Taller de diseño de software

Alumno: Nicolas Fraschetti 39327355

Introducción

El proyecto consistió en desarrollar un compilador (hasta la etapa de generación de assembly) para un lenguaje de programación muy simple estilo C.
El lenguaje permite declaración de variables globales, funciones, variables locales, uso de funciones externas, comentarios de una linea y, al igual que C, exige la declaración de un método main.
A continuación se detallan la estructura del proyecto, las etapas en el desarrollo, algunas decisiones de diseño y las limitaciones del lenguaje.

Resumen de los archivos del proyecto

types.h: Contiene las declaraciones de las estructuras usadas, las principales son: Info, en donde se almacena información relativa a una variable, función o valor inmediato; ListNode: Elementos de la tabla de símbolos, variables locales, globales y funciones; TreeNode: nodos del árbol sintáctico; Cod3D: Elementos de la lista de código 3 direcciones; Op: Enumerado con todos los códigos de operaciones utilizados para generar assembly.
calc-léxico.l: Archivo flex, utilizado para realizar el análisis léxico.
calc-sintáxis.y: Archivo Bison. Se encarga de “parsear” los tokens enviados por calc-lexico y, en base a la gramatica, construir el arbol sintáctico del programa.
A su vez, realiza también algo de análisis semántico (por ejemplo, detecta si un “statement” quiere usar una variable no declarada).
ast_utilities: Provee los métodos para la construcción del árbol sintáctico. Es usado por calc-sintaxis para construir el arbol.
Provee también un método para construir un archivo dot con el arbol generado, y el método que realiza el chequeo semántico del programa previo a generar la lista cod3D y el assembly.
symbol_table_utilities: Provee las funcionalidades relativas a la tabla de símbolos. Variables globales y funciones tienen “vida” durante todo el programa, las variables locales son removidas de la tabla cuando culmina su bloque.
cod_3D_list_utilities: Provee las funcionalidades relativas a la creación de la lista de código 3 direcciones, y a la de generación de código assembly.
offset_generator: Se encarga de generar los offsets adecuados para variables locales y parametros.

Etapas

1. **Análisis léxico, sintáctico y obtención del árbol sintáctico:** Las herramientas utilizadas para leer el programa fuente y obtener el AST fueron flex (para obtener los “tokens” del programa fuente) y bison (para procesar dichos tokens y estructurarlos de acuerdo al lenguaje diseñado). Para la obtención del AST se usó el archivo auxiliar ast_utilities.c, y se fue construyendo en forma ascendente al procesar las distintas reglas en bison. Del mismo modo, se usó una tabla de simbolos (symbol_table_utilities.c) para llevar informacion de las variables (solo nombre hasta ese momento). Cada nodo correspondiente a una variable tiene una referencia al mismo campo Info, que el que tiene la variable en la tabla de simbolos.
   Hasta el momento, el lenguaje permitía solo variables enteras, y operaciones como
   suma y resta. Ademas, era un especie de intérprete, ya que podía evaluar expresiones una a una.

2. **Generación de la lista de codigo 3 direcciones:**
   Luego de haber implementado las funcionalidades para obtener el árbol del programa fuente (y de extender el lenguaje con otras operaciones como multiplicación, división, modulo e instrucciones para imprimir) se comenzó a diseñar las de la lista de código 3 direcciones. Esta lista sirvió luego como un código intermedio entre el árbol y el assembly generado. Para esto se crea otro archivo C (Cod3D_list_utilities), y se implementa un método que obtiene la lista tomando como parámetro de entrada el árbol.
   Cada elemento de la lista tiene 4 componentes: El código de operación, y 3 componentes de tipo info. Estos pueden ser referencias directas a componentes de un nodo o a nuevos generados para guardar el resultado de una operación. De acuerdo al código de operación, estos componentes de tipo

Info podían tener información almacenada o ser nulos.

3. **Generacion de código assembly:** Luego de tener construida la lista de codigo de 3 direcciones se comenzó finalmente con la generación de codigo assembly. La lista de cod 3D resulta particularmente util debido a que cada instrucción (elemento de la lista) puede ser traducida directamente a una instrucción de assembly.
   Aquí es donde cobró importancia que cada variable tenga un offset para poder identificarla en assembly. Debido a esto se debieron hacer modificaciones en el tipo Info, en los métodos de inserción en la tabla de símbolos, y se agregó un archivo nuevo que se encargó exclusivamente de la generación de offsets. El método que se encarga de tomar un AST, generar la lista cod3D y, a partir de esta, generar el codigo assembly se encuentra también en el archivo cod3D_list_utilities.

4. **Primera extension del lenguaje: tipo booleano, posibilidad de declarar variables booleanas y nuevas operaciones que retornan un valor booleano (menor, mayor, igual, and , or, not):** Se debió extender la gramática, y los métodos de generación de cod 3 direcciones y assembly. Para el campo Info de los nodos booleanos, el valor 1 se corresponde con verdadero, y 0 con falso.

5. **Segunda extension del lenguaje: Sentencias de control if, if else y while, y aparición de la noción de un scope para las variables:** Aquí cobró importancia la noción de alcance de una variable, y esto obligó a extender las funcionalidades de la tabla de símbolos. Afortunadamente, la había implementado como una pila, por lo que no tuve que modificar demasiado los métodos que tenia. Lo único que cambió fue la introducción del concepto de nivel, un

método que busque si ya había una variable en ese nivel (en caso de declarar dos variables iguales en un mismo scope) para informarlo, y limpiar las variables de un nivel una vez que se salió de éste.

6. **Ultima extensión: Declaración y definición de funciones, variables globales y funcion main:** Se debió modificar considerablemente la gramática, ya que el lenguaje pasó a ser un conjunto de funciones, siendo la ultima de ellos la función main. A nivel de assembly esto provocó la aparición de muchos "frames" distintos (antes había uno solo), por lo que el modo de tratar las variables hasta el momento se hizo solo válido para variables locales.
   Esto obligo a extender la gramática también para las variables globales, y a modificar un poco la generación de offsets para los parámetros y variables locales.

7. **Chequeo de tipos:** Tener un método que realice un chequeo de tipos se hizo necesario cuando se introdujo el tipo booleano. Ésto obligo a tener un método que chequee por ejemplo, que a una variable entera se le asigne una expresión de tipo entero.
   En las siguientes etapas este método debió ser extendido: Con la aparición de sentencias de control había que chequear, por ejemplo, que las condiciones sean lógicas. Con la aparición de funciones hubo que empezar a chequear que la lista de parámetros formales tenga la misma aridad que la de los actuales, y que a su vez, tengan el mismo tipo uno a uno.

Algunas decisiones de diseño y limitaciones:

1. El lenguaje es estilo C, en el sentido que debe tener una función main de tipo int (sin parámetros), el inicio y fin de cada bloque se indica con { y } respectivamente y los operadores definidos son los mismos (&& para and, || para or, etc.)
2. Algunas diferencias: La declaración de variables debe tener la palabra reservada var antes. Ademas se permiten solamente comentarios de una linea (//).
3. El lenguaje no permite la declaración de bloques aislados. Cada bloque tiene que corresponderse a una función, o a una sentencia de control.
4. En un mismo bloque, primero deben estar las declaraciones y después las sentencias, NO se puede declarar una variable después de, por ejemplo, haber llamado a una función.
5. Las funciones deben si o si estar declaradas con un tipo de retorno int o bool. El lenguaje no permite void.
6. Las variables globales pueden estar inicializadas sólo con un valor inmediato (entero o lógico). Por ejemplo, una variable global no puede ser inicializada con una suma.
7. Las variables globales están inicializadas por defecto con 0 (falso para los booleanos). Es decir, a diferencia de las variables locales, el compilador no alerta si se quiere usar una variable global a la que no se le ha asignado un valor.
8. El compilador NO chequea nada acerca de la instrucción return en una función. (es algo que me quedó pendiente). Ésta puede estar o no, por lo que pueden haber errores en tiempo de ejecución en caso de olvidarlo. A su vez, tampoco se chequea nada respecto del tipo de retorno. Es decir, una funcion puede declarar devolver un entero, y sin embargo en el return devolver una variable booleana.
9. Las variables globales deben ser declaradas antes de las funciones.

10. Una función puede tomar cualquier número de parámetros.
11. Un parámetro puede ser sobrescrito por una variable local.

Ejemplos sobre las restricciones explicadas anteriormente:

3. A diferencia de otros grupos, este compilador no permite bloques aislados. El siguiente código, que me pasaron para hacer los test, no compila por eso:

```
1   extern int printi(int x);
2   // retorna el factorial de v, con v hasta 15
3   int factorial (int v){
4     var int limit;
5     limit = 15;
6     if ( v > limit){
7       return -1;
8     }
9     {//nuevo bloque
10      var int c;
11      var int fact;
12      c = 0;
13      fact = 1;
14      while (c<v){
15         c = c+1;
16         fact = fact*c;
17      }
18      return fact;
19    }
20  }
21  int main(){
22      return printi(factorial(5));
23  }
```

Al correrlo devuelve:

```
-> ERROR Sintactico en la linea: 9
```

La solución del problema fue introducir un if para que pueda crear un nuevo bloque:

```
1   extern int printi(int x);
2   // retorna el factorial de v, con v hasta 15
3   int factorial (int v){
4     var int limit;
5     limit = 15;
6     if ( v > limit){
7       return -1;
8     }
9     if (true){//nuevo bloque
10      var int c;
11      var int fact;
12      c = 0;
13      fact = 1;
14      while (c<v){
15         c = c+1;
16         fact = fact*c;
17      }
18      return fact;
19    }
20  }
21  int main(){
22      return printi(factorial(5));
23  }
```

resultado: 120

4. Las declaraciones deben ir antes de las sentencias. Por ejemplo el siguiente código falla:

```
1   extern int printi(int x);
2   int main(){
3       var int n;
4       n = 100;
5       var int m = n;
6       return printi(m);
7   }
-> ERROR Sintactico en la linea: 5
```

Para solucionarlo hay que declarar m antes:

```
1   extern int printi(int x);
2   int main(){
3       var int n;
4       var int m;
5       n = 100;
6       m = n;
7       return printi(m);
8   }
100
```

6. Las variables globales pueden estar inicializadas sólo con valores inmediatos. El siguiente código falla:

```
1   var int n = 2 * 100;
2   int main(){
3       return 0;
4   }
-> ERROR Sintactico en la linea: 1
```

10. Las funciones pueden tomar cualquier número de parámetros:

```
1   var int n;
2   extern int printi(int x);
3   int foo(int a, int b, bool c, bool d, int e, int f, int g, int h, int i, int j){
4       if (c && d){
5           return f + g + h + i + j;
6       }
7       else{
8           return -1;
9       }
10  }
11  int main(){
12      var bool x = true;
13      return printi(foo(1,2,true,x,1,1,10,2,5,21));
14  }
```

resultado: 39

11. Un parámetro puede ser sobrescrito:

```
1  var int n = 100;
2  extern int printi(int x);
3  int main(){
4      var int n = n + 1;
5      return printi(n);
6  }
```

resultado: `101`

Consideraciones finales

La forma incremental de trabajar en la materia me parece adecuada. Las mayores complicaciones las tuve al principio (me costo bastante entender a flex y bison, y la forma en que se comunicaban entre si), y al momento de implementar los distintos niveles de visibilidad de las variables, en el que por un mal análisis de la solución, perdí mas tiempo del esperado.
Otra dificultad que tuve fue lograr comprender la sintaxis del assembly de gnu, distinta a la de nasm que conocía.
No considero que la ultima etapa, la de introducir funciones, sea un cambio demasiado brusco, y de hecho, tuve menos complicaciones que las esperadas (fue bastante directo el reuso de las funcionalidades desarrolladas previamente).
Hubo algunas funcionalidades que me hubiese gustado que el lenguaje tenga, como la declaración de funciones que retornen void y su consecuente llamada “libre”, sin tener que estar atado a asignar su valor a una variable, o procesarlo de alguna forma; y la posibilidad de declarar variables en cualquier parte del código, algo que considero demasiado restrictivo de este lenguaje.