41 Jornadas Argentinas de Informática

Concurso de Trabajos Estudiantiles 2012

Categoría:

Teoría de la Computación, Modelos Formales y Compiladores.

Título:

Implementación de un compilador de C- -.

Autor:

Riberi, Franco Gaspar.

Docente:

Francisco Bavera.

Cátedra:

Taller de Diseño de Software.

Institución:

Universidad Nacional de Río Cuarto.

Implementación de un compilador de C- -.

16 de mayo de 2012

Resumen

Se presentará un compilador para un subconjunto de estructuras del conocido lenguaje de programación C, llamado C - -. Se describirán las distintas etapas de implementación, partiendo por el análisis léxico de un secuencia de caracteres y finalizando en la generación de código assembly, pasando por el análisis sintáctico y semántico, además de las representaciones internas y las distintas optimizaciones. Este proyecto contempla diversos conceptos de asignaturas previas, tales como gramáticas libres de contexto, expresiones regulares, autómatas, entre otros.

El lenguaje de implementación utilizado para el mencionado compilador fue C, bajo el sistema operativo GNU/Linux. Se utilizaron además diversas herramientas auxiliares, tales como Lex y Yacc para el análisis léxico y sintáctico (LR) respectivamente. Además se utilizó código de tres direcciones como lenguaje intermedio y se generó un assembly x86.

Palabras clave: C- -, análisis léxico, análisis sintáctico, análisis semántico, código 3 direcciones, assembly, generación de código.

1. Introducción

Los lenguajes de programación son notaciones que describen los cálculos a las personas y las máquinas. La percepción del mundo en que vivimos depende de los lenguajes de programación, ya que todo software que se ejecuta en una computadora se escribió en algún lenguaje de programación. Pero antes de poder ejecutar un programa, primero debe traducirse a un formato en el que una computadora pueda ejecutarlo. Los sistemas de software que se encargan de esta traducción se denominan **Compiladores**. A grandes rasgos un compilador es un programa que puede leer otro programa escrito en un lenguaje (fuente) y traducirlo en un programa equivalente.

Dada la importancia de tal herramienta en la ciencia de la computación y como condición necesaria de la cátedra Taller de Diseño de Software (código 3306) de la Universidad Nacional de Río Cuarto surgió la implementación de un compilador para un subconjunto del conocido lenguaje de programación C, al cual se denomina C- -. El producto de este proyecto deberá aceptar programas escritos en C - -, y efectuar su compilación produciendo un archivo ejecutable.

A continuación se detallarán algunos conceptos importantes y necesarios para tal implementación.

2. Compiladores

Un compilador [1] puede considerarse como una simple caja que mapea un programa fuente a un programa destino con equivalencia semántica, como se observa en la Figura 1.

Figura 1: Visión Abstracta de un compilador.

Si se inspecciona esta simple caja podemos observar que existen dos procesos importantes: análisis y síntesis.

La parte de **análisis** *(front-end)* divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Luego utiliza esta estructura para crear una representación intermedia del programa fuente. Además esta etapa recolecta información sobre el programa fuente y la almacena en una estructura de datos conocida como *tabla de símbolos*, la cual pasa a la siguiente etapa junto a una representación intermedia.

La parte de **síntesis** *(back-end)* construye el programa destino deseado a partir de la entrada provista por la parte de análisis. Más detalladamente, se puede observar que un compilador opera como una secuencia de fases, donde cada una de ellas transforma un representación del programa en otra. Esta secuencia se puede observar en la Figura 2.

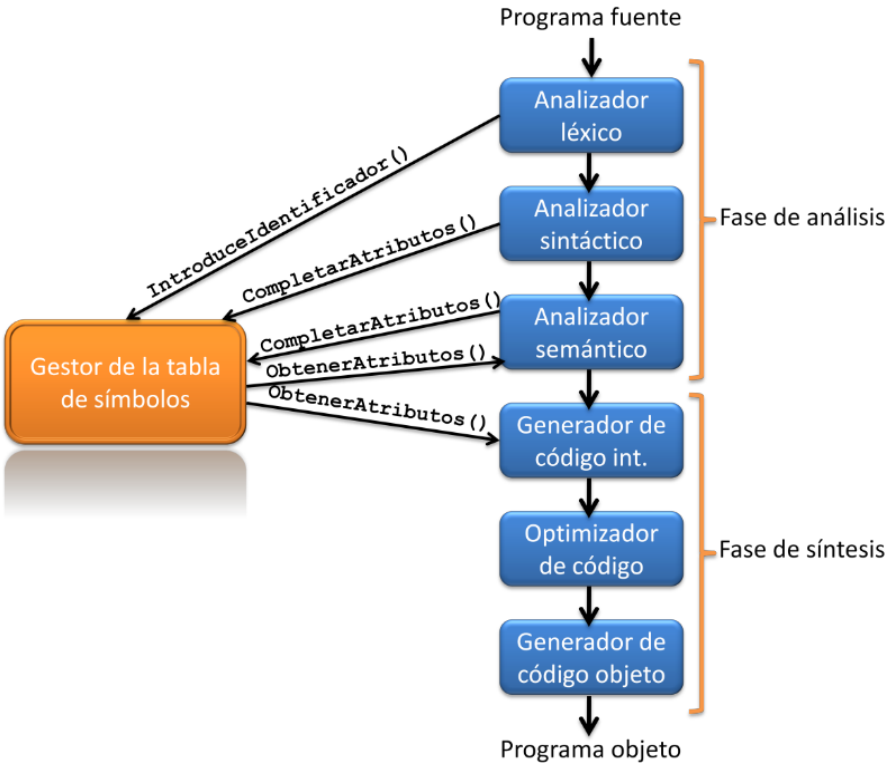

Figura 2: Fases de un compilador.

3. Un pequeño ejemplo

Supongamos que un programa fuente contiene la instrucción de asignación:

posicion := inicial + velocidad * 60

La traducción de esta instrucción realizada por un compilador se observa en la Figura 3.

4. El lenguaje C- -

En términos generales, C- - es un subconjunto de sentencias del conocido lenguaje de programación C, a diferencia del mismo no soporta recursión ni tampoco la definición de tipos por parte del usuario. Sólo maneja tipos básico como integer y float. La estructura y significado de las construcciones del lenguaje C- - se tratan en [8].

5. Lex y Yacc

Lex (o Flex) [2] es una herramienta de los sistemas UNIX/Linux que permite especificar un analizador léxico mediante la definición de expresiones regulares

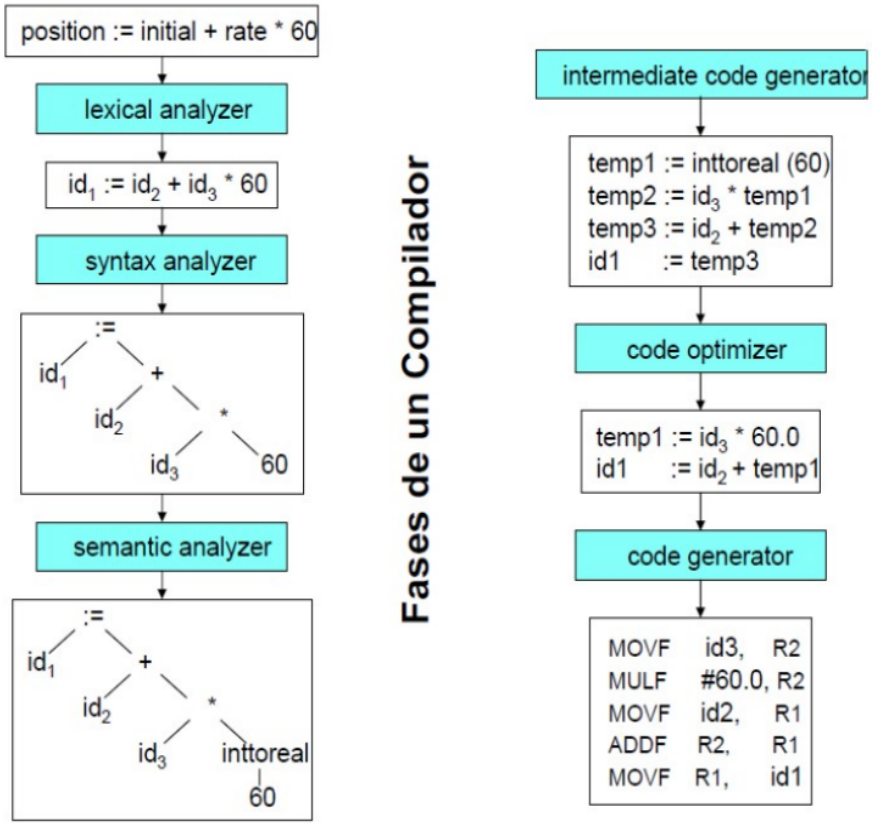

Figura 3: Fases de un compilador aplicadas al ejemplo.

[1][4] para definir patrones de los tokens. Lex posee su propio lenguaje y se encarga de transformar los patrones de entrada en un autómata que es simulado mediante un archivo llamado *lex.yy.c*. La estructura general de un archivo Lex es la siguiente:

```
declaraciones (variables, constantes, etcétera)
%%
reglas de traduccion (Patrón  {acción})
%%
funciones auxiliares
```

Por lo general, Lex se utiliza con el programa *Yacc (o Bison)* [5], que permite generar analizadores sintácticos LALR [3]. Un programa fuente Yacc tiene la siguiente estructura:

```
declaraciones
%%
reglas de traduccion
%%
soporte de las rutinas en C
```

Cuando el analizador sintáctico invoca al analizador léxico, éste empieza a leer la entrada, un carácter a la vez, hasta que encuentra el prefijo más largo de la entrada que coincide con uno de los patrones Pi. Después ejecuta la acción Ai asociada. Por lo general Ai regresará al analizador sintáctico, pero si no lo hace (por ejemplo para descargar saltos de línea) el analizador léxico procede a buscar lexemas adicionales, hasta que una de las acciones correspondientes provoque un retorno al analizador sintáctico. El analizador léxico devuelve al sintáctico un sólo valor, el nombre del token, pero utiliza la variable entera compartida **yylval** para pasarle la información adicional sobre el lexema encontrado, si es necesario.

A nivel de archivos, la comunicación entre Lex e Yacc puede observarse en la Figura 4.

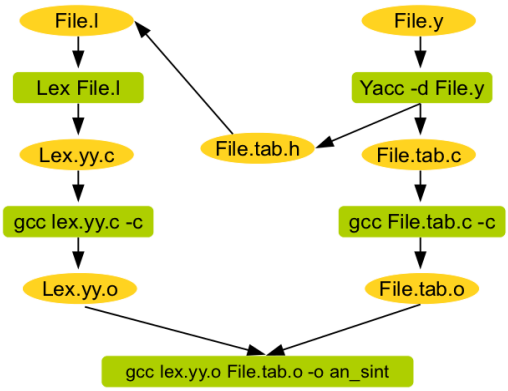

Figura 4: Comunicación a nivel de archivos.

A nivel de funcionamiento, la comunicación entre Lex e Yacc puede observarse en la Figura 5.

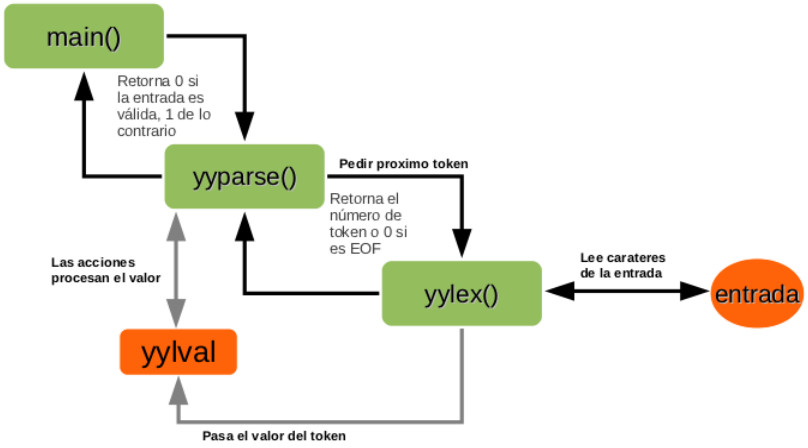

Figura 5: Comunicación a nivel funcionamiento.

6. Implementación Compilador C- -

6.1. Objetivo

Este trabajo de cátedra tiene como objetivo la implementación de un compilador para un subconjunto de expresiones del lenguaje C. Esto involucra, atravesar las distintas etapas que conforman un compilador, enfrentándose a un proyecto medianamente complejo, donde es necesario aplicar diversos conocimientos adquiridos previamente, tales como los conceptos de gramática, autómatas, estructuras de datos, entre otros. El compilador deberá ser capaz de recibir un programa fuente con extinción *.c-* y reportar errores en caso de que existan, ya sean de nivel léxico, sintáctico o semántico, de lo contrario deberá crear un archivo con código assembly.

6.2. Diseño

6.2.1. Análisis Léxico

La principal tarea de esta fase es leer los caracteres de la entrada del programa fuente, agruparlos en lexemas[1] y producir como salida una secuencia de token[2] para cada lexema. Este flujo de token se envía al analizador sintáctico para su análisis. Cuando descubre un lexema que constituye un identificador, almacena ese lexema en la tabla de símbolos.

La tabla de símbolos es una estructura de datos que posee dos funcionalidades principales que corresponden al chequeo semántico y la generación de código. Interactúa con casi todas las fases del compilador. Además, es importante destacar que sólo permanece en tiempo de compilación, no de ejecución, excepto en aquellos casos en que se compila con opciones de depuración.

Esta etapa es la encargada de eliminar comentarios y espacios en blanco además de correlacionar los posibles mensajes de error generados.

Para tal fin, es necesario escribir código con el objetivo de identificar cada ocurrencia de cada lexema en la entrada. Utilizando la herramienta Lex, se especificaron patrones[3] de lexemas. Algunos patrones definidos fueron:

- **Digito** [0-9]
- **FloatDenotation** (Digito)*“.”(Digito)*((“E”|“e”)(“+”|“-”)(Digito)$^{+}$)?

Un ejemplo de regla de traducción puede ser el siguiente:

- **{FloatDenotation}** { return FLOATDENOTATION[4]; }

6.2.2. Análisis Sintáctico

El análizador sintáctico debe obtener una cadena de tokens del analizador léxico, y verificar que la cadena pueda generarse mediante una gramática libre de

[1] Secuencia de caracteres en el programa fuente que coinciden con el patrón para un token y que el analizador identifica como una instancia de ese token.

[2] Es un par que consiste en el nombre del token que representa una unidad léxica y un valor de atributo opcional.

[3] Descripción formal de la forma que pueden adquirir los lexemas de un token.

[4] FLOATDENOTATION representa un token enviado al analizador sintáctico

contexto [1] para el lenguaje fuente. Este analizador debe construir un árbol de análisis sintáctico (AST) el cual debe ser pasado a la etapa de análisis semántico.

Para construir el analizador sintáctico para el compilador de C- - se utilizó la herramienta Yacc. Una versión reducida del analizador sintáctico para C- - puede ser la siguiente:

```
#include <stdio.h>
#include <stdlib.h>
extern FILE *yyin;   /* Referencia al archivo de entrada */
extern int yylineno; /* Para llevar el numero de linea */

%token INTDENOTATION    /* Tokens definidos en el archivo Lex */
%token FLOATDENOTATION
%token IDENTIFIER
...
%start Program
...
%%
Program : { TypeSpecifier IDENTIFIER '(' declPar ')' Compound };
declPar : ParameterDecl | ParameterDecl ',' declPar;
ParameterDecl : TypeSpecifier IDENTIFIER;
....
%%
```

6.2.3. Análisis Semántico

Este análisis utiliza el árbol sintáctico (AST) y la información de la tabla de símbolos para comprobar la consistencia semántica del programa fuente con la definición del lenguaje. También recompila información sobre el tipo e identificador para luego utilizarla en la etapa de generación de código intermedio.

Un aspecto muy importante de esta etapa es la comprobación de tipos, en donde el compilador verifica que cada operador tenga operandos que coinciden. Además, la especificación del lenguaje puede permitir ciertas conversiones de tipos (conocidas como coerciones) y manejo de alcances de los identificadores. En yacc, una acción semántica es una secuencia de instrucciones en C.

Para el manejo de errores se implementó un TAD lista, el cual contiene almacenado diferentes tipos de errores, los cuales se informarán durante el proceso de compilación en caso oportuno. Lo errores definidos son: error de tipos, identificador no declarado, redeclaración de identificador, error de retorno, tipos incompatibles, error en condición, símbolo no válido, entre otros.

A continuación se exhibe un ejemplo de chequeo de tipos en la operación suma:

```
Sum : Sum '+' Term { $$ = malloc (sizeof(struct tipoNum));
            if ($1->tipo == INT_TIPO && $3->tipo == INT_TIPO){
               $$->tipo = INT_TIPO;
            }else if ($1->tipo==FLOAT_TIPO && $3->tipo==FLOAT_TIPO){
               $$->tipo = FLOAT_TIPO;
            }else if ($1->tipo==INT_TIPO && $3->tipo==FLOAT_TIPO) {
               $$->tipo = FLOAT_TIPO;
               mostrarError(5,num_lineas); /* warnning */
```

```
}else if ($1->tipo==FLOAT_TIPO && $3->tipo==INT_TIPO) {
   $$->tipo = FLOAT_TIPO;
   mostrarError(5,num_lineas); /* warnning */
}
}
```

Aquí se puede observar como se pregunta por el tipo de los operandos, y si los dos poseen tipo entero, se sintetiza un tipo entero. Si los dos operandos son flotantes, se sintetiza un tipo flotante. En caso de que un operando sea entero y el otro flotante, se sintetiza un tipo flotante, pero se informa un warnning, y debe truncarse.

6.2.4. Generación de Código Intermedio

En esta etapa se genera una representación intermedia similar al código máquina, que podemos considerar como un programa para una máquina abstracta. La generación intermedia del compilador C- - consiste en la generación de código tres direcciones. Básicamente, consiste en la creación de una representación abstracta en un pseudo-lenguaje. Dicha representación será la entrada de la próxima etapa.

El código de tres direcciones consiste en trabajar con a lo sumo dos operandos y un resultado, además de la operación propiamente dicha. Por ejemplo, para la operación suma entre enteros se generará: **(sumi, op1, op2, res)**.

6.2.5. Generación de Código Objeto para Enteros

La generación del código assembler para enteros tiene como punto de partida la salida de la etapa anterior. Se definió un módulo assembly encargado de tomar la lista de código tres direcciones, y por cada instrucción que involucre operandos enteros, realizar el correspondiente assembler. Se utilizaron los registros de los procesadores de la familia de CPU 80x86. Continuando con el ejemplo de la suma:

```
int sumar (int a, int b){
  return a + b;
}

void printSumI (Tuple* tupla, FILE* file){
  if (tupla->op1->constVar == CONST) {
      if(tupla->op2->constVar == CONST){
         ... codigo assembly
      }else{
         ... codigo assembly
      }
   }else{
      if(tupla->op2->constVar == CONST){
         ... codigo assembly
      }else{
         ... codigo assembly
      }
  }
}
```

6.2.6. Generación de Código Objeto para Reales

De manera similar a la etapa anterior, la generación de código para reales o flotante, requiere como entrada el código tres direcciones, y haciendo patter matching sobre la operación, generar el código assembly correspondiente.

Es esta etapa se utilizó el co-procesador matemático 80x87.

6.2.7. Optimizaciones

Existen diversas optimizaciones que se pueden realizar a un compilador, dentro de ellas se pueden mencionar algunas, tales como: optimización de código local, optimización de mirilla (peephole), optimización de código global, propagación de copias, propagación de constantes, entre otras.

Propagación de constantes

Tiene como objetivo resolver expresiones cuando sea posible, es decir, sintetizar directamente el valor resultado de una operación cuando sus operandos son constantes. Un dato no menor, es que dicha optimización también la implementa el compilador de C.

A continuación se puede observar el assemby generado para la operación suma utilizando la optimización, y sin la optimización. Tomaremos como ejemplo la siguiente función:

```
int sumaConstantes(int a){
    return 5+5+5+5+5;
}
```

Assembly SIN optimización	Assembly CON optimización
pushl %ebp	pushl %ebp
movl %esp, %ebp	movl %esp, %ebp
subl $20, %esp	subl $4, %esp
movl $5, %eax	movl $25, %eax
addl $5, %eax	leave
movl %eax, -4(%ebp)	ret
movl -4(%ebp), %eax	
addl $5, %eax	
movl %eax, -8(%ebp)	
movl -8(%ebp), %eax	
addl $5, %eax	
movl %eax, -12(%ebp)	
movl -12(%ebp), %eax	
addl $5, %eax	
movl %eax, -16(%ebp)	
movl -16(%ebp), %eax	
leave	
ret	

7. Conclusión

El procesamiento de lenguajes es una tarea totalmente desafiante, no sólo desde el punto de vista profesional sino también desde lo humano. Este proyecto

demostró como toda una teoría es aplicable a la práctica de forma muy amigable y desmentificó este abismo.

Desde el punto de vista humano, este proyecto fue una experiencia muy enriquecedora, la cual permitió lograr una maduración muy importante en diversos conceptos vistos con anterioridad en otras cátedras, fomentó el trabajo y desempeño constante en un proyecto de desarrollo de software medianamente complejo. Además, requirió de mucha comunicación y soporte con otros grupos de la cátedra, permitiendo el intercambio de ideas y opiniones lo cual significó un proceso de retroalimentación para todos los integrantes de la misma.

Desde el punto de vista profesional, fue muy importante ya que, se adquirió conocimientos y experiencia en C, un lenguaje de programación no muy trabajado hasta entonces.

En posibles iteraciones del presente proyecto se podría pensar en la implementación de diferentes optimizaciones, además de posibles extensiones tanto simple como complejas. Entendiéndose por extensiones simples, por ejemplo la implementación de arreglos e incrementar la cantidad de tipos básicos. Y por extensiones complejas, la posibilidad de definición de tipos, recursión, concurrencia, entre otras.

Referencias

[1] *"Compiladores. Principios, técnicas y herramientas". Segunda Edición* Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, PEARSON EDUCACIÓN, México, 2008. ISBN: 978-970-26-1133-2.

[2] *"Página inicial de Flex: http://www.gnu.org/software/flex/".* Fundación de software libre.

[3] *"A Compact Guide to Lex & Yacc".* Paul A. Carter., T. Niemann, Disponible en: http://epaperpress.com/lexandyacc/index.html.

[4] *"Regular expression and state graphs for automata".* McNaugton R. y H.Yamada, 1960.

[5] *"Yacc- Yet Another Compiler Compiler".* Johnson, S. C., Computing Science Technical Report 32, Bell Laboratories, Murray Hill, Nj, 1975. Disponible en: http://dinosaur.compilertools.net/yacc

[6] *"Lenguaje Ensamblador para PC".* Paul A. Carter., 9 de agosto de 2009. Copyright c 2001, 2002, 2003, 2004.

[7] *"Procesador Intel 80386: Arquitectura y assembly para GNU/LINUX".* Marcelo Daniel Arroyo.

[8] *"An Introduction to Compiler Construction".* W. Waite,L. Carter and Harper Collins. College Publishers 1993.