Universitat de les Illes Balears

Documentación de la práctica

Compiladores

Autor/es
Mayol Matos, Sergi
Palmer Pérez, Rubén
Sevilla Marí, Jordi

Correo
sergi.mayol1@id.uib.es
r.palmer@uib.es
jordi.sevilla1@id.uib.es

`Chadpp`
7 de febrero de 2023

Índice

1. Introducción

Bienvenido a la documentación del compilador Chadpp. Este compilador es una herramienta diseñada para la asignatura Compiladores de GEIN-3 que permite convertir código escrito en el lenguaje imperativo Chadpp en código ensamblador; Específicamente al 68k de Motorola. Esta documentación abarca desde una descripción del lenguaje hasta cada una de las fases del compilador. Adicionalmente, es posible que se remarquen algunas estructuras de datos usadas o decisiones de diseño.

El compilador se ha implementado con Java, JFlex y CUP en el IDE VSC o IntelliJ. Adicionalmente, para compilar el código fuente del compilador será necesaria la versión 13 o superior de Java.

2. Características del lenguaje

Chadpp es un lenguaje imperativo inspirado en C++ y java centrado en el cómputo de operaciones logico-aritméticas.

Todo programa de chadpp debe usar la extensión `.chpp` o alternativamente `.txt`. La sintaxis de un programa en chadpp sigue siempre la siguiente estructura:

```
# Declaraciones globales
const int a = 4;

# Funciones
alpha int suma(int a, int b){
    return a + b;
}

# Programa principal
main{
    # Declaraciones
    tup v1 = [1,2,3];
    tup v2 = [3,2,1];
    BEGIN
    # Instrucciones
    output(suma(v1[0], v2[0]));
    output(suma(v1[1], v2[1]));
    output(suma(v1[2], v2[2]));
}
```

En ese exacto orden. Tanto las declaraciones globales como las funciones son elementos opcionales que pueden o no estar en el programa.

En primer lugar, se realizarán todas las declaraciones globales en la sección superior (si existen tales). Todas las variables (o constantes) declaradas al principio podrán ser accedidas desde cualquier parte del programa (main o funciones alpha).

Después, se declararán todas las funciones que podrán ser llamadas más adelante. Estas funciones pueden ser llamadas por otras funciones.

Algunas aclaraciones y restricciones del lenguaje:

- En las funciones no se pueden declarar variables fuera del bloque `BEGIN`. Si se quiere usar una variable a lo largo de la función, se debe declarar previamente en dicho bloque y darle un valor por defecto.
- Solo se permite acceso a tuplas mediante valores literales numéricos para evitar la "memory unsafety".
- Las tuplas declaradas globalmente se tratarán como constantes
- Hay un conjunto de warnings programados que informarán al programador de posibles casos en los que el programa pueda ejecutar con resultados no esperados. El proceso de compilación acabará, pero se recomienda crear programas que prescindan de estos warnings
- La división por 0 se permite aun sabiendo que no ejecutará correctamente. En caso de que se quiera utilizar el proceso de optimización, no se permitirá su compilación.
- No se contempla la recursividad de funciones
- Para que una función pueda ser llamada, debe haber sido declarada previamente.

3. Analizador léxico

Para esta fase se ha utilizado Jflex que nos permite discretizar el input del programa en tokens mediante el uso de patrones o reconocimiento de strings. El fichero por completo se puede encontrar en grammar/Scanner.flex.

3.1. Tokens

A continuación se nombrarán todos los tokens terminales y su respectivo string reservado que posee el lenguaje y por ende el compilador:

- `BEGIN` → *BEGIN*: Separa la parte de declaraciones de variables y el bloque de instrucciones para cada función.
- `MAIN` → *main*: Declaración del método principal.
- `ALPHA` → *alpha*: Declaración de una función.
- `CONST` → *const*: Declaración de una variable inmutable.
- `RETURN` → *return*: Valor de retorno de una función.

- `VBOL` → *true* o *false*: Literales booleanos.
- `INT` → *int*: Declaración de una variable o tipo de retorno de una función de carácter numérico.
- `BOL` → *bol*: Declaración de una variable o tipo de retorno de una función de carácter booleano.
- `TUP` → *tup*: Declaración de una variable o tipo de retorno de una función como tupla.
- `VOID` → *void*: Declaración de funciones sin valor de retorno.
- `IF` → *if*: if-statement.
- `WHILE` → *while*: While-loops
- `LOOP` → *loop*: Bucles a partir de un rango definido.
- `AND` → &&: Símbolo lógico AND.
- `OR` → ||: Símbolo lógico OR.
- `REQUAL` → ==: Identificar si dos expresiones son iguales.
- `LT` → <: Identificar si una expresión en menor que otra.
- `GT` → >: Identificar si una expresión es mayor que otra.
- `PLUS` → +: Suma.
- `MINUS` → -: Resta.
- `MULT` → *: Multiplicación.
- `DIV` → /: División.
- `OUT` → *output*: Salida por pantalla.
- `ININT` → *inputint*: Entrada de un valor de carácter numérico.
- `INBOL` → *inputbol*: Entrada de un valor de carácter booleano
- `EQUAL` → =
- `SEMICOLON` → ;
- `COMMA` → ,
- `LPAREN` → (
- `RPAREN` →)
- `LKEY` → {
- `RKEY` → }
- `LSKEY` → [
- `RSKEY` →]

Adicionalmente, tenemos unos tokens que no tienen una palabra reservada y necesitan de unos patrones para poder ser reconocidos (Mostrados en la siguiente sección). Se reconocen por el mismo nombre del patrón.

- `ID`
- `NUMBER`
- `COMMENT`
- `LINE_COMMENT`
- `WS`
- `ENDLINE`

3.2. Patrones

Estos patrones se expresan mediante una expresión regular.

$$
\begin{aligned}
ID &= [a-zA-Z][a-zA-Z\_0-9]^* \\
NUMBER &= 0|[\backslash+\backslash-]?[1-9][0-9]^* \\
COMMENT &= \$[\hat{}*]\ \text{"\$"}|\text{"\$""\$"} \\
LINE\_COMMENT &= \#[\hat{}\backslash r\backslash n]^*[\backslash r|\backslash n|\backslash r\backslash n|]? \\
WS &= [\backslash t]+ \\
ENDLINE &= [\backslash r\backslash n]+
\end{aligned}
$$

3.3. Rutinas

Definiremos 2 métodos para la creación de los tokens:

- **symbol(int type, Object value)**: Usado para la creación de símbolos `NUMBER`, `ID` y `VBOL`, ya que necesitamos saber el valor escrito en el programa i.e VBOL puede ser "false" o "true".
- **symbol(int type)**: Usado para el resto de símbolos.

Ambos métodos son usados para guardar la fila y columna del token en los parámetros `left` y `right` respectivamente.

Adicionalmente, usamos el método **error()**, para la gestión de tokens no válidos

4. Analizador sintáctico

Para esta fase se ha utilizado Cup que nos permite generar el parser para una gramática LALR como la que se ha diseñado para Chadpp.

Un aspecto a tener en cuenta es que para la gestión del compilador, se ha decidido crear un árbol sintáctico explícito, ya que nos proporciona mucho más control ante la gestión del programa y sus futuras fases.

A continuación se presentarán cada una de las producciones de la gramática con una breve explicación.

```
M           ::=;
```

Marcador usado para añadir un nivel de acceso a la tabla de símbolos (Ver Sección 6).

```
CHADPP      ::= M GDECLS:d  FUNCTIONS:fun  MAINFN:main
            |    M FUNCTIONS:fun  MAINFN:main
            ;
```

Nodo principal del árbol sintactico, Consta de 2 producciones distintas, una en caso de que se hayan encontrado declaraciones globales de variables o constantes (GDECLS) y otra sin declaraciones globales.

```
GDECLS          ::= GDECL:d GDECLS:ld
                | GDECL:d
                ;

DECLS           ::= DECL:d DECLS:ld
                | DECL:d
                ;
```

Lista de declaraciones globales (GDECLS) y no globales(DECLS).

```
GDECL           ::= CONST TYPEVAR:type DASIGNATION:assig
                |   TYPEVAR:type DASIGNATION:assig
                ;

DECL            ::= CONST TYPEVAR:type DASIGNATION:assig
                |   TYPEVAR:type DASIGNATION:assig
                ;
```

Declaración de variables (o constantes) tanto globales como no globales. En esta declaración se le indica el tipo (TYPEVAR) y toda su asignación.

```
TYPEVAR         ::= INT
                |   BOL
                |   TUP
                ;
```

Indica el tipo de variable a la hora de declarar las mismas, o el tipo de valor de retorno en de una función (en caso de que no sea `VOID`)

```
FUNCTIONS       ::= M FUNCTION:fn FUNCTIONS:fns
                |
                ;
```

Lista de cada una de las funciones del programa. Con la 2a producción, permitimos que el programa pueda ser compilado en caso de que no tenga ninguna función declarada (aparte de la función principal “main”).

```
FUNCTION        ::= ALPHA TYPEVAR:t ID:id LPAREN F_ARGS:args RPAREN LKEY
↪  DECLS:decls BEGIN R_INSTRS:instrs RKEY
                |   ALPHA TYPEVAR:t ID:id LPAREN F_ARGS:args RPAREN LKEY
                ↪  R_INSTRS:instrs RKEY
                |   ALPHA VOID ID:id LPAREN F_ARGS:args RPAREN LKEY DECLS:decls
                ↪  BEGIN INSTRS:instrs RKEY
                |   ALPHA VOID ID:id LPAREN F_ARGS:args RPAREN LKEY INSTRS:instrs
                ↪  RKEY
                ;
```

4 producciones posibles de declaración de funciones:

- Función con variable de retorno y declaraciones
- Función con variable de retorno sin declaraciones
- Función sin variable de retorno con declaraciones
- Función sin variable de retorno ni declaraciones

```
F_ARGS          ::= F_ARGS2:args
                |
                ;

F_ARGS2         ::= TYPEVAR:type ID:id COMMA F_ARGS2:args
                |  TYPEVAR:type ID:id
                ;
```

La primera producción nos permite generar declaraciones de funciones sin argumentos, mientras la segunda nos permite crear dichos argumentos.

```
ASIGNATION      ::= LID:lid EQUAL EXPRESION:e SEMICOLON;

DASIGNATION     ::= LID:lid EQUAL EXPRESION:e SEMICOLON;
```

Una asignación se compone de una lista de Identificadores (LID) y el valor que se le quiere asignar a cada una de estas variables (EXPRESION). ASIGNATION y DASIGNATION se diferencian en que DASIGNATION se usa en la declaración de variables y ASIGNATION se usa como una instrucción. Adicionalmente, debido a su ámbito de uso, solo ASIGNATION tiene un checkeo explícito de su semántica, ya que la semántica DASIGNATION se llama desde DECL.

```
LID             ::= ID:id COMMA LID:lid
                | ID:id
                ;
```

Lista de identificadores separados por una coma.

```
EXPRESION       ::= VALUE:v OP:op EXPRESION:e
                |   VALUE:v
                ;
```

Creación de las expresiones.

```
OP              ::= PLUS
                |   MINUS
                |   MULT
                |   DIV
                |   REQUAL
                |   LT
                |   GT
                |   AND
                |   OR
                ;
```

Cada una de las posibles operaciones que se pueden realizar, tanto aritméticas como relacionales y lógicas.

```
VALUE           ::= LPAREN EXPRESION:e RPAREN
                |   V_TUP:v
                |   A_TUP:a
                |   NUMBER:n
                |   VBOL:b
                |   CALLF:call
                |   ININT LPAREN RPAREN
                |   INBOL LPAREN RPAREN
                |   ID:id
                ;
```

Cada una de estas producciones son todos las posibles instancias de un VALUE, ya explicadas anteriormente.

```
A_TUP           ::= ID:id LSKEY NUMBER:num RSKEY
                ;
```

Indica el acceso a un elemento de una tupla. El identificador de la variable tupla (ID) y el indice al nelemento de la tupla (NUMBER).

```
V_TUP           ::= LSKEY ARGS:a RSKEY
                ;
```

Lista de cada uno de los elementos de la tupla (ARGS).

```
CALLF           ::= ID:id LPAREN ARGS:args RPAREN
                |   ID:id LPAREN RPAREN
                ;
```

Llamada a funciones, identificadas por su ID, con o sin parámetros.

```
ARGS            ::= EXPRESION:e COMMA ARGS:args
                |   EXPRESION:e
                ;
```

Lista de argumentos separados por comas. Permitimos que se escriban expresiones como argumentos. Semánticamente hablando, lo que se pasará como argumento será el resultado de la expresión ya calculado.

```
INSTRS          ::= INSTR:inst INSTRS:instrs
                |
                ;

R_INSTRS        ::= R_INSTR:inst R_INSTRS:instrs
                |
                ;
```

Lista de instrucciones de las funciones del programa para programas sin valor de retorno y aquellos con valor de retorno.

```
R_INSTR         ::= IF LPAREN EXPRESION:e RPAREN LKEY R_INSTRS:instrs RKEY
                |   WHILE LPAREN EXPRESION:e RPAREN LKEY R_INSTRS:instrs RKEY
                |   LOOP LPAREN EXPRESION:e1 COMMA EXPRESION:e2 RPAREN LKEY
                ↪   R_INSTRS:instrs RKEY
                |   OUT LPAREN EXPRESION:e RPAREN SEMICOLON
                |   ININT LPAREN RPAREN SEMICOLON
                |   INBOL LPAREN RPAREN SEMICOLON
                |   ASIGNATION:a
                |   CALLF:fn SEMICOLON
                |   RETURN EXPRESION:e SEMICOLON
                ;
```

Conjunto de instrucciones para las funciones con retorno.

```
INSTR           ::= IF LPAREN EXPRESION:e RPAREN LKEY INSTRS:instrs RKEY
                |   WHILE LPAREN EXPRESION:e RPAREN LKEY INSTRS:instrs RKEY
                |   LOOP LPAREN EXPRESION:e1 COMMA EXPRESION:e2 RPAREN LKEY
                ↪   INSTRS:instrs RKEY
                |   OUT LPAREN EXPRESION:e RPAREN SEMICOLON
                |   ININT LPAREN RPAREN SEMICOLON
                |   INBOL LPAREN RPAREN SEMICOLON
                |   ASIGNATION:a
                |   CALLF:fn SEMICOLON
                ;
```

Conjunto de instrucciones para las funciones sin retorno.

Con respecto a los métodos que se usan en el análisis sintáctico se presentan los siguientes:

- `void syntax_error(java_cup.runtime.Symbol cur_token)`: Sobreescribimos el tratamiento por defecto al encontrar un error sintáctico para que nos dé el mensaje correcto para nuestros errores.
- `void report_error(String message, Object token)`: Sobreescribimos el tratamiento por defecto para reportar un error para que use las funciones de nuestro gestor de errores.
- `void report_fatal_error(String message, Object info)`: Sobreescribimos el tratamiento de errores fatales para que use las funciones de nuestro gestor de errores.
- `String showExpectedTokenIds()`: Devuelve un string con todos los posibles tokens esperados para el token que acaba de ser pasado por el léxico. Se usa para los mensajes de errores sintácticos para poder dar más información sobre que es lo que esperaba el compilador.

5. Analizador semántico

Para este compilador se ha decidido ejecutar el análisis semántico paralelamente a la fase sintáctica; de esta manera podemos aprovechar el recorrido que se hace implícitamente en el análisis sintáctico para evaluar si el nodo a introducir al árbol sintáctico está bien formado.

Adicionalmente, se ha evitado la creación de gestores para cada nodo y se ha adoptado un gestor más modular; creando diferentes funciones que los diferentes métodos usarán.

A continuación se presentan los diferentes métodos que se han desarrollado:

- `boolean checkFunction(CallFn callFn)`: Verifica que la función que se está llamado esté declarada previamente. Devuelve true si no hay errores y false en caso contrario.
- `boolean checkCallFArgs(CallFn callFn)`: Verifica la función llamada y revisa la exactitud de sus argumentos (coincidencia de los tipos de los argumentos y los parámetros y número de argumentos. Devuelve true si no hay errores y false en caso contrario.
- `StructureReturnType checkExpresion(Expresion exp)`: Verifica la exactitud de una expresión. Devuelve el tipo de datos que devolverá la expresión evaluada.
- `boolean checkExp(Expresion e)`: Wrapper de `checkExpresion` para generar los errores pertinentes. Devuelve true si la expresión está bien formada y false en caso contrario.

- `StructureReturnType checkId(Id exp)`: Verífica el ID. Devuelve el tipo del ID si se ha encontrado y null en caso contrario.
- `StructureReturnType checkAsignation(Asignation assig, boolean InDeclaration)`: Verifica que la asignación o declaración sea correcta. Devuelve el tipo de la expresión si es correcta y null en caso contrario.
- `boolean checkTupleSize(Asignation ass)`: Comprueba que el tamaño de la tupla sea el mismo que el de la declaración inicial. Devuelve true si la cantidad de elementos asignados a la tupla son los mismos que en la declaración y false en caso contrario.
- `StructureReturnType checkReturnValue(Expresion e)`: Revisa que la expresión pasada por parámetro sea del mismo carácter que el valor de retorno de una función. Devuelve el `StructureReturnType` de la expresión y null en caso de que se encuentren errores.
- `void checkReturns()`: Verifica que el tipo de los retornos de una función sean del mismo tipo que el tipo de valor de retorno de dicha función.
- `boolean checkLoop(Expresion e1, Expresion e2)`: Verifica que las dos expresiones sean de carácter aritmético. Devuelve true si ambas lo són y false en caso contrario.
- `boolean checkLogicalExpresion(Expresion e)`: Verifica que la expresión pasada por parámetro sea de carácter lógico. Devuelve true si la expresión es lógica y false en caso contrario.
- `StructureReturnType typeVartoReturnType(TypeVar p)`: Se encarga de mapear cada uno de los posibles tipos de variables a su correspondiente `StructureReturnType`.
- `addtupleContentstoST(Asssignation assig)`: Itera por todos los elementos a añadir dentro de una tupla, gestiona su adición a la tabla de símbolos, gestiona su análisis semántico si es necesario e informa de posibles errores.

6. Tabla de símbolos

Para este compilador se ha decidido crear una estructura de datos personalizada para que se amoldase correctamente ante las decisiones de diseño del lenguaje; por lo que no sigue la estructura convencional de una tabla de símbolos.

Nuestra tabla de símbolos se divide en dos tablas:

- `Tabla de accesos`: Contiene los rangos de símbolos de cada función.
- `Tabla de símbolos`: No confundirla con la propia estructura, ya que esta contiene todos los símbolos del programa.

Con respecto a las tuplas, todos los símbolos tienen una lista llamada `content` donde residen todos los símbolos que contiene la tupla.

Una de las principales iniciativas para crear esta estructura más simple es la propia definición del lenguaje. Debido a que en chadpp no puedes definir variables en medio del código, no existen ámbitos internos para un método y si no se encuentra el símbolo en ese ámbito, siempre se puede coger el rango del acceso global e iterar por esos símbolos.

6.1. Funcionamiento

La tabla de accesos se usa para saber en qué función estamos y en que rango de la tabla de símbolos se encuentran los símbolos de dicha función, ya que la tabla de símbolos contiene todos los símbolos del programa.

A continuación se mostrarán un conjunto de ejemplos para clarificar su funcionamiento.

Truth Machine

```
main{
    int userInput = 0;
    BEGIN
    userInput = inputint();
    if(userInput == 0){
        output(0);
    }
    if(userInput == 1){
        while(true){
            output(1);
        }
    }
}
```

Por defecto, todos los programas tienen dos ámbitos: Global y main. En este programa, se puede ver como globalmente no se han declarado ninguna variable y en el main solo tenemos `userInput`. La tabla resultante de este código es la siguiente:

```
TA:
-1 -1
0 0
TS:
Symbol [name=userInput, structureType=VARIABLE, structureReturnType=INT,
↪  content=null, isGlobal=false, isConstant=false, line=26]
```

Al no tener ninguna variable global, su rango de ámbito es [-1, -1]. Con respecto al main tenemos el símbolo `userInput`, por lo que su rango de ámbito es [0, 0] al encontrar el símbolo en la línea 0 de la tabla de símbolos.

Factorial_loop

```
alpha int factorial(int number){
    int result = 1;
    int iter = number;
    BEGIN

    # No se permite factoriales de numeros negativos
    if (number < 0){
        return 0;
    }

    loop(number, 0) {
        result = result * iter;
        iter = iter - 1;
    }

    return result;
}

main {
    tup nums = [3, 5, 8];
    BEGIN
    output(factorial(nums[0]));
    output(factorial(nums[1]));
    output(factorial(nums[2]));
}
```

En este ejemplo mas complejo encontramos la definición de la función `factorial`, por lo que será necesario tener un nuevo ámbito.

```
TA:
-1 -1
0 5
6 6
TS:
Symbol [name=number, structureType=PARAMETER, structureReturnType=INT,
↪  content=null, isGlobal=false, isConstant=false, line=1]
Symbol [name=result, structureType=VARIABLE, structureReturnType=INT,
↪  content=null, isGlobal=false, isConstant=false, line=2]
Symbol [name=iter, structureType=VARIABLE, structureReturnType=INT, content=null,
↪  isGlobal=false, isConstant=false, line=3]
Symbol [name=Return8_15, structureType=RETURN, structureReturnType=INT,
↪  content=null, isGlobal=false, isConstant=false, line=8]
Symbol [name=Return16_11, structureType=RETURN, structureReturnType=INT,
↪  content=null, isGlobal=false, isConstant=false, line=16]
Symbol [name=factorial, structureType=FUNCTION, structureReturnType=INT,
↪  content=null, isGlobal=false, isConstant=false, line=1]
```

```
12 Symbol [name=nums, structureType=VARIABLE, structureReturnType=TUP,
   ↪  content=[Symbol [name=Number20_16, structureType=VALUE,
   ↪  structureReturnType=INT, content=null, isGlobal=false, isConstant=true,
   ↪  line=20], Symbol [name=Number20_19, structureType=VALUE,
   ↪  structureReturnType=INT, content=null, isGlobal=false, isConstant=true,
   ↪  line=20], Symbol [name=Number20_22, structureType=VALUE,
   ↪  structureReturnType=INT, content=null, isGlobal=false, isConstant=true,
   ↪  line=20]], isGlobal=false, isConstant=false, line=20]
```

Efectivamente, se crea una nueva instancia en la tabla de accesos que toma los valores desde 0 al 5 que son, respectivamente, el parámetro `number`, las variables `result` e `iter`, los valores de retorno de las líneas 8 y 16 y la propia función.

Adicionalmente, se puede ver como en el símbolo de tupla `nums` tiene populado su atributo `content` por el conjunto de elementos declarados en la línea 20.

6.2. Operaciones

Con este planteamiento, las operaciones necesarias son las siguientes:

- `int incrementTaIndex()`: Añadir +1 al índice de acceso para poder identificar en que funcion estamos añadiendo o mirando símbolos. Devuelve el índice actual una vez modificado.
- `boolean addAccess()`: El Access se crea con índices -1 en inicio y final para que quede reflejado el caso que no se añadan nuevos símbolos. En caso de que se añadan se modificarán acordemente. Devuelve true si la inserción ha sido correcta y false en caso contrario. Automáticamente, se añade +1 al índice de acceso.
- `boolean addSymbol(Symbol symbol)`: Añade un símbolo a la tabla de símbolos. Retorna true si la inserción ha sido correcta y false en caso contrario.
- `Symbol searchSymbolAtAccess(int access, String id)`: Busca en el Access pasado por parámetro el ID pasando por parámetro. Devuelve el símbolo cuyo id sea el mismo que el pasado por parámetro si se ha encontrado y null en caso contrario.
- `Symbol getSymbol(String id)`: Dado el acceso actual, busca en las definiciones de símbolos el símbolo pasado por parámetro. Si no se encontrase, busca en el acceso global. Devuelve dicho símbolo si se ha encontrado y null en caso contrario.
- `Symbol getFunction(String id)`: Busca en todas las funciones definidas en la tabla de accesos por una que tenga el mismo nombre que el pasado por parámetro. Esta función está parcialmente optimizada para que no se tenga que hacer un recorrido completo de la tabla de símbolos y aprovechar la información que tenemos de las funciones en la tabla de accesos para solo visitar los símbolos necesarios. Devuelve el símbolo si se ha encontrado y null en caso contrario.

- `Symbol getParameter(String functionId, int nParam)`: Busca la función pasada por parámetro de manera eficiente. Una vez encontrada mediante indexación encuentra el parámetro en concreto. En caso de que el parámetro indexado este fuera de límite de la tabla de símbolos, devolverá null. Devuelve el símbolo indexado si es un parámetro y null en caso contrario.
- `Stack<Symbol>getParameters(String functionId)Stack<Symbol>getParameters(String functionId)` Busca los parámetros de la función pasada por parámetro. Devuelve un stack con los parámetros si se ha encontrado y null en caso contrario.
- `Symbol getNTupleArgument(String id, int nArg)`: Devuelve el n-esimo argumento de la tupla pasada por parámetro. En caso de que no exista el símbolo o el n-ésimo argumento, devolverá null, en caso contrario devolverá el símbolo.

7. Generación del código intermedio

Con el objetivo de que el proceso de parseo del código fuente sea lo más rápido posible, se ha decidido aislar por completo la generación de código intermedio.

Por ende, cuando acaba el proceso de parseo, se itera por todo el árbol sintáctico generando el código intermedio pertinente para cada uno de los nodos. Comentar, que debido a que a no hay instrucciones de salida de bucles (como podría ser `break` o `continue` no se ha necesitado implementar un algoritmo de `backpatching`.

8. Optimización

Para la optimización del código intermedio se han implementado algunas optimizaciones de peephole, en concreto, las siguientes:

- Asignaciones diferidas.
- Normalización de operaciones conmutativas.
- Reducción de fuerza.
- Eliminación de código inaccesible.
- Eliminación de código no usado (tanto variables como funciones).
- Eliminación de etiquetas no usadas.
- Evaluación de expresiones.

9. Generación de código ensamblador

Para la generación del código ensamblador se decidido emplear el easy68k como arquitectura. En cuanto a librerías externas, no se ha incorporado ninguna para la generación código ensamblador.
Cabe destacar, que para la interacción con el sistema operativo se han creado unas subrutinas. En concreto:

- Salida por pantalla:
  - PRINT INT: Imprime un entero por pantalla.
  - PRINT STRING: Imprime un string por pantalla. Empleado para mostrar por pantalla el valor de un booleano como string.
- Entrada por teclado:
  - READ INT: Lee un entero por entrada de teclado.
  - READ STRING: Lee un entero por entrada de teclado. Empleado para leer como string un booleano.
- Auxiliares:
  - BOOLEAN TO STRING: Convierte un booleano a string.
  - STRING TO BOOLEAN: Convierte un string a booleano.

Estas subrutinas solo se incorporan en el código ensamblador sí y solo sí se emplean en el código intermedio.