Diseño de compiladores 2012

Informe Prácticos 3 y 4

Generador de código Intermedio y generador de código Assembler

GRUPO 15
Martín Cordischi (mcordischi@gmail.com)
AYUDANTE: José Massa

Introducción

En el siguiente informe se presenta la continuación de los trabajos prácticos 1 y 2, completando así el compilador desarrollado a lo largo de este segundo cuatrimestre. El documento consta de la explicación general del sistema y las decisiones más importantes tomadas para el desarrollo del programa.

Desarrollo

Temas Asignados

Trabajos prácticos 1 y 2:

- Punteros a función
- Unsigned integer
- Ciclo Do until
- Comentarios 1 línea
- Cadenas Multilíneas

Trabajos Prácticos 3 y 4:

- Polaca Inversa
- Control de recursividad en generación de código intermedio
- Generación de código assembler libre
- Chequeo en tiempo de ejecución de división por cero

Generación de código intermedio: polaca inversa

Una vez ya implementada la gramática y funcionando, se debieron realizar ciertos cambios menores en la misma para el correcto funcionamiento en la polaca inversa. La gramática es recursiva y ejecuta el código a medida que realiza la vuelta del backtracking, por lo que se tenía que editar la misma para que la ejecución sea la deseada. Por ejemplo, al inicio en la gramática se tenía

asignación: ID ASSIG exp {Código1}
expr : ID + ID {Código2}

Con el funcionamiento de la gramática, primero se ejecuta el código 2, luego el 1.
Ahora suponga un caso particular:

a:=b+c;

La polaca inversa se desea que tenga el orden:

<a,b,c,+,:=>

Por lo que se debe ejecutar el código 1 primero para obtener a al principio de la lista. Para este problema se propuso como solución añadir mayor nivel de backtracking y controlando así el orden de ejecución.Para la gramática ejemplo, el cambio a:

asignación: headerAssig exp {Código1}
expr : ID + ID {Código2}
headerAssig: ID ASSIG{Código3}

Otorga mayor control sobre el código, dando el orden de ejecución 3,2,1. Otra solución constaría en el uso de pilas para el manejo del ingreso a la polaca inversa, pero se ha descartado por la complejidad de la misma.

En cuanto al contenido polaca, cabe aclarar que para los items dentro de la misma se ha declarado una jerarquía de clases llamada *ReversePolishItem*, los cuales pueden ser de tipo puntero a tabla de símbolos o de control especial (saltos y labels). La clase *ReversePolish* maneja el ingreso y control a la polaca inversa, añadiendo de ser necesario marcas para luego traducirlas a labels, o comandos especiales ante situaciones de control específicas (jumps).

Para satisfacer la condición de las funciones, se han utilizado 3 polacas distintas - una para el programa principal y otras dos para las dos funciones posibles. Ante errores como enunciaciones de función dentro de otra o recursividad, la polaca anuncia mediante mensajes de error. En un principio se utilizaba la posición relativa de la polaca inversa para establecer el nombre del label y se marcaba la posición para luego ser traducida a un label. En la última versión, se ha decidido cambiar la estrategia y poner un tipo de elemento especial perteneciente a la clase *RPExtra* el cual contiene el label. Este es obtenido mediante la posición neta del programa (incluyendo las tres filas polacas) y logra así evitar posibles conflictos de labels.

Los retornos de las funciones se realizan mediante la variable auxiliar *rtn*, la cual se asigna inmediatamente a otra variable auxiliar del puntero que llamó a la función para evitar posibles sobreescrituras, por lo que una llamada a función posee los siguientes elementos en la polaca:

<nombreFunción,CALL,var_aux,rtn,:=,var_aux>

En cuanto al control de recursividad mutua, en realidad el sistema lo realiza automáticamente cuando chequea si la variable ya ha sido anunciada. Para mayor entendimiento se proveerá un ejemplo:

uint fun FunA{ *…* *FunB* *...* *}*	*uint fun FunB{* *…* *FunA* *...* *}*

En el caso que *FunA* esté enunciada primero que *FunB*, el chequeo de declaración de *FunB* responderá con error y pronta finalización del sistema.

Para finalizar, se ha escogido hacer la elección del tipo de salto en esta etapa para liberar de responsabilidad al generador de código assembler, por lo que los saltos ya se encuentran correctamente determinados y bajo la clase *RPExtra*.

Generación de código assembler

Una vez obtenida las 3 polacas, éstas se combinarán en el correcto orden y se utilizarán por el generador de código para obtener como salida código assembler.

Para esta última instancia, se han desarrollado dos clases con fuerte funcionalidad:

- **CodeGenerator:** entidad generadora de código. Interactúa con la polaca inversa para obtener los items y utilizarlos para apilarlos o transformarlos a assembler. Delega toda la responsabilidad de manejo de memoria al *MemoryManager,* pidiendo solo un dato tipo registro, memoria y/o inmediato.
- **MemoryManager:** Es el encargado de control completo sobre la memoria. A pedido del *CodeGenerator*, genera datos de memoria, asigna registros o devuelve valores inmediatos. También realiza movimientos de registros ante pedidos especiales. Al finalizar su ejecución devuelve las declaraciones de memoria necesarias.

Como el *MemoryManager* hace que el manejo de datos sea transparente, el tipo de pedido de memoria del *CodeGenerator* puede priorizar uso de registros o memoria. Por el momento y para mayor entendimiento del código assembler salida, el generador de código intentará maximizar el uso de memoria, exceptuando los casos donde se requiera obligatoriamente registros.

En cuanto al chequeo de división por cero, se ha hecho un control previo a la división que se inserta en la generación de código, resultando en un mensaje de error de ser necesario. Con respecto al chequeo de punteros se hizo algo similar, poniendo como valor inicial a todos los punteros un llamado a una función especial de error.

Vista general del sistema

Ya detallado todas las decisiones importantes, se procederá a realizar una vista general del sistema, observando los componentes funcionales más importantes.

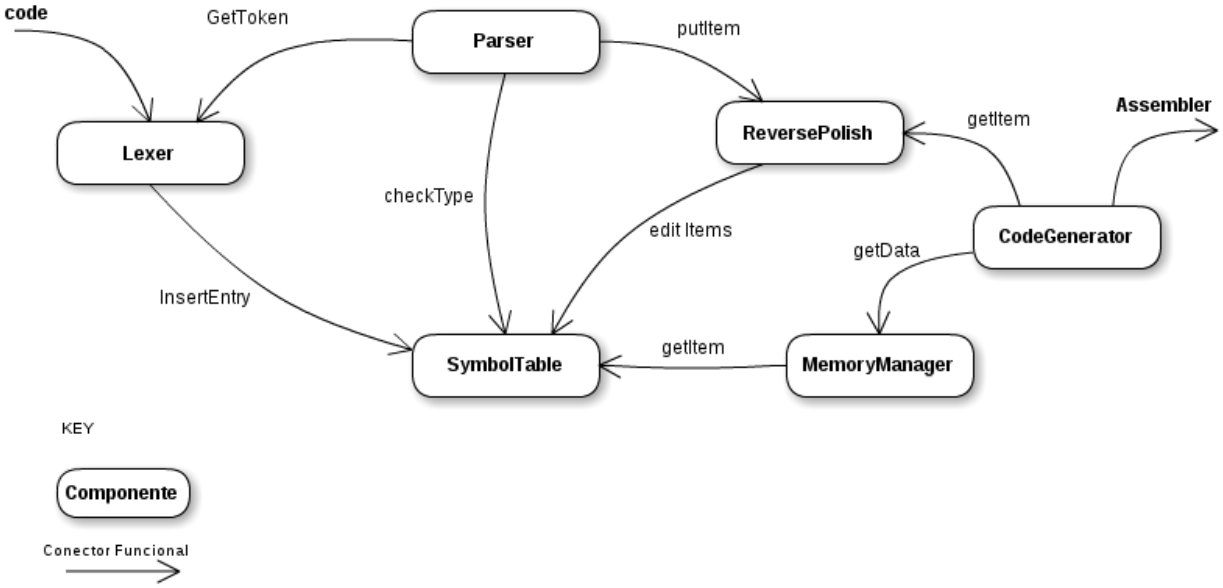

Fig1 *Diagrama de conectores*

Como se puede observar, el sistema está controlado en una primer etapa por el Parser, el cual es el encargado de pedir tokens al Lexer y luego insertarlos en la Polaca Inversa. Cabe destacar que en las tres etapas, **todos** los datos se referencian con entradas a la tabla de símbolos.

Una vez finalizada la ejecución del parser, el generador de código obtiene el control y realiza el manejo necesario sobre los componentes polaca inversa y controlador de memoria.

Cabe destacar que aunque no se detalla en el diagrama, existe un componente *Event* de interfaz con el usuario, el cual capta mensajes emitidos por todos los componentes ante todo tipo de eventos, dando la posibilidad de un nivel de detalle de la información alto. También, aunque no se ha realizado, se puede realizar una interfaz gráfica utilizando la interface *Event.*

Entregables

Junto a este informe se hace entrega de los siguientes elementos

- **Codigo fuente:** realizado en java, se encuentra dentro de la carpeta *src*
- **Gramática:** el archivo *grammar.y*
- **Ejecutable:** bajo el nombre *compiler.jar*, el cual funciona mediante consola y su único parámetro es la dirección al archivo que contiene el código a compilar. El mismo devolverá por *stdout*, la tabla de símbolos, la polaca inversa y/o errores o warnings. También devolverá un archivo con el nombre *out.asm* que contiene el código assembler solución.