Compiladors 2024-2025

Compilador per a un Llenguatge Imperatiu

Marc Llobera Villalonga
43461915B

1. Introducció

Per la construcció del compilador s'ha emprat el llenguatge Java amb les eines JFlex i CUP. Per al llenguatge assemblador s'ha emprat Motorola 68k i l'IDE EASy68K.

Per executar el compilador sobre un programa nou es pot introduir el codi dins l'arxiu anomenat **'programa.txt'** al directori **'./src/'**, dins el mateix directori es troba un programa (Executar.java) en Java que es pot executar per compilar el programa. D'aquesta execució es generaran tots els fitxers a les carpetes corresponents, entre ells es trobarà els arxius .X68 del codi assemblador tant optimitzat com fora optimitzar a la carpeta **'./src/compiler/c3a/'**.

Al mateix programa per executar es troba una variable booleana que si es posa a true compila els arxius JFlex i CUP, cosa que no es necessària ja que estan compilats i s'han generat els arxius corresponents. Per a l'execució d'aquest programa es necessari tenir tots els arxius essencials als directoris corresponents i tenir el JDK de Java descarregar i dins la variable de entorn PATH del dispositiu on s'executa. Es recomana executar amb el SO Windows 11 ja que és l'emprat per crear el compilador.

2. Definició del llenguatge

Una regla important del llenguatge que es compleix sempre és que tota línia de codi ha de acabar amb ";" excepte els inici i final de les declaracions de funcions i operacions com *if*, *while*, etc. També, per posar comentaris s'empra /// y tot el que vengui darrera serà un comentari fins trobar un salt de línia (*n*).

Tipus de dades implementades

- **Nombres enters**: aquests es declaren amb la paraula reservada ***integer*** i pot ser un nombre enter qualsevol en base 10.
- **Valors booleans**: aquests es declaren amb la paraula reservada ***logical*** i poden tenir el valor ***TRUE***, ***FALSE, true*** *o* ***false***.

Declaració de variables i constants

Les variables i constants sempre que es creen per primer pic s'ha de indicar de quin tipus son.

- **Variables**: aquestes s'indiquen amb la paraula clau ***val*** i després del nom de la variable sempre es posa "::", el tipus de la variable i finalment s'assigna el valor. Per assignar un altre valor a una variable ja declarada simplement es fa amb ***nomVariable = nouValor;***. Es pot assignar un valor a una variable a partir d'una operació aritmètica o lògica.

```
val x::integer = 3;  // x = 2;

val faSol::logical = TRUE;

val y::logical = 3 === 3;
```

- **Constants**: s'indiquen amb la paraula clau ***con***. Es declaren igual que una variable però no es pot modificar el valor una vegada declarats.

```
con max::integer = 10;
```

Tipus definits

Hi ha un tipus especial que és la tupla que funciona com una variable la qual esta definida per dues variables diferents que poden ser de qualsevol tipus. La tupla es declara amb la paraula reservada ***tuple*** i sempre s'ha de indicar el tipus de les seves dues variables dins "{}" i separades per ",". En aquest moment la tupla es buida així que s'ha de afegir el seu valor. Per fer la referència a les variables de la tupla es fa amb el nom de la tupla y l'indicador [posició variable]. D'aquesta manera només has de posar la posició per obtenir el valor, només es por emprar 0 o 1 per l'índex, emprar un altre valor no està permès.

```
tuple persona = {integer, logical};

persona[0] = 2;

persona[1] = TRUE;

val x::integer = persona[0];
```

Expressions aritmètiques i lògiques

- **Suma**: la suma s'indica amb el símbol **+** .

```
val x::integer = (3 + 4);
```

- **Resta**: la resta s'indica amb el símbol **-** .

```
val x::integer = (3 – 4);
```

- **Igual**: l'expressió lògica "igual a" s'indica amb el símbol **===** .

```
4 === 4
```

- **Diferent**: l'expressió lògica "diferent a" s'indica amb el símbol **/=** .

```
3 /= 4
```

- **AND**: l'expressió lògica "AND" o "i" s'indica amb el símbol ***&&*** .

```
(x === 3) && (esHome /= FALSE)
```

- **OR**: l'expressió lògica "OR" o "o" s'indica amb el símbol **||** .

```
(x  /= 3) || (x /= 4)
```

Precedència d'operadors (de major a menor prioritat):

- **Parèntesis**: (EXP)
- **Aritmètics**: +, - (avaluats de esquerra a dreta)
- **Relacionals/Lògics**: ===, /=, &&, || (avaluats de esquerra a dreta)

Operacions

- **Assignació**: com ja s'ha vist abans, l'assignació es realitza amb el símbol **=** .

```
val z::logical = FALSE;
```

- **Condicional**: per al condicional s'empra ***if():***, ***else:*** i ***endif***.

```
if(x === 12):
    /// Codi
else:
    /// Codi
endif
```

- **Bucle while**: es poden fer bucles de la forma ***while():*** i ***endwhile*** .

```
while(x === 12):
    /// Codi
endwhile
```

Tant al ***while*** com a l'***if*** les condicions han d'anar sempre entre "()".

- **Bucle for**: també hi ha els bucles for que es poden fer amb ***for _ to _ : endfor*** . Aquest bucle només serveix per variables que son nombres enters y es sumarà 1 al final de la iteració fins que el comparador vegi que la variable es igual al nombre al que es vol arribar, en tal cas no executarà el codi intern sinó que sortirà directament, així per el primer exemple mostrat *x* iterarà amb els valors: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (10 pics). L'inici del bucle sempre ha de declarar una variable tipus integer (no fa falta la sintaxis 'val', ja es suposa que es una variable) que es la que s'iterarà mentre que el final ha de ser una expressió de tipus integer (a les expressions es suposa que també es poden usar símbols ja declarats).

```
for x::integer = 0 to 10:
    /// Codi
endfor
////////////////////////////////////
con z::integer = 100;
for x::integer = (10 + 3) to z:
    /// Codi
endfor
```

Operacions d'entrada i sortida

- **Entrada**: es pot entrar valors (enters i booleans) des de teclat amb l'expressió ***in(tipusValor)***. Es pot assignar a una variable o retornar des d'una funció. El funcionament es com si sigues una funció on el que entres per teclat (si es vàlid) serà retornat. A la pantalla apareix aquest missatge: "Enter a *tipusValor*: ". Aquesta

operació suposa que es assignada o emprada en alguna expressió, per tant no es pot usar tot sola.

```
val x::integer = in(integer);

con y::logical = in(logical);

//////////////////////////////////////

NO !!!!!!

in(logical);
```

Pensa que si li has indicat un integer l'únic que pots escriure per teclat es un nombre enter i si has indicat booleà només pots entrar *FALSE* o *TRUE* (només majúscula). En qualsevol cas no acceptat retorna error i continua l'execució.

- **Sortida**: pots fer sortir per consola qualsevol variable, constant o valor directament amb ***out()***. Només es pot imprimir un enter o un booleà però si indiques una variable o constant no fa falta indicar el tipus (no es pot declarar una variable o constant, aquestes ja han d'estar creades). No funciona com una funció per tant no es pot usar per assignar un valor a una variable ni dins un bucle ni res similar, s'ha de usar sempre en solitari, però al seu paràmetre de sortida si es poden usar constants, variables i funcions.

```
out(x);

out(FALSE);

out(10);
```

Definició i crida de funcions

Per definir una funció emprem les paraules reservades ***fnct*** i ***endfnct***, i es poden usar paràmetres sempre que es defineixin amb el seu tipus dins els "()". Es obligatori que la funció retorni un valor i s'ha de indicar el tipus quan es declara la funció. També està la capacitat de retornar valors també definint el tipus a la funció amb la paraula ***rtrn*** i sempre entre "()". Per cridar a la funció es simplement el nom amb els "()" i els paràmetres corresponents i sempre acabant amb ";". La darrera instrucció de la funció ha de ser un return per assegurar que retorna alguna cosa. No es poden posar varis ***rtrn*** al cos, l'únic permès es el final. Exemple de definicions de funcions:

```
fnct integer suma(x::integer, y::integer):
        rtrn (x + y);
endfnct

//////////////////////////////////////

fnct logical imprimir1():
        out(1);
        rtrn (TRUE);
endfnct
```

```
//////////////////////////////////////

fnct logical sonIguals(x::integer, y::integer):

        con iguals::logical = (x === y);

        rtrn (iguals);

endfnct

//////////////////////////////////////

imprimir1()

//////////////////////////////////////

val x::integer = suma(1, 2);
```

Extra

Es poden emprar funcions en lloc de variables o constants en qualsevol lloc on es pugui emprar una variable o constant.

La localitat d'una variable només es dona a les funcions, en altres sentències com els if o for totes les variables son declarades globalment.

No es poden declarar funcions dins altres funcions.

El codi s'executa seqüencialment, no hi ha una funció main que comenci l'execució.

Algunes de les coses que estan permeses son:

```
fnct integer retornaIn():

        rtrn (in(integer));

endfnct

//////////////////////////////////////

out(suma(3, 5));

//////////////////////////////////////

suma(in(integer), 2)

//////////////////////////////////////

tuple persona = {integer, logical};

persona[0] = 2;

persona[1] = TRUE;

val x::integer = persona[0];

if (persona[1]):

   out(TRUE);

else:
```

```
        out(FALSE);

    endif
```

3. Lèxic

S'ha optat per uns tokens senzills que no necessiten patrons elaborats.

Token	Patró
fnct	"fnct"
endfnct	"endfnct"
rtrn	"rtrn"
val	"val"
con	"con"
if_t	"if"
else_t	"else"
endif	"endif"
while_t	"while"
endwhile	"endwhile"
for_t	"for"
to	"to"
endfor	"endfor"
in	"in"
out	"out"
tuple	"tuple"
integer	"integer"
logical	"logical"
assign	"="
plus	"+"
minus	"-"
equal	"==="
not_equal	"/="
and	"&&"
or	"||"
lparen	"("
rparen	")"
colon	":"
double_colon	"::"
semicolon	";"
comma	","
lbrace	"{"
rbrace	"}"
lbracket	"["
rbracket	"]"
integer_literal	[0-9]+
boolean_literal	[T][R][U][E] | [F][A][L][S][E] | [t][r][u][e] | [f][a][l][s][e]
id	[a-zA-Z][a-zA-Z0-9]*
whitespace	[\t\n\r]+
comment	"///".*

4. Sintàctic

L'analitzador sintàctic que correspon a l'arxiu .cup per a l'anàlisi sintàctic només té les funcions (sobreescrites) per a reportar els errors sintàctics que l'analitzador troba.

L'analitzador CUP empra el mètode LALR(1) que es una versió millorada del LR(1).

La gramàtica creada per al llenguatge és la següent:

S => P

P => SENT P

| FUNC_DECL P

| /* epsilon */

SENT => DECL

| ASIG_SENT

| IF_SENT

| WHILE_SENT

| FOR_SENT

| SAL_SENT

| CRID_FUNC

DECL => VAR_DECL

| CONS_DECL

| TUPLA_DECL

VAR_DECL => val id double_colon TIPO assign EXP semicolon

CONS_DECL => con id double_colon TIPO assign EXP semicolon

TUPLA_DECL => tuple id assign lbrace TIPO comma TIPO rbrace semicolon

ASIG_SENT => id assign EXP semicolon

| TUPLA_ACCESS assign EXP semicolon

IF_SENT => IF_ELSE COS endif

| IF_INIT COS endif

IF_ELSE => IF_INIT COS else_t colon

IF_INIT => if_t lparen EXP rparen colon

WHILE_SENT => WHILE_REP COS endwhile

WHILE_REP => while_t lparen MW: EXP rparen colon

MW => /* epsilon */

FOR_SENT => FORINIT COS endfor

FORINIT => for_t PARAM assign EXP to EXP colon

FUNC_DECL => FUNCINIT COS FINAL_RTN endfnct

| FUNCINIT FINAL_RTN endfnct

FUNCINIT => MF fnct TIPO id lparen PARAM_LIST rparen colon

MF => /* epsilon */

PARAM_LIST => PARAM

| PARAM comma PARAM_LIST

| /* epsilon */

PARAM => id double_colon TIPO

FINAL_RTN => rtrn lparen EXP rparen semicolon

COS => SENT COS

| SENT

SAL_SENT => out lparen EXP rparen semicolon

ENT_SENT => *in* lparen TIPO rparen

EXP => lparen EXP rparen

| LOG_TERM

| ARIT_TERM

| CRID_SIMB

| LIT

| ENT_SENT

CRID_SIMB => id

| CRID_FUNC

| TUPLA_ACCESS

ARGS => EXP ARGS_LIST

ARGS_LIST => comma EXP ARGS_LIST

| /* epsilon */

CRID_FUNC => id lparen ARGS rparen

| id lparen rparen

LIT => ENT_LIT

| BOL_LIT

ENT_LIT => POS_ENT_LIT

| minus POS_ENT_LIT

POS_ENT_LIT => integer_literal

BOL_LIT => boolean_literal

TUPLA_ACCESS => id lbracket integer_literal rbracket

TIPO => integer

| logical

LOG_TERM => EXP equal EXP

| EXP not_equal EXP

| EXP and EXP

| EXP or EXP

ARIT_TERM => EXP plus EXP

| EXP minus EXP

5. Semàntic

L'anàlisi semàntic es fa durant l'anàlisi sintàctic al mateix arxiu .cup mentre es fan les produccions. Emprant la taula de símbols y els objectes tipus Simbol es va guardant informació segons el que analitza la sintaxis. La més important per la part semàntica és el tipus del objecte (enter o booleà), el nom si es tracta d'un símbol al igual que la instancia (VariableConstant, Tupla, Funcio), en alguns casos el valor (per els literals) i l'àmbit.

També per la creació del codi intermig guardem valors com la posició en memòria, el desplaçament, etiquetes, procediments, etc. Segons la instancia del símbol aquestes variables poden variar.

Algunes variables generals de l'arxiu .cup que ens ajuden durant aquest anàlisi son:

- **DEBUG** (boolean): si s'activa es poden veure diferents missatges durant les produccions que ens ajuden a veure el recorregut de l'anàlisi sintàctic.
- **taulaSimbols** (TaulaSimbols): es una variable del tipus TaulaSimbols que conté la llista de símbols que es van guardant així com vàries funcions genèriques que ens ajuden a tractar aquesta llista.
- **error_detectat** (boolean): variable per indicar si hem trobat un error durant l'anàlisi semàntic. Si esta en true ja no es realitzarà l'anàlisi semàntic per lo que queda de codi.
- **àmbit** (String): es una variable que indica si en el moment de una producció estam dins una funció (variables locals) o en el bloc general (variables globals). Quan entrem dins una funció obtenim el valor de np (integer) quin es la posició de la darrera funció (o procediment) creat, també a part de la posició s'empra com id o nom de la funció durant la compilació i execució (es diferent al nom donat per l'usuari el qual s'empra en la taula de símbols).

- **c3@** (GenerarCodi): variable de tipus GenerarCodi que conté variables i procediments que s'encarreguen de generar el codi intermig i posteriorment optimitzar-lo i generar el codi assemblador.
- **nbytes** (integer): variable per indicar quants de bytes ocupen els enters.

També podem trobar funcions per imprimir els errors semàntics que trobem.

6. Taula Símbols

La taula de símbols conté una llista amb objectes tipus Símbols que a la vegada tenen una extensió de tipus VariableConstant (tant per variables com per constants), Tupla i Funcio.

- **Simbol**: conté les variables **value** (nom del símbol), **fila**, **columna**, **tipus** (tipus de la dada del símbol, tractem amb tipus INTEGER, BOOLEAN i NULL), **r** (posició en memòria), **d** (desplaçament), **valor**, **àmbit**, **etiqueta**, **etiquetafi** (per alguns casos com bucles on tenim més d'una etiqueta).
- **VariableConstant**: només conté una variable booleana per indicar si tracte d'una variable o d'una constant.
- **Tupla**: conté dues variables de tipus TipusDades on cada una indica el valor de cada objecte dins la tupla, al igual de dues variables integer que contenen els valors de la tupla. També tenim una variable que indica a quin dels dos objectes està apuntant la tupla (usat quan hi ha un accés a la tupla).
- **Funcio**: conté una llista tipus VariableConstant que representa els paràmetres que s'han indicat a la declaració de la funció. També destaca una funció per comprovar si una altre llista es igual a la llista de paràmetres de la funció (emprat durant la crida de funcions).

L'objecte TaulaSimbols també conté funcions per tractar la llista en les quals destaquen funcions per declarar nous símbols on es comprova la instancia i l'àmbit, si existeix un símbol que fa que el nou símbol no es pugui declarar retorna false, i també funcions per usar símbols de la taula que es crida quan hi ha una crida d'algun símbol, si no existeix a la taula es retorna un objecte Símbol buit.

7. Optimitzacions

A l'objecte GenerarCodi trobem una funció general que crida a funcions per optimitzar el codi intermig i generar un codi intermig optimitzat.

Entre les funcions que representen optimitzacions trobem:

- **Brancament Sobre Brancament**: es crida quan la instrucció es un skip y la següent es un goto. La funció recorrerà tot el codi intermig generat y si trobem un goto amb l'etiqueta de l'skip la canviem per l'etiqueta a la que apunta el goto.
- **Operacions Constants**: executada en instruccions amb operacions aritmètiques o lògiques. Si els dos operands son literals llavors es farà el càlcul corresponent y es posarà un copy, un goto o s'eliminarà la instrucció (depenent de la instrucció original).
- **Codi Inaccessible**: executat quan es troba un goto es mira si el codi següent es inaccessible fins un punt i s'elimina.

- **Assignació Diferida**: quan hi ha una instrucció copy es revisa tot el codi i si la variable es només assignada un sol pic es canvia tots els usos d'aquesta per el seu valor assignat inicialment.

8. Codi assemblador

El llenguatge assemblador elegit és el 68K amb l'IDE EASy68k, es recomana usar aquest IDE per executar els programes per assegurar al 100% el correcte resultat encara que no s'empra ninguna llibreria externa. Es crearà una sèrie de subrutines predeterminades al final de tots els fitxers que corresponen a la entrada i sortida de dades per consola emprats per les sentències 'in' i 'out'.

Per a cada instrucció del codi intermig podem trobar una funció que ho transforma a codi assemblador.

Quan s'empra una instrucció 'in' de tipus booleà (logical) només es permet entrar TRUE o FALSE tot en majúscules a diferència del llenguatge a alt nivell que permet de les dues formes. De la mateixa manera la instrucció 'out' imprimirà TRUE o FALSE encara que al programa s'hagi introduït true o false.

S'ha de remarcar que debut a la limitació del llenguatge al imprimir enters si aquests son nombres massa grans no s'imprimirà correctament. Això es pot solucionar modificant la subrutina que imprimeix enters per a que transformi les xifres del nombre a caràcters i imprimeixi caràcter a caràcter tot el nombre així com es fa per a les cadenes de TRUE o FALSE, però en aquest cas s'ha implementat la sortida de l'enter directament.