DISEÑO DE COMPILADORES

TRABAJOS PRÁCTICOS 3 Y 4, GRUPO 04

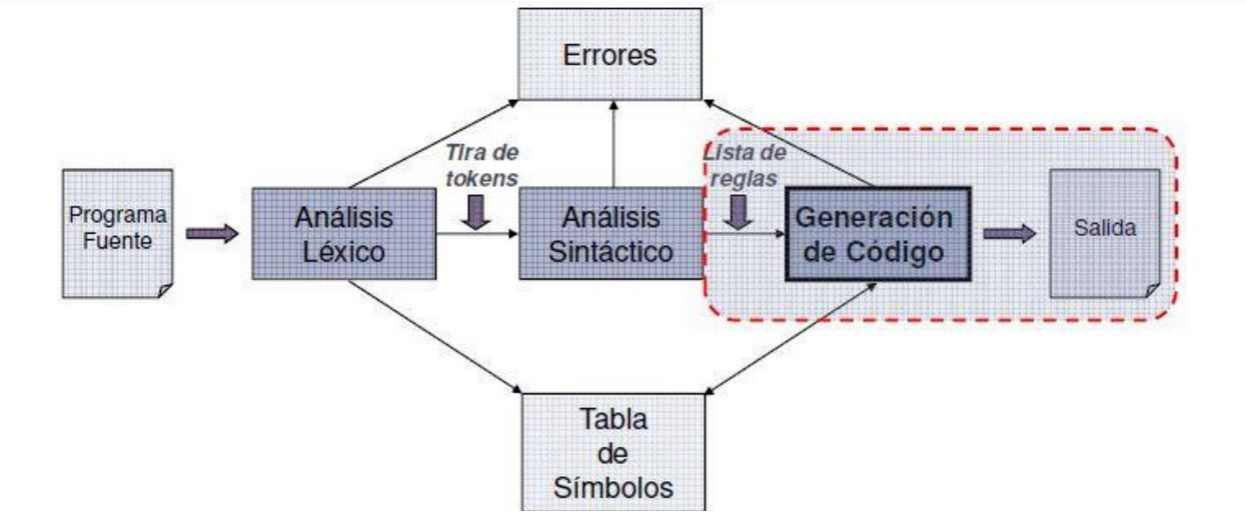
LUDMILA BALIÑO, TOMÁS GARCÍA, JULIÁN SPINELLI

11 de noviembre de 2017

Facultad de Ciencias Exactas, UNICEN

INTRODUCCIÓN

El siguiente informe contiene detalles acerca del desarrollo de los trabajos prácticos 3 y 4 de la materia Diseño de Compiladores. El objetivo de esta segunda parte era utilizar lo realizado en la entrega anterior para completar la compilación del código ingresado en el lenguaje especificado por la cátedra. Para lograrlo, se generó el código intermedio representado por árbol sintáctico y, en base a esto, generar el código assembler para Pentium de 32 bits. Para la generación de código fue necesario realizar cambios tanto a la tabla de símbolos como a la gramática. Dichos cambios serán explicados en detalle en este informe.



GENERACIÓN DE CÓDIGO INTERMEDIO

Después de los análisis sintácticos y semánticos, se generó una representación intermedia explícita del programa fuente. Se puede considerar esta representación como un programa para una máquina abstracta. Esta debe tener dos propiedades importantes: ser fácil de producir y fácil de traducir al programa objeto. El código intermedio permite que el compilador sea independiente de la máquina objetivo. Entonces, generar código para diferentes máquinas objetivo sólo requiere volver a escribir el traductor de código intermedio a código objetivo, sin necesidad de hacer un nuevo generador de código. Existen diferentes alternativas para generar código a partir de la lista de reglas, los cuales se nombran a continuación:

- Polaca Inversa
- Tercetos
- Árbol Sintáctico

Particularmente, desarrollaremos la representación Árbol Sintáctico.

IMPLEMENTACIÓN EN JAVA

1. ÁRBOL SINTÁCTICO

   Se optó por utilizar una estructura de Árbol, se implementó la clase ArbolSintactico en la cual se almacena el contenido, un boolean indicando si es de 16 o 32 bits y los punteros a derecha e izquierda; la cual provee una estructura dinámica de almacenamiento. Esta estructurará guardando (mediante el código de la gramática) las variables, constantes y operadores pertinentes en el código fuente de entrada.

2. CONVERSIONES EXPLÍCITAS

   Para realizar la conversión de UINT a ULONG se agrega un nodo CONV al árbol sintáctico el cual posee en su nodo izquierdo la expresión a convertir y se realiza la conversión de un registro de 16 a 32 bits.

3. SWITCH CASE

   Para implementar esta sentencia de control se agrega un nodo con el nombre SWITCH al árbol sintáctico, el cual tiene como hijo izquierdo el valor de la variable a comparar y a la derecha tiene el CASE, que a su vez, este tiene en la rama izquierda tiene el nodo CUERPO y a la derecha el próximo CASE. El nodo CUERPO tiene como hijo izquierdo el valor de la constante y en el derecho el bloque o sentencia a ejecutar.

4. LET

Para implementar el LET agregamos un nodo al árbol con el nombre LET, que tiene como hijo izquierdo la asignación. Además tenemos que verificar si la variable ya se encuentra repetida, es decir si anteriormente se re-declaró mediante otro LET o shadowing. Para este caso, se agrega a una estructura que controla los repetidos y se agrega delante de cada variable el símbolo '@' para controlar la cantidad de veces que se repitió.

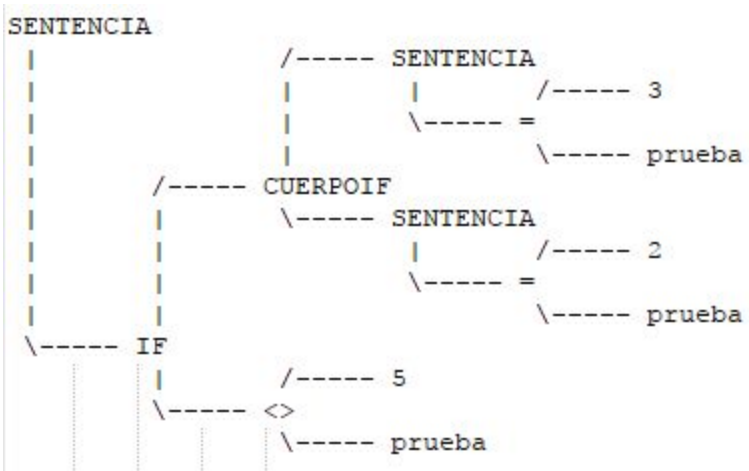
5. SHADOWING

Para implementar el shadowing, utilizamos una estructura auxiliar que almacena la cantidad de veces que se re-declara una variable ya que no se puede borrar de la tabla de símbolos. Entonces de esta forma se puede saber qué valor utilizar cuando aparece en el código,

6. SENTENCIA IF

Para la sentencia IF se agrega un nodo en el árbol que tiene como hijo izquierdo la condición y como hijo derecho el cuerpo del IF el cual tiene como hijo izquierdo el cuerpo THEN y como hijo derecho el cuerpo ELSE. En el siguiente ejemplo se puede ver más claro:

```
prueba : ULONG.
IF ( prueba <> 5) THEN
        BEGIN
        prueba=2.
        END.
ELSE
        BEGIN
        prueba=3.
        END.
END_IF
```



7. SENTENCIA OUT

   Para la sentencia OUT se agregó el nodo en el árbol y como hijo izquierdo la cadena de caracteres a imprimir por pantalla.

8. CHEQUEOS SEMÁNTICOS

   Para verificar si las variables utilizadas en el bloque de sentencias fueron declaradas previamente se verifica en la tabla de símbolos que exista la entrada correspondiente a esa variable. Las operaciones deben realizarse entre operandos del mismo tipo. Para lograr esto, se realiza la verificación en el assembler, preguntando por el valor del tipo almacenado en el árbol.

GENERACIÓN DE CÓDIGO *ASSEMBLER*

El objetivo del TP No 4 era generar código assembler a partir del código intermedio generado en el trabajo práctico anterior. El mecanismo de generación utilizado fue el de registros. A su vez, se incorporó el chequeo en tiempo de ejecución de los siguientes errores:

1. División por cero
2. Resultados de resta negativa
3. Overflow en productos

En esta última etapa del proyecto se realiza la generación de código Assembler. Se trata de un código de bajo nivel que es interpretado por el procesador. Para llevarlo a cabo, se utiliza el código intermedio (árbol sintáctico) y la tabla de símbolos, en la cual se encuentran las variables y constantes con sus valores y tipos. Se tienen en cuenta nuevos errores, los cuales se realizan en tiempo de ejecución, debido a que implican conocer valores de variables u operaciones. A continuación se explicará detalladamente cómo fue el proceso a seguir para llegar al código final.

COMPILADOR ASSEMBLER UTILIZADO

Para realizar la generación, se utilizó el compilador Assembler MASM32. El Microsoft Macro Assembler (MASM) es un ensamblador para la familia x86 de microprocesadores. Fue producido originalmente por Microsoft para el trabajo de desarrollo en su sistema operativo MS-DOS, y fue durante cierto tiempo el ensamblador más popular disponible para ese sistema operativo. El MASM soportó una amplia variedad de facilidades para macros y programación estructurada, incluyendo construcciones de alto nivel para bucles, llamadas a procedimientos y alternación (por lo tanto,MASM es un ejemplo de un ensamblador de alto nivel). MASM es una de las pocas herramientas de desarrollo de Microsoft para las cuales no había versiones separadas de 16 bits y 32 bits.

IMPLEMENTACIÓN EN JAVA

Para realizar la implementación fue necesario agregar una nueva clase al proyecto (GeneradorCod) en la que se desarrollan distintos métodos para poder construir el archivo final. El cuerpo básico de un archivo para MASM está principalmente formado por:

Encabezado: Se incluyen las librerías necesarias para el proyecto.

.DATA: Do nde se declaran las variables necesarias para nuestro programa.

.CODE: En el cual se llevan a cabo las operaciones tales como ADD,SUB,DIV, etc.

.START: Da comienzo al bloque de código.

Para chequear los errores nombrados anteriormente, se utilizaron tres subrutinas cada una con los siguientes labels:

Label 0 para verificar que el divisor no sea 0.

Label 1 para verificar que la resta de dos números positivos no de un resultado negativo.

Label 2 para verificar que el resultado de un producto no produzca overflow.

CONCLUSIÓN

Consideramos muy positiva la realización de este trabajo porque nos ayudó a entender las fases que atraviesa el código fuente de un programa durante su compilación. Al conocer la forma en que trabaja el compilador, resulta más sencillo entender qué procesos se ejecutan, y esto nos trae muchos beneficios aplicables a la hora de programar. Creemos que es importante haber comprendido mejor el manejo de errores de compilación y ejecución. Entender por qué suceden, qué es lo que sugieren o informan y por qué , en ocasiones, el orden de aparición de los mismos no es el más lógico o el esperable.

Finalmente, concluimos que la realización de este trabajo ha sido un gran aporte a nuestro conocimiento acerca del funcionamiento tanto de los compiladores como de los lenguajes que utilizamos.