquinas virtuales

Una máquina virtual es un software que emula el hardware de una PC. Existen dos tipos de máquinas virtuales: máquina virtual de sistema y máquina virtual de proceso.

DEFINICIÓN 3 (Bytecode.) *Es un lenguaje intermedio en el que se compilan las aplicaciones escritas en Java.*

Como el Bytecode es un lenguaje intermedio, este puede ser la entrada a una máquina virtual, que tenga conexión con un sistema operativo.

Figura 6: Diagrama del concepto de Bytecode.

Algunos lenguajes adoptan esquemas de traducción que incluyen tanto compilación e interpretación. Java se compila desde el código fuente en un formulario llamado bytecode, una representación compacta destinada a disminuir los tiempos de descarga para aplicaciones Java. Las aplicaciones Java se ejecutan ejecutando el bytecode en el máquina virtual Java correspondiente (jvm), un intérprete para bytecode. Al complicar aún más la imagen, muchas implementaciones de la jvm incluyen un compilador que se ejecuta en tiempo de ejecución, a veces llamado compilador just-in-time, o jit, que traduce secuencias de código

de bytes muy usados en código nativo para la computadora subyacente. Parecido a lo que se muestra en la figura 7 .

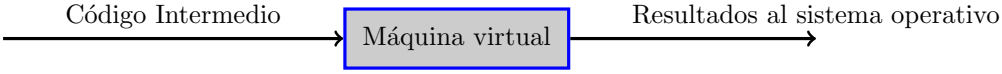

Figura 7: Diagrama del concepto de máquina vitual.

DEFINICIÓN 4 (Conjunto de instrucciones.) *El conjunto de operaciones soportadas por un procesador; el diseño general de un conjunto de instrucciones es a menudo llamado una arquitectura del conjunto de instrucciones o ISA. [5] .*

DEFINICIÓN 5 (Máquina virtual.) *Una máquina virtual es un simulador para algunos procesadores. Es un intérprete para ese conjunto de instrucciones de la máquina [5] .*

Las máquinas virtuales siven para:

1. Para poder probar otros sistemas operativos.
2. Para ejecutar programas antiguos.
3. Para usar aplicaciones disponibles para otros sistemas.
4. Para probar una aplicación en distintos sistemas.
5. Como seguridad adicional.
6. Para aprovechar su gran dinamismo.

Funcionalidad de las fases de un compilador

Para observar la funcionalidad de las fases del compilador, considerar una expresión como entrada al compilador: $posición := inicial + velocidad * 60$. y observemos que pasa a la salida de cada una de las fases del compilador hasta la salida del mismo compilador.

Análisis léxico

Generalmente en computación lo que el compilador debe analizar es un programa del cual debe reconocer las palabras que contiene. De esta manera se comienza a saber si el programa esta bien escrito. Al agrupar los caracteres se van formando las palabras con la aplicación de un conjunto de reglas, las cuales se expresan por medio de producciones. Si la palabra es válida, el reconocedor

asigna una categoría sintáctica, o parte del habla. Lo cual se muestra en la figura 8 , se observa que el analizador léxico entrega los tokens.

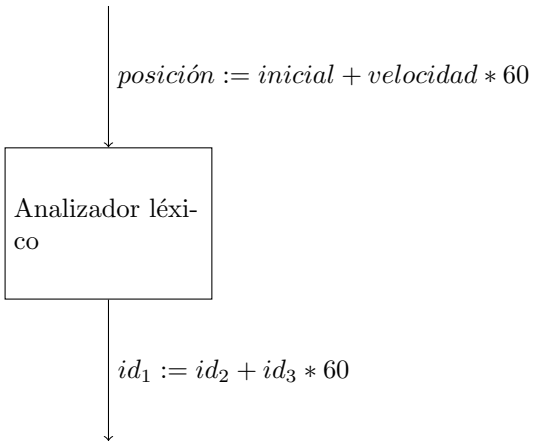

Figura 8: La entrada y la salida de la primera fase del compilador.

Existen herramientas automáticas para generar reconocedores. El proceso de la herramienta es una descripción matemática de la sintaxis léxica del lenguaje.

Para construir los reconocedores artesanales y los reconocedores generados, se aplican las mismas técnicas en ambos. Los compiladores comerciales y los compiladores de código abierto usan reconocedores artesanales. Los reconocedores artesanales son más rápidos que los reconocedores generados, porque la implementación optimiza una porción del encabezado que no se puede evitar en un reconocedor generado.

Modelo de reconocedor, programa que identifica palabras de una cadena de caracteres.

Reconocedor de palabras, la explicación más simple de un algoritmo para reconocer palabras es reconocer carácter por carácter. Por ejemplo reconocer la palabra clave **new**. Considerando la rutina SiguienteCaracter() que regresa el siguiente carácter y EsEnPan() que escribe en pantalla, se implementa como

sigue:

Algorithm 0.1: Reconocedor de palabras

```
1  c = SigCar();
2  if c == 'n' then
3  |  c = SigCar();
4  |  if c == 'e' then
5  |  |  c = SigCar();
6  |  |  if c == 'w' then
7  |  |  |  EsEnPan(Palabra Aceptada);
8  |  |  else
9  |  |  |  EsEnPan(Palabra No Aceptada);
10 |  else
11 |  |  EsEnPan(Falla);
12 else
13 |  EsEnPan(Falla);
```

Análisis Sintáctico

En la fase de análisis sintáctico, un compilador verifica si o no los tokens generados por el analizador léxico son agrupados de acuerdo a reglas sintácticas del lenguaje. Si los tokens en una cadena son agrupados de acuerdo a reglas de sintaxis del lenguaje, entonces las cadenas de tokens generados por el analizador léxico son aceptados como construcción válida del lenguaje; de otra manera un manejo de error es llamado. De ahí que dos casos son involucrados cuando se diseña la fase de análisis sintáctico de un proceso de compilación;

Todas las construcciones válidas de un lenguaje de programación deben ser especificadas: y para usar estas especificaciones un programa válido se forma. Aquellas, nosotros formamos una especificación de aquellos tokens que el analizador léxico regresará, y nosotros especificamos de que manera estos tokens son agrupados de modo que lo que resulte del agrupamiento sea una construcción válida del lenguaje.

Un reconocedor adecuado será designado para reconocer si una cadena de tokens generado por el analizador léxico es una construcción válida o no.

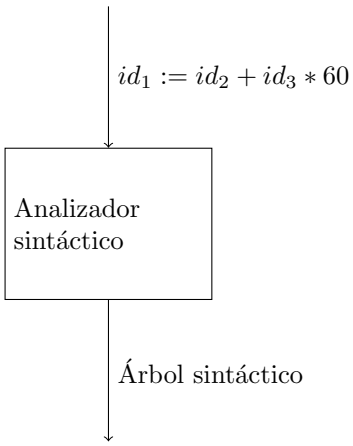

Figura 9: La entrada y la salida de la segunda fase del compilador.

Por ello una notación apropiada debe ser usada para especificar la construcción de un lenguaje. La notación para especificar la construcción debe ser compacta, precisa y fácil de entender. La especificación de la estructura sintáctica para el lenguaje de programación (es decir, la construcción válida del lenguaje) usa gramática libre de contexto (GLC), porque para ciertas clases de gramáticas, nosotros podemos automáticamente construir un analizador eficiente que determine si un programa fuente es sintácticamente correcto. De ahí que la notación GLC sea un tópico de estudio.

El análisis sintáctico es menos local en la naturaleza que el análisis léxico, donde se requieren métodos más avanzados. Se usa la misma estrategia: una notación más fácil para el entendimiento humano es transformada en una máquina semejante a una más sencilla de bajo nivel para una ejecución más eficiente. Este proceso es llamado generación del analizador.

Para encontrar la estructura del texto de entrada, el análisis sintáctico debe rechazar el texto invalido para reportar los errores de sintaxis. En la figura 9 se muestra que a la entrada del analizador sintáctico entran los tokens y a la salida se entrega un árbol sintáctico, el cual se muestra en la figura 10 .

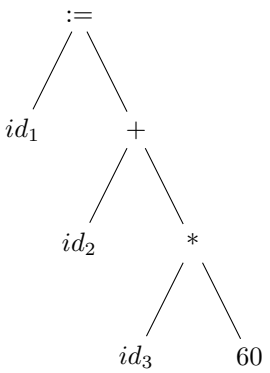

Figura 10: Árbol sintáctico, salida del analizador sintáctico.

Análisis Semántico

Un programa que es gramaticalmente correcto puede contener serios errores que evitarían su compilación. Para detectar tales errores, un compilador realiza un mayor nivel de verificación que involucra considerar cada sentencia en su contexto. Estas verificaciones encuentran errores de tipo y concordancia. En la figura 11 se muestra la entrada y la salida del analizador semántico. En la figura 12 se muestra el árbol adornado.

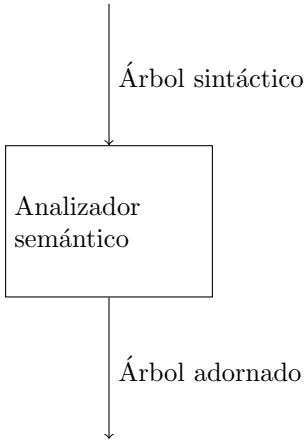

Figura 11: La entrada y la salida de la tercera fase del compilador.

Hay dos técnicas para verificar la sensitividad del contexto. Gramáticas de atributos, que son un formalismo funcional para especificar el cálculo sensible al contexto. Para esto la traducción dirigida por la sintaxis provee un marco de trabajo simple donde el escritor de compiladores puede colgar arbitrariamente fragmentos de código para desempeñar estas verificaciones.

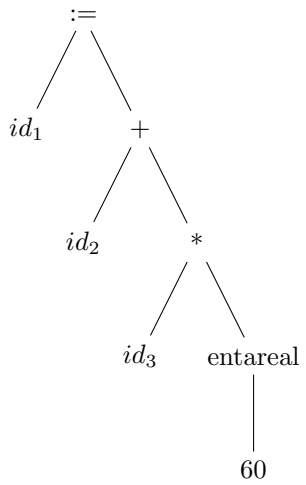

Figura 12: Árbol adornado, salida del analizador semántico.

La última tarea del compilador es traducir el programa de entrada en una forma que pueda ser ejecutado directamente en la máquina objeto. Para este propósito, se necesita saber acerca del programa de entrada que va bien acompañado de la sintaxis. El compilador debe construir una base grande de conocimientos de los códigos calculados detalladamente en el programa de entrada. Se debe saber que valores son representados, donde residen, y como fluyen de nombre a nombre. Se debe entender la estructura del cálculo. Se debe analizar como el programa interactúa con los archivos y los dispositivos externos. Todos estos hechos pueden ser derivados del código fuente, usando conocimiento contextual. Así, el compilador debe desempeñar un análisis más profundo que es típico de un escaner o un analizador.

Estas clases de análisis desempeñan uno u otro junto con el análisis o con un pase después que atraviesa la representación intermedia (IR) producida por el analizador. Llamemos a estos análisis entre ellos análisis sensible al contexto, a diferencia de la forma del analizador, o elaboración semántica, de la que se construye la representación intermedia. Se estudian dos formas: una aumentada basada en gramáticas de atributos y la otra ad hoc que utiliza conceptos similares.

Considerar un solo nombre usado en el programa que esta siendo compilado; llamemosla x. Antes de que el compilador pueda emitir un código de máquina ejecutable para el cálculo donde se involucra x, debemos tener respuesta a muchas preguntas.

¿Qué clase de valor esta almacenado en x? Los lenguajes de programación modernos usan una plétora de tipos de datos, incluyendo números, caracteres, valores booleanos, apuntadores a otros objetos, conjuntos (tales como {rojo, amarillo, verde }), y otros. Muchos lenguajes incluyen objetos compuestos que agregan valores individuales; estos incluyen arreglos, estructuras, conjuntos, y cadenas.

¿Qué tan grande es x? Porque el compilador debe manipular x, se necesita conocer la longitud de la representación de x en la máquina objeto. Si x es un número, este pudiera ser una palabra (un número entero o de punto flotante), dos palabras (un número de punto flotante de doble precisión o un número complejo), o cuatro palabras (un número de punto flotante de cuarta precisión o un número complejo de doble precisión). Para arreglos y cadenas, el número de elementos puede ser fijado a la vez en la compilación o puede ser determinado en el tiempo de ejecución.

¿Si x es un procedimiento, qué argumento se toma? ¿Qué clase de valor, si cualquiera, se retoma? Antes de que el compilador pueda generar código para involucrar un procedimiento, debe saber cuantos argumentos se codifican para la llamada esperada del procedimiento, donde se espera encontrar esos argumentos, y que clase de valor es esperado en cada argumento. Si el procedimiento regresa un valor, ¿dónde estará la rutina que se llama para encontrar el valor, y que clase de dato será ? (El compilador debe asegurar que la llamada a procedimiento usa el valor de una manera consistente y segura. Si la llamada a procedimiento asume que el valor regresado es un apuntador que se puede diferenciar, y la llamada a procedimiento regresa una cadena de caracteres arbitraria, los resultados pueden no ser predecibles, seguros, o consistentes.)

¿Cuánto debe valer x para ser preservado? El compilador debe asegurar que el valor de x permanece accesible para cualquier parte de los cálculos que deban hacer referencia de estos. Si x es una variable local en Pascal, el compilador

puede fácilmente sobrestimar el tiempo de vida de interés de x para preservar su valor durante la duración del procedimiento que declara x. Si x es una variable global que puede ser referenciada donde sea, o si esta es un elemento de una estructura explícitamente posicionado por el programa, el compilador puede tener más dificultades que determinan su tiempo de vida. El compilador puede siempre preservar el valor de x para el cálculo entero; sin embargo, la información más precisa acerca del tiempo de vida de x deja al compilador la reutilización del espacio para otros valores sin ningún conflicto en el tiempo de vida.

¿Quién es responsable de la gestión de espacio de x (y de la inicialización)? ¿Esta gestión de espacio para x implícita, o el programa gestiona explícitamente el espacio para x? Si la gestión es explicita, entonces el compilador debe asumir que la dirección de x no puede ser conocida hasta que el programa corra. Si, de otra manera, el compilador gestiona el espacio para x en un tiempo de ejecución de las estructuras de los datos que este maneje, entonces se sabe más de la dirección de x. Este conocimiento puede dejar código eficiente.

El compilador debe derivar las respuestas a preguntas, y más, del programa fuente y de sus reglas. En un lenguaje parecido al Algol, tal como Pascal o C, muchas de estas preguntas pueden ser respondidas al examinar las declaraciones para x. Si el lenguaje no tiene declaraciones, como en APL, el compilador debe ya sea derivar esta clase de información al analizar el programa, o debe generar código que pueda manejar cualquier caso que pueda surgir.

Muchos, si no todos, estas preguntas van más alla que la sintaxis del contexto libre del lenguaje fuente. Por ejemplo, el árbol de análisis para $x \leftarrow y$ y $x \leftarrow z$ definen solamente en el texto del nombre en el lado derecho de la asignación. Si x y y son enteros mientras z es una cadena de caracteres, el compilador puede necesitar emitir diferente código para $x \leftarrow y$ que para $x \leftarrow z$. Para distinguir entre estos casos, el compilador debe ahondar en el significado del programa. Tratar exclusivamente el escaneo y el análisis. Con la forma del programa; el análisis del significado es el reino del análisis sensible al contexto.

Para ver esta diferencia entre la sintaxis y el significado más claramente, considerar la estructura de un programa en muchos lenguajes parecidos al Algol. Estos lenguajes requieren que cada variable sea declarada antes de que sea usada y cada vez que se use sea consistente con su declaración. El escritor de compiladores puede estructurar la sintaxis y para asegurar que todas las declaraciones ocurran antes que cualquier sentencia ejecutable. Una producción como:

CuerpoProcedimiento → Declaraciones Ejecutables

donde los no terminales tienen el significado obvio, asegura que todas las declaraciones ocurran antes que cualquier sentencia ejecutable. Esta restricción sintáctica no verifica la regla más adentro (que el programa en el momento declara cada variable antes que su primer uso en una sentencia ejecutable. Nadie provee una forma obvia para manejar la regla en C++ que requiere declaraciones antes de usarse para algunas categorías de variables, pero permite que el programador mezcle declaraciones y sentencias ejecutables.

Hacer cumplir la regla "declarar antes de usar" requiere un nivel de mayor profundidad del conocimiento que pueda ser codificado con una gramática lebre de contexto. La gramática libre de contexto va de acuerdo con la categoría sintáctica más que con las palabras especificadas. Así, la gramática puede especificar la posición en una expresión donde el nombre de la variable puede

ocurrir, y esto nos puede decir una vez ocurrido.

Sin embargo, la gramática no tiene forma de acoplar un ejemplo de un nombre de variable con otro; que requeriría la gramática para especificar un nivel mucho más profundo de análisis, un análisis que puede contar con el contexto y que puede examinar y manipular información en un nivel más profundo que la sintaxis libre de contexto.

Introducción a los sistemas de tipo Muchos lenguajes de programación asocian una colección de propiedades con cada uno de los valores dato. Llamamos a esta colección de propiedades los tipos de valores. El tipo especifica un conjunto de propiedades que mantienen en común para todos los valores del tipo. El tipo puede ser especificado por afiliación; por ejemplo, un entero puede ser cualquier i en el rango $-2^{31} \leq i < 2^{31}$, o el rojo puede ser un valor de un tipo enumerado color, definido como el conjunto { rojo, naranja, amarillo, verde, azul, café, negro, blanco } . Los tipos pueden ser especificados por reglas; por ejemplo, la declaración de una estructura en C define un tipo. En este caso, el tipo incluye cualquier objeto con los campos declarados en un orden; el campo individual tiene tipos que especifican los rangos permitidos de valores y sus interpretaciones. (Representamos el tipo de una estructura como el producto de tipos de sus campos que lo constituyen) Algunos tipos son predefinidos por un lenguaje de programación; otros son constituidos por el programador. El conjunto de tipos en un lenguaje de programación, a lo largo de las reglas que usan tipos para especificar el comportamiento del programa, son colectivamente llamados un sistema tipo.

Los propósitos del sistema tipo Los diseñadores de lenguajes de programación introducen sistemas tipo de tal manera que ellos puedan especificar el comportamiento del programa en un nivel más preciso tanto como sea posible que una gramática libre de contexto. Los sistemas tipo crean un vocabulario para describir ambos la forma y el comportamiento de programas válidos. Analizando un programa desde la perspectiva de su rendimiento del sistema tipo la información que no puede ser obtenido usando las técnicas de escaneo y análisis. En un compilador, esta información es típicamente usado para tres propósitos distintos: seguridad, expresividad, y eficiencia en tiempo de ejecución.

Asegurar la seguridad en tiempo de ejecución Un sistema tipo bien diseñado auxilia al compilador a detectar y evitar errores en tiempo de ejecución. El sistema tipo asegura que los programas son bien comportados, aquello es, el compilador y el sistema tiempo de ejecución pueden identificar todos los programas mal formados antes de que ejecuten una operación que cause un error en tiempo de ejecución. En realidad, el sistema tipo no puede captar todos los programas mal formados no es calculable. Algunos errores en tiempo de ejecución, tal como desreferencia fuera de los limites de un apuntador, tiene obvios efectos (y frecuentemente catastróficos). Otros, tales como errores de interpretación de un entero como un número de punto flotante, puede tener y efectos acumulativos. El compilador debería eliminar muchos errores de tiempo de ejecución con técnicas de verificación de tipo.

Para completar esto, el compilador debe: **Primero**, inferir un tipo por cada expresión. Estos tipos inferidos exponen situaciones en la cual un valor es inter-

pretado incorrectamente, tal como usar un número de punto flotante en lugar de un valor booleano.

Segundo, el compilador debe verificar los tipos de los operandos de cada operador contra las reglas que definen que permite el lenguaje. En algunos casos, estas reglas pueden requerir que el compilador convierta valores de una representación a otra. En otras circunstancias, ellos pueden olvidar tal conversión y simplemente declarar que el programa esta mal formado y, por ello, no ejecutable.

En muchos lenguajes, el compilador puede inferir un tipo por cada expresión. FORTRAN 77 tiene una particularidad su sistema tipo es simple con justamente un puñado de tipos. Los casos que pueden surgir para el operador +. Dada una expresión $a + b$ y los tipos de a y b, la tabla especifica el tipo de $a + b$. Para un entero a y un doble precisión b, $a + b$ debería ser ilegal. El compilador debería detectar esta situación y reportar antes de que el programa se ejecute, un ejemplo simple de seguridad tipo.

Para algunos lenguajes, el compilador no puede inferir tipos para todas las expresiones. APL, por ejemplo, carece de declaraciones, permite un tipo de variable para cambiar en cualquier asignación, y permite al usuario meter código arbitrario a la entrada de una indicación. Mientras que esto hace que APL sea potente y expresivo, esto asegura que la implementación debe hacer alguna cantidad de inferencia en tiempo de ejecución tipo y verificación. La alternativa, por su puesto, es asumir que el comportamiento del programa bien e ignore tal verificación. En general, esto lleva a mal comportamiento cuando un programa va torcido. En APL, muchas de las características avanzadas realmente fuertes sobre la disponibilidad de tipo y la dimensión de la información.

La seguridad es una razón fuerte para usar lenguajes con tipos, Una implementación del lenguaje que garantice captar más errores relacionados con los tipos antes de la ejecución puede simplificar el diseño e implementación de programas. Un lenguaje en el cual a cada expresión se le puede asignar un tipo ambiguo es llamado un lenguaje fuertemente tipeado. Cada expresión puede ser tipeada a la vez que se compila, el lenguaje es tipeado estáticamente; si alguna expresión puede solamente ser tipeada en tiempo de ejecución el lenguaje es tipeado dinámicamente. Dos alternativas existen: un lenguaje monotipeado, tal como código ensamblador o BCPL, y un lenguaje débilmente tipeado , uno con un peor sistema de tipo.

Generación de código intermedio

Traducción a notación postfija. La notación postfija para una expresión E se define inductivamente como sigue:

1. Si E es una variable o constante, entonces la notación postfija para E es E mismo.

2. Si E es una expresión de la forma $E1$ *op* $E2$, dónde *op* es cualquier operador binario, entonces la notación postfija para E es $E'1E'2op$, dónde $E'1$ y $E'2$ son la notación postfija de $E1$ y $E2$, respectivamente.

3. Si E es una expresión de paréntesis de la forma $(E1)$, entonces la notación postfija de E es la misma que la notación postfija para $E1$.

En la figura 13 se muestra la entrada y la salida del generador de código intermedio. En el listado 1 se presenta el código intermedio, salida del generador de código intermedio.

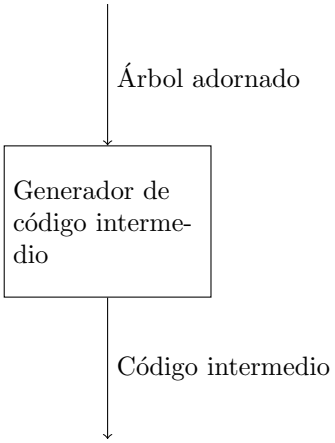

Figura 13: La entrada y la salida de la fase de generación de código intermedio.

Listing 1: Código intermedio.

```
1 temp1 := entareal(60)
2 temp2 := id3 * temp1
3 temp3 := id2 + temp2
4 id1 := temp3
```

Optimización de código

Un sistema de tipos bien diseñado provee al compilador con una información detallada de cada expresión en el programa , la información que puede ofrecer se usa para producir más traducciones eficientes. Considerar la implementación de la suma en FORTRAN 77. El compilador puede determinar completamente los tipos de todas las expresiones, de modo que este puede consultar una tabla.

La figura 14 muestra la entrada y la salida de la fase optimizador de código.

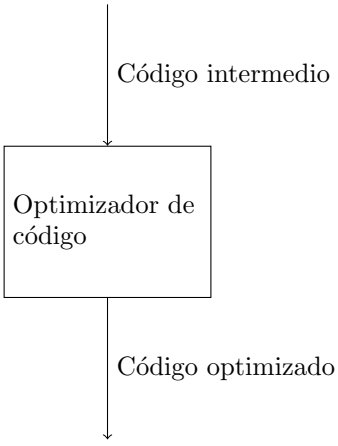

Figura 14: La entrada y la salida de la fase de optimización de código.

En el listado 2 se muestra la salida del optimizador de código.

Listing 2: Código optimizado.

```
1 temp1 := id3 * 60.0
2 id1 := id2 + temp1
```

En un lenguaje con tipos que no pueden ser totalmente determinados durante el timpo de compilación, algunas de estas verificaciones pueden ser desreferenciadas en el tiempo de ejecución. Para complementar esto, el compilador necesitaría emitir código para dos tipos numéricos, entero y real. Una implementación actualizada necesita cubrir el conjunto entero de posibilidades. Mientras estas aproximaciones aseguran un tiempo de ejecución seguro, estas sumas significan gastos generales para cada operación. Un objetivo en el tiempo de compilación es verificar esto para proveer tal seguridad sin costo de tiempo de ejecución.

Notar que el tiempo de ejecución de la verificación de tipos requiere un tiempo de ejecuciones representativas por tipo. Así, cada variable tiene ambos un campo de valor y un campo de etiqueta. El código que desempeña el tiempo de ejecución de verificación realiza dos el campo de etiqueta, mientras la aritmética usa el valor de los campos. Con etiquetas, cada elemento dato necesita más espacio, que es, más bytes de memoria. Si una variable es almacenada en un registro, ambos su valor y una etiqueta debe ser inicilizada, leído, comparado, y escrito en tiempo de ejecución. Todas las actividades suman gustos generales a una simple operación de suma.

El tiempo de ejecución de verificación de tipos imponen un gran gasto general sobre una simple aritmética, o una conversión y una suma, con el código anidado if then else fuertemente sugiere que los operadores tales como la suma son implementados como procedimientos y tienen cada instancia de un operador ser tratado como una llamada de procedimiento. En un lenguaje que requiere tiempo de ejecución de verificación, puede fácilmente abrumar el costo de las operaciones que se realizan.

El desempeño de la inferencia y verificación de tipos en el compilador a la vez elimina esta clase de gasto general.

Generación de código objeto

El proceso de traducción directo es relativamente sencillo. Cada instrucción del código intermedio puede traducirse en una secuencia de instrucciones de máquina. n la figura 15 se muestra la entrada y la salida de la fase generador de código.

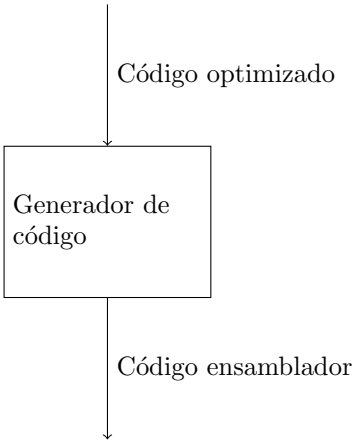

Figura 15: La entrada y la salida de la fase de generación de código.

En el listado 3 se muestra el código ensamblador que es la salida del generador de código y que es también la salida del compilador.

Listing 3: Código ensamblador.

```
1 MOVF id3, R2
2 MULF #60.0, R2
3 MOVF id2, R1
4 ADDF R2, R1
5 MOVF R1, id1
```

La tabla de símbolos

La tabla de símbolos es la que guarda la información y debe tener el formato que se muestra en el cuadro 1 .

1	Posición	$\cdots$
2	Inicial	$\cdots$
3	Velocidad	$\cdots$
4		

Cuadro 1: Sugerencia sobre la tabla de símbolos.

Bibliografía

[1] Thomas W. Parsons, Introduction to compiler construction, Computer Science Press, 1992.

[2] Ralph Grishman, Computational Linguistics An Introduction, Cambridge University Press, 1986.

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compiladores Principios, técnicas y herramientas, Addison Wesley, 1990.

[4] Torben Aegidius Mogensen, Basics of compiler design, DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF COPENHAGEN, 2009.

[5] Linda Torczon and Keith D. Cooper, Engineering a Compiler, Second Edition, Elsevier 2012.

[6] Bernard Teufel, Stephanie Schmidt, Thomas Teufel, Compiladores, conceptos fundamentales, Addison-Wesley Iberoamericana, 1995.

[7] Jean Paul Tremblay, Paul G. Sorenson, The theory and practice of compiler writing, McGraw Hill Book Company, 1985.

[8] Mark P. Jones, jacc: just another compiler compiler for Java A Reference Manual and User Guide, Department of Computer Science Engineering OGI School of Science Engineering at OHSU 20000 NW Walker Road, Beaverton, OR 97006, USA February 16, 2004.

[9] www.lcc.uma.es/ galvez/ftp/tci/TutorialYacc.pdf