PRÁCTICA DE PROCESADORES DE LENGUAJES

Curso 2009 – 2010

Entrega de Septiembre

NOMBRE Y APELLIDOS: Juan Antonio Elices Crespo

DNI: 12411633-M

CENTRO ASOCIADO: Palencia – Nueva York(estoy en USA con una beca).

MAIL DE CONTACTO: juanantonio.elices@gmail.com

TELÉFONO DE CONTACTO: (+1) 505-489-0649

983357086 (familiar)

IDENTIFICADOR: jelices1

GRUPO (A ó B): B(implementada entera)

1 Cambios realizados en el analizador léxico y semántico

El analizador léxico no se ha modificado en nada, a excepción de añadir las nuevas palabras claves para incluir todo el lenguaje no solo la parte B. En el analizador sintáctico a parte de las nuevas reglas para añadir las nuevas expresiones, ha habido una ligera modificación en la sentencia for para simplificar el análisis semántico.

2 El analizador semántico y la comprobación de tipos

2.1 Descripción de la Tabla de Símbolos

La tabla de símbolos se ha implementado con una tabla Hash, para optimizar la búsqueda de símbolos comparando con una lista enlazada. También podemos comentar que no hemos hecho mucho hincapié en el algoritmo de la función Hash o resumen, por lo que no aseguramos que sea óptima.

Las funciones o procedimientos se guardan con su nombre mas "( )", de esta manera no se confundirán nunca con una variable (los paréntesis no están permitidos en un identificador).

3 Generación de código Intermedio

3.1 Descripción del código Intermedio generado

Vamos a comentar las instrucciones creadas en el código intermedio y el significado que le hemos dado:

Instrucciones de Código Intermedio	
LABEL etiqueta	Genera una etiqueta(sin símbolos no aceptados), la instrucción NOP que se añade es simplemente para que si hay dos etiquetas seguidas no genere un error.
END	Final del código intermedio, el compilador sabe que tras esta instrucción debe colocar las cadenas de texto en memoria (directiva DATA).
HALT	El programa se parará si llega a esta instrucción
MOVE op1 op2	Copia op1 a op2
ADD op1 op2 res	Suma op1 y op2 y lo coloca en res
SUB op1 op2 res	Resta op1 menos op2 y lo coloca en res
DEC op1	Decrementa op1 (añadida al final para aprovechar las instrucciones de esta arquitectura)
INC op1	Incrementa op1 (añadida al final para aprovechar las instrucciones de esta arquitectura)
COMP op1 op2	Compara op1 con op2, para conocer si son iguales o cual es mayor
OR op1 op2 res	Operación OR entre op1 y op2 y el resultado lo coloca en res
AND op1 op2 res	Operación AND entre op1 y op2 y el resultado lo coloca en res
WRTINT op1	Escribe por pantalla el entero op1
WRTLN	Escribe el carácter salto de línea

WRTSTR cadena	Escribe por pantalla la cadena
ASP op1 op2	Copia en op2 la dirección de memoria de op1 (Notación de PasQal: op2 :=@op1).
CPP op1 op2	Asigna a la dirección de memoria apuntada por op2 el valor de op1, (op2^=op1)
REF op1 op2	Asigna a op2 el valor apuntado por op1, (op2:=op1^)
BP etiqueta	Salto condicionado a que la última operación fuese 0 o mayor
BN etiqueta	Salto condicionado a que la última operación fuese menor de 0
BZ etiqueta	Salto condicionado a que la última operación fuese 0
BNZ etiqueta	Salto condicionado a que la última operación fuese distinto de 0
BR etiqueta	Salto incondicional
ACTUALIZAR-SP	Mueve el puntero de pila a donde de verdad se encuentra la pila, muchas veces no actualizamos el puntero SP al usar la memoria.
PUSH op1	Añade el operando1 a la pila.
VINC-ACC-HIJO	Se crea un vinculo de acceso a un procedimiento hijo, es decir se copia el puntero al registro de activación actual en la pila.
VINC-ACC-HERMANO	Se crea un vinculo de acceso a un procedimiento hermano, es decir se copia el vinculo de acceso actual en la pila.
VINC-CONTROL	Se copia el puntero al registro de activación actual en la pila.
CAMBIAR-FP	Se modifica el puntero al registro de activación, para que apunte a donde acaba la pila en este momento
CALL funcion	Copia el Contador de programa a la pila, y modifica este para que continúe la ejecución en la función.
COPIAR FP-EN-SP	Disminuye la pila hasta el registro de activación.
VINC-CONTROL-A-FP	Restaura el registro de activación al anterior.
SET-VALOR-DEVUELTO op1	Guarda op1 en un sitio accesible por el registro de activación previo, concretamente el registro 9.
CAMBIAR-SP-EXTRAER-ARGUMENTOS op1	Disminuye la pila en op1 palabras, se utiliza para eliminar el contexto saliente.
GET-VALOR-DEVUELTO op2	Copia el valor guardado en op2.
RET	Lee el valor de la pila y lo coloca en el contador de programa, se utiliza para retornar de una función al punto donde se llamó.

4 Generación de código final

4.1 Descripción del registro de activación

El registro de activación generado es (con pila decreciente):

Temporales locales
Variables locales
Dirección de retorno
Enlace de control
Enlace de acceso
Parámetros (si los hay)

El puntero de cuadro apunta a la dirección de retorno, cuando este es el registro de activación en uso.

También es importante resaltar que se ha optado porque el programa principal tenga su registro de activación sin ninguna diferencia, las variables globales solo son variables locales de este registro de activación por lo que para acceder a ellas desde una función se tiene que realizar a través de los enlaces de acceso.

5 Indicaciones especiales

En las interpretaciones del enunciado dudosas se ha optado por seguir en lo más posible las soluciones reales de Pascal, sobretodo en la accesibilidad a funciones. Es decir:

- Una función hijo no puede llamar a la función padre (sería un tipo de recursividad indirecta). La decisión se ha tomado por ser consistente con Pascal original.

- Una función solo puede llamar a funciones hermano que se hayan definido previamente. Es decir que es valido el primer programa pero no el segundo

```
procedure padre();
        procedure hijo1();
        begin
                write("Hijo1");
        end;
        procedure hijo2();
        begin
                hijo1();
                write("Hijo2");
        end;
begin
        hijo2();
end;

procedure padre2();
        procedure hijo1();
        begin
                write("Hijo1");
                        hijo2();
        end;
        procedure hijo2();
        begin
                write("Hijo2");
        end;
begin
        hijo1();
end;
```

- El paso de un puntero por referencia puede hacerse pero el operador de indirección(^) no hará lo esperado, no se implementa la doble indirección.

- Nuestro programa acepta el paso de conjuntos o registros completos por valor, copiando entero dicha variable en la zona de parámetros del registro de activación. A pesar de que el enunciado dice lo contrario, nos dimos cuenta cuando ya estaba implementado.

- Se aceptan los operadores = y <> entre expresiones booleanas, debido a que en los ejemplos del enunciado aparecen, a pesar de que en el texto parece lo contrario.

- No permitimos una variable por referencia, pasarla a otra función por referencia.

```
//Programa erroneo
procedure incrementar (var valor: integer);
begin
	valor:=valor+1;
end;

procedure sumar(var valor:integer; cantidad:integer);
var i:integer;
begin
	i:=cantidad;
	repeat
		incrementar(valor);
		i:=i-1;
	until (i=0) ;
end;
```

- También es de aclarar que el código generado es poco eficiente tanto en la cantidad de memoria usada, como en el número de instrucciones ejecutadas.

    - Cada vez que hay una etiqueta genera una instrucción NOP para evitar problemas

    - Un código entero1:=entero2; genera un temporal, mueve entero2 a este temporal y luego mueve este temporal a entero1, usando una posición de memoria no necesaria y 3 instrucciones de ensamblador no necesarias.

- El compilador nuestro está preparado para que funcione con dirección de crecimiento de pila descendente, es decir la opción por defecto. Si esto se decide cambiar están anotados en el código fuente los cambios que se deberían hacer (básicamente es cambiar los sub por add cuando se añaden elementos a la pila).

- También me gustaría comentar que en la carpeta doc/test/pruebas existen ejemplos de programas, tanto en código fuente como compilados(ensamblador), cubriendo casi todas las características del lenguaje.

6 Conclusiones

Esta práctica a pesar de ser muy laboriosa, creo que ha sido imprescindible para comprender realmente el proceso de creación de un compilador.

A mi entender las fases que me han servido más para entender este proceso, son las de generación de código, tanto intermedio como final, ya que las otras partes se pueden ver mejor al entender la teoría.

7 Gramática (solo para el análisis sintáctico no semántico)

```
Axiom ::= programa;
```

```
programa ::= PROGRAM IDENTIFICADOR:id SEMICOLON declaraciones subprogramas cuerpo
    POINT | error ;

declaraciones ::= seccionConstantes seccionTipos seccionVariables;

seccionConstantes ::= CONST declaracionConstantes| ;

declaracionConstantes     ::=     declaracionConstante     |     declaracionConstante
    declaracionConstantes ;

declaracionConstante ::= listaIdentificadores EQUAL valorConstante SEMICOLON | error
    SEMICOLON ;

listaIdentificadores ::= IDENTIFICADOR:id COMMA listaIdentificadores| IDENTIFICADOR
    | ;

valorConstante ::= TRUE | FALSE |  LITERALENTERO;

seccionTipos ::= TYPE declaracionTipos | ;

declaracionTipos ::= declaracionTipo | declaracionTipo declaracionTipos;

declaracionTipo ::= IDENTIFICADOR EQUAL definicionTipo SEMICOLON | error SEMICOLON;

definicionTipo ::= definicionRegistro | definicionConjunto ;

definicionConjunto ::= SETOF LITERALENTERO TWOPOINTS LITERALENTERO;

definicionRegistro::= RECORD camposRegistro END;

camposRegistro ::= campoRegistro| campoRegistro camposRegistro;

campoRegistro ::= listaIdentificadores COLON tipoPrimitivo SEMICOLON;

tipoPrimitivo ::= INTEGER | BOOLEAN | CARET INTEGER;

seccionVariables ::= VAR declaracionVariables| ;

declaracionVariables      ::=      declaracionVariable      |      declaracionVariable
    declaracionVariables;

declaracionVariable ::= listaIdentificadores COLON tipoVariable SEMICOLON | error
    SEMICOLON;

tipoVariable ::= tipoPrimitivo | IDENTIFICADOR;

subprogramas ::= subprograma subprogramas | ;

subprograma ::=  procedimiento | funcion ;

procedimiento  ::=  PROCEDURE  IDENTIFICADOR  OPENPARENTHESIS  definicionArgumentos
    CLOSEPARENTHESIS  SEMICOLON  declaraciones  subprogramas  cuerpo  SEMICOLON  |
    PROCEDURE error SEMICOLON declaraciones subprogramas cuerpo SEMICOLON ;

definicionArgumentos ::= argumentos |  ;

argumentos ::= argumento | argumento SEMICOLON argumentos;

argumento  ::=  listaIdentificadores  COLON  tipoVariable  |  VAR  listaIdentificadores
    COLON tipoVariable;

funcion    ::=    FUNCTION    IDENTIFICADOR    OPENPARENTHESIS    definicionArgumentos
    CLOSEPARENTHESIS COLON tipoPrimitivo SEMICOLON declaraciones subprogramas cuerpo
```

```
    SEMICOLON | FUNCTION error SEMICOLON declaraciones subprogramas cuerpo SEMICOLON
    | FUNCTION error BEGIN listaSentencias END SEMICOLON;

cuerpo ::= BEGIN listaSentencias  END ;

listaSentencias ::= sentencia listaSentencias | |error;

sentencia  ::=  sentenciaAsignacion  SEMICOLON    |  sentenciaIf  |  sentenciaFor  |
    sentenciaRepeat SEMICOLON | llamadaFuncionOProcedimiento SEMICOLON | sentenciaES
    SEMICOLON | error SEMICOLON ;

sentenciaAsignacion ::= referencia ASSIGMENT expresion;

referencia  ::=  IDENTIFICADOR  |  IDENTIFICADOR  POINT  IDENTIFICADOR  |  IDENTIFICADOR
    CARET;

expresion  ::=  valorConstante  |  OPENPARENTHESIS  expresion  CLOSEPARENTHESIS  |
    referencia | llamadaFuncionOProcedimiento | expresionPuntero | expresion operador
    expresion | expresionConjunto | expresionPertenencia | error ;

operador ::= MINUS| OR | GREATERTHAN | EQUAL | NOTEQUAL| AND| LESSTHAN | PLUS;

llamadaFuncionOProcedimiento  ::=  IDENTIFICADOR  OPENPARENTHESIS  valorParametros
    CLOSEPARENTHESIS | IDENTIFICADOR OPENPARENTHESIS CLOSEPARENTHESIS ;

valorParametros ::= expresion COMMA valorParametros | expresion;

expresionPuntero ::= AT IDENTIFICADOR;

expresionPertenencia ::= expresion IN IDENTIFICADOR;

expresionConjunto    ::=    OPENSQUAREBRACKET    expresion    TWOPOINTS    expresion
    CLOSESQUAREBRACKET | OPENSQUAREBRACKET CLOSESQUAREBRACKET ;

sentenciaIf ::= IF OPENPARENTHESIS expresion CLOSEPARENTHESIS THEN bloqueSentencias
    parteElse | IF error bloqueSentencias;

parteElse ::= ELSE bloqueSentencias | ;

bloqueSentencias ::= sentencia| BEGIN listaSentencias END SEMICOLON;

sentenciaFor   ::=   FOR   OPENPARENTHESIS   sentenciaAsignacion   TO   expresion
    CLOSEPARENTHESIS DO bloqueSentencias;

sentenciaRepeat  ::=  REPEAT  listaSentencias  UNTIL  OPENPARENTHESIS  expresion
    CLOSEPARENTHESIS;

sentenciaES ::= WRITE OPENPARENTHESIS mostradoPantalla CLOSEPARENTHESIS | WRITELN
    OPENPARENTHESIS CLOSEPARENTHESIS;

mostradoPantalla ::= CADENA | expresion;
```