PRÁCTICA FINAL

Compiladores - 21780

Curso 2021-2022

Jonathan Salisbury Vega
Joan Sansó Pericàs
Julián Wallis Medina
https://youtu.be/Soej4bxWSuM

Universitat de les Illes Balears

Índice

Analizador Léxico 2
Tokens 2
Generación del Escáner 3
Restricciones y Peculiaridades 3
Analizador Sintáctico 5
Gramática 5
Generación del Analizador 7
Restricciones y Peculiaridades 8
Analizador Semántico 9
Tabla de Símbolos 9
Recorrido del Árbol Sintáctico 10
Generación de Código 11
Código de Tres Direcciones 11
Generación de Código intermedio 13
Generación de Código Ensamblador 15
Restricciones y Peculiaridades 16
Optimización 16
Compilador 16
Gestión de Errores 17
Pruebas 18
Conclusiones 18
Referencias 19

Analizador Léxico

Tokens

Se han definido todos los *tokens* necesarios para la gramática del lenguaje de nuestro compilador, a continuación, se encuentra la lista completa:

- *Keywords*:
  - Int
  - Bool
  - Const
  - If
  - Else
  - While
  - Crotofunc
  - Break
  - Return
  - And
  - Or
  - Not
  - True
  - False
  - Main

En primer lugar, están las palabras reservadas por el lenguaje para funciones internas, estas palabras no podrán ser utilizadas como identificadores por el usuario ya que el escáner las detectará como *tokens* especiales.

- *Operators*:
  - =
  - +
  - -
  - *
  - /
  - ==
  - !=
  - >
  - <
  - >=
  - >=

En segundo lugar, encontramos los distintos operadores del lenguaje, estos incluyen los operadores aritméticos y de comparación.

- *Separators*:
  - (
  -)
  - {
  - }
  - ;
  - ,

Por último, tenemos los distintos separadores del lenguaje, estos tokens se utilizan para distinguir diferentes partes del código fuente como, por ejemplo, funciones, bloques de código o finales de línea.

Cabe destacar que también están definidos tokens para poder representar los valores literales de los distintos tipos de datos del lenguaje.

Generación del Escáner

Para el apartado del analizador léxico del compilador se ha utilizado la herramienta de generación de escáner *JFlex* [1]. Como se puede observar en el código del generador localizado en */jflex/Lexer.flex* existen tres grandes apartados para la generación del escáner que explicaremos a continuación.

- **Código de Usuario**:
  En este apartado inicial se puede encontrar código fuente de *Java* que se utilizara posteriormente en el archivo escáner generado. El código está formado en su totalidad de *imports* de clases para la creación de los símbolos correspondientes a partir de los *tokens*.

- **Opciones y Declaraciones**:
  En este apartado se indican diferentes parámetros a la herramienta *JFlex*, como por ejemplo el nombre final del escáner generado, el tipo de caracteres utilizados o que mantenga una cuenta de la línea y columna de cada *token*. Además, también se aplican los ajustes necesarios para que sea compatible con la herramienta *CUP* que se utiliza más adelante para el analizador sintáctico. A continuación, también definimos una sección de código *Java* donde creamos algunas funciones auxiliares para poder crear correctamente las instancias de los Símbolos y gestionar errores. Por último, se definen algunas variables que servirán para definir las reglas léxicas a continuación, algunos ejemplos son las expresiones regulares para los comentarios, identificadores o literales.

- **Reglas Léxicas**:
  En este último apartado se encuentran las reglas léxicas del escáner, donde se define como detectar y tratar los distintos *tokens* definidos anteriormente. En general se define la estructura de cada *token* utilizando su representación literal o utilizando las variables de expresiones regulares definidas en el apartado anterior. Una vez identificado el *token* en cuestión, se define el código *Java* a ejecutar en cada caso. Este código utiliza los métodos definidos anteriormente para generar las instancias de símbolo correctas para cada *token*, indicando el *token* en cuestión, el identificador del token y un valor en los casos necesarios.

Cabe destacar que uno de los métodos definidos en el apartado de declaraciones, es para la muestra correcta de los errores léxicos en caso de haber alguno. Este método utiliza la información de la línea y columna del token en cuestión para indicar al usuario de manera clara el error encontrado.

Restricciones y Peculiaridades

El lenguaje de este compilador se ha basado en la gramática del lenguaje *Java* por tanto la mayoría de los tokens necesarios para definir dicha gramática se comparten. Hay algunas excepciones:

- **Identificadores:** Los identificadores tienen más restricciones en este lenguaje. En primer lugar, deben comenzar obligatoriamente por un carácter del alfabeto y después pueden estar seguidos de más caracteres o dígitos numéricos. Se ha restringido el uso de "_" para gestionar internamente el apartado de generación de Código.

- **Comentarios:** Como se ha explicado anteriormente el lenguaje tiene un gran parecido con *Java* y en este caso comparte la misma definición de los distintos comentarios, tanto los de una línea, como los de múltiples líneas.

- ***Keywords*:**
    - **Operadores Lógicos:** los operadores *and, or y not* se representan con las mismas palabras, en lugar de utilizar los tokens de *Java* (&&, ||, !).
    - **Tipo Booleano:** para definir variables de tipo *booleano* se ha definido un *token* especial "*bool*". Además, se ha modificado la representación de los literales *booleanos* "*True*" y "*False*" para que comiencen por mayúscula de una forma parecida al lenguaje *Python*.
    - **Otros:** finalmente se han definido algunos tokens especiales del lenguaje como "*crotofunc"* para la definición de funciones o "*const*" para la definición de variables constantes.

Cualquier carácter que no esté definido dentro de un *token* o en una expresión regular lanzará un error de carácter no reconocido.

Analizador Sintáctico

Gramática

A continuación, se explica la gramática definida para el lenguaje del compilador. En primer lugar, se explica la estructura básica de un programa en CROTO.

```
1.  start with program;
2.  program         → main | methods main
3.  main            → CROTOFUNC MAIN LPAREN RPAREN LBRACE code_block RBRACE
4.  methods         → method | methods method
```

La unidad mínima de compilación definida es *program*, un programa puede estar formado por una lista opcional de métodos (*methods*), y obligatoriamente un método *main* al final.

La lista de métodos *methods* puede contener un único método o una lista de varios, el *main* tendrá la forma `crotofunc main(){ code_block },` donde *code_block* es un bloque de código que contiene todas las instrucciones a ejecutar.

La estructura de un método está definida en la gramática de la siguiente forma:

```
1.   method          → CROTOFUNC type id LPAREN param_list RPAREN LBRACE code_block RBRACE
```

Donde el *token* "*crotofunc*" es obligatorio para indicar el inicio de una función, *type* hace referencia al tipo de retorno de la función que puede ser *bool, int* o omitirlo en caso de que la función no retorne ningún valor. Seguidamente está el *string* identificador con el nombre de la función seguido de un paréntesis abierto, una definición opcional de parámetros y un paréntesis cerrado. Para finalizar hay un conjunto de llaves que abarca todo el *code_block* que contiene todas las instrucciones a llevar a cabo dentro del método.

```
1.  code_block      →   | instructions
2.  instructions    → instruction | instructions instruction
3.  instruction     → statement   | var_declaration
```

Como se puede observar en las reglas anteriores, un *code_block* puede estar **vacío**, o ser una lista de instrucciones. Una instrucción es una línea de Código que puede ser una declaración de variables o una sentencia. De esta manera dentro de cualquier bloque de código se pueden intercalar estos tipos de instrucciones y no separarlo en dos apartados diferentes como en algunos lenguajes por ejemplo *Ada*. A continuación, se puede ver la estructura de una declaración de variable

```
1.  var_declaration → CONSTANT type id SEMICOLON
2.                      | CONSTANT type id ASSIGNMENT expression SEMICOLON
```

En primer lugar, hay un *token* opcional para indicar si la variable es constante (solo se puede leer, después de declararla) o no, a continuación, está el tipo de datos de la variable que solo puede ser *bool* o *int*, seguido del *string* identificador con el nombre de la variable. Para finalizar, una declaración puede acabar aquí con un punto y coma, o asignar directamente un valor a la variable creada. Esto se haría indicado con un *token* "=" (llamado ASSIGNMENT) seguido de una expresión que devuelva un valor.

A continuación, tenemos la estructura del otro tipo de instrucción, las sentencias:

```
1.  statement       → id ASSIGNMENT expression SEMICOLON
2.                      | id LPAREN RPAREN SEMICOLON
3.                      | id LPAREN expression_list RPAREN SEMICOLON
4.                      | if_statement
5.                      | WHILE LPAREN expression RPAREN LBRACE code_block RBRACE
6.                      | RETURN SEMICOLON
7.                      | RETURN expression SEMICOLON
8.                      | BREAK SEMICOLON
9.                      | error;
```

Una sentencia es una instrucción para realizar alguna función específica del lenguaje. En la gramática están definidas las siguientes:

- **Asignación:**
  Sirve para dar o actualizar valores a variables declaradas. Comienza por el identificador de la variable, seguido del *token* de asignación ("=") y la expresión a calcular para obtener el valor.
- **Llamada a funciones:**
  Sirve para realizar llamadas a funciones que no tengan un valor de retorno o que no se quiera asignar a alguna variable. Su estructura es: el identificador de la función a llamar seguido de un paréntesis abierto, una lista opcional de argumentos que pasar a la función en caso de que se haya definido con parámetros y por último el cierre de paréntesis.
- ***If:***
  Sirve para realizar saltos condicionales en el código, tiene la siguiente estructura:

```
1.  if_statement    → IF LPAREN expression RPAREN LBRACE code_block RBRACE
2.                  | IF LPAREN expression RPAREN LBRACE code_block RBRACE ELSE if_statement
3.                  | IF LPAREN expression RPAREN LBRACE code_block RBRACE ELSE LBRACE |
                      code_block RBRACE LBRACE code_block RBRACE
```

  Empieza con el *token* especial *if* para indicar la sentencia, seguido de la expresión a evaluar entre paréntesis. A continuación, puede seguir opcionalmente un *code_block* dentro de un *else if* con otra expresión a evaluar, o un *code_block* en un *else* que se ejecutaría solo si no se cumple ninguna de las condiciones anteriores, pudiendo formar así cadenas de condicionales.

- ***While***:
  Sirve para realizar un bucle condicional en el código, tiene una estructura muy parecida al *if* básico, pero el código del *code_block* se ejecuta hasta que deje de cumplirse la condición definida en la expresión.
- ***Return*:**
  Sirve para retornar de una llamada a función, se debe utilizar con una expresión en las funciones que tengan declaradas un tipo de retorno.
- ***Break:***
  Por último, tenemos la sentencia *break* que sirve para salir de bucles en cualquier momento, su estructura es el *token* especifico *break* seguido de un punto y coma para terminar la línea.

Por último, tenemos las expresiones, un elemento muy necesario en la gramática, que se evalúan para obtener un valor. La gramática tiene las siguientes:

```
1.  expression      → expression ADDITION expression
2.                      | expression SUBTRACTION expression
3.                      | expression MULTIPLICATION expression
4.                      | expression DIVISION expression
```

```
5.                      | LPAREN expression RPAREN
6.                      | boolean_expression
7.                      | id
8.                      | id LPAREN expression_list RPAREN
9.                      | id LPAREN RPAREN
10.                     | literal
11.                     | SUBTRACTION expression
12.                     | error;
```

- **Aritméticas:**
  Son expresiones que contienen un operador aritmético, estos pueden ser suma (“+”), resta (“-”), división (“/”), multiplicación (“*”) y paréntesis (“()”). Estos operadores siguen el orden matemático de las operaciones combinadas indicando la precedencia. También está incluida la operación de negación, todas estas operaciones deben tener operandos del tipo *Int* y siempre devuelven un valor del mismo tipo.
- **Expresiones Booleanas:**
  Son expresiones que solo se pueden evaluar a un valor booleano (*True* o *False*), la gramática consta de las siguientes:

```
1.  boolean_expression → expression EQUAL expression
2.                      | expression DIFFERENT expression
3.                      | expression GREATER expression
4.                      | expression LOWER expression
5.                      | expression GREATER_EQUAL expression
6.                      | expression LOWER_EQUAL expression
7.                      | expression AND expression
8.                      | expression OR expresión
9.                      | NOT expresión
```

  Existen diferentes operadores de comparación y los operadores lógicos básicos. Este tipo de expresiones se puede usar como condición en los saltos y bucles, así como para asignar valores a variables de tipo *bool*.

- **Identificadores y Literales:**
  Son expresiones que se evalúan directamente, en el caso de los literales se obtiene el valor del símbolo, y en el caso de variables se accede a la tabla para obtener el valor.
- **Llamadas a Funciones:**
  Es un tipo de instrucción muy parecido al de las sentencias, aunque en este caso solo se puede utilizar en funciones que retornen un valor, dicho valor luego se puede utilizar como parte de una expresión.

Generación del Analizador

Para el apartado del analizador sintáctico del compilador se ha utilizado la herramienta de generación de *parser CUP* [2]. Esta herramienta hace uso de una clase *Symbol* para representar a los tokens, y se puede enlazar con la herramienta utilizada en el apartado léxico *JFlex.* Para poder tener más control sobre los símbolos hemos definido nuestra propia clase *CrotoSymbol* y un *Factory* para crear los objectos *CrotoSymbolFactory* localizados en el directorio */src/sintactic/symbols/*.

Como se puede observar en el código del generador localizado en */cup/Grammar.cup* también existen tres grandes apartados para la generación del *parser* que explicaremos a continuación.

- **Código de *Java*:**
  En este apartado inicial se puede encontrar código fuente de *Java* que se utilizara posteriormente en el archivo *parser* generado. El código está formado por *imports* de librerías utilizadas para la creación y manejo de símbolos, así como de funciones sobrescritas de *CUP* para poder gestionar los errores de una manera más limpia.

- **Declaraciones:**
  En este apartado se encuentran tanto las declaraciones de los terminales como la de los no terminales que usamos en la gramática del lenguaje. Por cada símbolo terminal definido se genera un identificador en el archivo */src/sintactic/sym.java* que se puede utilizar para identificar los *tokens* en el analizador léxico. Para poder realizar posteriormente el análisis semántico utilizando el método de gestión con árbol sintáctico explícito se ha creado una clase para cada símbolo no terminal y de esta manera poder ir generando el árbol a medida que se recorren las producciones.

- **Reglas Sintácticas:**
  Este apartado consta de todas las producciones que definen la gramática del lenguaje explicada anteriormente. Para cada posible producción tenemos un apartado de código *Java* donde creamos una instancia de la clase que representa la producción, de esta manera se va generando el árbol sintáctico. En cada instancia se almacena la línea y columna del inicio de la producción para poder mostrar información extendida en los errores semánticos (Se debe ejecutar el .cup con el *flag* (*-locations*).

Restricciones y Peculiaridades

- Debido a la definición de *program* la unidad mínima de compilación, el método principal (*main*) debe estar siempre después de cualquier declaración de método.
- En las declaraciones de métodos, siempre debe haber un *return* al final de la función, pero puede haber otros dentro del *code_block* si el usuario desea retornar en un punto anterior del codigo.
- También cabe destacar que no existe la posibilidad de declarar variables globales que se puedan utilizar
- Por último, una pecularidad del lenguaje es que puede haber bloques de codigo vacios, por tanto, se pueden declarar métodos sin codigo interno, o *if's* y *while's*.

Analizador Semántico

Para el análisis del código fuente se ha utilizado el método de gestión con árbol sintáctico explicito. En el apartado anterior se ha ido generando el árbol sintáctico a medida que se recorren las producciones, donde cada nodo del árbol es una instancia de una clase que representa la producción en cuestión.

Dentro del directorio */src/semantic/symbols* esta la Clase *Structure,* que representa cualquier nodo del árbol sintáctico, junto con todas sus clases hijas que representan las distintas producciones sintácticas. Para producciones complejas, como por ejemplo *Statement* o *Expression* se han hecho subclases para cada uno de los distintos casos específicos.

Además, está definida la interfaz *Visitor* que representa los métodos que debe tener una clase que quiera realizar el recorrido del árbol sintáctico con un método *visit* para cada clase distinta. En la clase *Structure* existe un método abstracto *check* que deben implementar todas las demás clases, este método recibe un objeto visitante y debe indicarle que visite ese nodo en cuestión. Se ha seguido este proceso basándose en un ejemplo del repertorio del *CUP* [3].

Tabla de Símbolos

Se han creado tres clases para implementar la Tabla de Símbolos:

- ***Variable***:
  Esta clase contiene tres atributos: un tipo (entero o booleano), un Valor de tipo *Object* (para así poder almacenar ambos tipos de valor) y un booleano que indica si es constante o no. Las instancias de esta clase se irán creando y almacenando en la tabla de variables a medida que vayamos identificando las distintas variables del Programa.
- ***Table***:
  Es una clase que contiene un *HashMap* de (*String*, *Variable)* y un puntero a otra ***Table*** que es su tabla padre. La "tabla padre" de la tabla contiene las declaraciones de variables del ámbito inmediatamente superior a esta. Tiene un método para insertar una variable y un método para buscar si una variable está en esa tabla en concreto.
- ***SymbolTable***:
  Contiene un puntero a la Tabla de variables del ámbito actual, un *HashMap* de parámetros (ya que vamos a tener **una tabla de símbolos para cada función**, al no tener variables globales), un contador de la profundidad de ámbito, y un tipo de retorno, que guarda el tipo de retorno de la función.
  Hay definida una función para insertar parámetros, otra para obtenerlos, una función para entrar en un ámbito (crea una nueva Tabla con la tabla actual como padre de la nueva), una función para salir del ámbito (cambia la tabla actual por la tabla padre de la actual). Finalmente, una función para añadir variables (añade la variable a la tabla actual), y una para buscar variables.
  La función para buscar variables se encarga de buscar la variable en el ámbito actual, y, si no la encuentra, se va al ámbito anterior (su tabla padre), y así hasta encontrar la variable o hasta llegar al último ámbito y no encontrarla, en ese caso devolverá *null*.

Recorrido del Árbol Sintáctico

Para hacer el recorrido del árbol sintáctico y realizar las comprobaciones necesarias en cada caso se ha creado la clase *SemanticAnalyzer* que implementa la interfaz *Visitor*. En la clase hay definidas dos variables globales que se utilizaran a lo largo del recorrido, en primer lugar, tenemos una tabla de métodos de tipo *HashMap* donde la llave es el nombre del método y el elemento es un objeto *SymbolTable.* La segunda variable es *string* que guarda el nombre del método actual.

El análisis comienza con el nodo más alto del árbol devuelto por el analizador sintáctico, que es un objeto *Program*. Para comprobar semánticamente este nodo solo se deben pasar a los nodos hijos que en este caso son los métodos declarados, en el caso de que haya, y el método *main.*

Para el análisis de los métodos, primero se comprueba que no haya un método ya declarado con el mismo identificador o utilice el nombre de las funciones reservadas *print* y *scan*. A continuación, se comprueban cada uno de los parámetros, y se actualiza el tipo de retorno. Por último, se comprueba el *CodeBlock* si después de la comprobación no se ha encontrado un *return* y la función no era *void* se muestra un error semántico. Para la comprobación de los parámetros se revisa que no haya ningún parámetro anterior definido con ese nombre en el mismo método.

El análisis semántico de un *CodeBlock* consiste solamente en recorrer la lista de instrucciones y realizar la comprobación de cada una. Las comprobaciones que se realizan en el primer tipo de instrucción *VarDeclaration* son, asegurar que no haya ya una variable con el nombre indicado y en el caso que se quiera inicializar directamente asignado un valor que este sea del tipo esperado.

A continuación, se explican las distintas instrucciones del segundo tipo *Statement*:

- ***Assignment*:**
  En primer lugar, se comprueba que la variable que se quiere actualizar este declarada correctamente, a continuación, se comprueba que no sea una variable constante, ya que estas no se pueden modificar. Por último, se comprueba que el valor que se quiere asignar sea del mismo tipo que la declaración de la variable.
- ***FunctionCall*:**
  En este caso primero se comprueba si el nombre de la función a ejecutar es alguna de las funciones reservadas del lenguaje *print* o *scan*. En el caso de *scan* se comprueba que tenga un único argumento y este sea el identificador de una variable donde almacenar el valor. En el caso de *print* se comprueba la expresión del argumento para obtener un valor. En cualquier otro caso primero se comprueba que el método con ese identificador exista, en caso contrario se muestra el error semántico. A continuación, se comprueba que el numero de argumentos sea el mismo que el número de parámetros en la declaración del método y que el tipo de cada uno corresponda. Además, si el nombre de la función es el mismo que el método actual se muestra un error de recursividad.
- ***If* y *While*:**
  Para las sentencias *if* y *while solo* se realiza la comprobación de que la expresión de condición sea de tipo booleano, en caso contrario se devuelve el error semántico correspondiente.
- ***Return:***
  En primer lugar, se comprueba si el *return* va seguido de una expresión, si no es el caso y el método actual tenía un tipo de retorno distinto a *void*, se muestra un error semántico. Si hay una expresión de retorno se comprueba que sea del tipo esperado.

- ***Break:***
  En este caso solo se comprueba que la sentencia este dentro de un bucle para poder salir, en caso contrario mostrar el error semántico correspondiente.

A continuación, se explican la gestión de las distintas expresiones del lenguaje:

- ***Arithmetic*:**
  Se comprueba que ambos operadores sean de tipo entero, en caso contrario se muestra un error.
- ***Boolean:***
  En primer lugar, se comprueba la expresión de la derecha al operador, si la operación es un *not* lógico se comprueba el resultado con este único operando. En cualquier otro caso se comprueba la expresión de la izquierda al operador y se asegura de que el resultado sea booleano.
- ***FunctionCall*:**
  Primero se comprueba que el método con ese identificador exista, en
  caso contrario se muestra el error semántico. A continuación, se comprueba que el numero de argumentos sea el mismo que el número de parámetros en la declaración del método y que el tipo de cada uno corresponda. Además, si el nombre de la función es el mismo que el método actual se muestra un error de recursividad.

- ***Literal*:**
  Se indica el tipo de retorno en función al valor del literal.
- ***Id*:**
  En primer lugar, se comprueba que la variable este definida, en caso contrario se muestra un error. A continuación, se comprueba si la variable tiene algún valor, si no es el caso se asigna el valor por defecto del tipo de dato (INTEGER: 0, BOOLEAN: false).

Generación de Código

La generación de código se divide en dos partes, la primera corresponde a la generación del código intermedio y la segunda la traducción al código ensamblador. Se han definido dos clases auxiliares *Variable* y *Procedure* para representar esos elementos en el código generado. Dichas clases contiene atributos específicos de cada elemento.

Para realizar la generación de ambas partes se ha definido la clase *CodeGenerator* y para el código intermedio se ha definido un lenguaje de tres direcciones que se describe a continuación.

Código de Tres Direcciones

Se ha definido la clase */code_generation/Instruction.java* (diferenciar del nodo *Instruction* para el recorrido semántico), que representa una instrucción en código de tres direcciones. Cada instrucción tiene un atributo *op* que representa la operación de esa instrucción. Las diferentes operaciones definidas están contenidas en un *Enum* y son las descritas a continuación.

- Operaciones aritmeticológicas:

Nombre	**Mnemónico**
Copia	_COPY
Suma	_ADD
Resta	_SUB
Multiplicación	_PROD
División	_DIV
Negación	_NEG
And	_AND
Or	_OR
Not	_NOT

- Operaciones de control de flujo de programa:

Nombre	**Mnemónico**
Etiqueta	_SKIP
Salto incondicional	_GOTO
Salto condicional	_IF
Menor	_LT
Menor o igual	_LE
Mayor	_GT
Mayor o igual	_GE
Diferente	_NE
Igual	_EQ

- Llamadas a procedimientos:

Nombre	**Mnemónico**
Inicialización	_PMB
Llamada	_CALL
Retorno	_RTN
Parámetro	_PARAM
Print	_PRINT
Scan	_SCAN

Cada instrucción puede tener hasta tres operadores (*dest, op1 y op2*) aunque no todas utilicen todos los disponibles. Dentro de la clase *Instrucción* se almacenan los distintos operadores y una representación en texto de la instrucción en tres direcciones. En la clase también existen funciones auxiliares para tratar de diferentes formas con las instrucciones.

- ***parse***:
  Este método estático crea una instancia de *Instruction* mediante el *Parsing* de un *String* que representa la instrucción.
- ***toAssembly***:
  Este método traduce la instrucción en cuestión a lenguaje ensamblador. Por defecto utiliza la traducción de cada operación al lenguaje ensamblador *NASM* [4].

Generación de Código intermedio

Para generar el código intermedio se hace un recorrido del árbol sintáctico utilizando la clase *CodeGenerator* que implementa la interfaz *Visitor*. En este apartado solo se genera el código de tres direcciones a medida que se recorre el árbol sintáctico ya que se asume que se ha analizado semánticamente el árbol y este es correcto.

La generación empieza, al igual que en el semántico, por el objecto *Program.* A continuación, se explicará de manera resumida en que consiste la generación de cada nodo del árbol:

- ***Method***:
  Se encarga de generar una etiqueta de inicio, que será a la que saltará el programa cuando se llame a dicho método. Generamos las operaciones de preámbulo, y comprobamos e introducimos los parámetros del método en la tabla de procedimientos. Finalmente, comprobamos el *CodeBlock* del método que genera el código intermedio de cada instrucción.
- ***Parameter*:**
  Se encarga de renombrar el parámetro concatenado el nombre del procedimiento en el que se encuentra con el nombre del parámetro. Esto se hace ya que luego en la generación del código ensamblador todas las variables serán globales. Al hacer esto se permite que haya variables con el mismo nombre en diferentes métodos. Por ejemplo, se tendría la variable *mainAUX* y *factorialAUX*.
- ***VarDeclaration***:
  De forma parecida a los parámetros, se concatena en el identificador de la variable el nombre del método actual para que no se confundan variables con el mismo nombre. Introduce la variable en la tabla de variables y si en la declaración había asignación se genera y se crea un *Copy* del valor a la variable.

A continuación, se explica en concreto la generación de las diferentes sentencias del lenguaje.

- ***Assignment*:**
  Se genera el valor de la expresión a asignar y una vez ha sido generado se copia dicho valor a la variable destino.
- ***FunctionCall*:**
  Existen tres posibilidades para esta sentencia:
  - Si se trata de un *Print* se genera la expresión de argumento y se crea la instrucción *_print* con el valor generado.
  - Si se trata de un *Scan* se genera la expresión de argumento (que solo puede ser un identificador de variable) y se crea la instrucción *_scan* con el valor generado.
  - Para cualquier función definida por un usuario se continua simplemente con el procedimiento normal.

  Independientemente de si es un *print*, un *scan* u otro tipo de función generaremos el código para guardar todos sus parámetros. Finalmente generamos la llamada a la etiqueta.

- ***If*:**
  En primer lugar, se evalúa la expresión de condición, se genera una etiqueta de salto para el caso en que no se cumpla dicha condición. A continuación, se crea la instrucción del if con dicha expresión y si no se cumple se salta a la etiqueta del *else*, se genera el *CodeBlock* del *if* y luego se salta a la etiqueta del final del *if* ya que se ha entrado a uno de los casos. En el caso

de que se haya saltado al *else* se comprueba si hay otro *if* y si es el caso se genera. En caso de que no haya otro *if* se comprueba si hay bloque de código del *else* y se genera. Por ultimo se pone la etiqueta de final del *if*.

- ***While*:**
  El caso del *while* es igual que un *if* simple, pero al final del bloque de código se genera una instrucción de salto a una etiqueta de inicio para que se siga iterando hasta que se deje de cumplir la condición. En cada bucle se guarda la etiqueta final en una variable global por si hay algún *break* dentro de las sentencias.
- ***Return*:**
  Si no tiene expresión simplemente se genera el mnemónico del *return*, si tiene expresión de retorno se genera el código de su expresión y luego el mnemónico del *return* con la variable que contiene el valor.
- ***Break*:**
  Se crea un salto a la etiqueta final del bucle actual en el que se encuentra.

A continuación, se explica la generación de las diferentes expresiones del lenguaje:

- ***Arithmetic*:**
  Se genera el código correspondiente a la parte izquierda de la operación, luego el de la derecha y finalmente se une mediante la operación correspondiente.
- **Boolean:**
  Si se trata de una operación *NOT* se genera la expresión de la derecha y se crea la instrucción. En los otros casos se comprueban ambas expresiones y se crea la instrucción con la operación correspondiente.
- ***FunctionCall*:**
  En primer lugar, se generan las instrucciones *_param* para guardar los parámetros de la función en la pila, esto se hace en orden invertido ya que luego se sacarán en orden. A continuación, se genera la llamada a la función correspondiente y se almacena el valor retornado en una variable temporal.
- ***Literal*:**
  Genera el código asociado con los valores literales. Se comprueba el tipo de variable y copia su valor en una variable temporal.
- ***Id*:**
  Genera el código asociado con los identificadores y copia el valor dentro de una variable temporal.

Para poder generar el código intermedio hay definidas algunas funciones auxiliares y variables globales. En primer lugar, se comentan las variables más importantes.

- **Tablas y Listas:**
  - **Tabla de Variables:** es un *HashMap* donde la llave es el nombre identificador de la variable y el elemento es una instancia de la clase *Variable*.
  - **Tabla de Métodos:** es un *HashMap* donde la llave es el nombre identificador del método y el elemento es una instancia de la clase *Procedure*.
  - **Lista de Etiquetas:** es un *ArrayList* que contiene todas las etiquetas definidas en el código.

    - **Lista de Instrucciones:** es un *ArrayList* donde se almacenan todas las *Instructions* generadas del codigo de tres direcciones.
- **Variables:**
    - **Contadores:** existen dos contadores para saber el numero de variables temporales y etiquetas creadas.
    - **Indicadores:** existen dos variables para almacenar indicadores. En primer lugar, tenemos *currentProc* que almacena el nombre del método actual para tener contexto y *varName* que almacena el nombre de la variable donde se ha almacenado el ultimo valor calculado o generado. Por último, también existe *endLoop* que almacena el nombre de la etiqueta final del bucle actual en caso de que haya algún *break*.

A continuación, se comentan las funciones auxiliares definidas

- ***generate3ac()*:**
  Este método inicia el proceso de generación de código intermedio y es el que debe utilizar la clase principal del compilador para este proceso. Simplemente comienza generando el nodo padre del árbol sintáctico (*Program*) y continua con el recorrido.
- ***write3ac():***
  Este método escribe el conjunto de instrucciones intermedias generadas en un archivo especificado.
- ***generate()*:**
  Este método crea la instrucción correspondiente al *string* indicado utilizando el método *parse* de la clase *Instruction* y la añade a la lista de instrucciones.
- ***newVar(), newTempVar() y newLabel()*:**
  Estos métodos crean respectivamente, variables con el nombre indicado, variables temporales utilizando el contador global, etiquetas utilizando el contador global y las introducen en sus respectivas tablas. Devuelven el nombre del elemento creado para que se pueda usar en el código generador.

Generación de Código Ensamblador

Para el ultimo paso del compilador se ha decidido utilizar el lenguaje *NASM* [4] para la generación de código ensamblador, la generación de este también se realiza en la clase *CodeGenerator* en este caso se utilizan los siguientes métodos.

- ***generateAssembly()*:**
  En primer lugar se llama a la función auxiliar *generateData()* para generar el encabezado y las definiciones de variables en la sección correspondiente del ensamblador.
    - ***generateData()*:**
      define la sección correspondiente a los datos el programa, y declara cada variable de la tabla de variables. A continuación, recorre la tabla de procedimientos y declara los parámetros de cada uno.

  Una vez generada la sección de datos, se define la sección de código y la etiqueta *main.* En el caso que durante el código haya operaciones de entrada y salida (*scan* y *print*) se definen los formatos necesarios y los *imports* a las funciones correspondientes de *C* [5]. Para finalizar se recorre la lista de instrucciones de código intermedio y se llama a la función *toAssembly()* que traduce la instrucción a código ensamblador y la añadimos a la lista de instrucciones de ensamblador.

    - ***toAssembly()*:**
      Es un método de la clase *Instruction* que traduce una instrucción de código intermedio a código ensamblador. Dependiendo de la operación de la instrucción pone uno o ambos operadores en un registro y realiza la instrucción correspondiente en *NASM* [4].
- ***writeAssembly()*:**
  Este método escribe el conjunto de instrucciones ensamblador generadas en un archivo especificado.

Restricciones y Peculiaridades

Una restricción del compilador es que debido a la forma en que se generan las instrucciones de código intermedio utilizando el método *generate()* que a su vez utiliza el *parse*() de *Instruction*, no se podían diferenciar las variables que tuvieran un nombre de operación con la operación en concreto. Por tanto, se han definido los mnemónicos con el carácter inicial "_" y se ha eliminado este de los identificadores de variables.

Debido a las limitaciones de tiempo, el lenguaje no soporta la posibilidad de definir funciones recursivas ya que esto implicaría introducir todas las variables en la pila.

Optimización

En la clase *CodeGenerator* hay definido un método *optimize()* que intenta optimizar el código de tres direcciones antes de traducirlo al ensamblador para obtener un ejecutable mas eficiente. Las optimizaciones que se han intentado tener en cuenta son algunas del tipo *peephole* y cada una tiene un método. El método general realiza un numero de pasadas (por defecto 10) por si hay cambios y pueden seguir habiendo optimizaciones. Lo ideal seria poder trasladar todas estas funcionalidades a otra clase que reciba la lista de instrucciones de código intermedio y genere una optimizada, pero por falta de tiempo no se ha podido implementar.

No están hechas todas las optimizaciones *peephole* y las que hay implementadas no es seguro que funcionen al cien por cien. Por tanto, la clase principal del compilador no intenta optimizar el resultado final a no ser que se modifique explícitamente.

Compilador

La clase principal del compilador (*/src/main/Compiler.java*) se encarga de integrar los diferentes apartados explicados anteriormente para poder compilar de forma satisfactoria un archivo fuente. Para ello es necesario crear una instancia de *Compiler* y llamar al método *compile()* indicando la dirección del archivo fuente, y la esperada para la salida.

El método *compile()* llama a un método para cada fase de la compilación, si alguna de estas fases devuelve error, no se pasa a la siguiente y se termina la compilación. Para tratar los distinto errores tiene una instancia de la clase *ErrorManager* que genera un *.txt* con el mismo nombre y en el mismo directorio que el archivo de salida con los errores encontrados. Los archivos de salida del proceso de

compilación son un archivo con el nombre indicado *.3ac* con el código intermedio generado y otro archivo *.asm* con el código ensamblador generado.

Gestión de Errores

Se ha definido el paquete */errors/* que contiene todo el código para la gestión de errores. En primer lugar, está definida la clase *ErrorManager* que se encarga de gestionar los errores, cuando haya un error se imprimirá por consola y se escribirá en un archivo de texto. Para poder utilizar el *ErrorManager* primero hay que llamar al método *open()* que inicializa el buffer de escritura al archivo de salida, a continuación se puede utilizar el método *writeError()* tanto con *String* como con *Exception* y se imprime por los dos *streams*. Por ultimo hay que cerrar el stream llamando al método *closeManager*().

Para un mejor control de los errores se han definida una serie de excepciones en el paquete. En primer lugar, se ha definido una clase *CrotoExcepcion* que representa cualquier error del compilador, este contiene la línea y columna donde ha surgido el error. A continuación, se encuentran diferentes clases para los diferentes tipos de errores:

- ***LexicException*:**
  Representa los errores que pueden ocurrir durante la fase de análisis léxico. Principalmente serán lecturas de caracteres no válidos.
- ***SyntaxException*:**
  Representa los errores que pueden ocurrir durante la fase de análisis sintáctico. Estos suelen ocurrir al no encontrar una producción en la gramática que tenga la misma estructura que la entrada. Muestran diferentes datos como por ejemplo los tokens esperados en caso de error.
- ***SemanticException*:**
  Representa los errores que pueden ocurrir durante la fase de análisis semántico. Estos errores ocurren principalmente cuando no se cumple una comprobación semántica, por ello existe una subclase para cada tipo de error y con un mensaje específico del error, de esta manera se puede tener una gestión de los errores mas uniforme y clara de cara al usuario. A continuación, se muestra una lista de todas las excepciones semánticas declaradas.

  - *MethodAlreadyDeclaredException*
  - *MissingReturnException*
  - *UnexpectedTypeException*
  - *VariableNotDeclaredException*
  - *IncorrectArgumentSizeException*
  - *RecursionException*
  - *MustInitializeConstantException*
  - *MethodNotDeclaredException*
  - *ParameterAlreadyDeclaredException*
  - *VariableAlreadyDeclaredException*
  - *AssignToConstantException*
  - *ScanArgumentException*
  - *UnexpectedBreakException*

- ***CompilationException***:
  Representa los errores que pueden ocurrir durante el proceso de compilación de un archivo sin ser de ninguno de los apartados específicos.

Pruebas

Para realizar distintas comprobaciones del funcionamiento del compilador, se han diseñado un conjunto de diferentes pruebas dentro del directorio */test/* con dos carpetas diferentes, una para las pruebas con compilación satisfactoria y las funcionalidades del lenguaje, y otras para mostrar los errores gestionados. Para poder obtener un ejecutable y comprobar el resultado se ha hecho uso de una consola con *WSL* [6] para poder compilar con *nasm.* Utilizando el siguiente desde el directorio con el archivo *.asm* se puede ejecutar:

```
nasm -f elf -o Programa1.o Programa1.asm && gcc -m32 Programa1.o -o Programa1 && ./Programa1
```

En los archivos de prueba correctos se ha intentado abarcar el máximo numero de producciones de la gramática. En cambio, en los incorrectos se muestran errores comunes y el output de ellos.

Conclusiones

La realización de esta práctica nos ha hecho entender muy bien todo el proceso de compilación de un archivo de código. Tener que hacer todas las partes, tanto de *Frontend* como de *Backend* ha aumentado gratamente nuestra comprensión de los compiladores, incluso cuando no hemos aplicado todos los pasos (apenas hemos hecho optimización) que se aplican en los compiladores actuales.

Es una práctica larga y pesada, pero al acabarla y ver como todo el código escrito funciona como debe, proporciona una satisfacción difícil de alcanzar por otras prácticas.

Nos gustaría haber implementado más elementos opcionales (*strings*, *arrays*, optimizaciones, etc), pero no hemos podido por falta de tiempo. Además, algunos conceptos y partes de la practica nos han llevado mas tiempo del esperado para implementarlos y acabar de entenderlos, cosa que tal vez no nos ha permitido implementar más sentencias como por ejemplo *for*, *switches*, etc.

En general nos ha gustado mucho y hemos aprendido bastante sobre como funciona un computador moderno hoy en día.

Referencias

[1] [En línea]. Available: https://www.jflex.de/.

[2] [En línea]. Available: http://www2.cs.tum.edu/projects/cup/.

[3] [En línea]. Available: https://versioncontrolseidl.in.tum.de/parsergenerators/cup/-/tree/master/testgrammars/minijava2/src/minijava.

[4] [En línea]. Available: https://www.nasm.us/xdoc/2.15.05/html/nasmdoc0.html.

[5] [En línea]. Available: https://devdocs.io/c/.

[6] [En línea]. Available: https://docs.microsoft.com/en-us/windows/wsl/.