Diseño de compiladores 1

Informe Compilador

Trabajo Práctico 3 y Trabajo Práctico 4

Grupo 26

Alvarez, Maximiliano

maxi25294@gmail.com

Attilio, Nicolas

nattilio@alumnos.exa.unicen.edu.ar

Gonzalez, Francisco

gonzalezfranl10@gmail.com

Introducción

El objetivo de este trabajo es realizar un compilador completo. En esta segunda entrega, nos enfocaremos en la generación de código intermedio y la salida a Assembler.

Para la generación de código intermedio, se nos asignó como estructura Tercetos, esta posee un operador, y dos operandos, de la forma: (Operador, Operando 1, Operando 2).

Para la generación de código Assembler se utilizará el mecanismo de variables auxiliares a lo largo de todo el proyecto, todo esto se realizará para la ejecución en Pentium de 32 bits.

Descripción de la Generación de Código Intermedio

Estructura utilizada para el almacenamiento del código intermedio

Para la generación del código intermedio se implementó la clase Terceto que posee la información individual de cada Terceto necesario, por ejemplo, para una suma como es a + b, almacenaría (operador, operando 1, operando 2), siendo (+,a,b). Además de estos datos también se almacena el número correspondiente al lugar que ocupa en el orden de Tercetos generados, el resultado de la operación (se utilizará en Assembler) y el tipo del resultado.

Para administrar y contener a todos los Tercetos necesarios, se utiliza la clase AdministradorTercetos la cual tiene una ArrayList de Tercetos, de esta manera se asigna el número de cada Terceto correspondiente a la posición que tiene en la lista. Dentro de esta clase también se implementó un ArrayList de Integers para simular la pila utilizada para las bifurcaciones. En esta pila se almacenan los números (referencias) de los Tercetos incompletos para luego desapilarlos y completarlos.

Uso de notación posicional de Yacc ($$, $n)

Se utilizó la notación posicional de Yacc dentro de la gramática para poder acceder dentro de ella a los operandos y operadores de cada una de las reglas. Siendo $$ el valor del no terminal que se está definiendo, es decir, el valor “a la izquierda” de la gramática y $n, siendo n la posición del elemento “a la derecha” en la gramática.

- **`Declaración : tipo lista_de_variables ‘;’`**
    - Para tomar el tipo de la declaración y setearlo en la tabla de símbolos en caso de que el lexema no exista.Se utilizó $1.sval p
    - Para saber si el nuevo lexema existe o no en la tabla de símbolos, se debe recorrer la lista_variables, que se toma utilizando (ArrayList<String>)$2.obj e iterando sobre ella.

- TYPEDEF

  Declaracion: TYPEDEF IDENTIFICADOR '=' tipo funcion_type ';'

  - Se usa $2.sval para agregar el lexema en la tabla de símbolos, seteando su uso como un “TypeDef” y su tipo como $4.sval.

  funcion_type: FUNC '(' tipo ')' ';' bloque_type

  - Se chequea mediante el $3.sval y $6.sval que el tipo y el bloque_type no sean nulos.
  - Se retorna el valor de bloque_type mediante $$ = new ParserVal($6.sval)

  bloque_type: IDENTIFICADOR lista_de_variables

  - Por medio del $3.sval se accede a todas las variables, y para cada una se agrega a la tabla de símbolos junto con su ámbito. Además al agregarlo se seteó el tipo con el $1.sval.
  - Se retorna el nombre del tipo, es decir el identificador, por medio de $$ = new ParserVal($1.sval)

- **tipo: ULONG**

  | DOUBLE

  - Se utiliza tanto para ULONG y DOUBLE la notación posicional $$ = new ParserVal(“LONG”) -> “Para retornar como salida de la regla el string del tipo correspondiente.”

- REPEAT

  control: REPEAT'(' asignacion_repeat ';'condicion_repeat';' CTE_ULONG')' bloque_repeat

- Se utiliza $3.sval y $5.sval para acceder a la asignación y la condición respectivamente y verificar que sean distintas de NULL.
- Se utilizan $3.sval y $7.sval para formar el terceto que incrementará el iterador.

asignacion_repeat: IDENTIFICADOR ASIGNACION CTE_ULONG

- Se utiliza $1.sval para verificar si el Identificador es alcanzable en el ámbito, es decir, si se puede utilizar la variable en el ámbito actual;
- Se utiliza $3.sval para crear el nuevo terceto de asignación, con el ámbito previamente mencionado y la constante.
- Se devuelve $$ = new ParserVal(ambitoVariable), siendo ambitoVariable el lexema en la tabla de símbolos (variable@ambito)

condicion_repeat: expresion1

- Se utiliza el $1.obj para verificar que sea válido, en caso de que sea válido se arma el terceto de bifurcación por falso “BF”, se agrega el terceto a la lista, se apila el terceto incompleto y se devuelve el valor de expresion1 con $$, en caso de que no sea válido devuelve un valor por defecto con $$.

- **asignacion : IDENTIFICADOR ASIGNACION expresion2**
  - Se utiliza $1.sval para verificar si el Identificador es alcanzable en el ámbito.
  - Se guarda el operando correspondiente con $3.obj, si es válido, se crea el terceto de asignación y se agrega a la lista. Si no es válido, se devuelve un error.

- IF

```
seleccion: IF condicion_if  THEN bloque_then ENDIF

    | IF  condicion_if  THEN bloque_then ELSE bloque_if
```

  - Se utiliza $2.sval para verificar si la condición no es nula. Si no lo es, se desapila el terceto.

```
condicion_if: '(' condicion ')'
```

  - Se utiliza $2.sval para verificar que la condición sea válida, en caso de que sea válida se arma el terceto con la bifurcación por falso, se agrega a la lista de tercetos y se apila.
  - $$ = new ParserVal($2.sval) se retorna la condición para el posterior chequeo y desapilado.

- **impresion: PRINT '(' CADENA ')' | PRINT '(' ')'**
  - Se utiliza en el primer caso $3.sval para armar el terceto de impresión con la cadena, en el segundo caso no se utiliza porque se imprime la cadena vacía.

- **invocacion : IDENTIFICADOR '(' CTE_ULONG ')'**

```
        | IDENTIFICADOR '(' CTE_DOUBLE ')'

        |IDENTIFICADOR '(' IDENTIFICADOR ')'
```

  - Se utiliza $3.sval para ver si la constante es válida, si lo es, utiliza $1.sval para verificar el ámbito como en los casos anteriores. En caso de que el ámbito sea alcanzable y el tipo corresponda, se genera el terceto de invocación.
  - Se retorna $$ = new ParserVal($1.sval) , es decir el nombre de la función invocada (IDENTIFICADOR) para luego buscarlo en la tabla de símbolos .

- **lista_de_variables : lista_de_variables ',' IDENTIFICADOR |IDENTIFICADOR**
    - Se toma la lista de variables con el casteo (ArrayList<String>) y el uso posicional $1.obj, se agrega el identificador a la lista, tomando el identificador con $3.sval, y se devuelve con el $$.
    - En caso de que sea un solo identificador, simplemente se agrega con el $1.sval a la lista y se devuelve la lista con el $$.

- Función

  funcion : declaracion_funcion bloque_funcion

    - Chequea que haya una declaración y un bloque por medio de $1.sval y $2.sval, y si ambos no son nulos agrega el terceto correspondiente al fin de función. El nombre de la función se obtiene también del $1.sval, y es lo que finalmente retorna con el $$.

      declaracion_funcion: FUNC IDENTIFICADOR'('parametro')'

    - Se chequea que el nombre de la función ($2.sval) no se encuentre en la tabla de símbolos, y en caso de que así sea se agrega (junto con el ambito) y se crea un terceto indicando el comienzo de esta.
    - Se usa el $$ para retornar el lexema de la función que se agregó.

      parametro : tipo IDENTIFICADOR

    - Se agrega el parámetro a la tabla de simbolos utilizando el $2.sval y agregando el ámbito. El tipo se setea mediante el $1.sval.
    - Se retorna mediante $$ el identificador ($2.sval)

      bloque_funcion : bloque_declarativo bloque_ejecucion_funcion

      | bloque_ejecucion_funcion ..

- Como puede no haber declaraciones, solo se chequea que el bloque de ejecución no sea NULL (con $2.sval), si esto se cumple se retorna este con $$.

- Función Type

  funcion_type: FUNC '(' tipo ')' ';' bloque_type

  - Se verifica que el tipo ($3.sval) y el bloque_type ($6.sval) sean distinto de NULL y, si eso se cumple, retorna el valor del bloque_type con $$.

  bloque_type: IDENTIFICADOR lista_de_variables

  (Ya explicado anteriormente)

  bloque_funcion : bloque_declarativo bloque_ejecucion_funcion ..

  | bloque_ejecucion_funcion ..

  (Ya explicado anteriormente)

bloque_ejecucion_funcion: BEGIN bloque_sentencias RETURN '('condicion')' ';' END

| BEGIN pre_condicion ';' bloque_sentencias RETURN '('condicion')' ';' END

| BEGIN pre_condicion ';' RETURN '('condicion')' ';' END

- Evalúa que la condición no sea nula con el $n.sval, siendo n la posición de la condición. La retorna mediante $$.

- **pre_condicion: PRE ':' '(' condicion ')'**
    - Se verifica que la condición no sea NULL. Si esto se cumple agrega el terceto incompleto de la bifurcación por falso y apila el número de este.
    - Mediante $$ retorna el valor de la condición.

- **condicion : expresion**

  | condicion OR expresion

    - Verifica que expresión no sea nulo ($1.sval en el primero y $3.sval en el segundo)
    - Para el segundo caso agrega el terceto correspondiente a la operación OR con los operandos $1.sval y $3.sval. Retorna mediante $$ el número del terceto agregado.

- **expresion: expresion1**

  | expresion AND expresion1

    - Verifica que expresión no sea nulo ($1.sval en el primero y $3.sval en el segundo)
    - Para el segundo caso agrega el terceto correspondiente a la operación AND con los operandos $1.sval y $3.sval. Retorna mediante $$ el número del terceto agregado.

- **expresion1: expresion2**

  | expresion1 comparador expresion2

    - Verifica que expresión no sea nulo ($1.sval en el primero y $3.sval en el segundo)

- Para el segundo caso agrega el terceto correspondiente a la operación utilizando $2.sval (ya sea <, >, <>, <=, >=) con los operandos $1.sval y $3.sval. Retorna mediante $$ el número del terceto agregado.

- **expresion2: termino**

  | expresion2 '+' termino ..

  | expresion2 '-' termino

  - Verifica que expresión no sea nulo ($1.sval en el primero y $3.sval en el segundo)
  - Para el segundo y tercer caso agrega el terceto correspondiente a la operación $2.sval (suma o resta) con los operandos $1.sval y $3.sval. Retorna mediante $$ el número del terceto agregado.

- **termino : factor**

  |termino '*' factor

  | termino '/' factor

  - Verifica que expresión no sea nulo ($1.sval en el primero y $3.sval en el segundo)
  - Para el segundo y tercer caso agrega el terceto correspondiente a la operación $2.sval (división o multiplicación) con los operandos $1.sval y $3.sval. Retorna mediante $$ el número del terceto agregado.

- **factor : '-' factor**

  | CTE_DOUBLE

  | CTE_ULONG

```
| IDENTIFICADOR

| invocacion
```

- Se chequea en todos que el operador $1.sval ($2.sval para el primero) no sea nulo y lo retorna con $$.

- **comparador :** `'<'`

```
| '>'

| IGUAL_IGUAL

| MAYOR_IGUAL

| MENOR_IGUAL

| DISTINTO
```

  - En todos los casos se retorna el string con el valor del comparador, utilizando $$.

Algoritmo usado para la generación de las bifurcaciones en sentencias de control

Para la generación de las bifurcaciones en sentencias de control (en nuestro caso el Repeat) se siguió el siguiente esquema de generación de tercetos, tanto para los del bloque interno de la sentencia como para los de la condición, saltos, etiquetas e incrementos:

1. Terceto de asignación
2. Label para destino del salto incondicional
3. Terceto comparación
4. Terceto salto por falso (hacia punto 8.)
5. Tercetos de bloque interno de for

6. Terceto de incremento
7. Terceto salto incondicional hacia la comparación (punto 2.)
8. Label para destino del salto por falso

Como puede apreciarse en los pasos, durante la generación de Tercetos se agregan las etiquetas, evitando así hacer más pasadas.

Inicialmente se agrega el terceto de la asignación del Repeat. Luego se genera una etiqueta que será utilizada como dirección de destino del salto incondicional para generar el bucle de la sentencia de control. Al volver se vuelve a controlar la condición. La dirección de la etiqueta se almacena en una lista que se utiliza a modo de pila, y de esta manera al generar el terceto de salto incondicional se busca en dicha estructura la dirección a la etiqueta de la condición del Repeat.

Luego se genera el terceto de comparación y, de forma incompleta, el salto por falso; es decir,se crea un terceto que en la primera posición tiene el salto por falso (BF), en la segunda el número de terceto propio de la comparación y la tercera posición queda sin completa. La dirección de dicho terceto incompleto se almacena en un arreglo dinámico donde luego se irá a buscarla para ser completada en el momento de conocer el final del bloque de sentencias.

Después de la cabecera del repeat se agregan los tercetos correspondientes a las sentencias de ejecución del cuerpo de este, agregando al final el terceto de incremento.

Finalmente, se genera el terceto de salto incondicional, el cual se completa con la dirección al label de comparación que, tal como se citó anteriormente, se obtiene de la pila de enteros donde se guardan direcciones de tercetos a los que se necesita saltar o completar.

Nuevos errores considerados

Para los siguientes errores, al detectarlos se los agrega a una lista (junto con los errores léxicos y sintácticos) que luego se imprime por pantalla. Si esta lista está vacía se puede proceder a realizar el código assembler.

- Operaciones entre tipos distintos
- Uso de variables, o invocación de funciones que no están al alcance
- Uso de variables o funciones que no fueron declaradas
- Invocación a una función utilizando un parámetro de distinto tipo
- Redeclaración de funciones y/o funciones en un mismo ámbito

Descripción de la Generación de Código Assembler

Mecanismo utilizado para la generación de Código Assembler

Para la generación del código assembler creamos una clase “Assembler” que se encarga de crear un archivo con extensión .asm, este posee todas las instrucciones del código fuente.

Si bien el .data debe ir antes que el .code en el archivo .asm, se genera primero el .code recorriendo la lista de tercetos generada en la formación del código intermedio pasando por un case que analiza, dependiendo el operador, las instrucciones de assembler que se irán agregando al archivo. Una vez finalizadas las instrucciones del los tercetos, se genera el .data con los datos de la tabla de símbolos que también poseen las variables auxiliares utilizadas.

Se declaran también las etiquetas de control y constantes necesarias para el control en tiempo de ejecución del assembler, dado que estas no están en la tabla de símbolos, ni en la lista de tercetos generada por la gramática.

Mecanismo utilizado para efectuar cada una de las operaciones aritméticas

Para realizar las operaciones aritméticas de la consiga se tuvo en cuenta la operación y las posibilidades en cuanto al órden del terceto, es decir, si era entre registro y registro, variable y registro o registro y variable, etc. Como indica la consigna se realizaron las operaciones utilizando variables auxiliares para ambos tipos. Por último, las operaciones de DOUBLE se realizaron utilizando el co-procesador 80x87.

Para todas las operaciones suma y resta se utilizó el registro EBX. Lo que se realizó con las sumas y restas en ULONG fue guardar el valor de uno de los operandos en el registro y luego sumar/restar el valor de ese registro con el otro operando, teniendo en cuenta para las operaciones no conmutativas el orden dado. Luego de esto se realizaron los chequeos para evitar el overflow en tiempo de ejecución cuando corresponda.

Para la multiplicación y división se realizó algo similar, en la división se controla en tiempo de ejecución la división por 0. En este caso se utiliza el registro EAX para estas operaciones, y para las operaciones de tipo DOUBLE, al igual que en las sumas y restas, se utilizó el co-procesador 80x87.

Mecanismo utilizado para la generación de las etiquetas

En cuanto a la generación de etiquetas, se utilizaron para las sentencias de control y selección, en nuestro caso para el IF y el REPEAT. Sin embargo, también se utilizaron para los controles y avisos en tiempo de ejecución como por ejemplo en el overflow en sumas. El código assembler deberá controlar el resultado de la operación en caso de que en la comparación el número de tipo DOUBLE sea mayor al límite, se saltará al final con un mensaje de error, sucede lo mismo para el caso de las divisiones por 0.

Modificación a las etapas anteriores

Los cambios en relación a las etapas anteriores fueron realizados mayormente en la gramática. Dichos cambios son:

- Manejo de etiquetas/labels y bifurcaciones (por falso BF o incondicionales BI) para sentencias de selección y de control.
- Modificaciones necesarias para la aplicación de Name mangling, tales como modificación de nombre en símbolos y actualizaciones de alcance, para obtener referencia del ámbito correspondiente a cada identificador.
- Chequeos de declaraciones de variables y procedimientos.
- Chequeos de igualdad de tipos.

Resolución de temas particulares (17 a 25)

Tema 18: Precondiciones con CATCH.

A continuación de los encabezados de declaraciones de funciones, se podrá incluir una precondición con la siguiente estructura:

PRE: (<condición>);

Ejemplo Declaración:

```
FUNC INT f(INT x)

SINGLE y, z;

BEGIN

PRE: (x > 3);

RETURN (x + 1);

END;

Ejemplo Invocación:

TRY
```

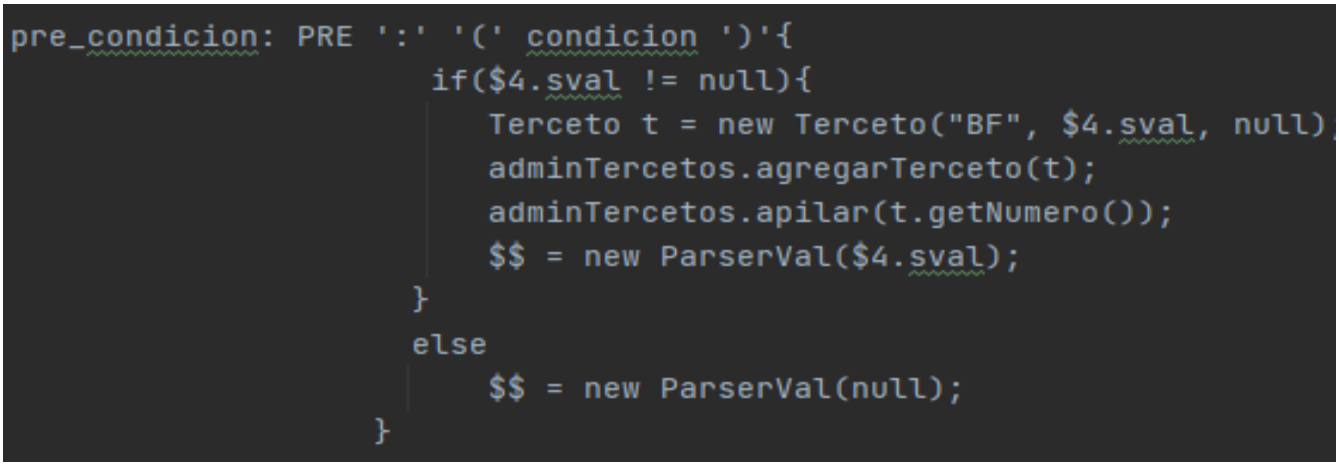
```
x := f(z) * 4;

CATCH

BEGIN

x := 10;

END;
```

Código intermedio:

Se deberá generar código para la evaluación de la precondición, considerando que si no se cumple, se debe abortar la ejecución de la función, y ejecutar el cuerpo del CATCH.

Luego, se debe continuar con la ejecución del resto del programa.

Para lograr esto, en la gramática, en la declaración de la precondición se crea y agrega un terceto incompleto de bifurcación por falso (BF), que va a ser desapilado posteriormente (en este caso en la ejecución del TRY). El desapilado incluye también la creación del terceto que contiene la etiqueta del salto (esto último para el assembler). Así mismo en la sentencia TRY se agrega una bifurcación incondicional (BI) para el caso de que la sentencia del TRY no falle y no tenga que entrar al CATCH, es decir, se lo salte.



```
pre_condicion: PRE ':' '(' condicion ')'{
                    if($4.sval != null){
                        Terceto t = new Terceto("BF", $4.sval, null);
                        adminTercetos.agregarTerceto(t);
                        adminTercetos.apilar(t.getNumero());
                        $$ = new ParserVal($4.sval);
                    }
                    else
                        $$ = new ParserVal(null);
                }
```

A continuación se muestra un ejemplo de una ejecución:

```
Contenido del archivo:

pruebaTryCatch
ULONG x,z;

ULONG FUNC f (ULONG a)
ULONG y;
BEGIN
PRE: (a < 3);
RETURN (a+1);
END;

BEGIN
x:=11;
x:= 12;
TRY
x := f(z) + 4;
CATCH
BEGIN
x := 10;
END;

z:=6;

END
```

```
------ CÓDIGO INTERMEDIO ------
6. (:=, x@main, 11)
7. (:=, x@main, 12)
8. (InvocacionFuncion, f@main, null)
9. (:=, a@main@f, z@main)
0. (ComienzaFuncion, f@main, null)
1. (<, a@main@f, 3)
2. (BF, [1], 13)
3. (+, a@main@f, 1)
4. (RetornoFuncion, [3], null)
5. (FinFuncion, f@main, null)
9. (:=, a@main@f, z@main)
10. (+, [4], 4)
11. (:=, x@main, [10])
12. (BI, 15, null)
13. (Label13, null, null)
14. (:=, x@main, 10)
15. (Label15, null, null)
16. (:=, z@main, 6)
```

Tema 23 : Asignación de funciones a variables con TYPEDEF.

Sintaxis:

TYPEDEF ID = <encabezado_de_funcion>;

Ejemplo declaración:

```
TYPEDEF f1 = INT FUNC (INT);

f1 x, y, z;

INT a, b, c;

FUNC INT f(INT p)

BEGIN

p := p +1;

RETURN (p);

END;

Ejemplo uso:

x := f;

z := x;

a := x(b) + z(3);
```

Para poder permitir este tema particular se agrega una modificación en los datos de la tabla de símbolos, más específicamente se agrega un campo llamado FuncionReferenciada, que inicialmente está vacío y se completa cuando a una variable TypeDef se le asigne una función correspondiente a su tipo.

Al detectar variables que sean de tipo TYPEDEF, como es el caso de a, b y c en el ejemplo anterior se agregan en la tabla de símbolos seteando su uso como “ VariableTypeDef ”. Luego, cuando estas

variables se incluyan en asignaciones, o en otras sentencias de ejecución se chequea que el campo “FuncionReferenciada” contenga en nombre de una función válida y ya declarada. Para evitar que cualquier variable pueda setear dicho campo, siempre se controla que su uso sea “VariableTypeDef”.

Conclusiones

En esta parte del trabajo pudimos poner en práctica los conceptos teóricos vistos a lo largo de la materia, pudimos desarrollar el código intermedio y posteriormente la transcripción a código assembler.

Como grupo nos pareció una idea interesante ver y entender cómo funciona un compilador, debido a que durante la carrera utilizamos varios y por fin podemos tener un panorama bastante amplio de su funcionamiento interno.

Creemos que fue una experiencia positiva debido a que, a pesar de las dificultades, pudimos resolver los problemas que se nos pusieron en el camino.