View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Servicio de Difusión de la Creación Intelectual

IBURG: su aplicación para la generación de generadores de código en un proyecto de compiladores

Marcelo Arroyo
Univ. Nac. de Río Cuarto,
marroyo@exa.unrc.edu.ar

Jorge Aguirre
Univ. Nac. de Río Cuarto. UBA
email:jaguirre@dc.uba.ar

Palabras clave: generación de código, generadores de generadores de código, programación dinámica, reconocimiento de patrones sobre árboles, compiladores.

Resumen

En numerosas carreras de Ciencias de Computación se incluye la realización de un proyecto de diseño e implementación de un compilador. La etapa de generación de código, por su complejidad y por la extensión global del proyecto, suele simplificarse adoptando como lenguaje objeto el mismo lenguaje en que se realiza el desarrollo o el de alguna máquina abstracta definida ad hoc. Sin embargo desarrollar esta etapa en condiciones reales, para una arquitectura existente, es de gran valor formativo y quizás la mejor forma de enfrentar al alumno con la práctica de la programación a bajo nivel. En este trabajo se discute la aplicación de las técnicas usadas en la construcción de generadores de código y el uso de una herramienta **iburg,** cuya utilización permitió, generar código assembler optimizante para el ambiente linux como última etapa de un compilador Mini c. Trabajo realizado en una asignatura cuatrimestral.

Generalmente un compilador genera una representación intermedia del programa fuente para realizar las operaciones requeridas en la generación de código optimizante. Esto permite además abstraerse de la plataforma (hardware y sistema operativo) sobre el cual se desea generar el código objeto, permitiendo desarrollar un front-end portable. El front_end utiliza un (o viarios) back_end que se encarga de la generación de código final para una plataforma determinada.

El código intermedio generado, generalmente tiene la forma de árbol, ya que se asemeja a la forma del árbol de parsing construido por el reconocedor sintáctico (parser). Para producir código objeto a partir de esta representación es conveniente utilizar alguna herramienta que permita realizar un reconocimiento de patrones (subárboles) que se relacionan con las entidades semánticas mínimas y seleccionar las instrucciones de máquina más apropiadas que las implementen, esto es, secuencias de instrucciones que tengan el menor costo (según un criterio de uso de recursos preestablecido).

Un recorrido posible para un árbol no necesariamente es el que lleva a producir código de menor costo (en términos de uso de recursos), por lo cual es necesario evaluar todos los posibles recorridos para determinar el óptimo.

En este artículo se analizará la herramienta, ***iburg***, que toma como entrada una gramática libre de contexto que describe el árbol de representación elegido (aumentada con información de costos) y genera como salida un reconocedor de patrones (parser o BURS authomata) sobre el árbol. El reconocedor generado realiza una cobertura óptima en término de los costos asociados para un nodo del árbol dado como entrada.

La determinación de recorridos (covers) se realiza en el momento en que se construye el autómata BURS (Bottom-Up Rewrite System) utilizando programación dinámica. Esto permite que el autómata generado sea extremadamente rápido y simple.

Se ven ejemplos de aplicaciones para selección de instrucciones, inferencia de tipos y simplificaciones de árboles y se describe la experiencia realizada.

1 Introducción

En este trabajo se describe el uso de una herramienta utilizada para la generación de código en el proceso de enseñanza aprendizaje en la asignatura Compiladores, de la Licenciatura en Ciencias de la Computación en la Universidad Nacional de Río Cuarto.
La asignatura Compiladores esta organizada como un taller o laboratorio cuyos objetivos son los siguientes:

- Capacidad de aplicar los conocimientos adquiridos hasta el momento en la totalidad de las asignaturas cursadas hasta el momento y específicamente los adquiridos en Autómatas y Lenguajes a la construcción de soluciones concretas en el campo del diseño e implementación de compiladores.
- Capacidad autónoma de adquirir destreza en el uso de nuevos sistemas de desarrollo de software.
- Habilidad en el manejo del ambiente de desarrollo UNIX.
- Habilidad en el uso del lenguaje C.
- Capacidad de abordar el diseño de sistemas complejos.
- Capacidad de modularizar un proyecto y de trabajar en grupo.
- Habilidad en el uso de herramientas para la construcción de compiladores y traductores.
- Capacidad de elaboración de documentación, tanto técnica como de uso.
- Capacidad de elaborar y respetar cronogramas de trabajo.

Un compilador está compuesto de diferentes subsistemas claramente definidos. Entre ellos se tienen el analizador lexicográfico (scanner), el analizador sintáctico (parser), y el generador de código. Seguramente existen otros módulos de soporte como son manejadores de tablas de símbolos, analizador semántico, etc. Este curso sigue la modalidad de un taller, durante el cual se desarrolla un compilador completo para algún lenguaje determinado. La teoría necesaria para la construcción de compiladores se dicta en una asignatura previa denominada Autómatas y Lenguajes. Se utilizan herramientas como lex y yacc para generar los analizadores lexicográficos y sintácticos respectivamente. En el último curso se introdujo una herramienta adicional a las ya mencionadas para aplicar en la etapa de generación de código. Esta herramienta se denomina *iburg*. En este trabajo se describen sus conceptos, su aplicación y los resultados de la experiencia.

El lenguaje elegido a compilar fue ***C—***extendido, subconjunto de C que contiene dos tipos de datos (enteros y reales en punto flotante) y consta de una única función (la cual puede ser invocada recursivamente). El objetivo fue generar código assembly para Linux.

2 Generación de código y representaciones intermedias

Generalmente se genera una representación intermedia del programa fuente, lo cual permite abstraerse de la máquina destino sobre el cual se genera el código final, permitiendo una mayor portabilidad del compilador a otros sistemas. Una de las representaciones intermedias más utilizadas consiste en una estructura de árbol, esta es particularmente adecuada para el uso de técnicas de generación de código optimizante. Esta forma fue la elegida para el desarrollo del compilador.

Cada subárbol de la representación intermedia está asociado a una operación o instrucción de máquina (la cual puede ser virtual). Ciertos subárboles o patrones representan operaciones.

La generación de código se realiza recorriendo el árbol y generado código a medida que se van reconociendo los patrones correspondientes a cada operación, la forma de recorrido es elegida con criterios optmizadores del código generado.

Cada operación o patrón tiene un costo asociado, generalmente con respecto al uso de recursos (registros o ciclos de reloj) requeridos. El problema principal radica en encontrar una cobertura del árbol por un conjunto de patrones (operaciones) cuyo costo sea mínimo. Esto soluciona en forma directa el problema de selección de instrucciones (óptimas) para un árbol dado.

La figura 1 muestra una especificación de patrones que representan instrucciones de máquina.

Patrón #	Patrón	Costo
1	objetivo → reg	0
2	reg → Reg	0
3	reg → Entero	1
4	reg → Carga ↓ dir	2
5	reg → Suma ↙ ↘ reg reg	2
6	dir → reg	0
7	dir → Entero	0
8	dir → Suma ↙ ↘ reg Entero	0

Figura 1: Ejemplo de patrones de instrucciones de máquina

En la figura 2 se describen dos coberturas del mismo árbol. La cobertura de la figura 2.a tiene un costo de 7 (la suma de los costos de los patrones #2, #3, #5, #6, #4, #4), mientras que la cobertura de la figura 2.b tiene un costo de 4 (suma de costos de los patrones #2, #8, #4, #4).

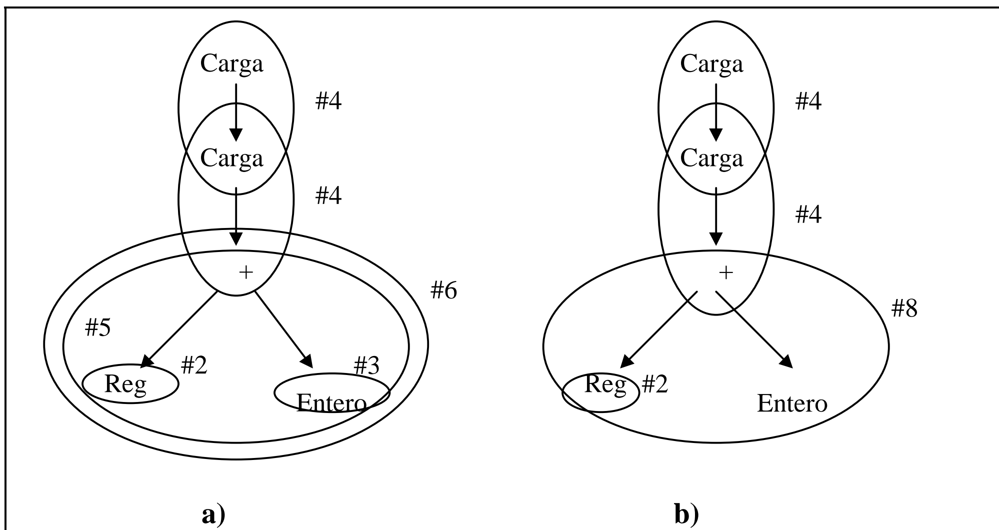

Figura 2: Dos posibles coberturas del mismo árbol

De lo anterior se deduce que si se cubre el árbol como en la figura 2.b se puede generar código más eficiente (instrucciones de menor costo) que si se cubre como en la figura 2.a.

3 Reconocimiento de patrones en árboles y programación dinámica

El problema de encontrar una cobertura mínima del árbol es un problema de optimización combinatoria y puede ser resuelto combinando reconocimiento de patrones en árboles (tree patern matching) y programación dinámica. El problema puede enunciarse como:

$$P: \text{minimizar } f(x), \text{ sujeto a } x \in S$$

donde S es un conjunto de soluciones factibles que satisfacen las restricciones impuestas.
La función $f: S \rightarrow Z$, donde Z es el conjunto de los números enteros y f la función objetivo. Una solución factible $x \in S$ es óptima si ninguna otra solución factible y satisface $f(y) < f(x)$. El conjunto S puede representarse como un conjunto de estados. Una decisión d aplicada a un estado $q \in S$ induce una transición a un estado $q' \in S$. Cada transición tiene un costo asociado. Una secuencia de transiciones (decisiones) $p=d_1d_2 \,..\, d_n$ se denomina una política. Una política lleva desde un estado a otro por medio de aplicación de la secuencia de transiciones. El costo de una política es la suma de los costos de las transiciones que la componen. Generalmente los problemas de optimización combinatoria pueden verse como problemas de encontrar políticas óptimas en la representación del espacio de estados que representan posibles soluciones.

La programación dinámica requiere que se construya un autómata finito determinístico que representa todos los estados posibles del espacio de solución. Las transiciones representan las decisiones tomadas para el estado dado. Una vez construido el autómata, es posible encontrar la secuencia de decisiones o políticas que llevan a una solución óptima. El principio de optimalidad de la programación dinámica establece lo siguiente:

> Una política óptima $x=x1x2$ que transforma el estado inicial q_0 en el estado final q_f tiene la propiedad de que si q_i es el estado resultante de aplicar la política $x1$, sucede que cualquier política $x1'$ que transforme q_0 en q_i con costo mínimo, también proporciona una política óptima $x'=x1'x2$ que transforma q_0 en q_f.

El principio de optimalidad establece que una política óptima tiene la propiedad de que cualesquiera que sean el estado inicial y la decisión, el resto de las decisiones deberían constituir políticas óptimas con respecto al estado resultante de la primera decisión.

La representación del espacio de estados y sus transiciones o decisiones se puede representar como un autómata finito determinístico con información de costo asociada a cada transición. El alfabeto sería el conjunto de decisiones y el conjunto de estados finales serían los estados que representan soluciones factibles.

La herramienta utilizada se basa en la técnica denominada ***BURS***[1] que combina reconocimiento de patrones en árboles y programación dinámica para generar autómatas que descubren en tiempo lineal, una cobertura óptima del árbol.

4 iburg

iburg[2] es un programa que toma una *gramática de árbol*[3], aumentada con información de costos y genera un parser sobre el árbol usando la técnica BURS (Bottom Up ReWrite System). El parser generado es un programa en lenguaje C que determina en tiempo lineal una cobertura óptima. Para poder realizar la función mencionada se deben determinar todas las coberturas posibles y seleccionar la de menor costo. *iburg* construye un autómata finito determinístico aumentado (con información de costo de cada transición) que representa el

[1] Bottom-Up Rewrite System.
[2] La versión C de iburg se puede obtener vía ftp desde *ftp.cs.princeton.edu*.
[3] Una gramática de árbol es una gramática libre de contexto que permite especicar una estructura de árbol, la cual se describe en la figura 2.

espacio de soluciones y aplica técnicas de programación dinámica para encontrar coberturas óptimas. Se realizan dos pasadas sobre el árbol. La primer pasada es ascendente (bottom_up) y encuentra un conjunto de patrones para cubrir el árbol con costo mínimo. La segunda pasada es descendente (top-down) y ejecuta las acciones semánticas asociadas a los patrones reconocidos con mínimo costo. La programación dinámica se realiza en tiempo de construcción del parser, trasladando el costo de la programación dinámica al momento de compilación-compilación. Lo cual permite que en tiempo de compilación la generación se ejecute en tiempo lineal, siempre que las rutinas escritas por el usuario ejecuten en un tiempo acotado.

3.1 Especificaciones

Una especificación *iburg* es similar a una especificación YACC. Contiene una sección de configuración (declaraciones C), declaraciones de terminales y símbolo inicial de la gramática y un conjunto de reglas que describen un árbol.

La figura 3 muestra la EBNF de una especificación *iburg*.

gram	: *conf* {*decl*} **%%** {*regla*} [**%%** *texto*]
conf	: **%{** *declaraciones_c* **%}**
decl	: **%start** *noterminal*
	| **%term** { *ident* = *ident* }
regla	: *noterminal* : *árbol* = *entero* [*costo*] ;
costo	: (*entero*)
árbol	: *terminal* (*árbol, árbol*)
	| *terminal* (*árbol*)
	| *terminal*
	| *noterminal*

Figura 3: EBNF de una especificación iburg

iburg no acepta acciones semánticas embebidas como *yacc*. En lugar de ello se enumeran las reglas y se usa el número de regla para notificar su selección al modulo de generación que debe escribir el usuario.

La figura 4 especifica un selector de instrucciones muy simple. La sección de configuración requiere que se definan NODEPTR_TYPE, OP_LABEL(p), LEFT_CHILD(p), RIGTH_CHILD(p), STATE_LABEL(p) y PANIC. Estas funciones o macros son usadas en el parser generado por *iburg*, por lo tanto el usuario tiene que definirlas en base a su representación de árbol implementado. En cada nodo del árbol debe existir al menos un atributo de tipo entero, el cual es accedido por la macro STATE_LABEL(p). Las declaraciones de la sección de configuración son tratadas textualmente, lo mismo que el texto que sigue al segundo símbolo (opcional) **%%**, en el cual generalmente se definen funciones de usuario. STATE_LABEL(p) debe ser una macro ya que el parser generado lo usa como un lvalue.

La gramática de árbol es una gramática libre de contexto, tiene reglas, símbolos terminales y no terminales. La parte derecha de una regla, llamada un *patrón*, denota un árbol. Cada no terminal denota un árbol. Una *regla de encadenamiento (chain rule)* es una regla cuyo patrón es otro no terminal.

```
%{
#define NODEPTR_TYPE treepointer
#define OP_LABEL(p)              ((p)->op)
#define LEFT_CHILD(p) ((p)->left)
#define RIGTH_CHILD(p)           ((p)->right)
#define STATE_LABEL(p)           ((p)->label)
#define PANIC    printf
%}
%start objetivo
%term Reg=1 Entero=2 Carga=3 Suma=4
%%
objetivo : reg                   = 1 (0);
reg      : Reg                   = 2 (0);
reg      : Entero          = 3 (1);
reg      : Carga(dir)            = 4 (2);
reg      : Suma(reg,reg)         = 5 (2);
dir      : reg                   = 6 (0);
dir      : Entero          = 7 (0);
dir      : Suma(reg,Entero)      = 8 (0);
%%

/* Rutinas de usuario (label, reduce, etc) */
```

Figura 4: Un selector de instrucciones simple

El costo de una derivación es la suma de los costos de todas aquellas reglas que fueron aplicadas en la derivación. El parser encuentra la derivación con menor costo de todas las posibles derivaciones.

3.2 Parser generado

El parser (autómata BURS, denominado BURM[4]) recorre el árbol dos veces. La primer pasada o rotulador (labeller) se realiza en forma ascendente y de izquierda a derecha, visitando cada nodo exactamente una vez. Cada nodo es rotulado con un estado, valor entero que codifica todos los machings óptimos (completos o parciales) posibles en este nodo y el número de regla al cual corresponde el matching óptimo. El estado de un nodo está rotulado como (M,C), donde M es el número de regla (patrón) con que hace matching y C es el costo (mínimo) asociado. El costo de cada nodo se computa como la suma de los costos de las reglas de encadenamiento más el costo de la regla actual. En la siguiente figura se describe el proceso de la aplicación de la programación dinámica al árbol que representa Carga(Carga(Suma(Reg,Int)))[5]. La programación dinámica prueba todos los patrones coincidentes para cada nodo, estableciendo las reglas y costos (mínimos) asociados para cada posible no terminal. Esa información se codifica en el "estado" del nodo.

[4]BURM: Bottom Up Rewrite Machine.
[5] Es el árbol representado en la figura 2.

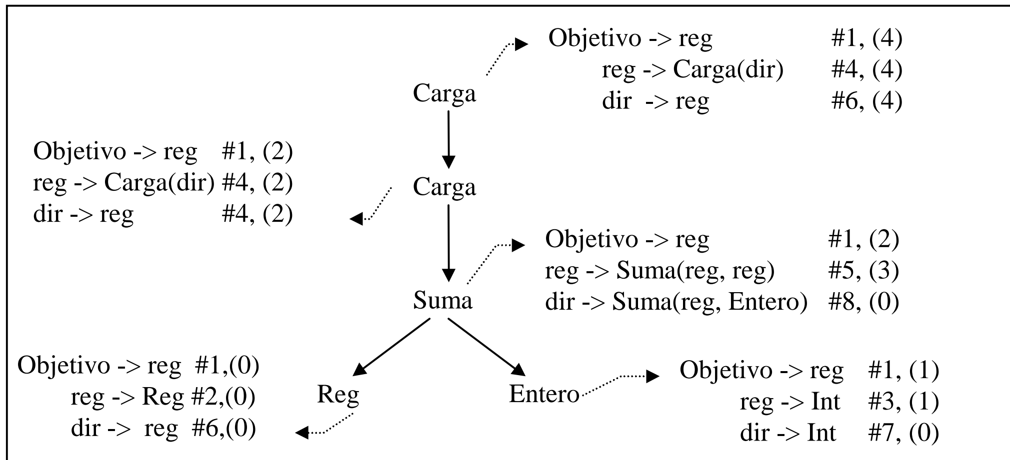

Figura 5: Programación dinámica aplicada al árbol de la figura 2

La segunda pasada o reductor (reducer), recorre el árbol en forma descendente. El reductor acepta un rótulo de estado y un no terminal objetivo (inicialmente el rótulo de estado de la raíz y el símbolo de comienzo) y determina la regla a ser aplicada en este nodo. Por construcción, la regla tiene el no terminal dado como objetivo como parte izquierda en la regla correspondiente en la gramática. El patrón de la regla identifica subárboles y no terminales objetivos para todas las visitas recursivas. Los subárboles especificados en un patrón no necesariamente se circunscriben a aquellos cuyas raíces son hijos inmediatos del nodo corriente. Patrones con operadores interiores causan que el reductor salte a los nodos correspondientes (hijo de hijo o hijo de hijo de hijo, y así sucesivamente). Por otro lado, las reglas de encadenamiento causan que el reductor revisite el nodo corriente con un nuevo no terminal objetivo. De esta manera puede resultar que *x* sea un subárbol de sí mismo.

En este segundo recorrido, cada vez que un nodo es visitado, el código provisto por el usuario debe ejecutar las acciones semánticas asociadas y controlar el orden en que visitan los subárboles.

El rotulador es generado completamente por *iburg*, pero el reductor combina código provisto por el usuario y el código generado.

El código generado contiene primitivas para el rotulado y la reducción de árboles. El usuario puede combinar esas primitivas para poder recorrer árboles de manera arbitraria.

3.3 Rutinas generadas

Rotulador:

extern int burm_label(NODEPTR_TYPE p);

Rotula el árbol apuntado por p y retorna el rótulo de estado del nodo raíz. Retorna 0 en el caso que el árbol no se corresponda con la especificación.

extern int burm_state(int op, int leftstate, int rightstate);

Esta función retorna *burm_label* sin ejecutar el código para recorrer el árbol ni el código para leer y escribir sus campos. Esta función puede ser usada para realizar un rotulado dentro de código provisto por el usuario. El prototipo de esta función varía si la gramática contempla

operadores unarios (se ignora el último argumento) o para las hojas (ignora los dos últimos argumentos).

Reductor

El usuario debe escribir la cascara del reductor, pero los módulos generados por BURM dan soporte a esta etapa implementando los procesos más complejos.

extern int burm_rule(int state, int goalnt);

acepta un rótulo de estado y un no terminal objetivo (*goalnt)* y retorna el número de regla con que hace matching (la regla tiene como patrón un árbol reconocido y como parte izquierda *goalnt*). Esta función retorna 0 cuando el árbol rotulado con *state* no hace matching con el no terminal dado. Para el no terminal objetivo inicial, éste debe ser 1.
BURM exporta un arreglo que identifica los valores para los sucesivos llamados anidados:

*extern short *burm_nts[ ]={...};*

es un arreglo indexado por número de regla. Cada elemento apunta a un vector (terminado en cero) de enteros, mediante los cuales se codifica la regla, de derecha a izquierda.

Con lo expuesto, el usuario ya tendría todo lo necesario para escribir el reductor, pero una tercera función simplifica aún más algunas aplicaciones:

*extern NODEPTR_TYPE *burm_kids(NODE_PTRTYPE p, int eruleno, NODEPTR_TYPE kids [ ]);*

burm_kids acepta la dirección de un árbol p, un número de regla y un vector de punteros a árboles. El procedimiento asume que p hizo matching con alguna regla, entonces llena el vector *kids* con los subárboles (en el sentido descripto anteriormente) de p. *kids* no termina en cero.

A continuación se presenta un ejemplo que realiza un parsing sobre el árbol. El reductor simplemente imprime el recorrido. En general una aplicación emitirá código relacionado con la regla *ruleno*. Para cada no terminal *x*, *iburg* emite una directiva de preprocesador de la forma *#define burm_x_NT n* (donde *n* es un número natural comenzando desde 1).

```
parse (NODEPTR_TYPE p)
 {
   burm_label(p);/* rotulado */
   reduce(p,1);                /* reducción */
 }
reduce (NODEPTR_TYPE p, int goalnt)
 {
   int ruleno = burm_rule(STATE_LABEL(p), goalnt);        /* nro regla matched */
   short *nts = burm_nts[ruleno];   /* no terminales raíces de los subárboles */
   NODEPTR_TYPE kids[10];      /* punteros a subárboles */
   int i;

   printf(“%s\n”, burm_string[ruleno] );      /* imprime la regla */
   burm_kids(p, ruleno, kids);                 /* inicializa punteros a subárboles */
   for (y=0; nts[i]; y++)                      /* accedemos a subárboles de izq. a der. */
     reduce(kids[i], nts[i]);            /* reduce subárboles recursivamente */
 }
```

Figura 6: Un ejemplo de un parser con el reductor

4 Aplicación

Herramientas de este tipo se pueden aplicar a casi cualquier problema que requiera el uso de reconocimiento de patrones óptimos en árboles. En el caso específico de la construcción de compiladores o cualquier otro tipo de procesadores de lenguajes ha sido usado principalmente en asignación de registros, selección de instrucciones, inferencia de tipos y simplificación de árboles. La representación intermedia de un programa C—elegida es un árbol, lo que permite usar *iburg* para reconocer patrones de reglas y emitir las instrucciones assembly correspondientes. A continuación se muestran algunos ejemplos muy simples para ilustrar la forma en que fue aplicado en el proyecto.

```
int     :  ADDI(int, int)  = 1 (1);
float   :  ADDF(float, float) = 2 (1);
float   :  ITOF(int) = 3 (1);
int     :  TRUNC(float) = 4 (1);
```

Figura 7: Representación intermedia de C--

La figura 7 muestra un pequeño fragmento de las reglas que describen la representación intermedia de C--.

El sistema de tipos de C-- es muy simple, por lo que se exigió que se verificara durante el parsing, aunque podría haber sido realizado usando *iburg* como lo muestra la figura 8.

```
integer   : ADD(integer, integer) = 1 (0);
real      : ADD(real, real) =2 (0);
real      : integer = 3 (0);
```

Figura 8. Reglas de inferencia de tipos de C--

Las expresiones en una representación intermedia pueden ser simplificadas. Por ejemplo, la expresión *a+a* puede ser simplificada por *shl(a)* (lo cual es equivalente a *2a*). Se podría hacer que cada vez que se reconozca un patrón de la primera forma, se genere un árbol con ese subárbol reemplazado por el segundo patrón. Esto podría formar parte de un optimizador de código. La siguiente figura muestra otro ejemplo de reglas que permiten simplificar el árbol de una representación intermedia.

```
int       : ADDI(int, int) =1 (1);
int       : NEGI(int) =2 (1);
int       : ADDI(int, NEGI(int)) =3 (1);
int       : ADDI(NEGI(int), int) =4 (1);
```

Figura 9. Reemplazo de substracción por adición

El parser elegirá las reglas 3 o 4 en lugar de la composición de 1 y 2 (si componemos 1 y 2 se obtiene un costo de 2, en cambio si se aplica 3 o 4 el costo es 1). Este esquema permitiría reemplazar la substracción por una adición.

Otra aplicación de *iburg* es la de chequeo de estructuras de datos. Para el caso de compiladores se puede usar para verificar que el generador de código intermedio genere árboles legales (el parser generado será legal solo si el parser generado por *iburg* encuentra un recorrido).

5 Gramáticas divergentes

La programación dinámica se realiza en tiempo de compilación-compilación. Esto requiere que se calculen todos los posibles árboles y se generen todos los estados del autómata finito determinístico que representa el problema de optimización. Para que esto sea posible deben existir un número finito de estados. Existen gramáticas que generan un número infinito de estados. Estas gramáticas de denominan *divergentes*.

Una gramática diverge cuando es posible que existan dos no terminales que están en un mismo estado tengas costos diferentes. *iburg* resuelve el problema haciendo uso de un test de divergencia. Este test determina la mayor diferencia de costo entre los no terminales dentro de un mismo estado. Si la diferencia sobrepasa un valor máximo preestablecido, la gramática se rechaza por ser “posiblemente divergente”.

Afortunadamente, las gramáticas para generación de código no divergen, ya que generalmente los no terminales describen valores de datos (registros, constantes, direcciones, etc).

La gramática mostrada en la siguiente figura es divergente, dado que los no terminales *green_reg* y *red_reg* derivan infinitos árboles idénticos con diferentes costos. Si el costo de la regla 6 fuese cambiado por 1, la gramática no sería divergente.

```
reg          : GreenFetch(green_reg) = 1 (0);
reg          : RedFetch(red_reg) = 2 (0);
green_reg    : Const =3 (0);
green_reg    : Plus(green_reg,green_reg) = 4 (1);
red_reg      : Const = 5 (0);
red_reg      : Plus(red_reg,red_reg) = 6 (2);
```

Figura 10. Una gramática divergente

6 La experiencia

El proyecto se pudo completar completamente en un cuatrimestre y todos los grupos (5 en total con tres integrantes cada uno) entregaron en el plazo preestablecido. Para mantener un cierto control sobre el desarrollo del proyecto se pusieron fechas límite para cada etapa. La siguiente tabla muestra el cronograma de etapas de desarrollo.

Etapa de desarrollo	**Duración (en semanas)**
Descripción del proyecto	1
Analizador lexicográfico	2
Analizador sintáctico (incluye eliminación de conflictos LALR)	2
Analizador semántico y recuperación de errores	3
Generación de código intermedio	2
Generación de código assembly	2
Prueba	1
Entrega del sotware y documentación	1

Los alumnos no encontraron inconvenientes con el uso de la herramienta, posiblemente por su similitud a especificaciones lex-yacc. Si bien la falta de experiencia en el uso de estas herramientas hizo que en principio les resultara dificultosa la asignación de costos a los diferentes patrones para lograr una generación de código óptimo. Con el pasar de los días se fueron dando cuenta de cómo se podía aplicar *iburg* para realizar otras optimizaciones y también se utilizó para validar los árboles generados por el generador de código intermedio, lo cual ahorró mucho tiempo de depuración de los programas.

Los autores de este artículo estiman conveniente el uso integrado de las herramientas lex-yacc e iburg ya que las especificaciones tienen mucho en común lo que evita multiplicar el esfuerzo de aprendizaje de utilización. Además el uso de herramientas para cada etapa de desarrollo de un producto permite aumentar la productividad, potenciando los conceptos de ingeniería de software.

7 Conclusiones

A consideración de los autores, la incorporación de la herramienta coadyuvo a lograr el cumplimiento de los objetivos incorporando una nueva herramienta de desarrollo, permitiendo cubrir una etapa mas del proyecto con el uso de un producto de alto nivel, dejando espacio para complicar la calidad del producto final, como así mismo requerir que se genere código en línea, a nivel de maquina real. El uso de la herramienta también permitió abordar la etapa de testing de la primera fase con un enfoque de verificación automática.

Las ventajas destacadas de *iburg* son su facilidad de uso, ya que las especificaciones son similares a las otras dos herramientas mencionadas. La compacidad y legibilidad del código generado.

Como desventaja se podría decir que obliga al usuario a escribir un modulo C, pero como se mostró en la figura 4 esta presenta un trabajo ínfimo frente al requerido para la escritura integra de un generador de calidad similar.

Otro inconveniente presentado al usar *iburg* consiste en la posibilidad de que la gramática de especificación resulte divergente. Aunque esto difícilmente suceda en aplicaciones de procesadores de lenguajes. Otra desventaja es que los costos deben ser establecidos en forma estática (valor constante en la especificación), mientras que otros reconocedores de patrones en árboles (como Twig), permiten que los costos sean asignados dinámicamente, lo cual permite realizar optimizaciones dependientes de un contexto, no obstante estos procesadores carecen a su vez de otras ventajas de *iburg*.

7 Referencias

- “Engineering a Simple, Eficient, Code Generator Generator”. C. Fraser, D. Hanson, T. Proebsting. ACM Toplas. September 1192
- “BURG - Fast Optimal Intruction Selection and Tree Parsing”. C. Fraser, R. Henry, T. Proebsting. ACM Toplas. December 1991.
- “BURS Automata Generation”. T. Proebsting. ACM Toplas. May 1995.
- “Code Generation Using Tree Patern Matching and Dynamic Programming”. T. Proebsting. ACM Toplas. October 1989.
- “Compilers: Principles, Tachniques and Tools”. Aho, Sethi, Ulmann. Addison Wesley. 1988.
- “A Retargetable Compiler for Ansi C”. Fraser. SIGPLAN Notices. October 1991.
- “Rewrite Systems. Pattern Matching, and Code Generation”. Ph. D Thesis. Computer Science Division, Univ. of California, Berkeley. 1988.
- “A Retargetable C Compiler: Design and Implementation”. Fraser and Hanson. Benjamin-Cummings. 1995.