Del propósito de la materia de compiladores en la formación del ingeniero informático

JosuKa Díaz Labrador, José Mª Sáenz Ruiz de Velasco
Dpto. de Ingeniería del Software
Universidad de Deusto
Apartado 1 - 48080 Bilbao
e-mail: josuka@eside.deusto.es, jmsaenz@eside.deusto.es

Resumen

Se presenta una reflexión sobre el papel que puede jugar la materia de compiladores en el plan de estudios de la ingeniería informática. Se argumenta que, además del objetivo obvio de enseñar a construir un compilador de un lenguaje de programación, pueden extraerse otras aportaciones. Se ha identificado que una de ellas es completar la formación que hasta el momento ha adquirido el estudiante sobre la tarea de programar y su herramienta (el lenguaje de programación), y que se trata de una aportación relevante y necesaria para este propósito. En segundo lugar, se considera que la aplicación de los conocimientos de compiladores no se restringe al problema de implementar un compilador en el sentido clásico, sino que al extender su concepto, pueden también aparecer nuevas aplicaciones, que incluso se dan con más frecuencia que la original.

1. Introducción

La materia de *Compiladores* es una de las troncales de 2º ciclo, con una carga lectiva de 9 créditos, en el Plan de Estudios de Ingeniero en Informática fijado por el Ministerio de Educación y Ciencia [10]. A pesar del título («*Procesadores de lenguaje*»), la descripción que aparece a continuación («*Compiladores, Traductores e Intérpretes. Fases de Compilación. Optimización de código. Macroprocesadores*») permite suponer que se pretendía un enfoque clásico (según el texto de Aho, Sethi y Ullman [1], referencia aún obligada pese a su antigüedad), en el que el objetivo prácticamente único del compilador es procesar lenguajes de programación.

Sin embargo, a la luz de los ya más de 10 años transcurridos desde el establecimiento de dicho plan de estudios, parece necesario (si no lo es en todo momento) volver a preguntarse sobre el papel que cumple la materia de compiladores en la formación del ingeniero en informática, sobre su orientación, y sobre sus contenidos teóricos y prácticos.

Muchas de las ideas que se recogen en este trabajo se desarrollaron en el año 2000 como parte del proyecto docente del primer autor para las asignaturas de compiladores en la Universidad de Deusto. Sin embargo, hemos constatado en otros docentes preocupación por estas cuestiones, y ha sido para nosotros una grata sorpresa encontrar un trabajo muy reciente de Debray [4], que realiza un análisis similar, y llega a conclusiones en parte comunes: sus ideas se irán desgranando en las sucesivas secciones.

En la sección 2 se presentan los resultados de un estudio (no exhaustivo) que se ha realizado de la implementación de la asignatura en varias universidades españolas. En la sección siguiente se plantean las cuestiones que (además de la razón obvia de mostrar qué es y cómo se construye un compilador) pueden justificar el interés de la materia.

En la cuarta, se argumenta que la misma aporta adicionalmente una serie de conocimientos y perspectivas que completan o afianzan los que el estudiante ha adquirido en relación con el concepto de programar. Después, la sección quinta muestra que la aplicación de los conocimientos de compiladores se extiende más allá del problema concreto de construir un traductor de un lenguaje de programación, y que la clase de estos problemas afines es bastante más común (al menos desde hace unos años) que el

problema anterior. Las conclusiones recogen una redacción tentativa de los objetivos de la asignatura, a la luz de lo discutido, y se analiza su efecto sobre los contenidos.

2. Situación actual

Una revisión de los planes de estudios de algo más de la mitad (30 sobre 44) de las universidades españolas que ofrecen la asignatura (como troncal de 2º ciclo), permite alcanzar varias conclusiones.

1. Aunque existen algunos casos, no se suele aumentar la carga lectiva fijada por la troncalidad (23 de las 30 analizadas mantienen los 9 créditos, y solo en un caso se llega a 15 créditos). Esto puede explicarse porque muchos centros deciden seguir la troncalidad, o porque el segundo ciclo en que se encuentra la materia no permite fácilmente la adición de créditos extra, pero también porque no se percibe necesidad de una mayor carga. Esto contrasta, sin embargo, con otras materias, por ejemplo, sistemas operativos (aunque es de primer ciclo), en que solo dos universidades siguen los 6 créditos troncales, siendo al menos 17 los centros analizados que dan más de 12 créditos, entre troncales y obligatorias.
2. Salvo muy contadas excepciones, la implementación de la materia sigue el enfoque que hemos llamado clásico, de compilación de los lenguajes de programación. Entre los centros de los que se ha podido recabar información, solo se han encontrado dos aspectos afines adicionales: procesamiento de lenguajes naturales (Camacho [3]) y lenguajes de la tecnología *web* (Barrutieta [2]).
3. Existe un enfoque de diseño prácticamente unánime en la presentación de la materia: el objetivo final es que el estudiante *sea capaz de construir* un compilador para un lenguaje de programación. Este enfoque, que se contrapone a una perspectiva analítica o descriptiva de la asignatura, es totalmente adecuado en la medida que se están formando ingenieros.
4. También es casi unánime incluir unas prácticas más o menos extensas, en que se hace uso de las conocidas herramientas de generación de compiladores en sus diversas versiones, algunas de las cuales han sido mostradas en ediciones anteriores de las JENUI (por ejemplo Mascaró y Orrell [9]).

Sin embargo, es interesante añadir algo más (impresiones subjetivas esta vez) a los datos precedentes. Pensamos que los estudiantes (lamentablemente, también algunos docentes) observan esta asignatura como una especie de "isla" en su formación. La idea de fondo es similar a la que se tiene respecto a la teoría de autómatas y lenguajes formales o la metodología de la programación: todo el mundo tiene claro que "parecen necesarias", pero pocos (quitando a los propios docentes y honrosas excepciones) les conceden mayor relevancia. La materia de compiladores parece verse como necesaria por el mero hecho de ser el sistema informático que permite en la práctica la tarea de programar, pero no se aporta mucho más acerca de su papel, ni se pone en relación con otras materias anteriores o posteriores de los estudios.

Por lo tanto, pensamos que resulta preciso profundizar en la razón de ser de los compiladores, no con el objetivo de una posible actualización de la materia (que creemos que en principio no se necesita), sino más bien para ser capaces de motivar a nuestros estudiantes, de situar la materia en el edificio de conocimientos que están construyendo, y prepararles para el aprovechamiento futuro de los mismos.

3. Interés de los compiladores

A primera vista, parece obvio pensar que las razones de que haya que enseñar compiladores a los estudiantes de informática son dos: primero, el compilador es un sistema informático suficientemente importante como para que sea conocido en detalle, y segundo, una de las hipotéticas tareas del profesional informático puede ser construir un compilador.

Adoptar un enfoque de diseño en la presentación de la asignatura es entonces doblemente adecuado, pues permite contestar el *qué* y el *cómo*, que son los objetivos planteados antes.

Ahora bien, si profundizamos en las razones anteriores, se plantean inmediatamente dos cuestiones. Primera: ¿por qué es importante lo que debe conocerse del compilador? Es decir, ya sabemos que casi todas las materias aportan una formación, un "poso de conocimientos", que justifican por sí mismos su interés, pero ¿podemos ser más explícitos? ¿Qué aportan los procesadores

de lenguajes concretamente a *la formación que hasta ahora* ha tenido el estudiante?

La segunda cuestión, expresada llanamente, es la siguiente: en verdad, ¿quién se dedica (sin necesidad de restringirse a España, incluso) a construir compiladores? Es decir, ¿en algún momento de su vida profesional van nuestros alumnos a poner en juego esos conocimientos adquiridos? Este argumento "utilitarista" suele repelernos a los académicos (nosotros al menos compartimos las ideas de Giner [6]), y desde luego, no es siquiera necesaria una respuesta si somos capaces de contestar adecuadamente a la primera pregunta. Pero hemos de recordar que estamos formando ingenieros (quizás también a algunos científicos), por lo que esta cuestión, bien entendida, se convierte en relevante.

Además, identificamos en ella una de las posibles razones por las que el estudiante no se encuentra suficientemente motivado ante asignaturas como compiladores (u otras similares como sistemas operativos). El alumno percibe (a partir de su ya amplia experiencia como usuario de esos sistemas) que son enormemente complejos, que muy pocos se dedican a ellos, y por tanto, no se conciben a sí mismos como posibles futuros participantes en su desarrollo.

Debray [4] hace exactamente este análisis (como muchos otros trabajos de mejora docente en compiladores, como el de Vegdahl [13], por citar uno de los más recientes que hemos conocido), y veremos que sus conclusiones son ciertamente similares a las nuestras.

Las dos cuestiones planteadas se desarrollan en sendas secciones a continuación.

4. Importancia del conocimiento de los compiladores

Un compilador es, en sentido estricto, un sistema intrínsecamente ligado a la programación, que a su vez, es una de las tareas fundamentales del informático. Por lo tanto, el conocimiento de los compiladores influirá en la maestría con que se lleve a cabo esta tarea.

En efecto, durante el primer ciclo de la carrera de informática se dedica un considerable número de créditos a los aspectos teóricos y prácticos relacionados con el concepto de programar: fundamentos de la programación y de los lenguajes, metodología, paradigmas, estructuras de datos y algoritmia, acompañados del conocimiento de (normalmente) varios lenguajes de programación. Se supone que el estudiante que termina el primer ciclo (o la carrera técnica, por poner el caso) es diestro al menos en programar y usar las herramientas de programación (entornos de desarrollo y, precisamente, compiladores).

Sin embargo, nuestra conclusión a este respecto es que todavía *falta* algo: un cierto conjunto de conocimientos que han de añadirse a los ya disponibles para que la maestría en programar pueda considerarse completa.

4.1. Comprensión de la definición de los lenguajes

Como primera aportación, destacamos la *sistematización* de la definición de los lenguajes de programación. El uso de modelos matemáticos formales (autómatas, expresiones regulares, gramáticas independientes del contexto, gramáticas de atributos, etc.) para definir rigurosamente el léxico, la sintaxis y la semántica de los lenguajes de programación aporta un sustento único al que afianzarse a la hora de comprender el lenguaje de programación.

Digamos que hasta ahora el conocimiento de los lenguajes era "intuitivo": el estudiante domina la escritura en ellos como un niño aprende a hablar, pero carece del edificio necesario para comprender la estructura de la herramienta. Después de ver cómo los modelos matemáticos formales ayudan a expresar la definición de un lenguaje, el futuro informático estará preparado para comprenderlo de forma organizada.

Esta idea es importante en relación con otro aspecto, que es la evolución de los lenguajes de programación. Hoy enseñamos Pascal, C, Ada, Modula, C++, Java, Eiffel, Lisp, Scheme, ML, Haskell... pero el día de mañana estas serán "lenguas muertas", y surgirán nuevos lenguajes, por lo que es importante que los conocimientos que transmitimos no sean caducos, y puedan aplicarse igualmente en los retos venideros. Cuando un ingeniero informático se enfrente a la tarea de aprender un nuevo lenguaje, el sustento de los compiladores le ayudará a sistematizar la mirada (elementos léxicos, estructuras sintácticas, aspectos semánticos), comprender rápidamente sus entidades (con ayuda de los modelos formales

mencionados), y fijarse en los detalles que puedan ser importantes.

Hay incluso una tercera ventaja, que es el mejor aprovechamiento de los mensajes de error. Los estudiantes han tenido ya una dilatada experiencia de uso de los compiladores, y probablemente están unánimemente de acuerdo en que los mensajes de error resultan muchas veces crípticos, cuando no equivocados, y no acaban de comprender la razón de que se señale cierto error, cuando claramente (según su percepción) es otro. El conocimiento de las fases de análisis, su funcionamiento y la gestión de errores resulta determinante para que se puedan comprender mejor los mensajes que emite el compilador, con el resultado indudable de favorecer el ciclo de composición de programas.

4.2. Comprensión de la implementación de los lenguajes de programación

En segundo lugar, encontramos una serie de aspectos relacionados con la generación de código. Todos ellos se dirigen a que el estudiante sea consciente de qué es un programa traducido, cuando se ejecuta (objetivo final de programar), y cómo puede mejorarse su calidad.

De nuevo, o quizá más que en el caso anterior, los estudiantes tienen una serie de ideas más bien difusas sobre el asunto: alcanzan a saber como mucho que hay una pila (que tiene direcciones de retorno) y un montículo (para la memoria dinámica), que se dice que "los datos están en memoria" (sin ser plenamente conscientes de lo que esto significa o relacionarlo con lo anterior), que se usan los registros de la máquina, que se pueden aplicar optimaciones, etc.

Mientras, es indudable que solo un conocimiento pleno de estos aspectos puede ayudar a explicar lo bueno o lo malo que es un programa (queremos decir, una vez que se han asegurado todas las cuestiones algorítmicas, metodológicas y de estructuras de datos involucradas, evidentemente).

La estructura del programa objeto, el uso de la pila y el montículo, el concepto de registro de activación, las secuencias de llamada y retorno, la asignación estática de datos locales en la pila, la importancia de la existencia o no de anidamiento de bloques en el lenguaje, la inicialización de datos globales y locales, el paso de parámetros, las estrategias de asignación de registros para la evaluación de expresiones y sentencias, las optimaciones que pueden practicarse, etc. forman un bloque de aspectos ineludibles para comprender finalmente los programas.

4.3. Comentarios adicionales

En definitiva, nuestra postura es que la enseñanza de los compiladores aporta al futuro informático una serie de conocimientos que afianzan, o terminan de fijar, una de las tareas que se supone que este ha de realizar, como es la de programar, y pensamos que ello viene dado por dos aspectos principales: primero, la sistematización o estructuración del concepto de lenguaje de programación, y segundo, la comprensión de la implementación de los mismos.

Ambos aspectos pueden hacerse aparentes sin necesidad de cambiar el temario o enfoque tradicionales de la materia. El primer aspecto puede surgir al presentar las fases del *front-end* del compilador, mientras que al segundo se le suele dedicar una lección (capítulo 7 de Aho, Sethi y Ullman [1]).

Una de nuestras estrategias docentes al respecto es plantear durante el curso múltiples cuestiones sobre los lenguajes de programación conocidos por los estudiantes, del tipo:

- ¿es `integer` una palabra reservada de Pascal?
- ¿cuesta lo mismo inicializar variables locales y variables globales en C?

cuyas respuestas (negativas en estos casos) sorprenden a muchos, lo que les lleva a reflexionar y darse cuenta que es precisa otra visión (a la vez sistemática y de implementación) para comprender *plenamente* la programación y los lenguajes que la permiten.

Hay que tener en cuenta que en algunos planes de estudio se encuentra (normalmente en tercer curso) una asignatura que, entre otras cosas, trata justamente de algunos de estos temas (principalmente el segundo, esto es, la implementación de los lenguajes): lo que podría llamarse quizá "lingüística de programación", tal como se refleja, entre muchos otros, en los textos de Sethi [12] o MacLennan [8].

Su presencia en esos planes de estudios responde indudablemente a los motivos

expresados antes respecto a compiladores, y se supone que, entonces, en dichos planes esta última asignatura se dedica a profundizar más en los aspectos complejos propios del problema.

5. Aplicación de las técnicas de compilación

Recordemos la segunda cuestión planteada al final de la sección 3: ¿tiene utilidad práctica, en cuanto a desarrollo de la profesión, el conocimiento de los compiladores? Desde luego, puede decirse que un porcentaje realmente pequeño de los informáticos construye alguna vez en su vida un compilador (en sentido estricto, es decir, para procesar un lenguaje de programación), pero de la misma forma, aún es menor el número de quienes desarrollan un sistema operativo o una base de datos, así que este argumento debe ser rebatido a quienes (normalmente alumnos poco motivados) lo exponen.

Nos referimos a este aspecto incluso recordando que normalmente enseñamos a *diseñar e implementar* un compilador, es decir, se completa la cuestión anterior con esta otra: ¿por qué es importante este enfoque de diseño?

Debray [4], ante este problema, sostiene una idea básica que apoyamos plenamente: «*the principles, techniques, and tools discussed in compiler design courses are nevertheless applicable to a wide variety of situations that would generally not be considered to be compiler design*». Sin embargo, reconsideramos este argumento a través de dos facetas: primero, los métodos y herramientas de compiladores resultan relevantes en sí mismos como nuevos utensilios para la resolución de problemas (tarea final de la informática), y segundo, su aplicación a problemas que a primera vista no parecen de compilación es más habitual de lo que parece.

5.1. Nuevos métodos y herramientas de solución de problemas

La informática busca la solución de problemas. Los estudiantes llegan al segundo ciclo de la carrera con un notable repertorio de teorías, métodos y herramientas dirigidos a este propósito, pero es evidente que aún existen muchas más. Nuestra postura es que el diseño de compiladores aporta un conjunto de ellas que son relevantes y de aplicación en un amplio rango de problemas: se ejercitarán en la asignatura de compiladores bajo el objetivo concreto de construir un compilador, pero debemos recalcar a los alumnos que su aplicación excede ampliamente este problema concreto, y que esa es precisamente una de las razones por las que resulta importante la materia.

En primer lugar, encontramos los modelos matemáticos formales típicos (expresiones regulares, autómatas finitos y con pila, máquinas de Moore y de Mealy, gramáticas independientes del contexto). No suele ser habitual, creemos, que se dedique mucho tiempo, en la asignatura correspondiente del primer ciclo, a la implementación práctica de estos modelos, por lo que hacerlo en compiladores (al explicar las fases de análisis léxico y sintáctico) aporta ya de por sí una nueva herramienta de trabajo al informático.

En segundo lugar, aparecen los algoritmos propios de análisis sintáctico (sobre todo descendente, factible sin disponer de herramienta de generación), y los mecanismos de procesamiento semántico (esquemas de traducción y gramáticas de atributos), que se convierten en útiles adicionales.

Finalmente, pero como aspecto más importante, disponemos de los archiconocidos programas de generación de compiladores como lex y yacc (o sus variantes; ver Levine *et al.* [7]), ANTLR/PCCTS (ver Parr y Quong [11]), y similares. Nosotros, una vez más, los enseñamos en el contexto concreto de implementación de compiladores, pero nunca se insistirá lo suficiente en la capacidad que tienen dichos programas para resolver problemas cuya "apariencia" no es la de un compilador.

Hay un aspecto que muchas veces se olvida al plantear la bondad de estas herramientas de generación de compiladores. Desde un punto de vista reduccionista, no hacen realmente nada nuevo aparte de implementar un autómata o una gramática (de la misma forma que los programadores de verdad no necesitan más que el ensamblador). Sin embargo, la *rapidez* con la que podemos resolver cada problema y, sobre todo, la *confianza* (casi) plena que podemos depositar en los programas que generan hacen que las consideremos herramientas insustituibles entre los utensilios del informático.

5.2. Problemas que no son de compilación

Todos los textos de compiladores hablan de las aplicaciones de las técnicas de compilación, pero la inmensa mayoría está relacionada de una u otra forma con lenguajes de programación. Se citan algunos otros ejemplos de lenguajes que tienen otros propósitos, principalmente TEX y similares (aunque es difícil discernir si TEX *no* es un lenguaje de programación), pero poco más. Además, como la presentación de la materia de compiladores se realiza tomando el procesamiento de los lenguajes de programación como ejemplo único, la mayor parte de los estudiantes saca la conclusión de que los conocimientos de la asignatura solo se aplicarán si alguien les manda construir un compilador de un lenguaje de programación.

Como dijimos al principio, es claro que este problema concreto se presenta a un número bien reducido de profesionales. Sin embargo, pensamos que existen más problemas en que pueden aplicarse las técnicas de compilación.

De nuevo, Debray [4] presenta dos ejemplos concretos de problemas (una descripción textual de grafos que genera imágenes Postscript y un conversor de LATEX a HTML) que no son estrictamente de compilación, pero que se resuelven claramente con las técnicas de compilación, y en particular, con las herramientas mencionadas.

Sin embargo, Debray aboga como conclusión por «*consider compilers as just one instance of translators, broadly, from (almost) any arbitrary source language to (almost) any arbitrary target language*». Nosotros planteamos que esta generalización de la compilación como traducción, primero, siendo relevante, es casi un lugar común en la presentación inicial de la materia (véase cualquier apunte de la asignatura, como el de Eguíluz y Díaz [5]), y segundo, se encuentra todavía demasiado "cerca" del concepto original, apartada entonces de muchas de sus aplicaciones.

Nuestra postura es que la condición para plantearse una posible aplicación de las técnicas de compilación a un problema es únicamente la intervención de un *lenguaje (formal) artificial* en su especificación (normalmente como datos de entrada del problema), y que esta no necesita estar elaborada específicamente como una traducción (aunque en el fondo lo sea).

El ejemplo típico que solemos comentar es el navegador de Internet, en su faceta de presentación textual y/o gráfica de contenido HTML. A pesar de que nosotros identificamos formalmente que este programa es un *traductor* de HTML a secuencias de posicionamiento de elementos gráficos en la pantalla, es difícil a primera vista hacer esta interpretación del problema. Realmente, sin embargo, bastaría con darse cuenta que interviene HTML como entrada: desde el momento en que es un lenguaje artificial suficientemente rico, debería sospecharse que se pueden aplicar los conocimientos de compiladores, y ello sea lo que sea lo que se quiere hacer con esa entrada.

Esta idea, además, cobra gran relevancia desde hace pocos años, no solo por el advenimiento de Internet y el conjunto de lenguajes asociados, sino en la medida que la información se almacena cada vez menos en forma binaria, y cada vez más de forma textual. El siguiente ejemplo se nos presentó hace pocos meses en un contexto no relacionado con nuestra docencia de compiladores.

Imaginemos que nos proporcionan una (enorme) base de datos supuestamente codificada en XML, con la salvedad de que los datos (obtenidos de una fuente no estructurada) no han sido bien codificados, es decir, el documento XML resultante no está bien formado. Concretamente, entre otras dificultades, ciertas etiquetas no tienen su cierre y otras no están correctamente anidadas, como se ve en el siguiente ejemplo:

```
<a> xxx <b> yyy </a> zzz </b>
```

Lamentablemente, ocurre que no es posible modificar el programa de codificación para obtener un documento bien formado (o aunque se pudiera, la fuente de la información original no estructurada ya no se encuentra disponible), es decir, solo puede tratarse el documento pseudo-XML original para corregir los errores.

Este ejemplo es, creemos, paradigmático del *procesamiento de información textual* no relacionado *a priori* con los compiladores, ni siquiera desde la perspectiva generalizada de traducción: «oye, es que tengo un documento

XML mal formado que quiero corregir» es lo que nos plantearon. De nuevo, la mera aparición de XML como entrada es lo que debe encender la bombilla de las técnicas de compilación.

Pero incluso el ejemplo nos parece interesante porque muestra que no es necesaria siquiera una traducción estricta:

- entrada: XML mal formado;
- salida: XML bien formado;
- método: hacer el *parser* de XML mal formado y generar código de XML bien formado.

como solución.

Nuestra estrategia fue analizar someramente las modificaciones necesarias, y observar que no requerían la construcción de un *parser* completo de XML mal formado (lo que llevaría un considerable esfuerzo, incluso con técnicas de compilación). Al contrario, las partes problemáticas seguían patrones muy claros, y se podían identificar por el contexto. Con ello, nos fue suficiente recurrir a lex, particularmente haciendo uso de los estados de análisis que permite esta herramienta.

Por tanto, el ejemplo muestra que ni en la especificación ni en la solución resulta muy útil un enfoque de traducción, por mucho que lo que hagamos en el fondo con el programa lex no sea más que eso exactamente. El enfoque que funciona es el de procesamiento: si la entrada es un lenguaje formal, habrá que procesarla (para lo que sea), y ello lleva a considerar las técnicas o herramientas de compilación como posible vehículo de la solución.

5.3. Comentarios adicionales

Esta perspectiva hace que cobren nueva importancia los aspectos del *front-end* del compilador, y en particular, las herramientas de generación de compiladores, percibidas más bien como programas de ayuda al diseño rápido, efectivo y seguro de soluciones para problemas de procesamiento de información. Ello es una razón más para insistir como docentes en la parte práctica de la materia, tal como ya se ha visto (en la sección 2) que se lleva a cabo en la mayor parte de los centros. Quizá sería conveniente incluir como práctica adicional algún pequeño problema del tipo presentado antes, además de los consabidos lenguajes de programación simples.

Por otro lado, es cierto que bajo este enfoque resultan de menor aplicabilidad los aspectos del *back-end* del compilador clásico (generación de código intermedio y objeto, optimación). Ello no justifica en ningún modo (al menos debido al interés planteado en la sección 4) su posible marginación en el temario.

Ahora bien, según la importancia que cada docente quiera conceder a las ideas aquí expuestas, pueden considerarse dos presentaciones alternativas de la materia de compiladores a este respecto (nuestro interés sería realmente no tener que perder ningún aspecto, pero ya se ha visto lo difícil que resulta superar el límite de los 9 créditos):

- la presentación tradicional, en que prevalece el interés de la implementación de los lenguajes de programación, con fuerte profundización en los aspectos señalados del *back-end*;
- la presentación alternativa, en que prevalece la aplicación a problemas de procesamiento de información textual, más sencilla (descriptiva y no tanto sintética) en los aspectos del *back-end*, y focalizada en la resolución de problemas como los mostrados.

6. Conclusión: objetivos de la materia de compiladores

La discusión precedente nos permite elaborar de una forma profunda los objetivos de la asignatura de compiladores.

- El objetivo genuino (y preponderante) de la materia es ciertamente *ser capaz de diseñar y desarrollar un compilador* de un lenguaje de programación: esto incluye la teoría, métodos e implementación de cada fase del compilador, y en particular, las herramientas de generación de compiladores.
- El segundo objetivo es *afianzar la tarea de programar*, es decir, añadir una serie de perspectivas y conocimientos (quizá no fundamentales, pero sí relevantes y necesarios) al edificio que ya tiene el estudiante. Se enfatizan dos factores: primero, la sistematización de la definición de los lenguajes de programación (y artificiales), y segundo, la comprensión de los modelos de

implementación de los lenguajes de programación.

- El tercer objetivo es *ser capaz de identificar y resolver problemas que no son de compilación* (de lenguajes de programación). Aquí, será importante, primero, comprender que la aplicación de los conocimientos de compiladores incluye el concepto general de lenguaje formal artificial, segundo, ser capaz de identificar problemas que parecen alejados de la compilación como problemas de procesamiento de lenguajes, y tercero, comprender que los algoritmos y las herramientas de generación de compiladores se convierten entonces en utensilios relevantes para el profesional informático.

Se ha visto que, dados estos objetivos, los contenidos de la asignatura (tal como se explican de forma casi unánime en nuestras universidades) no tienen que experimentar ninguna modificación significativa, en la medida que siguen siendo adecuados al primero y principal, y que mediante sencillas técnicas docentes se puede asegurar el cumplimiento de los otros dos. En todo caso, si se desea una orientación más fuerte al tercer objetivo, se puede simplificar ligeramente la explicación de la fase de generación de código.

Referencias

[1] Aho, A.V.; Sethi, R.; Ullman, J.D. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986 (traducción al castellano: *Compiladores. Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, 1990)

[2] Barrutieta Anduiza, G. "Procesadores de lenguaje". Escuela Politécnica Superior, Universidad de Mondragón, sin fecha (`http://www.mondragon.edu/bin/tusestudios/pdfs/asignaturas/IF2403.pdf`).

[3] Camacho Fernández, D. "Procesadores de Lenguaje II (13-10830)". Universidad Carlos III de Madrid, revisión 04/04/2002 (`http://www.uc3m.es/uc3m/gral/ES/ESCU/1310830.html`).

[4] Debray, S. "Making Compiler Design Relevant for Students who will (Most Likely) Never Design a Compiler". *Proceedings of the 33rd SIGCSE technical symposium on Computer Science Education*, pp. 341-345, Covington, Kentucky, EE.UU., 2002.

[5] Eguíluz Morán, A.; Díaz Labrador, J. *Teoría de compiladores. Primer Parcial.* Universidad de Deusto, 1995.

[6] Giner Sanjulián, S. "La Universidad y la falacia utilitarista". *Conferencia: Los objetivos de la Universidad ante el nuevo siglo*, Universidad de Salamanca, 17 y 18 de noviembre de 1997 (`http://www.crue.org/pginersa.htm`)

[7] Levine, J.R.; Mason, T.; Brown, D. *lex & yacc (second edition)*. O'Reilly & Associates, 1992.

[8] MacLennan, B.J. *Principles of Programming Languages. Design, Evaluation, and Implementation (second edition)*. Harcourt Brace Jovanovich College Publishers, 1987.

[9] Mascaró Portells, M.; Orell Más, C. "Entorno de programación Java para la docencia y desarrollo de procesadores de lenguajes". En José Miró Julià, editor, *Actas de las VII Jornadas de Enseñanza Universitaria de Informática, Jenui 2001*, pp. 204-209, Palma de Mallorca, julio 2001.

[10] Ministerio de Educación Cultura y Deporte *Troncales de Ingeniero en Informática.* 1990, revisión 06/05/2002 (`http://www.mec.es/consejou/titulos/troncal/inginfor.html`).

[11] Parr, T.J.; Quong, R.W. "ANTLR: A Predicated-LL(k) Parser Generator". *Software Practice & Experience*, 25, 7, 1995, pp. 789-810.

[12] Sethi, R. *Programming Languages. Concepts and Constructs.* Addison-Wesley, 1989 (traducción al castellano: *Lenguajes de programación. Conceptos y constructores*. Addison-Wesley Iberoamericana, 1992).

[13] Vegdahl, S.R. "Using visualization tools to teach compiler design". *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, pp. 72-83, Beaverton, Oregon, EE.UU., 2000.