Trabajo Práctico

Jorge Mozzino Alexander Campos Francisco Pagliaricci

23 de noviembre de 2012

Índice

1. Consideraciones realizadas

Se decidió generar un programa para un lenguaje muy parecido a C. El lenguaje es, en realidad, un subconjunto de C que permite definir funciones, los tipos de datos char e int, entre otras cosas.

El compilador recibe un archivo y como salida genera un archivo .c listo para ser compilado por el gcc o equivalente.

2. Desarrollo del T.P.

Para la realización del compilador, separamos las dos funcionalidades básicas necesarias. Por un lado, el parseo del código, y por el otro el análisis sintáctico.

2.1. Parser

El parser se hizo en lex y consiste de unas reglas muy simples. Sólo se encarga de leer los tokens necesarios y devolverlos al analizador sintáctico. Entre los tokens más comunes se encuentran un id (que identifica una variable o una función), los paréntesis y las llaves.

2.2. Analizador sintáctico

El analizador sintáctico se implementó en yacc. Aparte de analizar sintácticamente el archivo de entrada, se agregó la funcionalidad de verificar la declaración de funciones y variables, es decir, no compila un programa donde no se declaró una variable o función y tampoco compila un programa donde se redeclara alguna variable o función.

Además, el analizador optimiza cuentas matemáticas que se realicen entre constantes, es decir, si existe la linea

```
int i = 1 + 3;
```

el compilador la reemplazará por

```
int i = 4;
```

Finalmente, se aceptan también múltiples funciones en un mismo archivo.

3. Dificultades encontradas

Uno de los mayores problemas encontrados fue el de desarrollar la gramática para el programa ya que ésta no era simple y además había que asegurarse que no fuera ambigua, y que tampoco hubiera conflictos, para que yacc la pudiera resolver sin problemas.

También se presentaron problemas al permitir la posibilidad de leer y escribir por entrada y salida estándar. Decidimos incluir directivas al preprocesador de C en nuestro lenguaje para así poder soportar las funciones printf y scanf. Sin embargo, no se realiza ninguna verificación al respecto (en cuanto a la inclusión de las librerías correspondientes, o los parámetros).

Otro problema importante fue el de querer optimizar los cálculos matemáticos.

En una primera instancia, todo lo asignable (constantes, variables, etc) se englobó dentro del mismo conjunto de producciones (con el lado izquierdo igual). El problema es que para poder realizar las optimizaciones, los casos en los que se tenían constantes afectadas por un operador matemático eran casos especiales. Pero agregarlos casos especiales no bastó, ya que la gramática pasó a ser claramente ambigua: se pasó de tener:

```
 ASSIGN -> ID
| CONST
| LEFT_PARENTHESIS ASSIGN RIGHT_PARENTHESIS
| ASSIGN PLUS ASSIGN
| ASSIGN MINUS ASSIGN
| ASSIGN TIMES ASSIGN
| ASSIGN DIVIDE ASSIGN
| FUNCTION_CALL
```

a tener

```
 ASSIGN -> ID
| LEFT_PARENTHESIS ASSIGN RIGHT_PARENTHESIS
| ASSIGN PLUS ASSIGN
| ASSIGN MINUS ASSIGN
| ASSIGN TIMES ASSIGN
| ASSIGN DIVIDE ASSIGN
| ASSIGN PLUS CONST
| ASSIGN MINUS CONST
| ASSIGN TIMES CONST
| ASSIGN DIVIDE CONST
| CONST PLUS ASSIGN
| CONST MINUS ASSIGN
| CONST TIMES ASSIGN
| CONST DIVIDE ASSIGN
| CONST PLUS CONST
| CONST MINUS CONST
```

```
| CONST TIMES CONST
| CONST DIVIDE CONST
| FUNCTION_CALL
```

4. Futuras extensiones

Existen muchas extensiones posibles. En primer lugar, una idea que tuvimos fue permitirle al usuario elegir en qué lenguaje de salida quiere su código (entre C, ASM y JAVA, por ejemplo).

También se le pueden agregar funciones al compilador en sí como puede ser la optimización de condiciones lógicas, o manejar mejor la interacción con el preprocesador.