Lenguaje: Pascal-TDS - 2017 - Taller de Diseño de Software (Cod. 3306)

Primera etapa: Analizador sintáctico y léxico.

Organización del grupo:

- Para comenzar, en cuanto a la comunicación entre Lex y Yacc, se decidió transmitir toda la información mediante tokens. A partir de allí, Boselli realizó una implementación inicial del Analizador Léxico mientras que González y Etcharren realizaban la correspondiente al Analizador Sintáctico.

- Al finalizar las implementaciones "alfa", se solucionaron en conjunto los conflictos shift/reduce y se refinaron ambos analizadores. Luego, para los casos de testing: comentarios, "end" y "begin" balanceados, a cargo de Boselli, if anidados, funciones, while, a cargo de González y Etcharren a cargo de la implementación para descartar correctamente los comentarios largos. Se realizó una última reunión donde se revisaron en conjunto los puntos a mejorar para dar por finalizada esta etapa.

Decisiones de Diseño:

- Comunicación entre Lex y Yacc mediante tokens. Esto será útil en un futuro para el analizador semántico.

- Crear una librería *"types.c"* donde se definieron todos los tipos a utilizar. Para el análisis léxico y sintáctico se crearon dos de ellos: TokenStr, utilizado para enviar el nombre de una variable declarada con su numero de línea y TokenVal utilizado para literales enviando su valor y número de línea.

- Para el resto de los tokens se enviará solo el número de línea, ya que podrán identificarse directamente por su nombre.

Detalles de implementación interesantes:

- Implementación de una función error que permitirá especificar correctamente el error en casos de:

* Un caracter extraño.

* Un entero mayor a 32 bits.

* Un comentario largo no cerrado.

El resto de los errores serán notificados mediante yyerror().

- Implementación correcta de comentarios largos mediante condiciones de entrada de Flex.

****

Segunda etapa: Analizador semántico.

Organización del grupo:

- Para comenzar, se definió la representación de la tabla de símbolos y el árbol semántico mediante estructuras. Luego, González y Etcharren realizaron la carga del árbol mientras que Boselli, modificó las funciones ya creadas en la etapa anterior para poder reutilizarlas. A partir de allí se comenzó con una primera instancia de chequeo de tipos, continuada por González y optimizada por Etcharren. Boselli mejoró el script de ejecución tal como era solicitado en términos del proyecto y para finalizar, en conjunto se solucionaron los bugs de carga y chequeos donde también González realizó los casos de pruebas necesarios para el testing.

Decisiones de Diseño:

- La principal decisión de diseño fue crear un tipo *"item"*, el cual contiene: nombre, valor, tipo, tipo de retorno (en caso de ser una función), función y parámetros actuales de la misma. Según el tipo de item (VAR, CONSTANT, OPER_AR, OPER_LOG..) se almacenarán los datos necesarios en la estructura. Esto simplificará el armado del árbol ya que todo nodo tendrá la misma estructura que cada elemento de la tabla de símbolos, especificando el tipo de nodo a través del campo type.

- Máximo 20 anidamientos de bloque en la tabla de símbolos.

- Se manejará todo con punteros para no generar copias.

- No se podrán declarar funciones con el mismo nombre aunque tengan distinta cantidad de parámetros, se tomó esta decisión ya que se analizaron posibles complicaciones a futuro. Tampoco se permitirá definir variables con el mismo nombre que una función.

- Los parámetros podrán ser ocultos dentro de una función declarando una variable con el mismo nombre, esta decisión se debe a una cuestión de estructura, ya que necesitaremos guardar la lista de sus parámetros con sus tipos. Por lo cual, en un primer momento se creará el nivel correspondiente a los parámetros y luego se guardará en el nodo correspondiente a la función en el árbol. La otra alternativa, no permitir ocultarlo, no nos pareció adecuada.

- Para funciones cuyo retorno es de tipo entero o booleano, se chequeará que al menos contenga un *return*. Consideramos que lo ideal hubiese sido realizar un chequeo exhaustivo por las ramas del árbol, controlando en cada rama de ejecución la existencia de un return. Por la complejidad que esto significaba se tomó la decisión de realizarlo de una manera quizás menos elegante, la cual se podrá ver en la implementación.

Algunos problemas:

- No se pueden definir funciones recursivas. Para que esto funcione se deberá en primer lugar, insertar en la tabla de símbolos el nombre de la función con su tipo de retorno y párametros correspondientes. Luego, una vez recorrido el cuerpo, agregar el árbol que lo representa a dicha función.

****

Tercera etapa: Generador código intermedio.

Organización del grupo:

- Antes de comenzar esta etapa, se corrigieron algunos errores correspondientes a la etapa anterior y también se agregaron algunas pruebas adicionales más completas. Se realizaron modificaciones en yacc, necesarias para la nueva etapa. En la operación asignación se agregará como hijo derecho la variable a asignar, también chequeos de tipos en checkTree, entre otros.

- Para esta etapa se definió una idea inicial sobre la lista de operaciones a mostrar, lo cual puede observarse en el archivo *"CodIntermedio.xls"* . Una vez en claro el funcionamiento de nuestras instrucciones, se creó una librería *"intermediateCode.c"* la cual contiene la implementación de las operaciones necesarias.

- Se procede a la división de las tareas: Etcharren a cargo de la inicialización de las estructuras necesarias, los métodos *generateIntercode*, *initListTreeDir*, *insertOperation*, *showOperation.* González a cargo de la implementación de las instrucciones de los operadores aritméticos y lógicos *(generateOpLog*, *generateOpArit..).* Boselli a cargo de la implementación de las condiciones, ciclos y llamadas a funciones *(generateFunctionCall*, *generateIf*, *generateWhile..)* para las cuales se utilizaron los labels y temporales donde fueron necesarios.

- Una vez terminadas las tareas, en conjunto se realizaron los chequeos necesarios para verificar el correcto funcionamiento de los métodos implementados, finalmente se procede a la etapa de pruebas.

Decisiones de Diseño:

- Crear una librería *"intermediateCode.c"* donde se definieron los tipos a utilizar, como se puede observar, cada operación tendrá una instrucción (tipo enumerado), los respectivos operadores y el resultado (punteros del tipo item definidos y utilizados en "structures.c")

- Se utilizaron *Labels y Temporales*, para los indicadores de salto y resultados de operaciones respectivamente. Se decidió en la instrucción asignación no utilizar temporales para las variables a asignar.

- En el caso de las funciones, se definieron dos labels nuevos: IC_BEGIN_FUNCTION, IC_END_FUNCTION los cuales indican el comienzo y fin de una función. Se utilizará una nueva función general *generate(item *func)* la cual creará los labels cuando sean necesarios.

- Para mostrar la lista de operaciones completa, se pensó en utilizar variables globales en yacc pero debido a complicaciones con el pasaje de parámetros, se utilizaron variables estáticas en intermediateCode. Así finalmente, se recorrerá el nivel correspondiente mostrando de manera completa todas las funciones e instrucciones correspondientes a cada programa.

****

Cuarta etapa: Generador código objeto.

Organización del grupo:

- Se realizaron modificaciones en las librerías *structures.c* y *ptds-sintaxis.y*, para poder tener la información sobre la cantidad de memoria necesaria en una función, como así tambien la localización de variables y parámetros en memoria. Estas correcciones a cargo de Etcharren y González, mientras Boselli comenzaba con la idea inicial sobre el paso a assembler, chequeando como genera cada instrucción el lenguaje c.

- Luego se generó la librería que contendrá la generación del código objeto, dividiendo las tareas: Boselli a cargo de generateAssign, generateIf, generateWhile, generateLabel, generateJump, generateLoad. Etcharren: generateRetInt, generateRetBool, generateRetVoid, generatePushParam, generateCallFunc, generateLoad, generateBeginFunc, generateEndFunc. González: generateSub, generatePlus, generateDiv, generateMod, generateAnd, generateOr, generateNot, generateEqAr, generateEqLog, generateNeg, generateMinnor, generateMajor.

- Una vez terminadas las tareas, Etcharren realizó los chequeos necesarios para verificar el correcto funcionamiento de los métodos implementados. Boselli continuó con la modificación de la *Interfaz de la línea de comandos* especificada en la Descripción del proyecto. También se realizaron las modificaciones necesarias en el script para poder diferenciar entre las compilaciones en Linux y Mac, debido a algunos conflictos debido a la diferencia de sistemas operativos. Los comandos de compilación se encuentran detallados en el archivo *README.md* del proyecto. González realizó las modificaciones necesarias para la creación del archivo *assemblyCode.s* y permitir que se almacenen en él todas las sentencias correspondientes al código assembler generado, como así también los casos de prueba para el testing. Para finalizar, Etcharren definió un nuevo label IC_PRINT para permitir la impresión de valores.

Decisiones de Diseño:

- La principal decisión de diseño fue refinar el código intermedio para facilitar el paso al código objeto.

- En la librería *structures.c* se definió un nuevo atributo *stackSize* en el tipo *itemFunc* el cual contendrá la información sobre la cantidad de variables que contiene la función y el nuevo atributo *offSet* en el tipo *item* el cual contendrá la información necesaria para luego setear la posición del mismo.

- Crear una librería *"assemblyCode.c"* donde se definirán las funciones necesarias para generar de forma adecuada las sentencias de código assembler correspondientes a cada operación, las cuales también serán almacenadas en un archivo *assemblyCode.s.*

- Modificaciones en la librería *intermediateCode.c* para poder invocar una función dentro de otra. Para que esto funcione de forma adecuada, se resolverá primero la llamada a cada árbol, almacenando los resultados en temporales para luego pushearlos correctamente. Así se podra evitar la sobreescritura de los registros y posibles pérdidas de información que puedan ocasionar errores

en los cálculos. Además se corrigió el bug donde las variables se seteaban en temporales siendo realmente no necesario.

- Se tomo la decisión de crear una nueva sentencia *(print expresion;)* modificando todas las instancias necesarias.

****

Etapa de optimización.

Optimización de constantes.

- Se recorrerá el árbol con depth first search en busca de sentencias del tipo expresiones conformadas por constantes para poder optimizar recursivamente, para ello controla que la operación sea de tipo aritmeticos, logicos o relacionales. Obtenido el resultado en el caso de una optimización, sube el resultado a la cabeza actual y libera sus hijos.

- Se podrá observar esta optimización en casos como las condiciones de if y while, operaciones aritméticas, lógicas y relacionales (add,sub,prod,div..)

Otras optimizaciones.

- Sentencias *return*: al encontrar una sentencia return en un bloque, considera código muerto las sentencias que lo proceden, por lo tanto las poda. No se ignorá el resto del código si el return se encuentra dentro de un bloque if o while

- No se utilizarán temporales para la lectura de constantes como se hacía anteriormente.

-*Variables globales:* se permitió el uso de variables globales a partir del uso de una nueva constante VARGLOBAL para distinguir el tipo de valor del item en la tabla de símbolos.

- Actualmente en caso de las sentencias statement, en cuanto a estructura, sus hijos pueden ser nulos. Esto nos permite poder realizar podas en casos como un < if(true) then .. else > podando la rama else, < if(false) then .. else> podando la rama then, < while(false) ..> podando la rama while, quedándonos únicamente con los bloques necesarios.

If-Then: Intenta optimizar la condición. Luego, si la misma es falsa constante elimina completamente el código del if; caso contrario, verdadera constante, sube el codigo correspondiente al then a la cabeza actual y elimina la instrucción if.

If-Then-Else: Intenta optimizar la condición. Luego, si la misma es falsa constante sube el código correspondiente al else a la cabeza actual y elimina la instrucción if; caso contrario, verdadera constante, sube el código correspondiente al then a la cabeza actual y elimina la instrucción if. Si la condición no es constante, intenta optimizar los bloques then y else.

While: Intenta optimizar la condición. Luego, si la misma es falsa constante elimina completamente el código del ciclo. Caso contrario (sea o no constante la condición), intenta oprtimizar el bloque del ciclo.

Sentencias: Se intentará optimizar su hijo izquierdo, si encuentra un return poda su hijo derecho. Si su hijo izquierdo queda con código muerto, lo poda. Intenta optimizar su hijo derecho y si queda con código muerto también lo poda.

- A continuación se mostrarán algunos ejemplos, la imagen del lado izquierdo será la correspondiente a la versión anterior mientras que la imagen del lado derecho corresponde a la versión optimizada.

Operación suma.

```
program
  begin
    integer  main()
      begin
        integer x;
        x = 4+3;
        return x;
      end
  end
```

```
BEGIN FUNCTION    (null)      (null)      main
LOAD              4           (null)      Temp 0
LOAD              3           (null)      Temp 1
ADD               Temp 0      Temp 1      Temp 2
ASSIGN            x           (null)      Temp 2
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

```
BEGIN FUNCTION    (null)      (null)      main
ASSIGN            x           (null)      7
IC_PRINT          (null)      (null)      x
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

Operación sustracción.

```
program
  begin
    integer  main()
      begin
        integer x;
        x = 8-4;
        return x;
      end
  end
```

```
BEGIN FUNCTION    (null)      (null)      main
LOAD              8           (null)      Temp 0
LOAD              4           (null)      Temp 1
SUB               Temp 0      Temp 1      Temp 2
ASSIGN            x           (null)      Temp 2
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

```
BEGIN FUNCTION    (null)      (null)      main
ASSIGN            x           (null)      4
IC_PRINT          (null)      (null)      x
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

Sentencia if(true) then .. else.

```
program
  begin
    integer  main()
      begin
        integer x;
        if (true && true) then
          begin
            x =1;
          end
        else
          begin
            x =2;
          end
        return x;
      end
  end
```

```
BEGIN FUNCTION    (null)      (null)      main
LOAD              true        (null)      Temp 0
LOAD              true        (null)      Temp 1
AND               Temp 0      Temp 1      Temp 2
IF                Label_0     Label_1     Temp 2
LOAD              1           (null)      Temp 3
ASSIGN            x           (null)      Temp 3
JUMP              (null)      (null)      Label_1
LABEL             (null)      (null)      Label_0
LOAD              2           (null)      Temp 4
ASSIGN            x           (null)      Temp 4
LABEL             (null)      (null)      Label_1
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

```
BEGIN FUNCTION    (null)      (null)      main
ASSIGN            x           (null)      1
IC_PRINT          (null)      (null)      x
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

Sentencia if(false) then .. else.

```
program
  begin
    integer  main()
      begin
        integer x;
        if (true && false) then
          begin
            x =1;
          end
        else
          begin
            x =2;
          end
        return x;
      end
  end
```

```
BEGIN FUNCTION    (null)      (null)      main
LOAD              true              (null)      Temp 0
LOAD              false              (null)      Temp 1
AND               Temp 0      Temp 1      Temp 2
IF                Label_0      Label_1      Temp 2
LOAD              1           (null)      Temp 3
ASSIGN            x           (null)      Temp 3
JUMP              (null)      (null)      Label_1
LABEL             (null)      (null)      Label_0
LOAD              2           (null)      Temp 4
ASSIGN            x           (null)      Temp 4
LABEL             (null)      (null)      Label_1
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

```
BEGIN FUNCTION    (null)      (null)      main
ASSIGN            x           (null)      2
IC_PRINT          (null)      (null)      x
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

Sentencia while(false).

```
program
  begin
    integer  main()
      begin
        integer x;
        x = 2;
        while (true&&false)
          begin
            x = x+1;
          end
        return x;
      end
  end
```

```
BEGIN FUNCTION    (null)      (null)      main
LOAD              2           (null)      Temp 0
ASSIGN            x           (null)      Temp 0
LABEL             (null)      (null)      Label_0
LOAD              true              (null)      Temp 1
LOAD              false              (null)      Temp 2
AND               Temp 1      Temp 2      Temp 3
WHILE             Label_1      (null)      Temp 3
LOAD              1           (null)      Temp 4
ADD               x      Temp 4      Temp 5
ASSIGN            x           (null)      Temp 5
JUMP              (null)      (null)      Label_0
LABEL             (null)      (null)      Label_1
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

```
BEGIN FUNCTION    (null)      (null)      main
ASSIGN            x           (null)      2
IC_PRINT          (null)      (null)      x
RETINT            (null)      (null)      x
END FUNCTION      (null)      (null)      main
```

Sentencia return.

```
program
  begin
  integer main()
    begin
      integer a, b, c, d, e;
      a = -5;
      b = 2;
      c = 10;
      d = 3;
      e = a + b;
      print e;
      return e;
      e = b + a;
      print e;
      e = a - b;
      print e;
      e = b - a;
      print e;
      e = a * b;
      print e;
      e = a * d;
      print e;
      print e;
    end
  end
```

```
BEGIN FUNCTION    (null)      (null)      main
LOAD              5           (null)      Temp 0
NEG               Temp 0      (null)      Temp 1
ASSIGN            a           (null)      Temp 1
LOAD              2           (null)      Temp 2
ASSIGN            b           (null)      Temp 2
LOAD              10           (null)      Temp 3
ASSIGN            c           (null)      Temp 3
LOAD              3           (null)      Temp 4
ASSIGN            d           (null)      Temp 4
ADD               a      b      Temp 5
ASSIGN            e           (null)      Temp 5
IC_PRINT         (null)      (null)      e
RETINT            (null)      (null)      e
ADD               b      a      Temp 6
ASSIGN            e           (null)      Temp 6
IC_PRINT         (null)      (null)      e
SUB               a      b      Temp 7
ASSIGN            e           (null)      Temp 7
IC_PRINT         (null)      (null)      e
SUB               b      a      Temp 8
ASSIGN            e           (null)      Temp 8
IC_PRINT         (null)      (null)      e
PLUS              a      b      Temp 9
ASSIGN            e           (null)      Temp 9
IC_PRINT         (null)      (null)      e
PLUS              a      d      Temp 10
ASSIGN            e           (null)      Temp 10
IC_PRINT         (null)      (null)      e
IC_PRINT         (null)      (null)      e
END FUNCTION      (null)      (null)      main
```

```
BEGIN FUNCTION    (null)      (null)      main
ASSIGN            a           (null)      5
ASSIGN            b           (null)      2
ASSIGN            c           (null)      10
ASSIGN            d           (null)      3
ADD               a      b      Temp 0
ASSIGN            e           (null)      Temp 0
IC_PRINT         (null)      (null)      e
RETINT            (null)      (null)      e
END FUNCTION      (null)      (null)      main
```

Uso de temporales.

```
program
  begin
  integer main()
    begin
      integer a, b, c;
      a = 5;
      b = 2;
      c = 22;
      return c;
    end
  end
```

```
BEGIN FUNCTION    (null)      (null)      main
LOAD              5           (null)      Temp 0
ASSIGN            a           (null)      Temp 0
LOAD              2           (null)      Temp 1
ASSIGN            b           (null)      Temp 1
LOAD              22           (null)      Temp 2
ASSIGN            c           (null)      Temp 2
RETINT            (null)      (null)      c
END FUNCTION      (null)      (null)      main
```

```
BEGIN FUNCTION    (null)      (null)      main
ASSIGN            a           (null)      5
ASSIGN            b           (null)      2
ASSIGN            c           (null)      22
RETINT            (null)      (null)      c
END FUNCTION      (null)      (null)      main
```