Práctica de Compiladores II:
PARTE III: Generación de código intermedio
Curso 2003/2004

Web de Compiladores: http://www.dsic.upv.es/asignaturas/facultad/com2

v. 04.3

Índice

7. La máquina virtual Malpas

El objetivo final del proyecto que se está desarrollando es la generación de un código intermedio de tres direcciones. Este código intermedio que se presentará en la sección 7.2 podrá ejecutarse en una máquina abstracta llamada *Malpas*.

Para gestionar la memoria en tiempo de ejecución la máquina virtual Malpas emplea un segmento de código (donde se almacena el código intermedio a ejecutar), un *display* y una *pila de activación*. La no existencia de objetos de tamaño dinámico hace que esta máquina no necesite un "heap".

Para almacenar los objetos globales se emplea la base de la pila. De esta manera, podemos considera al Programa Principal como un bloque más (de nivel 0) que dispondrá de su propio *registro de activación* que contendrá únicamente variables globales y temporales.

7.1. Estructura del registro de activación (RA)

En la Fig. 1 se muestra la estructura de un registro de activación tal y como debe construirse para funcionar correctamente con la máquina abstracta Malpas.

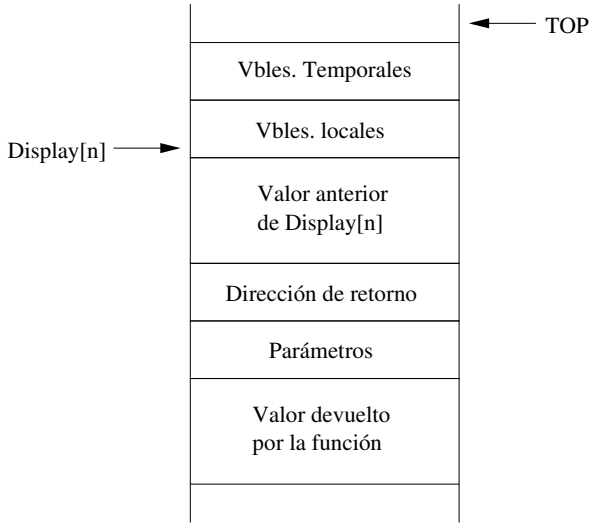

Figura 1: Estructura de un registro de activación.

7.1.1. Acceso a las variables.

El acceso a las variables se realiza de forma homogénea para todos los niveles de anidamiento (incluyendo el nivel 0 del programa principal), tanto para las variables declaradas por el usuario como para variables temporales definidas por el compilador. El acceso a la dirección absoluta de memoria es responsabilidad del intérprete. Sea un objeto x, con un nivel de anidamiento (niv_x) y un desplazamiento relativo en el segmento correspondiente (des_x), el cálculo de la dirección de x vendrá dado por:

$dir_x = display[niv_x] + des_x$

Cuando un argumento de una instrucción de código intermedio es de tipo *posición* la máquina virtual ya realiza estos cálculos sin más que indicar el nivel y el desplazamiento relativo del argumento.

El esquema del registro de activación empleado por la máquina virtual permite que el desplazamiento de la primera variable definida en cada nivel de anidamiento sea cero.

7.1.2. Acceso a los parámetros.

Como el tipo de todos los parámetros es el mismo (tipo simple) su talla será idéntica ($talla = 1$). Suponiendo que el primero de los parámetros se

declara con desplazamiento relativo cero, si una función tiene k parámetros, el acceso al parámetro i-ésimo se realiza de igual forma que el acceso a una variable que tuviera el desplazamiento relativo

$.des = -(k+2) + i$

Cuando una función desea dejar su valor de retorno en la base de su registro de activación, bastará con que ejecute una instrucción de asignación cuyo argumento resultado sea una posición de memoria con el nivel de anidamiento de la función y como desplazamiento relativo:

$.des = -(k+3)$

7.2. Juego de instrucciones

En esta sección se presenta el juego de instrucciones de la máquina abstracta Malpas. En la descripción mostrada, I indica que el operando es de tipo entero, y P que es de tipo posición. En las instrucciones aritméticas los argumentos serán $arg1$ y $arg2$, y el resultado de la ejecución se almacenará en la posición indicada por res.

Operaciones aritméticas

OP	arg1	arg2	res	Significado
ESUM	I/P	I/P	P	Suma
EDIF	I/P	I/P	P	Resta
EMULT	I/P	I/P	P	Multiplicación
EDIVI	I/P	I/P	P	División entera
RESTO	I/P	I/P	P	Resto división entera
ESIG	I/P		P	Cambio de signo
EASIG	I/P		P	Asignación

Operaciones de salto

OP	arg1	arg2	res	Significado
GOTOS			E	Salto incondicional a E
EIGUAL	I/P	I/P	E	si arg1=arg2 salto a E
EDIST	I/P	I/P	E	si arg1<>arg2 salto a E
EMEN	I/P	I/P	E	si arg1<arg2 salto a E
EMAY	I/P	I/P	E	si arg1>arg2 salto a E
EMENEQ	I/P	I/P	E	si arg1<=arg2 salto a E
EMAYEQ	I/P	I/P	E	si arg1>=arg2 salto a E

Operaciones de asignación con matrices

OP	arg1	arg2	res	Significado
EAV	P	I/P	P	Asigna un elemento de una matriz a una variable: $res := arg1[arg2]$
EVA	P	I/P	P	Asigna una variable a un elemento de una matriz: $arg1[arg2] := res$

Operaciones de entrada/salida

OP	arg1	arg2	arg3	Significado
EREAD			P	Lectura
EWRITE			I/P	Escritura

Operaciones de llamada

OP	arg1	arg2	res	Significado
FIN				Fin del programa
RET				Desapila la dirección de retorno y transfiere el control a dicha dirección
CALL			E	Apila la dirección de retorno y transfiere el control a *res*

Operaciones de manejo de pila y display

OP	arg1	arg2	res	Significado
EPUSH			I/P	Apila un elemento en la cima de la pila
EPOP			P	Desapila la cima y la deposita en *res*
PUSHDISP			I	Apila el display de nivel *res*
DISPPOP			I	Desapila la cima y la deposita en display de nivel *res*
DISPTOP			I	Copia el tope de la pila en el display de nivel *res*
TOPDISP			I	Actualiza el valor del tope de la pila con el display de nivel *res*
INCTOP			I	Incrementa el tope de la pila en *res* posiciones
DECTOP			I	Decrementa el tope de la pila en *res* posiciones

8. Generación de código intermedio

8.1. Estructuras de datos y variables globales

En las secciones anteriores hemos visto que en el fichero *header.h* podemos encontrar la definición de las estructuras, constantes simbólicas, variables globales y cabeceras de funciones necesarias para escribir las acciones semánticas. De este fichero podemos extraer las siguientes declaraciones útiles para generar código intermedio:

```
extern int si;       /* Referencia a la siguiente instrucción de código      */
                     /* intermedio a generar (SIGINST)                       */

typedef struct tipo_posmem
{                       /* Estructura para almacenar una posición de memoria */
  int pos, niv;                  /* Posición relativa y nivel de anidamiento */
} TIPO_POSMEM;

typedef struct tipo_arg
{              /* Estructura para los argumentos del código tres direcciones */
  int tipo;                       /* Tipo del argumento */
```

```
  union {
    int i;                  /* Variable para argumento entero               */
    TIPO_POSMEM p;          /* Variable para argumento posición de memoria  */
    int e;                  /* Variable para argumento dirección de memoria */
  } val;
} TIPO_ARG;
```

La variable global *si*, inicializada a 0, contiene el número de la siguiente instrucción de código intermedio que se va a generar. La propia función *emite* se encarga de incrementarla.

Por otro lado, se puede observar que $TIPO\_ARG$ es un tipo de dato definido por el usuario que se corresponde con una estructura que contiene en su campo *val* una unión. De esta forma, al definir una variable o atributo de tipo $TIPO\_ARG$ (por ejemplo $arg1$) será posible almacenar un argumento de cualquier tipo. Por ejemplo, para almacenar un argumento de tipo entero con valor 5, bastaría con realizar las asignaciones $arg1.tipo = T\_ENTERO; arg1.val.i = 5$. No obstante, para facilitar la creación de las estructuras de tipo argumento, el alumno puede emplear las funciones $cr\_arg\_nulo()$, $cr\_arg\_entero()$, $cr\_arg\_etiqueta()$ y $cr\_arg\_posicion()$ que se presentan en la siguiente sección.

8.2. Funciones de ayuda

Se presenta a continuación el listado de funciones que deben emplearse para generar código intermedio.

```
TIPO_ARG cr_arg_nulo ();
/* Crea un argumento de un instrucción tres direcciones de tipo nulo.        */

TIPO_ARG cr_arg_entero (int valor);
/* Crea un argumento de un instrucción tres direcciones de tipo entero
   con el "valor".                                                           */

TIPO_ARG cr_arg_etiqueta (int valor);
/* Crea un argumento de un instrucción tres direcciones de tipo
   etiqueta con el "valor".                                                  */

TIPO_ARG cr_arg_posicion (int n, int valor);
/* Crea un argumento de un instrucción tres direcciones de tipo
   posición con el nivel de anidamiento "n" y el desplazamiento
   "valor".                                                                  */

void emite (int cop, TIPO_ARG arg1, TIPO_ARG arg2, TIPO_ARG res);
/* Crea una instrucción tres direcciones con el código de operación
   "cod" y lo argumentos "arg1", "arg2" y "res", y la pone en la
   siguiente posición libre (indicada por "si") del segmento de
   código. A continuación incrementa "si".                                   */
```

```
int crea_var_temp ();
/* Crea una variable temporal, de talla 1, en el segmento de variables
   del bloque actual y devuelve su desplazamiento relativo. Además,
   incremente "dvar = dvar + 1".                                          */

void vuelca_codigo_ascii ();
/* Vuelca, en modo texto, el código generado a un fichero cuyo nombre
   es el de entrada con la extensión ".txt".                              */

void vuelca_codigo_binario ();
/* Vuelca, en modo binario, el código generado a un fichero cuyo
   nombre es el de entrada con la extensión ".bin".                       */


/*************************************************************************/
/*******          Operaciones para la manipulación de las LANS         *******/
/*************************************************************************/
int crea_lans (int d);
/* Crea una lista de argumentos no satisfechos para una instrucción
   incompleta cuya dirección es "d", y devuelve su referencia.        */

int fusiona_lans (int x, int y);
/* Fusiona dos listas de argumentos no satisfechos cuyas referencias
   son "x" e "y", y devuelve la referencia de la lista fusionada.     */

void completa_lans ( int x, TIPO_ARG arg);
/* Completa con el argumento "arg" el campo "res" de todas las
    instrucciones incompletas de la lista "x".                        */
```

8.3. Ejemplo de producciones generadoras de código

Para que el alumno pueda hacerse una idea de la forma en la que se puede generar código empleando las funciones anteriores, se presentan unas cuantas producciones con sus acciones semánticas de generación de código asociadas. El objetivo de estas producciones es generar el código intermedio para asignaciones del tipo $id := cte$.

Evidentemente será necesario definir los tipos semánticos necesarios tal y como se vió en las secciones anteriores. Además, para simplificar el ejemplo, *se ha omitido todo el código relacionado con las comprobaciones de tipo*, como la comprobación de que el ID está declarado, o la compatibilidad de tipos en la asignación.

```
inst_simple: ID_ ASIGNACION_ expresion
        {  SIMB sim;
 /* Acciones sin comprobaciones de tipo */

           sim = obtener_simbolo(busca_id($1, nivel));
           switch (sim.categoria) {
           case VARIABLE:
             res = cr_arg_posicion(sim.nivel, sim.desp);
             break;
```

```
            case PARAMETRO:
              res = cr_arg_posicion(sim.nivel,sim.desp-(sim.tasegpar+2));
              break;
            case FUNCION:
              res = cr_arg_posicion(nivel,-(sim.tasegpar+3));
              break;
            }
            arg2 = cr_arg_nulo();
            emite(EASIG, $3.pos, arg2, res);
           }

expresion: CTE_ { $$.tipo = T_ENTERO;
                  $$.pos = cr_arg_entero($1);
              } ;
```

9. Ejemplo de programa en código intermedio

En esta sección mostramos como ejemplo de generación de código, el código intermedio generado para un pequeño programa que calcula el factorial de un número. Hemos de destacar que se trata solo de un ejemplo de cómo se podría generar el código intermedio, y por lo tanto, distintos compiladores podrán generar código diferente pero igualmente válido.

```
program factorial;
var
    x:integer;

function factorial(n: integer):integer;
  begin
     if n <= 0 then factorial:= 1
     else factorial := n * factorial(n-1);
  end;

begin
   read(x);
   write(factorial(x));
end.
```

0	GOTOS			e: 22	Salta al begin del programa principal
1	PUSHDISP			i: 1	Apila DISPLAY[1]
2	DISPTOP			i: 1	DISPLAY[1] = TOP
3	GOTOS			e: 4	Salta al begin de la función
4	INCTOP			i: 4	Reserva espacio para área de datos
5	EASIG		i: 1	p: 1,0	t0=1
6	EMENEQ	p: 1,-3	i: 0	e: 8	si n¡=0 salta a 8
7	EASIG	i: 0		p: 1,0	t0=0
8	EIGUAL	p: 1,0	i: 0	e: 11	si t0=0 salta a 11
9	EASIG	i: 1		p: 1,-4	factorial=1
10	GOTOS			e: 19	
11	EPUSH			i: 0	Reserva espacio para valor devuelto
12	EDIF	p: 1,-3	i: 1	p: 1,1	Evalúa parámetro (en t1)
13	EPUSH			p: 1,1	Apila parámetro: t1
14	CALL			e: 1	Llama a factorial
15	DECTOP			i: 1	Desapila parámetro
16	EPOP			p: 1,2	Desapila valor devuelto en t2
17	EMULT	p: 1,-3	p: 1,2	p: 1,3	t3=n*t2
18	EASIG	p: 1,3		p: 1,-4	factorial=t3
19	TOPDISP			i: 1	TOP=DISPLAY[1]
20	DISPPOP			i: 1	DISPLAY[1]=Desapilar
21	RET				
22	INCTOP			i: 2	Reserva espacio para área datos globales
23	EREAD			p: 0,0	read (x)
24	EPUSH			i: 0	Reserva espacio para valor devuelto
25	EPUSH			p: 0,0	Apila parámetro x
26	CALL			e: 1	Llama a factorial
27	DECTOP			i: 1	Desapila parámetro
28	EPOP			p: 0,1	Desapila valor devuelto en t1
29	EWRITE			p: 0,1	write(t1)
30	FIN				