TALLER DE DISEÑO DE SOFTWARE

2018

Gaido Laureana - González Gabriel

Introducción

Los compiladores son programas de computadora que traducen de un lenguaje a otro. Un compilador toma como entrada un programa escrito en lenguaje fuente y produce un programa equivalente escrito en lenguaje objeto. Generalmente al lenguaje fuente se lo asocia como lenguaje de alto nivel, mientras que al lenguaje objeto se lo conoce como código objeto escrito específicamente para una maquina objeto. A lo largo del proceso de traducción, el compilador debe informar la presencia de errores en el lenguaje fuente.

Diseñar y desarrollar un compilador, no es tarea fácil pero como los compiladores se utilizan en casi todas las formas de la computación y cualquiera involucrado en esta área debería conocer la organización y el funcionamiento básico de un compilador nos involucramos en este proyecto que consiste en implementar un compilador para un lenguaje imperativo simple, similar a C o Pascal.

El proyecto consiste en el diseño e implementación de un compilador para un lenguaje de programación simple, denominado TDS18.

Análisis léxico sintáctico

El Analizador Léxico toma como entrada un archivo con código fuente TDS18 y retorna tokens. Un token representa a una clase de símbolos del lenguaje.

El Analizador Sintáctico, toma como entrada la secuencia de tokens y verifica que esta secuencia sea una secuencia válida, es decir, que cumpla con la especificación sintáctica del lenguaje. La verificación controla que, por ejemplo, los paréntesis y llaves estén balanceados, la presencia de operadores, etc. Verificaciones de tipado, nombres de variables y funciones no son realizados en esta etapa. La salida de esta etapa puede ser el árbol sintáctico (o de parsing) o simplemente si la entrada es correcta o no (sintácticamente). La gramática (especificación sintáctica) del lenguaje TDS18 se presenta en otro documento. Es necesario separar la especificación del Analizador Léxico de la especificación del Analizador Sintáctico.

Las herramientas usadas para realizar estas actividades son: lex/flex y yacc/bison.

Lo primero que hicimos fue definir todas las palabras reservabas en el Yacc (tokens) ya que es mucho más prolijo trabajar con tokens que con los símbolos propios, sobretodo más legible cuando se deban pasar como parámetro. Primeramente habíamos trabajado todo con los símbolos, pero lo cambiamos por lo dicho anteriormente. Luego empezamos a modificar la parte de expresiones que se había realizado en el preproyecto, adaptándolo a esta nueva parte. Aquí empezaron los problemas. Surgieron varios conflictos shift-reduce, algunos por falta de precedencia explicita y otros por problemas de implementación de Yacc, que fuimos arreglando cambiando de lugar la recursión, agrupando gramáticas, estableciendo precedencia (%prec).

Seguimos con el Lex, también modificando el del preproyecto, agregando las reglas que nos hacían falta.

En esta parte no hubo mucha división de trabajo, se trabajó conjuntamente y cooperativamente en el Análisis léxico sintáctico en general, para lo único que hubo división, fue para realizar los test.

El main se trató con la gramática, porque nos pareció correcto hacerlo ahora para testear los programas completos. Contemplamos que haya un solo main, por cómo se definió la gramática.

Análisis Semántico

Esta etapa verifica las reglas semánticas del lenguaje, compatibilidad de tipos, visibilidad y alcance de los identificadores, etc. En esta etapa se implementó una tabla de símbolos para mantener la información de los símbolos (identificadores) de un programa. El análisis semántico se realiza sobre el árbol de parsing, utilizando la información almacenada en la tabla de símbolos.

Para la implementación de la tabla de símbolos, se utilizó una pila (stack) en donde la estructura implementada, no sigue estrictamente el comportamiento de una pila. Cada elemento de la pila, es de tipo data_stack, que se definió de la siguiente manera:

data: Este campo nos permite representar tanto datos de variables como de funciones (su nombre, tipo de retorno, etc).
es_extern: Nos indica si una función es externa o local.
tipoOp: Campo utilizado para representar datos en el árbol y saber qué tipo de operación se va a realizar.
tipo: Tipo de la variable (integer, bool) o tipo de retorno (VOID, integer, bool) en caso de ser una función
linea: Numero de línea.
es_funcion: Nos indica si el data_stack esta representando una función.
nParams: En caso de representar una función, este campo nos indica la cantidad de parámetros formales de la misma.
stack_size: Nos indica el tamaño del stack utilizado por la función, entre parámetros y variables locales (luego se agregan los temporales).
params: Lista de parámetros actuales de una función.
formalParams: Lista de parámetros formales de una función
block: Cuerpo de la función representado con un árbol.
next: Puntero al próximo elemento del scope.

```
typedef struct data_stacks{
  data_gen *data;
  bool es_funcion;
  bool es_extern;
  int tipoOp;
  int nParams;
  int stack_size;
  node *block;
  paramList *params;
  formalParam *formalParams;
  struct data_stacks *next;
} data_stack;
```

El scope se manejó, buscando el id de elemento, primero, en el nivel en el que se está (los niveles fueron enumerado) y si no se encuentra en dicho nivel, se procede a realizar una búsqueda por niveles. Para ello se realizó una función buscar_por_niveles, a la cual se le pasa el nivel actual y si allí no encuentra el id, realiza la búsqueda en los niveles que contienen a dicho nivel.

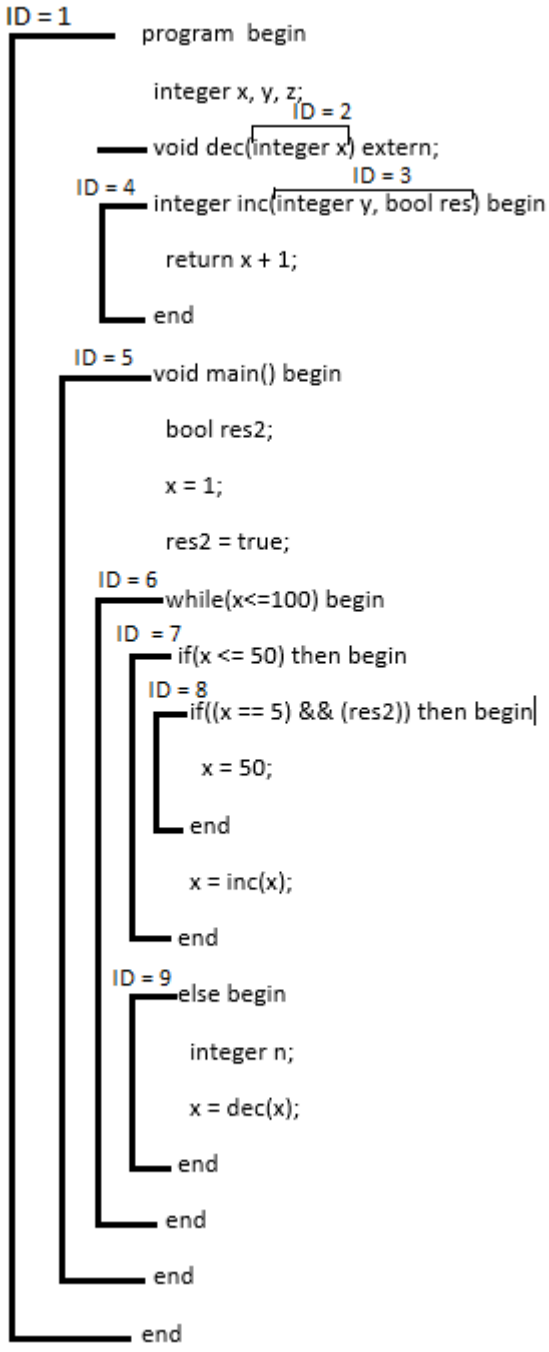
Decidimos implementar el Scope (alcance o visibilidad de variables) mediante niveles (representados por una lista), dentro de cada nivel, tendremos nuevamente una lista, que representa a todos los identificadores visibles dentro del nivel. Para representar el alcance dinámico (cuando una variable no se encuentra dentro del nivel corriente, se hace una búsqueda en los niveles que lo contienen), cada nivel contiene un atributo ID(propio) y un atributo PARENT(ID del bloque que lo contiene), de este modo podemos hacer una búsqueda por niveles de un identificador (necesario para las asignaciones o definiciones de variables.).

Ejemplo de funcionamiento de la pila (Scope).

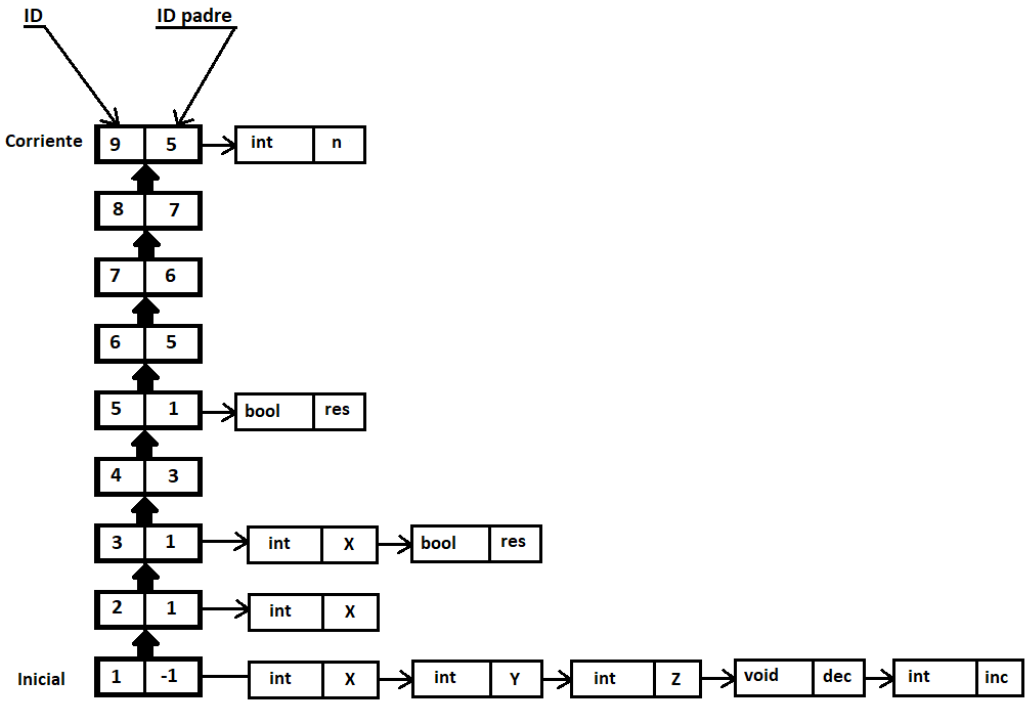
Programa ejemplo:

```
program  begin
  integer x, y, z;
  void dec(integer x) extern;
  integer inc(integer y, bool res) begin
    return x + 1;
  end
  void main() begin
    bool res2;
    x = 1;
    res2 = true;
    while(x<=100) begin
      if(x <= 50) then begin
        if((x == 5) && (res2)) then begin
          x = 50;
        end
        x = inc(x);
      end
      else begin
        integer n;
        x = dec(x);
      end
    end
  end
end
```

Análisis del Scope y Alcance:

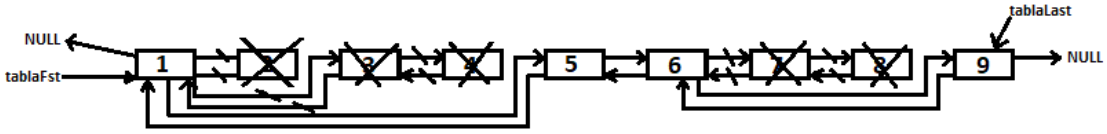


Resultado teórico del Stack:



Traza de la lista de Indexs:

Nota: Los niveles tachados representan niveles cerrados.



Traza de ejecución:

Se crea el nivel 1 -> se **crea** el nivel 2-> se **cierra** el nivel 2-> se **crea** el nivel 3 -> se **crea** el nivel 4 -> se **cierra** el nivel 4 -> se **cierra** el nivel 3 -> se **creó** el nivel 5 -> se **creó** el nivel 6 -> se **creó** el nivel 7 -> se **creó** el nivel 8 -> se **cerró** el nivel 8 -> se **cerró** el nivel 7 -> se **creó** el nivel 9. Hasta aquí podemos ver la creación del último nivel. Luego, antes de finalizar la ejecución del programa, deberían **cerrarse** los niveles 9, 6 y 5

Resultado de la corrida mediante el compilador:

```
 ------------------------------
| ID nivel:    | 1             |
| ID Padre:    | -1            |
| Es funcion:  | 0             |
| Nombre:      | x
| Tipo:        | 1             |
| Linea:       | 3             |
-------------------------------
 ------------------------------
| ID nivel:    | 1             |
| ID Padre:    | -1            |
| Es funcion:  | 0             |
| Nombre:      | y
| Tipo:        | 1             |
| Linea:       | 3             |
-------------------------------
 ------------------------------
| ID nivel:    | 1             |
| ID Padre:    | -1            |
| Es funcion:  | 0             |
| Nombre:      | z
| Tipo:        | 1             |
| Linea:       | 3             |
-------------------------------
 ------------------------------
| ID nivel:    | 1             |
| ID Padre:    | -1            |
| Es funcion:  | 1             |
| Nombre:      | dec
| Valor:       | -1            |
| Tipo:        | 4             |
| Linea:       | 4             |
-------------------------------
 ------------------------------
| ID nivel:    | 1             |
| ID Padre:    | -1            |
| Es funcion:  | 1             |
| Nombre:      | inc
| Valor:       | -1            |
| Tipo:        | 1             |
| Linea:       | 8             |
-------------------------------
 ------------------------------
| ID nivel:    | 2             |
| ID Padre:    | 1             |
| Es funcion:  | 0             |
| Nombre:      | x
| Valor:       | -1            |
| Tipo:        | 1             |
| Linea:       | 4             |
-------------------------------
 ------------------------------
| ID nivel:    | 3             |
| ID Padre:    | 1             |
| Es funcion:  | 0             |
| Nombre:      | res
| Valor:       | -1            |
| Tipo:        | 8             |
| Linea:       | 6             |
-------------------------------
 ------------------------------
| ID nivel:    | 3             |
| ID Padre:    | 1             |
| Es funcion:  | 0             |
| Nombre:      | j
| Valor:       | -1            |
| Tipo:        | 1             |
| Linea:       | 6             |
-------------------------------
NIVEL ID: 4 VACIO CON PADRE ID: 3
 ------------------------------
| ID nivel:    | 5             |
| ID Padre:    | 1             |
| Es funcion:  | 0             |
| Nombre:      | res2
| Tipo:        | 8             |
| Linea:       | 11            |
-------------------------------
NIVEL ID: 6 VACIO CON PADRE ID: 5
NIVEL ID: 7 VACIO CON PADRE ID: 6
NIVEL ID: 8 VACIO CON PADRE ID: 7
 ------------------------------
| ID nivel:    | 9             |
| ID Padre:    | 6             |
| Es funcion:  | 0             |
| Nombre:      | n
| Tipo:        | 1             |
| Linea:       | 25            |
-------------------------------
```

Se generó el AST, donde cada nodo del árbol es un data_stack (explicado anteriormente) y el árbol puede tener, máximo, tres hijos:
fst: Nodo que representa el primer hijo en caso de tenerlo, sino es NULL.
snd: Nodo que representa el segundo hijo en caso de tenerlo, sino es NULL.
trd: Nodo que representa el tercer hijo en caso de tenerlo, sino es NULL.

Para recorre el árbol se implementó una función llamada evalExpr, la cual evalúa una expresión que es pasada como parámetro y retorna su tipo. Si encuentra algún conflicto con el tipo pasado como parámetro, genera un error y lo carga en la lista.

En esta parte, en principio se realizó división de tareas, pero tuvimos muchos problemas, no nos funcionaba nada, fue una etapa que costó mucho y fuimos tratando de arreglar las dificultades entre los dos, no vimos posible solucionar los problemas individualmente, ya que al no saber dónde estaba el problema ni de dónde venía, implicaría, quizá terminar trabajando sobre las mismas funciones.

Generador Código Intermedio

Esta etapa del compilador retorna una representación intermedia (IR) del código. A partir de esta representación intermedia, se generará el código assembler y luego el código objeto.

En esta etapa se utilizará como código intermedio: Código de Tres Direcciones para las operaciones con tipos enteros y lógicos y para las operaciones de control de flujo.

Lo primero que hicimos, fue definir dos tipos que íbamos a utilizar como structs.

El primero es el código de tres direcciones, compuesto por la operación, op (Código del tipo de operación a realizar), *op1 (primer operando), *op2 (segundo operando) y *res (el resultado).

```
typedef struct codTresDirs {
  int op;
  data_gen *op1;
  data_gen *op2;
  data_gen *res;
  struct codTresDirs *next;
}tresDir;
```

El segundo struct, es una lista de éste último, con acceso al primer y último elemento de la lista.

```
typedef struct tresDirList{
  int stackSize;
  char nombre[32];
  bool is_gv;
  int tipo;
  tresDir *fst;
  tresDir *last;
  struct tresDirList *next;
} tresDirL;
```

Para generar el código de tres direcciones se implementaron las siguientes instrucciones:

CONSTANTEE
ASIGNACIONN
IGUALDADD
RESTAA
SUMAA
PRODD
DIVV
MODD
ANDD
ORR
MAYORR
MENORR
IFTHENN
IFTHENELSEE
WHILEE

Cuando creamos cada una de estas instrucciones, generamos un temporal o label donde se guarda el resultado de esa instrucción.

Otras instrucciones implementadas fueron:

BLOCK
STATEMENTS
RETURNN
INVOCC

Generador Código Objeto

En esta etapa se genera código assembly x86-64 (sin optimizaciones) a partir del código intermedio.
Luego de generar el código intermedio y almacenarlo en una lista, recorrimos la misma obteniendo cada una de las instrucciones del código intermedio y traduciéndolas a su instrucción equivalente en Assembler.

Un ejemplo de instrucción traducida del código intermedio al assembler:

```
|CTE_INSTRUCCION  int_cte  _ TMP1
|
|ASIGN_INSTRUCCION  TMP1  _ x


movq  $3, -40(%rbp)
movq  -40(%rbp), %rax
movq  %rax, -8(%rbp)
```

Tanto el -40 como el -8 son offset. Los offset comienzan del -8 y se decrementan de 8 en 8.

Para la traducción a assembler, tomamos como referencia, la manera de hacerlo del leguaje C. Analizamos como lo hacía generando código assembler con el comando gcc –S. Y así fuimos implementando cada una de las instrucciones.

En esta etapa comenzamos dividiendo el trabajo sin problema, cada uno implementado sus instrucciones y cuando tuvimos inconvenientes trabajamos conjuntamente para tratar de solucionarlos.

Optimización

En esta parte solo realizamos dos optimizaciones. La primera, diferenciar constantes. Cuando creamos las instrucciones de assembler, chequeamos si es contante o no, de esta manera no generamos temporales de más.

La segunda, algo muy similar a las contantes. Chequeamos si una variable es global o no. En el caso de serlo, accedemos a la variable de la siguiente forma: _nombreVar(%rip).

Ejemplos de optimización:

Programa ejemplo:

```
program begin
  void main() begin
    integer a,b;
    a=1;
    b=2;
    if(10<100)then begin
      a = a + b;
    end
    else begin
      b = b + a;
    end
  end
end
```

Código assembler generado antes de la optimización:

```
   .globl  _main
_main:                  ## -- Begin function main
  .cfi_startproc
  enter $80, $0
  movq  $1, -24(%rbp)
  movq  -24(%rbp), %rax
  movq  %rax, -8(%rbp)
  movq  $2, -32(%rbp)
  movq  -32(%rbp), %rax
  movq  %rax, -16(%rbp)
  movq  $10, -40(%rbp)
  movq  $100, -48(%rbp)
  movq -40(%rbp), %rax
  cmpq %rax, -48(%rbp)
  setg %dl
  andb $1, %dl
  movzbl %dl, %esi
  movq %rsi, -56(%rbp)
  cmpl  $0, -56(%rbp)
  je LBB_1
  movq  -8(%rbp), %rax
  addq  -16(%rbp), %rax
  movq  %rax, -64(%rbp)
  movq  -64(%rbp), %rax
  movq  %rax, -8(%rbp)
  jmp LBB_2
LBB_1:
  movq  -16(%rbp), %rax
  addq  -8(%rbp), %rax
  movq  %rax, -72(%rbp)
  movq  -72(%rbp), %rax
  movq  %rax, -16(%rbp)
LBB_2:
  leave
  retq                  ## -- End function
  .cfi_endproc
```

Número de Líneas: 36

Código assembler generado luego de la optimización:

```
   .globl  _main
_main:                  ## -- Begin function main
  .cfi_startproc
  enter $48, $0
  movq  $1, -8(%rbp)
  movq  $2, -16(%rbp)
  movq  $10, %rax
  cmpq  $100, %rax
  setl %dl
  andb $1, %dl
  movzbl %dl, %esi
  movq %rsi, -24(%rbp)
  cmpl  $0, -24(%rbp)
  je LBB_1
  movq  -8(%rbp), %rax
  addq  -16(%rbp), %rax
  movq  %rax, -32(%rbp)
  movq  %rax, -8(%rbp)
  jmp LBB_2
LBB_1:
  movq  -16(%rbp), %rax
  addq  -8(%rbp), %rax
  movq  %rax, -40(%rbp)
  movq  %rax, -16(%rbp)
LBB_2:
  leave
  retq                  ## -- End function
  .cfi_endproc
```

Número de líneas: 28

Diferencia entre las asignaciones con constantes:

Ineficiente	Eficiente

movq $1, -24(%rbp)
movq -24(%rbp), %rax
movq %rax, -8(%rbp)
movq $2, -32(%rbp)
movq -32(%rbp), %rax
movq %rax, -16(%rbp)
movq $10, -40(%rbp)
movq $100, -48(%rbp)
	movq $1, -8(%rbp)
	movq $2, -16(%rbp)
	movq $10, %rax
	cmpq $100, %rax

Comentarios

Nos hubiese gustado poder manejar mejor el uso de jumps para las condiciones del if, while. Utilizando los distintos tipos de jump que brinda assembler (JGE, JLE, etc).