PRÁCTICA DE
PROCESADORES DEL LENGUAJE II
Curso 2019 – 2020
Entrega de Junio

APELLIDOS Y NOMBRE: PALMA SEBASTIÀ, ISRAEL

IDENTIFICADOR: ipalma13

DNI: 20236985K

CENTRO ASOCIADO MATRICULADO: C.A. VALÈNCIA

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: C.A. VALÈNCIA

EMAIL DE CONTACTO: ipalma13@alumno.uned.es

TELÉFONO DE CONTACTO: 610101060

GRUPO: B

¿REALIZAS LA PARTE OPCIONAL?: NO

ÍNDICE

0.Introducción
1.El analizador semántico y la comprobación de tipos
1.a. Descripción del manejo de la Tabla de Símbolos y de la Tabla de Tipos
2.Generación de código intermedio
2.a. Descripción de la estructura utilizada
3.Generación de código final
3.a. Descripción de hasta dónde se ha llegado
4. Indicaciones especiales y conclusión

0. Introducción

Este documento es la memoria de la práctica de procesadores del lenguaje 2. En ella voy a comentar el proceso de realización de la práctica, dificultades encontradas, decisiones tomadas y otros que sean de relevancia.

1. El analizador semántico y la comprobación de tipos

El primer paso realizado para esto fue el de buscar los puntos donde abrir y cerrar los ámbitos. En la práctica encontramos los posibles ámbitos: global, de procedimiento o función, de la sentencia if-else y de la sentencia for. He declarado en el parser un atributo entero que sirve para la creación de los nombres de los ámbitos de las sentencias if-else y for.
A continuación, se añaden al ámbito global los tipos primitivos INTEGER y BOOLEAN.
Luego se crean las clases con los métodos que irán "ligados" a los no terminales para la correcta implementación de la práctica. Algunos no terminales usarán clases que ya existen, como podría ser vBooleano que usará la clase Boolean de Java o intOBool que usará la clase TypeSimple de la práctica.
Se deberá propagar los no terminales para que los no terminales superiores puedan hacer uso de ellos.
Por último, pasaremos a implementar las comprobaciones semánticas que explicaremos en el siguiente punto (1.a).

a. Descripción del manejo de la Tabla de Símbolos y de la Tabla de Tipos

En cuanto a los tipos, encontramos 5 clases usadas en la práctica:
-**TypeSimpleBoolean**: representa el tipo BOOLEAN de la práctica, hereda de TypeSimple.
-**TypeSimpleInteger**: representa el tipo INTEGER de la práctica, hereda de TypeSimple.
-**TypeArray**: representa el tipo Vector de la práctica, tipo compuesto.
-**TypeProcedure**: reprensta el tipo para los procedimientos en la práctica.
-**TypeFunction**: representa el tipo para las funciones en la práctica.

Respecto a los símbolos, encontramos 5 clases usadas en la práctica:
-**SymbolConstant**: representa los símbolos constantes en la práctica.
-**SymbolVariables**: representa los símbolos de las variables en la práctica.
-**SymbolParameter**: representa los símbolos de los parámetros en la práctica.
-**SymbolFunction**: representa los símbolos de las funciones en la práctica.
-**SymbolProcedure**: representa los símbolos de los procedimientos en la práctica.

Todos los identificadores se añaden en mayúscula porque el lenguaje ModulUned no disitingue entre mayúsculas y minúsculas.

A continuación, voy a comentar las comprobaciones semánticas tenidas en cuenta:

No terminal	Comprobación semántica	Explicación
expConst	¿Ya existe el identificador?	Se comprueba que no exista el identificador en la tabla de símbolos para añadir el símbolo. En caso de existir, salta un error semántico.
expTipo	¿El rango del vector es correcto?	Se comprueba que el inicio del vector sea menor que el fin. En caso contrario, salta un error semántico.
expTipo	¿Ya existe el identificador?	Se comprueba que no exista el identificador en la tabla de símbolos para añadir el símbolo. En caso de existir, salta un error semántico.
expVar	¿Ya existe el identificador?	Se comprueba que no exista el identificador en la tabla de símbolos para añadir el símbolo. En caso de existir, salta un error semántico.
tipoVar	¿Está declarado el tipo?	Se comprueba que el tipo esté declarado en la tabla de tipos. En caso negativo, salta un error semántico.
stmSubprogram	¿Coinciden tipoRetorno y la expresión de la sentencia return?	Se comprueba que la función devuelve una expresión del mismo tipo que el tipo que tiene escrito en su implementación. En caso negativo, salta un error semántico.
stmSubprogram	¿Existe la sentencia return?	Se comprueba que la función tenga un tipo de retorno. En caso negativo, salta un error semántico.
cabProcedure	¿Ya existe el identificador?	Se comprueba que no exista el identificador en la tabla de símbolos para añadir el símbolo. En caso de existir, salta un error semántico.
cabProcedure	¿Ya existe el identificador del parametro?	Se comprueba que no exista el identificador de los paramétros en la tabla de símbolos para añadir el símbolo. En caso de existir, salta un error semántico.
exprArit	¿Las expresiones son de tipo INTEGER?	Se comprueba que las dos expresiones sean de tipo INTEGER. En caso negativo, salta un error semántico.
exprLogica	¿Las expresiones son de tipo BOOLEAN?	Se comprueba que las dos expresiones sean de tipo BOOLEAN. En caso negativo, salta un error semántico.
sentAsign	¿Se le está haciendo una asignación a una constante?	Se comprueba que no se esté tratando de hacer una asignación a una constante. En caso afirmativo, salta un error semántico.
sentAsign	¿Se está haciendo una asignación entre tipos que no concuerdan?	Se comprueba que variables y expresiones sean del mismo tipo. En caso negativo, salta un error semántico.
sentIf	¿La expresión condicional es de tipo BOOLEAN?	Se comprueba que la expresión que condiciona el IF sea de tipo booleano. En caso negativo, salta un error semántico.
sentFor	¿Las expresiones que indican el principio y el fin de la sentencia son enteros?	Se comprueba que las dos expresiones que indican el principio y el fin de la sentencia FOR son de tipo INTEGER. En caso negativo, salta un error semántico.
sentFor	¿Ya existe el identificador?	Se comprueba que no exista el identificador en la tabla de símbolos para añadir el símbolo. En caso de existir, salta un error semántico.
sentProcedure	¿Existe la sentencia return?	Se comprueba que se está haciendo una llamada a un símbolo existente. En caso negativo, salta un error semántico.

sentProcedure	¿El símbolo al que se está llamando es una función o un parámetro?	Se comprueba que se está llamando a un procedimiento o a una función. En caso negativo, salta un error semántico.
sentProcedure	¿Coinciden los parámetros?	Se comprueba que los parámetros de la llamada coinciden con los parámetros de la función o procedimiento que se llama. En caso negativo, salta un error semántico.
sentProcedure	¿Son correctos los parámetros?	Se comprueba que el tipo de los parámetros de la llamada coinciden con los parámetros de la función o procedimiento que se llama. En caso negativo, salta un error semántico.
sWriteInt	¿Se va a imprimir un entero?	Se comprueba que la expresión sea de un entero. En caso negativo, salta un error semántico.
variables	¿Ya existe el identificador?	Se comprueba que no exista el identificador en la tabla de símbolos para añadir el símbolo. En caso de existir, salta un error semántico.
variables	¿El acceso al vector está dentro del rango?	Se comprueba que se accede a una posición existente del vector. En caso negativo, salta un error semántico.
variables	¿El acceso al vector es con un entero?	Se comprueba que se accede al vector mediante un entero. En caso negativo, salta un error semántico.
variables	¿Coinciden los parámetros?	Se comprueba que el tipo de los parámetros de la llamada coinciden con los parámetros de la función o procedimiento que se llama. En caso negativo, salta un error semántico.
variables	¿Son correctos los parámetros?	Se comprueba que el tipo de los parámetros de la llamada coinciden con los parámetros de la función o procedimiento que se llama. En caso negativo, salta un error semántico.
entOid	¿El identificador es de tipo INTEGER?	Se comprueba que el identificador sea de tipo INTEGER. En caso contrario, salta un error semántico.
entOid	¿Existe el identificador?	Se comprueba que el identificador exista. En caso negativo, salta un error semántico.

Para la comprobación de tipos he creado un método en el parser, llamado getTypeExpresion que devuelve el tipo de una Expresion.

2. Generación de código intermedio

Como ya sabemos, el código intermedio se genera con cuádruplas, estas cuádruplas se componen de [INSTRUCCIÓN, RESULTADO, OPERACIÓN 1, OPERACIÓN2].
En este caso he localizado los no terminales donde se debe generar código intermedio, este código intermedio luego pasará a ser código final que se ejecutará en el simulador ENS2001.
En estos no terminales, generaremos las cuádruplas.
Habrá puntos en los que tendremos que propagar las cuádruplas de sus hijos para que todas las cuádruplas acaben llegando al código intermedio del axiom.
A continuación, haré un resumen de todos los tipos de cuádruplas que he usado.

a. Descripción de la estructura utilizada

Cuádrupla	Resultado	Operando1	Operando2	Explicación
VAR	Variable	0		Asigna las variables globales a una dirección.
HALT				Indica el fin del programa
MV	Temporal	Value		Mueve el Value al resultado
MVA	Temporal	Variable		Mueve la dirección del operando al resultado
REF	Temporal	Variable		Carga el valor de una varible tras un MVA
SUB	Temporal	Temporal	Temporal	Resta los temporales de los dos operandos y lo guarda en el resultado
MUL	Temporal	Temporal	Temporal	Multiplica los temporales de los dos operandos y lo guarda en el resultado
GR	Temporal	Temporal	Temporal	Hace la función mayor que con los dos operandos y lo guarda en el resultado
EQ	Temporal	Temporal	Temporal	Hace la función igual que con los dos operandos y lo guarda en el resultado
OR	Temporal	Temporal	Temporal	Hace la función OR con los dos operandos y lo guarda en el resultado
NOT	Temporal	Temporal		Hace la función NOT con el operando y lo guarda en el resultado.
STP	Temporal	Temporal		Guarda el valor del operando al resultado
BRF	Temporal	Label		Salta a la etiqueta si el resultado es falso
BR	Label			Salta a la etiqueta
INL	Label			Inseta la etiqueta
INC	Variable			Incrementa en 1 el valor del resultado
WRITESTRING	Temporal	Label		Imprime lo que hay en la etiqueta del resultado
CADENA	Label	Label		Crea una label de texto
WRITEINT	Temporal			Imprime el resultado
WRITELN				Imprime salto de linea

3. Generación de código final

La generación del código final consiste en la traducción de las cuádruplas generadas en el código intermedio para que puedan ser ejecutadas con el simulador ENS2001.
Para esto, hemos implementado en la clase ExecutionEnvironmentEns2001 dos métodos que hacen posible esta traducción.
Por una parte, tenemos el método translate, que irá comprobando la operación que recibe de cada cuádrupla y hará su correspondiente traducción.
Por otro, el método operacion comprobará cada elemento de la cuádrupla y hará su traducción en función de si es un Variable, un Value, un Temporal, o una Label.

Es importante también comentar el código implementado para la distribución de las direcciones. En el no terminal program he implementado un código que recorre los scopes. En cada scope recorre sus símbolos para asignar una dirección desde la 65535 hacia atrás y resta a la dirección el tamaño de dicha variable para saber la dirección de la siguiente variable. A su vez, crea las cuádruplas para inicializar las variables globales cuando se está recorriendo el scope global. Luego hace esto mismo pero con los temporales. Por último, añade la cuádrupla de finalización del programa y las de texto.

a. Descripción de hasta dónde se ha llegado

He realizado todo lo que corresponde con la parte obligatoria de la práctica, es decir, el análisis semántico completo y la implementación del código intermedio y final a excepción de los subprogramas.
Trasladándolo a los 11 tests sugeridos, el análisis semántico funciona en todos ellos.
En cuanto al código final, de los 8 test que compilan, todos generan el código final esperado a excepción del testCase06, ya que este contiene subprogramas.

4. Indicaciones especiales y conclusión

Sobre fallos conocidos, los comentarios no funcionan correctamente si contienen acentos, salta una excepción. He modificado las letras que contenían acentos de los tests para que funcionen correctamente.

En la carpeta de test se encuentran los 8 ens generados por si fuera necesario evaluarlos.
Se ha evitado entrar en más detalle y en la inclusión de código en la memoria para lograr una memoria amena, tal y como se recomienda en el enunciado.

La conclusión de la práctica es que logra totalmente su objetivo, ya que consigues conocer de principio (incluyendo también la práctica de PL1) a fin el funcionamiento de los lenguajes de programación. Ahora tengo otra visión a la hora de afrontar el estudio de un nuevo lenguaje de programación que me ayuda a comprenderlos mejor. Si tuviera que indicar alguna parte negativa, mencionaría la dificultad de la práctica y lo larga que es, sobre todo al inicio cuesta mucho, pero una vez empiezas a comprender cada parte, todo es bastante más orgánico. A pesar de todo esto, puedo afirmar que es una de las prácticas que más me han hecho aprender de la materia de todas las realizadas en todo el grado.