PRÁCTICA DE PROCESADORES DEL LENGUAJE I - Curso 2012 – 2013 - Entrega de Septiembre

September 6, 2013

APELLIDOS Y NOMBRE: **Martínez García, Octavio**
IDENTIFICADOR: **omartinez185**
DNI: **71280002-N**
CENTRO ASOCIADO MATRICULADO: **CA Burgos**
CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: **CA Burgos**
MAIL DE CONTACTO: **omartinez185@alumno.uned.es**
TELÉFONO DE CONTACTO: **690 36 11 55**
GRUPO (A ó B): **A**

1 Analizador Semántico y Comprobación de Tipos

Las comprobaciones semánticas, que inicialmente se realizaban directamente en el parser, se han trasladado a los constructores de los no terminales, por claridad, mejor encapsulación y facilidad de debug, y éstas se ejecutan cuando el parser llama al constructor de la clase correspondiente bajo la regla semántica que corresponda.

1.1 Tabla de Símbolos

La tabla de símbolos como tal viene dada en la arquitectura de referencia mediante la clase SymbolTable, la cual es una estructura de tabla hash que relaciona nombres de símbolos con símbolos propiamente dichos, representados mediante las clases que heredan de la interfaz SymbolIF y extienden la clase base SymbolBase, siendo estas las siguientes:

- SymbolConstant
  - en la declaracion de constantes simbólicas (clase SymbolicConstantDeclaration) se obtiene la tabla de símbolos del scope actual, y se comprueba que dicha tabla no contenga previamente los nombres de las constantes, para, en su caso, crear una nueva SymbolConstant con ese nombre e insertarla en la tabla de símbolos del scope
- SymbolVariable
  - en la declaracion de variables globales (y locales a subprogramas, ya que se usa la misma clase GlobalVarDeclaration) se obtiene la tabla de simbolos del scope actual, y en función del tipo de token pasado al constructor (IDENTIFIER - vble de tipo compuesto, INTEGER - vble entera, BOOLEAN - vble booleana) se crea una nueva SymbolVariable del tipo adecuado y se inserta en la tabla en caso de que no existiera previamente
- SymbolParameter - *(tratado en el epígrafe del SymbolProcedure)*
- SymbolProcedure
  - cuando se procesa la cabecera de un subprograma dado se obtiene la tabla de simbolos del ambito padre del subprograma y se la añade el SymbolProcedure (o Function segun sea) para que pueda ser llamado desde fuera, además de lo cual se abre un nuevo scope (el correspondiente al ambito del subprograma) y se añade asimismo dicho SymbolProcedure a la tabla del ámbito interno, para que pueda ser llamada recursivamente desde dentro.
  - del mismo modo, en la tabla de símbolos del ámbito interno del subprograma se añaden los SymbolParameters de este, los cuales se ha pasado previamente al constructor de la cabecera de procedimiento (CabeceraProc)
- SymbolFunction
  - igual que en el caso de SymbolProcedure, solo que además se define el tipo de retorno de la función (CabeceraFunc hereda de CabeceraProc, añadiendo el att. TypeIF tipoRetorno)

2 Generación de código intermedio (CI)

2.1 Estructura utilizada

A partir del juego de instrucciones de ENS2001 se ha definido un lenguaje intermedio que consta de un juego de 21 tipos de instrucciones (20 YA QUE "MVP" PROBABLEMENTE LA ELIMINE) que se muestran a continuación:

CÓDIGO	***OPERANDOS***	***SIGNIFICADO***
ADD	3 (r,o1,o2)	r = o1 + o2
AND	3 (r,o1,o2)	r = o1 && o2
ARG	1 (r)	push argumento en la stackTop
BR	1 (r)	salta a la etiqueta r
BRF	2 (r,o1)	if !o1 salta a la etiqueta r
CALL	1 (r)	llama a la funcion r
DATA	0 / 1 (r)	inicializa memoria, pila y registros y salta al mainLabel
HALT	0	finaliza la ejecución del programa
INL	1 (r) / 2(r,o1)	introduce la etiqueta de if o while / subprograma
LT	3 (r,o1,o2)	r = o1 < o2
MUL	3 (r,o1,o2)	r = o1 * o2
MV	2 (r,o1)	r = o1
MVA	2 (r,o1)	r = &o1
MVArray	3 (r,o1,o2)	r(i) = o2 / r = o2(i) / o1 es el offset
MVP	2 (r,o1)	&r = o1 (r es un parametro y o1 una exp)
NE	3 (r,o1,o2)	r = o1 /= o2
NOP	0	no operación
RET	0 / 1 (r)	retorno de proced / función - r valor de ret
SUB	3 (r,o1,o2)	r = o1 - o2
WRINT	1 (r)	escribe en consola el entero r
WRSTR	1 (r)	escribe en consola la cadena r

Para la generación del CI se han añadido en cada uno de los no terminales de la gramática (y en SymbolParameter por requerimientos posteriores) un método generateCI() que se encarga de añadir las cuádruplas de CI necesarias en cada caso, el cual es llamado en el parser una vez que las comprobaciones semánticas son satisfechas. No obstante existen casos, como las reglas "exp := ID; exp := TRUE;" y otras similares, en que las acciones de generación de CI aun se llevan a cabo directamente en el parser.

3 Generación de código final

3.1 Registro de Activación (RA)

Se ha implementado un RA basado en pila con enlace de control y enlace de acceso para el direccionamiento de variables no locales, el cual responde a la estructura descrita a continuación:

vbles locales
temporales
FP o enlace de control (.IX)
direccion de retorno
valor de retorno
parametros

enlace de acceso (Display)

- los parámetros son introducidos en la pila por el llamante en orden PUSH P1, PUSH P2, ... PUSH Pn, de modo que puedan ser rescatados de ella por el llamado en el orden correcto según las direcciones asignadas a cada uno durante la resolución de referencias simbólicas (clase Auxil) justo antes de la generación del código final

- se deja una posición libre para almacenar el valor de retorno (se deja siempre, pero solo se usa en caso de funciones), de modo que el valor de retorno está en +2[FP]. Se utiliza el registro índice IY para almacenar el FP del llamante y el valor de retorno se almacena y se rescata con relación a este índice.

- la dirección de retorno se introduce en la pila por medio de la instrucción CALL de ENS2001 (inserta el PC en la pila y mueve una posición, decreciente en este caso, el SP). Posteriormente se recoge en el registro R7 durante el retorno de subprograma y se inserta finalmente en la base del RA, es decir, en este caso en la posicion mas alta (sentido decreciente de la pila), desde donde se recoge la direccion de retorno y se vuelve al PC siguiente del llamante

- enlace de control [.IX] - el FP o enlace de control esta en la posicion central (0[.IX]) del RA, en el cual se almacena el FP del llamante, almacenado previamente (durante CALL) en el indice .IY

- enlace de acceso [.R8] - para el acceso a referencias no locales se utiliza un Display, que es un array donde se almacenan los FP de los ambitos en orden de anidamiento. Este array está almacenado desde la posicion de memoria 61000 en adelante (61000 es el Display[0], 61001 es Display[1], y asi sucesivamente). Dicho FP se almacena en el registro R8 temporalmente para realizar el acceso a la variable en cuestión segun su direccion relativa a ese FP

- display [.R9] - para almacenar y recuperar las direcciones del display (para la gestion del display propiamente dicho) se utiliza el registro R9

- la asignacion de las direcciones a los temporales, variables locales y parámetros se hace a través de la clase Auxil y su método estático symbolicReferenceTranslation(), el cual va obteniendo los diferentes ambitos y dentro de cada uno asigna

a cada elemento mencionado su direccion partiendo de un Offset (lOffset +1 para temporales y locales, y pOffset +3 para parámetros) que indica donde deben empezar respecto al FP teniendo en cuenta la colocación del propio FP, dirección de retorno y valor de retorno.

- por homogeneidad el RA del main se trata de modo idéntico a los RA del resto de subprogramas
- la llamada a un subprograma y por tanto la gestión del RA se articula en dos pasos, CALL (secuencia de llamada) y RET (secuencia de retorno). A continuación se muestra un fragmento de la generación de código final para estos dos casos, articulado en TranslatorCALL y TranslatorRET respectivamente:

```
// movemos el FP del llamante (.IX) al vinculo de control (.IY)
translator.append("MOVE " + ".IX" + " , " + ".IY" + "\n");
// dejamos un espacio para la direccion de retorno
translator.append("PUSH " + "#0" + "\n");
// hacemos que el puntero de marco (FP) apunte a la cima de pila (SP) --> 1ª posicion libre de la p
translator.append("MOVE " + ".SP" + " , " + ".IX" + "\n");

// dejamos una posicion libre para el valor de retorno (direccion de retorno ya en pila por llamante
translator.append("DEC " + ".IX" + "\n");
// colocamos el enlace de control (FP del llamante)
translator.append("MOVE " + ".IY" + " , " + "[.IX]" + "\n");
// movemos el FP al display de este ambito
translator.append("INC " + ".R9" + "\n");
translator.append("MOVE " + ".IX" + " , " + "[.R9]" + "\n");
// introducimos en la pila el espacio para variables locales y temporales
// de esto se encarga el TranslatorINL, es decir cuando se inserta la etiqueta del subprograma
//translator.append("SUB " + ".SP" + " , " + "#" + Auxil.getScopeSize(proc.getScope().getLevel()+1)
//translator.append("MOVE " + ".A" + " , " + ".SP" + "\n");

translator.append("CALL /" + label + "\n");


// FUNCIONES (valor de retorno de tipo Temporal)
int scopeLvl = ((Temporal) res).getScope().getLevel();
// movemos el FP del llamante a .IY
translator.append("MOVE " + "[.IX]" + " , " + ".IY" + "\n");
// movemos el resultado a su posicion en el RA llamante (FP almacenado en .IY), valor ret en posicio
//translator.append("INC " + ".IY" + "\n");
translator.append("MOVE " + r + " , " + "#2[.IY]" + "\n");
// destruimos el RA del llamado (decrementar .SP)
translator.append("ADD " + ".SP" + " , " + "#" + Auxil.getScopeSize(scopeLvl) + "\n");
translator.append("MOVE " + ".A" + " , " + ".SP" + "\n");

// recogemos la direccion de retorno en el registro R7
translator.append("POP " + ".R7" + "\n");
// decrementamos (en este caso aumentamos, ya que la direccion de crecimiento de la pila es decrecie
// en el tamaño de los parametros + 1, donde dejaremos la direccion de retorno finalmente
translator.append("ADD " + ".SP" + " , " + "#" + (Auxil.getParamSize(scopeLvl)+1) + "\n");
translator.append("MOVE " + ".A" + " , " + ".SP" + "\n");
// insertamos la direccion de retorno
translator.append("PUSH " + ".R7" + "\n");
// restauramos el FP al del llamante
translator.append("MOVE " + ".IY" + " , " + ".IX" + "\n");
// retornamos al PC del llamante
translator.append("RET" + "\n");

// actualizamos el currentScope para comprobar las vbles no locales
Auxil.setCurrentScopeLevel(((Temporal) res).getScope().getLevel()-1);
int currentScope = Auxil.getCurrentScopeLevel();
translator.append("; Current Scope [" + currentScope + "] \n");
```

4 Indicaciones especiales

- Partes incompletas
  - TranslatorMVA y TranslatorMV no se si cubren la totalidad de los casos posibles para la gramatica definida ya que existen muchas combinaciones de Temporales y Variables de tipos distintos (isParameter, isNoLocal, etc) que se pueden dar en esas instrucciones. Los casos de prueba de la carpeta test, testCaseA01 - 12, testA y testOctavio, si que los cubren pero no estoy del todo seguro que cubran todos los posibles.
  - En las especificaciones se dice que el lenguaje no debe ser case-sensitive, para lo cual se añaden los nombres a las tablas de símbolos y tipos siempre en minúscula en Java (toLowerCase() después de cada getLexema()) para comparar homogéneamente minúsculas con minúsculas. (testCase07 vectorE vs VectorE deben ser iguales - no case sensitive). Esto lo he modificado al final y no he realizado tests exhaustivos, de modo que quizás me haya dejado alguna situación sin comprobar.
  - el Display se construye en las llamadas (TranslatorCALL) de modo que no estoy seguro de que lo que almacene realmente no sean los FP de los RA segun orden de llamada y no de anidamiento. Quizas deberia trasladarlo a TranslatorINL, pero desde alli no puedo saber cual va a ser el FP del subprograma ya que el RA se construye durante la llamada (y por tanto es en ese momento cuando se conoce la direccion del FP).
- Interpretaciones y Soluciones
  - Axiom - es Abstract, de modo que se ha creado Axioma como clase hija concreta
  - Equivalencia de tipos - se articula a traves de isCompatible de TypeSimple (no se puede sobreescribir el equals de TypeBase y al usarlo no funciona como se espera ya que TypeBase es Abstract pero no asi sus métodos que estan declarados como finales)
  - Array_Access - se utiliza tanto para el acceso a un elemento de un vector como para la llamada a un proc de 1 solo parámetro. No se verifica que la expresión de acceso a un vector esté dentro de los límites (se debería hacer en t^{o} de ejecución)
  - Paso por Referencia - la implementación se parece a un paso por copia-valor, ya que un paso por referencia puro (trabajar con la indireccion de registros para el param y todos sus temporales asociados) no sabia bien como implementarlo y no encontré documentacion suficiente a este respecto. De este modo se trabaja con la direccion real del parámetro al obtenerle inicialmente y al asignarle nuevo valor, pero entre medias se trabaja con los temporales asociados como valores.
  - en la instrucción INL, el comentario de scope size (temporal + local vbles) incluye +1 de lOffset (lOffset es inicializado en 1 ya que es la primera direcion respecto al FP del RA), y ocurre igual con pOffset que se inicializa en 3 (ver clase Auxil)

  - TranslatorMVP - solo se usa en sentAssign para mover el resultado de una expresion a la direccion del parámetro (quizás se podría factorizar en otra instrucción preexistente)

- Restricciones del Compilador
  - en las sentencias Put_line("String") se han omitido los caracteres ( ) : en los archivos .ha de los tests porque generaban error en las etiquetas, ya que el TranslatorWRSTR crea la etiqueta a partir de la cadena y si existen caracteres como espacios blancos, paréntesis o dos puntos, el ens2001 no los reconoce como una etiqueta y da error (la instruccion WRSTR usa como operando la direccion - etiqueta - desde donde se almacena secuencialmente la cadena a escribir)
  - en el testCaseA11 el resultado de este test es -6 (5-11), y no 6 como se especifica en la salida (en el Put_line previo)
  - R7 se usa para la gestión de la direccion de retono en la secuencia de retorno de un procedimiento / función. .R8 se usa para acceso a vbles no locales, y .R9 gestiona el diaplay
  - Tal como está estructurado solo se permiten subprogramas con un máximo de 7 parámetros, ya que hace uso de los registros de propósito general R0 - R6 para la gestión de las direcciones (paso por referencia) de los parámetros, de modo que si excede ese número no se le podría asignar un registro al octavo parámetro, lo que generaría errores imprevistos. RegisterDescriptor con su método AllocateRegisters() se encarga de asignar los registros del R0 al R6 a los parámetros de los subprogramas según el nombre de su ámbito
  - está activo el filtro de mensajes para info, debug y error. Esto se puede modificar desde la subregla "cabeza_main" de axiom, descomentando las sentencias que se requieran.

5 Conclusiones

El compilador lee archivos en lenguaje HAda y produce archivos ejecutables en ensamblador (ens2001). Pasa con éxito los 12 casos de prueba de tipo A, más algunos casos adicionales probados por mi. (pasa todos los de la carpeta test de la arquitectura), sin embargo no puedo asegurar que pase todos los tests posibles de programas tanto correctos como incorrectos en HAda. Reconoce errores léxicos, sintáticos y semánticos, informando del luegar del código donde se produjeron además de la causa del error.

6 Gramática

```
start with program;

program ::= axiom:ax;

// ------------------------------------- estructura general del programa HAda ------------------------

axiom ::= cabeza_main:head cuerpo_main:body;

cabeza_main ::= PROCEDURE IDENTIFIER:id LPAREN RPAREN;

cuerpo_main ::= IS declarations:d BEGIN fun_st_block:fsb END IDENTIFIER:idf SEMICOL
                | IS declarations:d BEGIN END IDENTIFIER:idf SEMICOL
                | IS BEGIN fun_st_block:fsb END IDENTIFIER:idf SEMICOL
                | IS BEGIN END IDENTIFIER:idf SEMICOL
                | error;


// ------------------------------------- declaraciones iniciales -------------------------------------
declarations ::= symbolic_constant_declaration:scd
                 | symbolic_constant_declaration:scd declarations:ds
                 | tvs:t;

tvs ::= global_types_declaration:gtd
        | global_types_declaration:gtd tvs:t
        | vs:v;

vs ::= global_var_declaration:gvd
       | global_var_declaration:gvd vs:v
       | sp:s;

sp ::= subprog_declaration:sd
       | subprog_declaration:sd sp:s;


// ------------------------------------- declaracion de constantes simbolicas ------------------------
symbolic_constant_declaration ::= list_ids:li COL CONSTANT ASSIGN literal_constant:lc SEMICOL
                                  | error SEMICOL;

literal_constant ::= LITERALINT:i
                     | TRUE
                     | FALSE ;


// ------------------------------------- declaracion de tipos globales -------------------------------

// declaracion de tipos (ARRAY EN ESTA ESPECIFICACION)
global_types_declaration ::= TYPE IDENTIFIER:id IS ARRAY LPAREN array_indexes:ai RPAREN OF array_va
                             | error;

array_indexes ::= a_index:f PNTPNT a_index:l;

a_index ::= LITERALINT:cint
            | IDENTIFIER:id ;

array_values ::= INTEGER
                 | BOOLEAN;


// ------------------------------------- declaracion de vbles globales -------------------------------
global_var_declaration ::= var_decl:vd IDENTIFIER:it SEMICOL
                           | var_decl:vd INTEGER:it SEMICOL
                           | var_decl:vd BOOLEAN:it SEMICOL
;
var_decl ::= list_ids:lid COL;
```

```
list_ids ::= IDENTIFIER:id COMMA list_ids:lid
                        | IDENTIFIER:id;


// ------------------------------------ declaracion de subprogramas ------------------------------------
subprog_declaration ::= proced:p
                                            | func:f;

proced ::= cabecera_proc:cab cuerpo_proc:cue;

cabecera_proc ::= PROCEDURE IDENTIFIER:idi LPAREN ref_params:rp RPAREN
                | PROCEDURE IDENTIFIER:idi LPAREN RPAREN ;

cuerpo_proc ::= IS local_declarations:ld BEGIN st_block:sb END IDENTIFIER:idf SEMICOL
                                    | IS local_declarations:ld BEGIN END IDENTIFIER:idf SEMICOL
                                    | IS BEGIN st_block:sb END IDENTIFIER:idf SEMICOL
                                    | IS BEGIN END IDENTIFIER:idf SEMICOL;

ref_params ::= param:p SEMICOL ref_params:rp
                        | param:p;

param ::= var_decl:vd io INTEGER
            | var_decl:vd io BOOLEAN
            | var_decl:vd io IDENTIFIER:id;

// ver si necesitamos el tipo de parametro en algun momento (in / out)
// en principio son de tipo out todos (de momento dejo el in por si es necesario mas tarde)
io ::= IN | OUT;

local_declarations ::= tvs:t;

st_block ::= sent:s st_block:sb
                    | sent:s;


// ------------------------------------------ expresiones ------------------------------------------
// esta variedad de expresiones las diferenciamos en el semántico
exp ::= exp:e1 op:o exp:e2
        | LPAREN exp:ep RPAREN
        | func_invoc:fi
        | array_access:aa
        | IDENTIFIER:i
        | LITERALINT:li
        | TRUE:id
        | FALSE
        | error;

op ::= LESSER:lt
        | NOTEQUAL:ne
        | AND:a
        | MINUS:m ;


// acceso mediante expresiones
array_access ::= IDENTIFIER:id LPAREN exp:e RPAREN;

func_invoc ::= IDENTIFIER:id act_params:ap
                        | IDENTIFIER:id LPAREN RPAREN;

act_params ::= LPAREN IDENTIFIER:id COMMA list_ids:lids RPAREN
                        |  LPAREN IDENTIFIER:id RPAREN;

// ------------------------------------------ sentencias ------------------------------------------

sent ::= sent_assign:sa
                | sent_if:sif
                | sent_ctrl:sc
```

```
                | sent_io:sio
                | func_invoc:fi SEMICOL
                | array_access:aa SEMICOL
            | error SEMICOL ;


// sentencias de asignacion
sent_assign ::= IDENTIFIER:id ASSIGN exp:e SEMICOL
                        | array_access:aa ASSIGN:id exp:e SEMICOL;


// sentencias if
sent_if ::= IF:id exp:e THEN st_block:sb END IF SEMICOL
                | IF:id exp:e THEN st_block:sbt ELSE st_block:sbe END IF SEMICOL;


// sentencias de control de flujo (solo while en esta espec-A)
sent_ctrl ::= WHILE exp:e LOOP st_block:sb END LOOP SEMICOL;


// sentencias de entrada/salida (putline solo)
sent_io ::= PUTLINE:id LPAREN exp:e RPAREN SEMICOL
                | PUTLINE LPAREN LITERALSTRING:ls RPAREN SEMICOL;

// sentencia return en funciones
sent_return ::= RETURN:id exp:e SEMICOL;


// ------------------------------------ declaracion de funciones ------------------------------------
// (el bloque de sentencias no puede ser vacio ya que return es obligatorio)

func ::= cabecera_func:cab cuerpo_func:cue;

cabecera_func ::= FUNCTION IDENTIFIER:idi LPAREN ref_params:rp RPAREN RETURN INTEGER
                        | FUNCTION IDENTIFIER:idi LPAREN RPAREN RETURN INTEGER;

cuerpo_func ::= IS local_declarations:ld BEGIN fun_st_block:fsb END IDENTIFIER:idf SEMICOL
                        | IS BEGIN fun_st_block:fsb END IDENTIFIER:idf SEMICOL;

fun_st_block ::= f_sent:fs fun_st_block:fsb
                        | f_sent:fs;

// incluimos array_access ya que es como la invocacion de un procedimiento con 1 solo param?
f_sent ::= sent_assign:sa
            | sent_if:sif
            | sent_ctrl:sc
            | sent_io:sio
            | sent_return:sr
            | func_invoc:fi SEMICOL
            | array_access:aa SEMICOL;
```