Introducción

Pedro O. Pérez M., PhD.

Diseño de compiladores
Tecnológico de Monterrey

pperezm@tec.mx

07-2021

1.1 - 1.3

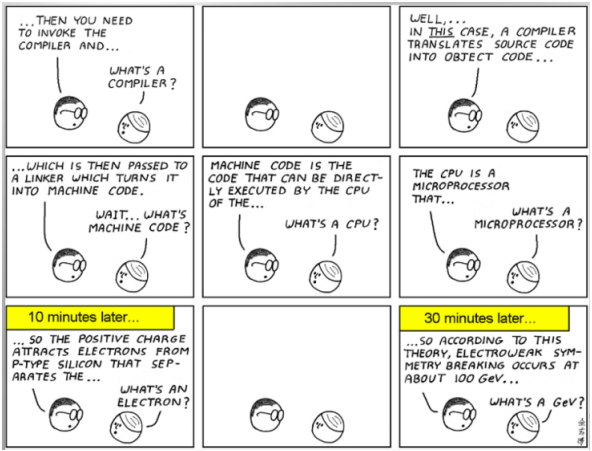
- 1.1 Procesadores de lenguajes
- 1.2 La estructura de un compilador
- 1.3 La evolución de los lenguajes de programación

1.4 - 1.6

- 1.4 La ciencia de construir un compilador
- 1.5 Aplicaciones de la tecnología de los compiladores
- 1.6 Conceptos básicos de los lenguajes de programación

1.1 Procesadores de lenguajes

¿Qué es un compilador?



Un compilador es una herramienta que traduce software escrito en un lenguaje a otro. Para realizar esta traducción, la herramienta debe ser capaz de comprender el lenguaje original (sintaxis, semántica). Además, necesita comprender el lenguaje final. Finalmente, debe tener un esquema que le permita hacer la traducción.

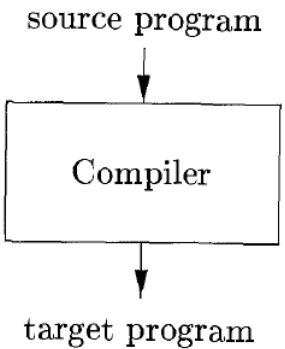


Figure 1.1: A compiler

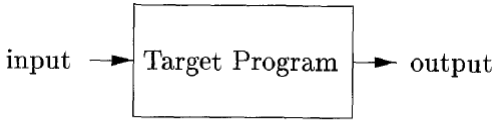


Figure 1.2: Running the target program

Un interprete es otro tipo de herramienta común. En vez de producir un programa destino (o traducción), un interprete ejecuta directamente las operaciones especificadas en el programa fuente con las entradas dadas por el usuario.

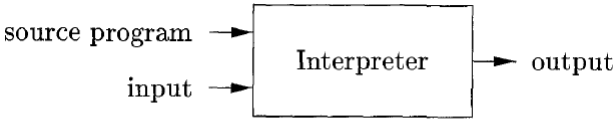


Figure 1.3: An interpreter

Un código destino producido por un compilador es usualmente más rápido que una código que está siendo interpretado. Sin embargo, un interprete puede, usualmente, dar mejor diagnóstico de errores que un compilador, porque ejecutar el código fuente instrucción por instrucción.

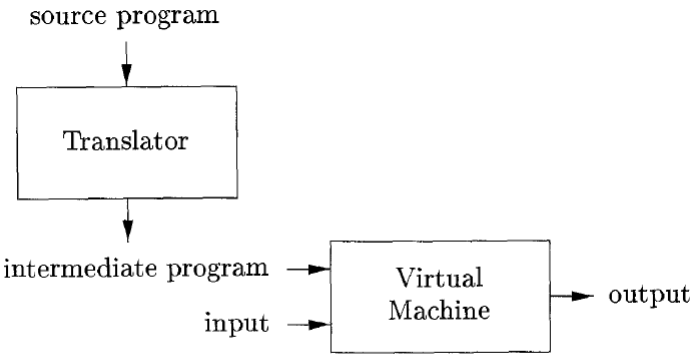


Figure 1.4: A hybrid compiler

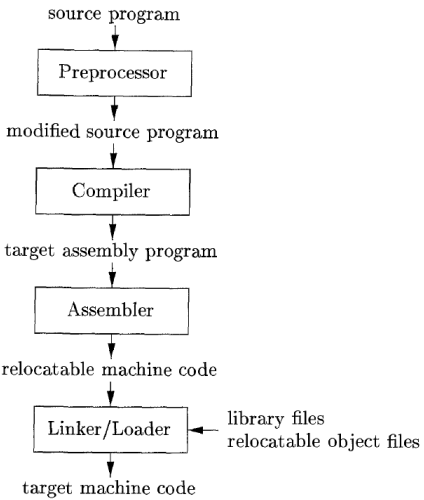
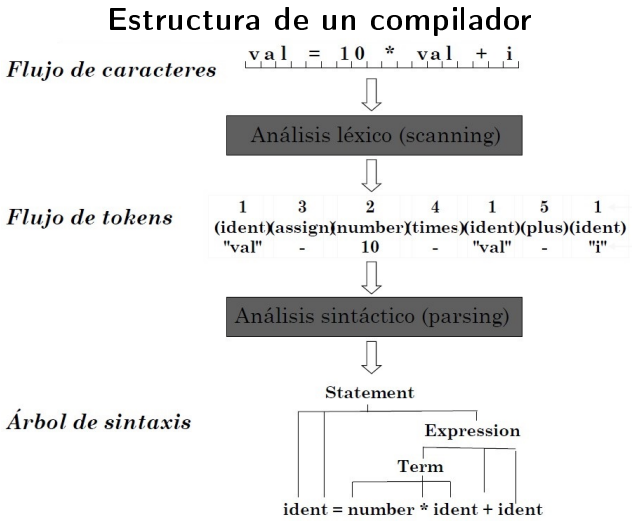
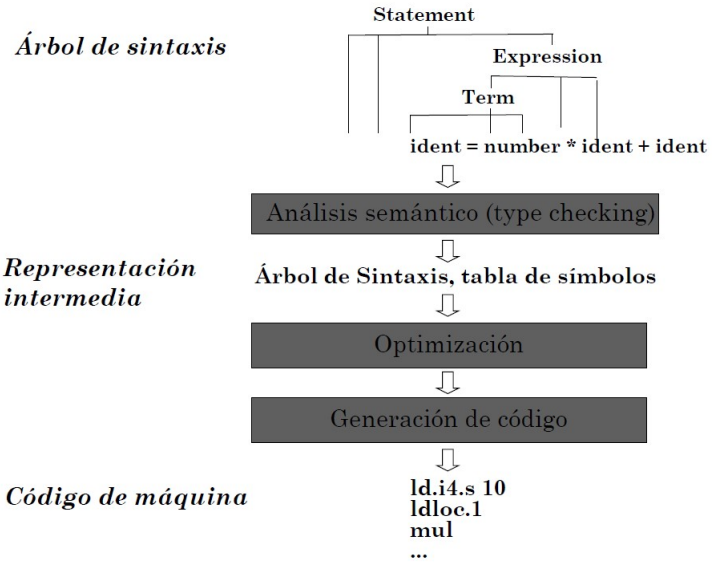


Figure 1.5: A language-processing system

1.2 La estructura de un compilador




Árbol de sintaxis
Statement
Expression
Term
ident = number * ident + ident
Análisis semántico (type checking)
Representación intermedia
Árbol de Sintaxis, tabla de símbolos
Optimización
Generación de código
Código de máquina
ld.i4.s 10
ldloc.1
mul
...

1.2.9 Herramientas de construcción de compiladores

Existen herramientas especializadas definir e implementar componentes específicos:

- *Parser generators*: Automáticamente produce analizadores léxicos a partir de una descripción gramatical de un lenguaje de programación.
- *Scanner generators*: Producen analizadores léxicos a partir de la expresiones regulares que definen los "tokens" de un lenguaje de programación.
- *Syntax-directed translation engines*: Producen colecciones de rutines para recorrer un árbol de análisis sintáctico para la generación de código intermedio.
- *Code-generator generators*: Producen generadores de códigos a partir de una colección de reglas de traducción de código intermedio a código fuente.

- *Data-flow analysis engine*: Facilitan la recuperación de información acerca de cómo los valores son transmitidos de una a otra parte del programa. El análisis de flujo de datos es una parte importante de la optimización de código.
- *Compiler-construction toolkits*: Proveen de una conjunto de rutinas para la construcción de las varias fases de un compilador.

Ejercicio 1.1

Instrucciones:

- Lee el documento que se encuentra en el ejercicio 1.1 que se encuentra en Canvas.
- Genera un breve resumen de las ideas que ahí se describen, enviálo como evidencia de esta actividad y discútelo en clase.

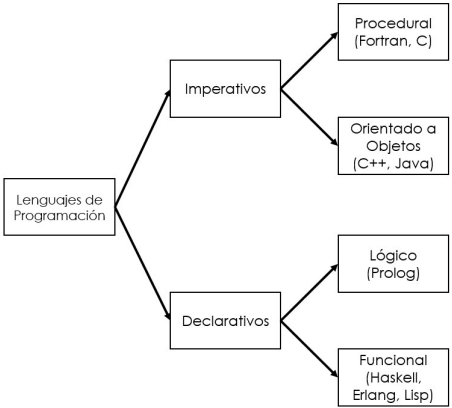
1.3 La evolución de los lenguajes de programación

Generaciones

- Primera generación:
  - Entendido directamente por las computadoras.
  - Utilizan instrucciones del procesador.
  - Dependientes del procesador.
  - En código binario.
- Segunda generación:
  - Siguen siendo dependientes del procesador.
  - Utilizan mnemónicos para representar instrucciones del procesador.
  - Más fáciles de recordar y leer.

- Tercera generación:
  - Independientes del procesador.
  - Utilizan variables y secuencias.
  - Incluyen ramas y ciclos.
- Cuarta generación:
  - Independientes del procesador.
  - Utilizan el llenado de formularios.
  - Gráficos asistidos por computadoras.
- Quinta generación:
  - Dependiente del procesador.
  - Utiliza técnicas de IA.
  - La computadora obtiene inferencias del código.

Paradigmas


Lenguajes de Programación
Imperativos
Declarativos
Procedural (Fortran, C)
Orientado a Objetos (C++, Java)
Lógico (Prolog)
Funcional (Haskell, Erlang, Lisp)

Ejercicio 1.2

Instrucciones:

- Reúnete en equipos de 3 personas.
- Discután cuáles de los siguientes términos: imperativo, declarativo, von Neumann, orientado a objetos, funcional, tercera generación, cuarta generación, scripting se pueden aplican a los siguiente lenguajes: C, C++, COBOL, Fortran, Java, Lisp, ML, Perl, Python, Visual Basic.
- Escribe un reporte con los resultados, sube tu reporte a Canvas en la sección de Tareas.

1.4 La ciencia de construir un compilador

Un buen compilador es un microcosmos del área de las ciencias computacionales: hace uso práctico de algoritmos codiciosos (asignación de recursos), técnicas de búsqueda heurística (programación de listas), algoritmos de grafos (eliminación de código muerto), programación dinámica (selección de instrucciones), autómatas finitos y push-down (escaneo y análisis), entre muchos otros temas. En otras palabras, trabajas en el diseño y programación de un compilador proporciona experiencia práctica en ingeniería de software que es muy difícil de obtener en sistemas más pequeños y menos complejos.

- Se emplean modelos matemáticos como expresiones regulares, maquinas de estados finitos, gramáticas libres de contexto.
- Optimización del código destino generado:
  - La optimización debe preservar el significado del programa compilado.
  - Debe mejorar el desempeño de muchos programas.

1.5 Aplicaciones de la tecnología de los compiladores

- Implementación de lenguajes de programación de alto nivel.
- Optimizaciones para arquitecturas computacionales:
  - Paralelismo a nivel de instrucciones.
  - Jerarquía de memorias.
- Diseño de nuevos arquitectura computacionales: Desde que los lenguajes de alto nivel son la norma, el desempeño de los sistemas computacionales está determinado no solo por su velocidad sino, también, por lo bien que los compiladores explotan sus características.
- Traducción de programas: traducción binaria (convertir código maquina a otro), hardware sintéctico (VHDL), interpretes de consultas a base de datos.

1.6 Conceptos básicos de los lenguajes de programación

- Distinción entre estático y dinámico: alcance estático/dinámico, declaración estático/dinámico.
- Ambientes y estados.
  - Enlace estático vs. dinámico del nombre a la localidad.
  - Enlace estático vs. dinámico de la localidad a los valores.

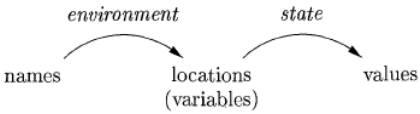


Figure 1.8: Two-stage mapping from names to values

- Alcance estático y/o bloque.
- Control de acceso explícito (ejemplo: público, privado, protegido).
- Paso de parámetros.

Ejercicio 1.3

- Indica cuál es el valor asignado a las variables `w`, `x`, `y` y `z`.

```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

(a) Code for Exercise 1.6.1

```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j;
{   int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

(b) Code for Exercise 1.6.2

- ¿Cuál el alcance de cada una de las variables declaradas?

```
#define a (x+1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n"), a; }
void main() { b(); c(); }
```