Trabajo Práctico Nº 6 – Transpiler

Generando código C

1. **Objetivos**
   a. Armar un compilador tipo transpiler para lenguaje mini que genera código en lenguaje C.
   b. Afianzar el conocimiento del analizador semántico.
2. **Temas**
   a. Analizador semántico.
   b. Generación de código.
3. **Tareas**
   a. Se pide armar un compilador tomando como base el TP5 al que le agregaremos las rutinas semánticas de control y generación de código.
   Seguiremos trabajando contra stdin y stdout, y redireccionando el archivo a probar al flujo stdin. Se proveen 2 archivos de entrada y sus correspondientes salidas.
   Comentarios:
      1. El archivo `entradaok.txt` es uno totalmente correcto, en tanto `entradaerr.txt` tiene errores léxicos, sintácticos y semánticos. El archivo `salidaerr.txt` se corresponde a `entradaerr.txt` y muestra los errores detectados. En tanto que el archivo `mainok.c` es el fuente que se genera a partir de `entradaok.txt`. Si además compilamos dicho archivo, y ponemos como entradas de `var1` y `var2` a `20` y `-3` respectivamente, el archivo `EjecuciónOk.txt` muestra que da al correrlo.
      2. Haremos las siguientes modificaciones.
         i. Controlaremos que las variables que se utilicen haya sido previamente declaradas.
         ii. Controlaremos que no se redeclare una variable.
         iii. Informaremos al final de la ejecución la cantidad de errores léxicos, sintácticos y semánticos, pero solo lo haremos si hubo errores. Si compiló sin errores no emitiremos ningún mensaje, de modo que si redireccionamos la salida a un archivo nos queda el fuente en lenguaje C.
      3. Agruparemos las rutinas que manejan el diccionario en `symbol.c,` no se requiere la rutina más eficiente, nos basta con una que sea clara y correcta
      4. Agruparemos las rutinas semánticas en `sematic.c`, las funciones del compilador micro que explicamos en clase son una buena base, con los ajustes pertinentes, y pueden necesitar nuevas funciones. Por supuesto, teniendo en cuenta que ahora generamos lenguaje C.
      5. Para las variables temporales usaremos _Tempi donde i es el número de temporal que iniciaremos en 1.
      6. Cuando haya errores semánticos usaremos YYERROR para dejar de analizar sintácticamente la sentencia o definición donde ocurrió el error.
      7. NO usaremos yyerrorok.
      8. Vamos a suponer que el identificador con el nombre del programa está en un espacio de nombre diferente del de las variables, por lo tanto es lícito que una variable se llame igual que el programa.

b. Programar usando los siguientes fuentes:
   1. `main.c` llama al parser y hace el informe final de la ejecución.
   2. `scanner.l` con la especificación para que flex arme los fuentes del scanner.
   3. `parser.y` con la especifiación para que bison arme los fuentes del parser.
   4. `makefile` para poder armar todo el proyecto, es decir, correr flex, bison y compilar.
   5. `semantic.c` y `semantic.h` para las rutinas semánticas.
   6. `symbol.c` y `symbol.h` para implementar el diccionario.

4. Productos

```
`24-002-xx                    //Repositorio del grupo
    |-- readme.md             // Carátula del grupo, ya hecha en TP1
    `-- TP5                   //Directorio para el TP2
        |-- readme.md         // Carátula del TP
        |-- main.c            // Inicio del programa
        |-- scanner.l         // Definición para flex
        |-- parser.y          // Definición para bison
        |-- makefile          // para armar el proyecto
        |-- semantic.c        // rutinas semánticas
        |-- semantic.h        // encabezado de runtinas semánticas
        |-- symbol.c          // funciones del diccionario
        |-- symbol.h          // encabezado de del diccionario
```

5. Fechas de entrega

a. Última fecha para primera entrega: 15/12/2024

b. Última fecha para segunda entrega: 30/12/2024