# PRÁCTICA DE PROCESADORES DE LENGUAJES

## Curso 2013 – 2014

## Entrega de Septiembre

APELLIDOS Y NOMBRE: Andrés Gil Rodríguez

IDENTIFICADOR: agil146

DNI:36155351H

CENTRO ASOCIADO MATRICULADO: Pontevedra

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: A Coruña

MAIL DE CONTACTO:andresgil@gmail.com

TELÉFONO DE CONTACTO:696773038

GRUPO (A ó B): B

# 1. Cambios realizados en los analizadores léxico y sintáctico

En el analizador léxico no he realizado modificaciones, ya que el reconocimiento de los Tokens ya era correcto.

En el analizador sintáctico realice varios cambios, que fueron principalmente:

- La introducción de las producciones para las referencias (que representa bien un identificador o bien el acceso al campo de un registro) ya que inicialmente no lo había desarrollado así en la gramática:

```
referencia ::= referencia POINT IDENTIFIER
             | IDENTIFIER;
```

- Adicionalmente, para las declaraciones del programa (constantes simbólicas, tipos, variables, subprogramas) inicialmente tenía reglas gramaticales que no contemplaban bien el orden obligatorio de dichas declaraciones. Se cambió la gramática para que las declaraciones fuesen consecutivas.

```
axiom          ::= PROCEDURE IDENTIFIER LEFTBRACKET RIGHTBRACKET IS declaraciones;
declaraciones  ::= seccionConstantes;
seccionConstantes  ::= declaracionConstante seccionConstantes | seccionTipos;
seccionTipos       ::= declaracionTipo seccionTipos | seccionVariables;
seccionVariables   ::= declaracionVariable SEMICOLON seccionVariables | seccionSubprogramas;
seccionSubprogramas ::= declaracionSubprograma seccionSubprogramas | seccionCuerpoPrograma;
```

- Así mismo, para la declaración de los subprogramas, se reutilizaron estas producciones, dado que un subprograma tiene, salvo las constantes, las mismas declaraciones.

```
declaracionSubprograma ::=
FUNCTION IDENTIFIER LEFTBRACKET parametrosFormales RIGHTBRACKET RETURN tipoPrimitivo IS seccionTipos
| PROCEDURE IDENTIFIER LEFTBRACKET parametrosFormales RIGHTBRACKET IS seccionTipos;
```

# 2. El analizador semántico y la comprobación de tipos

Se implementaron las siguientes clases de No Terminales:

```
non terminal program;
non terminal Axiom axiom;
non terminal Declaraciones declaraciones, seccionConstantes, seccionTipos, seccionVariables,
                           seccionSubprogramas;
non terminal BloqueSentencias bloqueSentencias;
non terminal SeccionCuerpoPrograma seccionCuerpoPrograma;
non terminal DeclaracionConstante declaracionConstante;
non terminal DeclaracionTipo declaracionTipo;
non terminal DeclaracionVariable declaracionVariable;
non terminal DeclaracionSubprograma declaracionSubprograma;
non terminal ParametrosFormales parametrosFormales;
non terminal ListaParametrosFormales listaParametrosFormales;
```

```
non terminal ValorConstante valorConstante;
non terminal TipoPrimitivo tipoPrimitivo;
non terminal ListaIdentificadores listaIdentificadores;
non terminal Sentencia sentencia;
non terminal SentenciaIO sentenciaIO;
non terminal SentenciaAsignacion sentenciaAsignacion;
non terminal SentenciaIf sentenciaIf;
non terminal SentenciaReturn sentenciaReturn;
non terminal LlamadaProcedimiento llamadaSubprograma;
non terminal SentenciaFor sentenciaFor;
non terminal BloqueCamposRegistro bloqueCamposRegistro;
non terminal ExpresionAccesoRegistro expresionAccesoRegistro;
non terminal Expresion expresion;
non terminal ExpresionAritmetica expresionAritmetica;
non terminal ExpresionLogica expresionLogica;
non terminal ParametrosActuales parametrosActuales;
non terminal Empty empty;
non terminal Referencia referencia;
```

Se podrían agrupar muchos elementos en clases únicas, pero no se hizo inicialmente por simplicidad y finalmente por falta de tiempo.

## 2.1. Descripción de la Tabla de Símbolos implementada

La tabla de símbolos implementada fue la proporcionada por la arquitectura de la práctica del equipo docente.

# 3. Generación de código intermedio

Tras la implementación de los aspectos léxicos, sintácticos y semánticos del compilador, se ha verificado que el código fuente no tiene errores.

Esta cuarta fase es la encargada de traducir el programa fuente a una representación de código intermedio, independiente de una arquitectura física concreta, para que luego pueda ser transformado en fases posteriores a un código final. Para ello, se añaden acciones semánticas en la especificación de Cup, de forma que el proceso de análisis sintáctico vaya generando una secuencia de instrucciones de código intermedio, que se irán acumulando en los atributos de los no terminales, y que llevarán a que al final del proceso, se alcance en el axioma una representación completa del código intermedio de todo el árbol de análisis sintáctico.

## 3.1. Descripción de la estructura utilizada

Para las instrucciones de código intermedio se ha utilizado una estructura de cuádruples (código de tres direcciones). El juego de instrucciones principales del código intermedio implementado ha sido el siguiente:

| Operación | Resultado | Operando 1 | Operando 2 | Descripción |
|---|---|---|---|---|
| INICIO | Variable | Valor | | Inicio del programa |
| FIN | | | | Fin del código del programa |
| DATA | | | | Escritura de las cadenas de texto del programa |
| LABEL | Etiqueta | | | Creación de etiqueta |
| MV | Temporal | Operando1 | | Mueve el operando 1 al elemento temporal |
| MVREG | Temporal | Operando1 | Operando2 | |
| ADD | Temporal | Operando1 | Operando2 | Suma operandos 1 y 2 y los deja en temporal |
| INC | Variable | | | Incrementa la variable en 1 |
| CMP | Operando1 | Operando2 | | Compara los operandos oper1 y oper2, restándolos |
| ASSIGN | Temporal | Value | | Asigna el valor al elemento temporal |
| WRITELN | Value | | | Escritura de valor |
| WRITEEXP | Temporal | | | Escritura de expresión |
| BP | Label | | | Si el resultado es positivo, salta a la etiqueta |
| BN | Label | | | Si el resultado es negativo, salta a la etiqueta |
| BZ | Label | | | Si el resultado es 0, salta a la etiqueta |
| BNZ | Label | | | Si el resultado no es 0, salta a la etiqueta |
| BR | Label | | | Salto incondicional al código de la etiqueta |
| CALL | Procedure | | | Llamada a función |
| RETURN | | | | Retorno de función |
| NOP | | | | No operación |

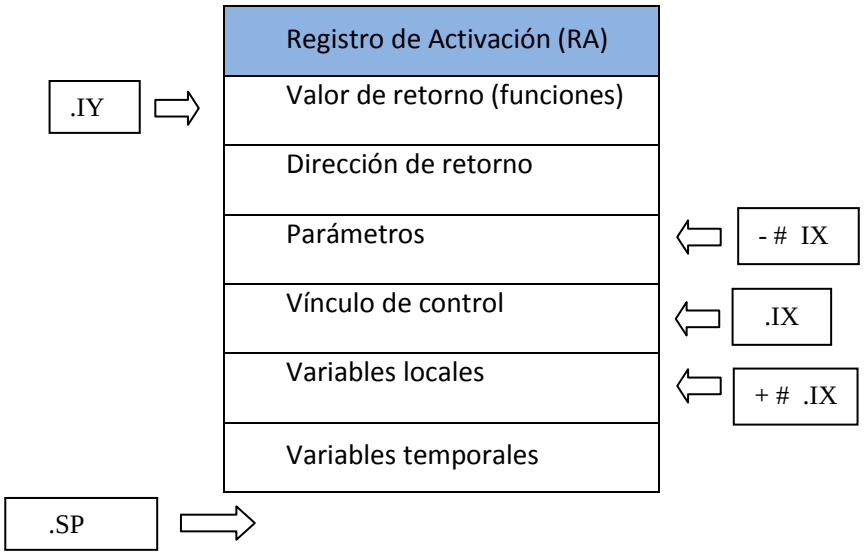
# 4. Generación de código final

En esta fase se traduce la secuencia de instrucciones de código intermedio en el código final listo para ejecutarse en una arquitectura física específica. En el caso de esta práctica el código final es código ensamblador del emulador ENS2001.

Para el proceso de traducción de cuádruplas se creó un objeto 'Translator', como clase abstracta en la que se implementa un método translate, y se implementó una subclase por cada cuádrupla: TranslatorAdd, TranslatorAssign, TranslatorBranch, TranslatorWriteLn, etc.. (paquete compiler.code.translate).

## 4.1. Descripción del registro de activación implementado

El registro de activación se implementa con los siguientes campos:




# 5. Indicaciones especiales

Partes incompletas en la práctica: No ha sido posible finalizar completamente la implementación de la práctica por falta de tiempo y han quedado partes sin terminar completamente, o que no funcionan correctamente como deberían, relacionadas con el tipo estructurado registro y con los subprogramas y los registros de activación. De dichas partes se ha entendido el concepto explicado en teoría, y se estaba intentando implementarlo a marchas forzadas para la fecha de entrega; por desgracia, a la hora de llevarlo a la implementación en código ha llevado más tiempo y se han generado errores, cambios y dudas con la implementación.
Aun así, y dado que esta es la última convocatoria de entrega de la práctica, he preferido entregar la práctica como está, considerando que el grueso de la práctica está realizado y que faltaría finalizar y completar lo indicado.

# 6. Conclusiones

La práctica de esta asignatura es realmente una práctica bonita. Aun así, guardo con ella una relación de amor/odio. La razón de ello es que por un lado es muy interesante ir construyendo un compilador de forma

progresiva y ver cómo van encajando todas las piezas como un puzzle; por el otro lado, la carga de trabajo es literalmente 'infernal' y me ha llevado todo el verano su realización. Es una práctica muy, muy larga, y que requiere una gran cantidad de horas de dedicación (desde luego muy superiores a los créditos de la asignatura). Este trabajo de implementación no es difícil como tarea en sí misma, sino que la dificultad está en el encadenamiento (y entendimiento) del funcionamiento de todas las partes de la construcción del compilador completo: análisis léxico, sintáctico, semántico, etc.., y en ir avanzando en cada parte lo mejor posible, para luego no tener que volver de nuevo demasiadas veces; aun así, el ir corrigiendo aspectos de partes anteriores en la elaboración del compilador, es inevitable.

La primera parte de la práctica, análisis léxico y sintáctico, es relativamente sencilla. Una vez comprendidos el funcionamiento de ambas partes, así como de Lex y de Cup en sí, el desarrollo se acelera. La parte más complicada es la realización de la gramática, dado que en el fondo, no se tiene la certeza completa de si la gramática está todo lo correctamente definida que debería, o se tienen dudas de si una implementación sería mejor que otra, y esto no se comprueba hasta que se llega a partes más avanzadas en la construcción del compilador, y se ve la necesidad de pequeñas correcciones en la gramática.

En cuanto a la segunda parte, la implementación es realmente muy larga. El trabajo más arduo ha sido la introducción de las acciones semánticas y el entender cómo encajaban todas las piezas del análisis semántico, código intermedio y código final.
Aunque ayuda el hecho de tener las tareas Ant de ejecución y prueba, el procedimiento de debug de errores termina haciéndose muy farragoso, teniendo que realizarse mediante múltiples sentencias de salida (como semanticErrorManager, etc..), ejecutando las tareas Ant, y visualizando a mano los resultados. Es decir, en múltiples ocasiones es un formato de depuración de prueba y error.

Como conclusión, quisiera agradecer al equipo docente la atención prestada, y muy especialmente al coordinador de la asignatura Anselmo Peñas, por su inestimable ayuda a la hora de conseguir una sesión de control iniciado ya el mes de Julio. Sin su ayuda no hubiera sido posible el conseguir una sesión de control para esta convocatoria de Septiembre.

## 7. Gramática

Se incluye de forma esquemática las producciones de la gramática generada (sin incluir ni acciones semánticas ni atributos)

```
program    ::= axiom;
axiom      ::= PROCEDURE IDENTIFIER LEFTBRACKET RIGHTBRACKET IS declaraciones;
declaraciones          ::= seccionConstantes;
seccionConstantes          ::= declaracionConstante seccionConstantes | seccionTipos;
seccionTipos               ::= declaracionTipo seccionTipos | seccionVariables;
seccionVariables           ::= declaracionVariable SEMICOLON seccionVariables |  seccionSubprogramas; seccionSubprogramas     ::=
declaracionSubprograma seccionSubprogramas | seccionCuerpoPrograma;
seccionCuerpoPrograma   ::= BEGIN bloqueSentencias END IDENTIFIER SEMICOLON;

listaIdentificadores     ::= listaIdentificadores COMMA IDENTIFIER
                           | IDENTIFIER;
declaracionConstante ::=  listaIdentificadores COLON CONSTANT ASSIGN valorConstanteSEMICOLON;

valorConstante   ::= TRUE
                   | FALSE
                   | NUMBER;

declaracionTipo    ::= TYPE IDENTIFIER IS RECORD bloqueCamposRegistro END RECORD SEMICOLON;

bloqueCamposRegistro ::=   bloqueCamposRegistro declaracionVariable SEMICOLON
                           | declaracionVariable SEMICOLON;
declaracionVariable ::= listaIdentificadores COLON tipoPrimitivo;
tipoPrimitivo ::= INTEGER
                | BOOLEAN
                | IDENTIFIER;
declaracionSubprograma  ::= FUNCTION IDENTIFIER LEFTBRACKET parametrosFormales RIGHTBRACKET RETURN tipoPrimitivo IS seccionTipos
                           | PROCEDURE IDENTIFIER LEFTBRACKET parametrosFormales RIGHTBRACKET IS seccionTipos;
parametrosFormales   ::= listaParametrosFormales
                       | empty;
listaParametrosFormales  ::=  listaParametrosFormales SEMICOLON listaIdentificadores COLON tipoPrimitivo
                            | listaIdentificadores COLON tipoPrimitivo;
bloqueSentencias  ::=    sentencia SEMICOLON bloqueSentencias
                       | empty;
```

```
sentencia  ::=    sentenciaAsignacion
                | sentenciaIf
                | sentenciaIO
                | sentenciaReturn
                | sentenciaFor
                | llamadaSubprograma;

expresion  ::=    expresionAritmetica
                | expresionLogica
                | LEFTBRACKET expresion RIGHTBRACKET
                | llamadaSubprograma
                | referencia;

expresionAritmetica ::=    expresion PLUS expresion
                         | NUMBER;
expresionLogica      ::=   expresion OR expresion
                         | expresion GREATTHAN expresion
                         | expresion EQUAL expresion
                         | TRUE
                         | FALSE;

referencia  ::=    referencia POINT IDENTIFIER
                | IDENTIFIER;

sentenciaAsignacion  ::=     referencia ASSIGN expresion;

sentenciaIO ::=     PUTLINE LEFTBRACKET expresion RIGHTBRACKET
                  | PUTLINE LEFTBRACKET STRING RIGHTBRACKET;
sentenciaIf::=    IF expresion THEN bloqueSentencias END IF
                | IF expresion THEN bloqueSentencias ELSE bloqueSentencias END IF;
sentenciaReturn ::=    RETURN expresion;

llamadaSubprograma  ::=    IDENTIFIER LEFTBRACKET parametrosActuales RIGHTBRACKET;
parametrosActuales ::=    parametrosActuales COMMA expresion | expresion | empty;

sentenciaFor::=    FOR IDENTIFIER IN expresion DOUBLEPOINT expresion LOOP bloqueSentencias END LOOP;

empty ::= ;
```