Compiladors

Hal

Héctor Ramón Jiménez

Alvaro Espuña Buxó

2 d'abril de 2014

Facultat d'Informàtica de Barcelona

1 Gramàtica

La nostra idea principal és que la sintaxi del llenguatge sigui molt neta. Per tant, el primer que vam decidir és desfer-nos de les claus '{' i fer que els blocs fossin definits per la seva propia indentació, com a Python. Per fer-ho, hem ampliat el `lexer` per tal de que emeti tokens relacionats amb la indentació (`INDENT` i `DEDENT`), tot utilitzant una pila que manté el nombre d'espais per cada nivell d'indentació definit:

```
@lexer::members
{
    public static final int MAX_INDENTS = 100;
    private int indentLevel = 0;
    int[] indentStack = new int[MAX_INDENTS];
    java.util.Queue<Token> tokens =
        new java.util.LinkedList<Token>();

    {
        // Compute first line indentation manually
        int i = 1;
        while(input.LA(i) == ' ') i++;

        int next = input.LA(i);

        // Ignore empty lines
        if(i > 1 && next != '\n' && next != '\r' && next != -1) {
            jump(Indent);
            indentStack[indentLevel] = i-1;
        }
    }

    @Override
    public void emit(Token t) {
        state.token = t;
        tokens.offer(t);
    }

    @Override
    public Token nextToken() {
        super.nextToken();

        if(tokens.isEmpty()) {
            // Undo all indentation
            if(indentLevel > 0) {
                jump(Dedent);
                return nextToken();
            }

            // End file with new line
            emit(new CommonToken(NEWLINE, ""));
            emit(Token.EOF_TOKEN);
        }
        Token t = tokens.poll();
```

```
        return t;
    }


    private void jump(int ttype) {
        String name;
        if(ttype == Indent) {
            name = "Indentation";
            indentLevel++;
        }
        else {
            name = "Dedentation";
            indentLevel--;
        }
        emit(new CommonToken(ttype, name));
    }
}
```

La indentació només pot succeïr quan hi ha un salt de línia. Per tant, afegim el comportament d'analitzar la indentació al token `NEWLINE`:

```
NEWLINE
    @init
    {
        int n = 0;
    }
    : NL (' ' {n++;} | '\t' {n += 8; n -= (n \% 8); })*
    {
        emit(new CommonToken(NEWLINE, "\\n"));

        int next = input.LA(1);
        int currentIndent = indentStack[indentLevel];

        // Skip if same indentation or empty line
        if(n == currentIndent || next == '\n' || next == '\r' || next == -1)
        {
            skip();
        }
        else if(n > currentIndent)
        {
            jump(Indent);
            indentStack[indentLevel] = n;
        }
        else
        {
            while(indentLevel > 0 && indentStack[indentLevel] > n)
            {
                jump(Dedent);
            }

            if(indentStack[indentLevel] != n)
                throw new RuntimeException("Unexpected␣indentation.");
        }
    }
```

Per la sintaxi ens hem basat en llenguatges com Ruby, Smalltalk, Python i Haskell. Volem que el llenguatge sigui còmode d'escriure, encara que la gramàtica sembli una mica més complicada:

```
prog
    :   (NEWLINE | stmt)* EOF -> ^(BLOCK stmt*)
    ;

stmt
    :   simple_stmt
    |   compound_stmt
    ;

simple_stmt
    @init{boolean conditional = false;}
    :   s1=small_stmt (
                (options {greedy=true;}:SEMICOLON small_stmt)* SEMICOLON?
            |   IF {conditional=true;} expr (ELSE s2=small_stmt)?
        )
        NEWLINE
        -> {conditional}? ^(IF_STMT expr ^(BLOCK $s1) ^(BLOCK $s2))
        -> small_stmt+
    ;

small_stmt
    : assign
    | expr
    ;

compound_stmt
    :   if_stmt
    |   for_stmt
    |   while_stmt
    |   fundef
    ;

if_stmt
    :   IF if_body -> ^(IF_STMT if_body)
    ;

if_body
    :   expr COLON! block if_extension?
    ;

if_extension
    :   ELIF if_body -> ^(BLOCK ^(IF_STMT if_body))
    |   ELSE! COLON! block
    ;

for_stmt
    :  FOR paramlist IN expr COLON block
        -> ^(FOR_STMT ^(PARAMS paramlist) expr block)
    ;
```

```
while_stmt
    :  WHILE expr COLON block -> ^(WHILE_STMT expr block)
    ;

block
    :   simple_stmt -> ^(BLOCK simple_stmt)
    |   multiline_block
    ;

multiline_block
    :   NEWLINE Indent (stmt)+ Dedent -> ^(BLOCK (stmt)+)
    ;

fundef
    :   DEF ID params COLON block -> ^(FUNDEF ID params block)
    ;

params
    :   paramlist? -> ^(PARAMS paramlist?)
    ;

paramlist
    :   ID (','! ID)*
    ;

funcall
    :   ID args -> ^(FUNCALL ID args)
    ;

args
    :   arglist? -> ^(ARGS arglist?)
    ;

arglist
    :   {space(input) && (!input.LT(1).getText().equals("-") ||
            directlyFollows(input.LT(1), input.LT(2)))}?
        expr (options {greedy=true;}: ','! expr)*
    ;


// Assignment
assign
    :   ID eq=EQUAL expr -> ^(ASSIGN[$eq,":="] ID expr)
    ;

expr
    :   boolterm (options {greedy=true;}: OR^ boolterm)*
    ;

boolterm
    :   boolfact (options {greedy=true;}: AND^ boolfact)*
    ;

boolfact
```

```
    :   num_expr (options {greedy=true;}:
            (DOUBLE_EQUAL^ | NOT_EQUAL^ | LT^ | LE^ | GT^ | GE^) num_expr)?
    ;

num_expr
    :   term (options {greedy=true;}: (PLUS^ | MINUS^) term)*
    ;

term
    :   factor (options {greedy=true;}: (MUL^ | DIV^ | MOD^) factor)*
    ;

factor
    :   NOT^ atom
    |   MINUS atom -> ^(MINUS["NEGATE"] atom)
    |   atom
    ;

atom
    :   INT
    |   (b=TRUE | b=FALSE)  -> ^(BOOLEAN[$b,$b.text])
    |   list
    |   funcall // An ID can be considered a "funcall" with 0 args
    |   LPAREN! expr RPAREN!
    ;

list
    :  LBRACK (expr (',' expr)*)? RBRACK -> ^(LIST expr*)
    ;
```

La part dels tokens és la següent:

```
// OPERATORS
EQUAL   : '=' ;
DOUBLE_EQUAL : '==';
NOT_EQUAL: '!=' ;
LT      : '<' ;
LE      : '<=';
GT      : '>';
GE      : '>=';
PLUS    : '+' ;
MINUS   : '-' ;
MUL     : '*';
DIV     : '/';
MOD     : '%' ;
// KEYWORDS
NOT     : 'not';
AND     : 'and' ;
OR      : 'or' ;
TRUE    : 'true';
FALSE   : 'false';
IF      : 'if';
ELIF    : 'elif';
ELSE    : 'else';
FOR     : 'for';
```

```
WHILE   : 'while';
IN      : 'in';
DEF     : 'def';
// SPECIAL SYMBOLS
COLON   : ':' ;
SEMICOLON : ';';
LPAREN  : '(';
RPAREN  : ')';
LBRACK  : '[';
RBRACK  : ']';

// Useful fragments
fragment DIGIT : ('0'..'9');
fragment LOWER : ('a'..'z');
fragment UPPER : ('A'..'Z');
fragment LETTER: (LOWER|UPPER);
fragment NL     : (('\r')? '\n')+;
fragment SP     : (' ' | '\t')+;

// Identifiers
ID  : (LETTER|'_') (LETTER|'_'|DIGIT)* ('!'|'?')?;

// Integers
INT : (DIGIT)+;

WS
    : SP {skip();}
    ;

COMMENT: '#' ~('\n'|'\r')* {skip();};
```

2 Exemples

2.1 Condicionals

```
# unlinis
if is_printable? a: do_something; print a;
print a if is_printable? a else print b

# multilinis
if true:
    do_a
elif false:
    do_b
else:
    do_c
```

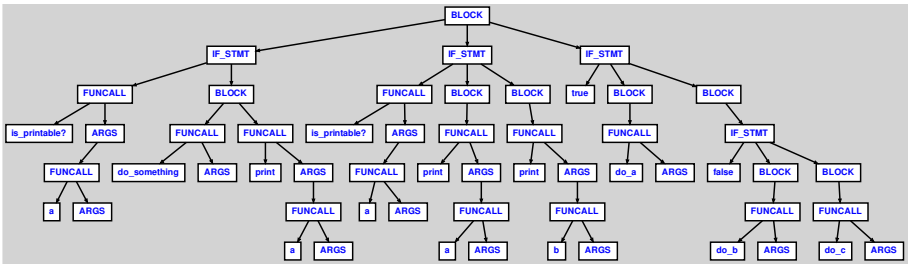

Figura 1: Conditionals. AST resultant. Fes zoom per veure la imatge

2.2 Definició de funcions

```
def f1! l:
    def f2 a:
        for x,y in a:
            print y
    f2 l
```

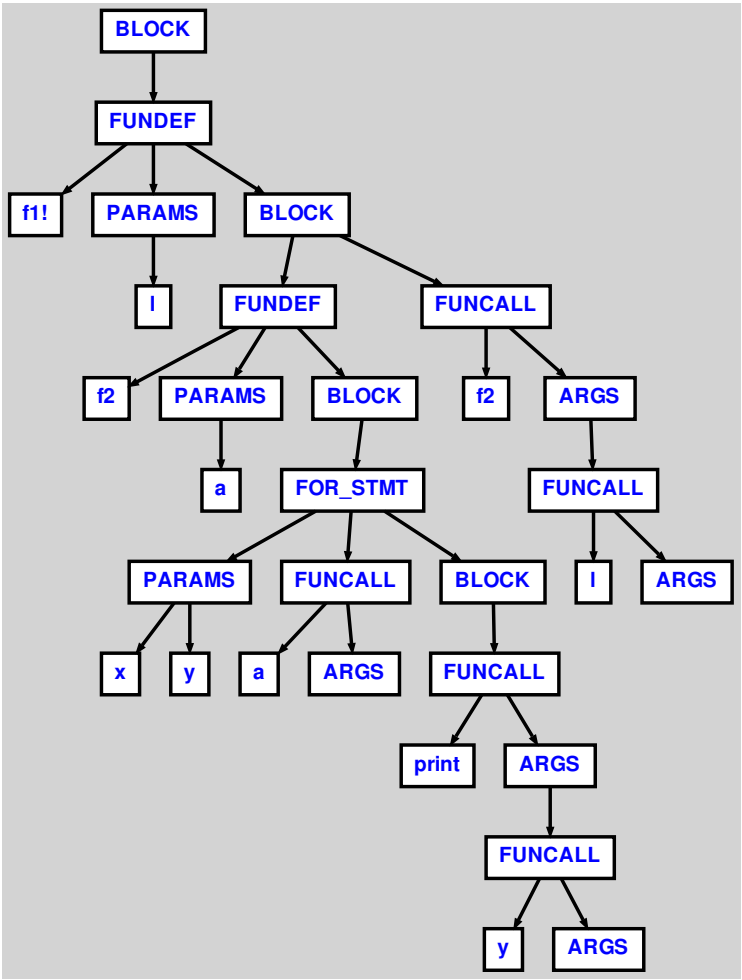

Figura 2: Definició de funcions. AST resultant

2.3 Crides a funcions

```
f1 f2 a,k
f1 (f2 a),k

# la separacio entre tokens es significativa
# en el cas de la negacio
f-3
f -3
f - 3
```

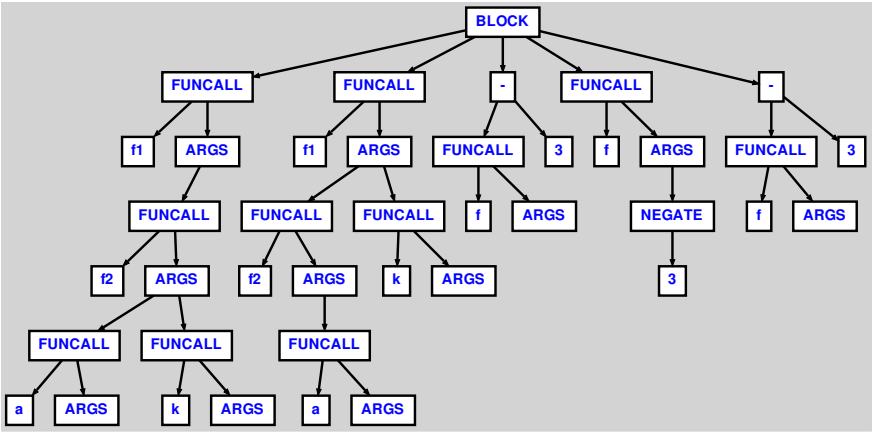

Figura 3: Crides a funcions. AST resultant

2.4 Bucles

```
# while
while boolea:
    if i == 0:
        p

# for
for key, value in map:
    print key, value
```

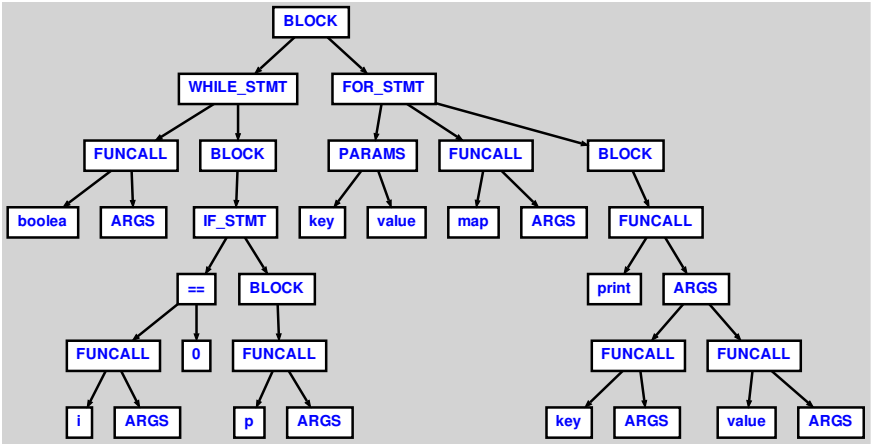

Figura 4: Bucles. AST resultant

2.5 Llistes

```
l = [1, 3 * 5, e + 15, f]
```

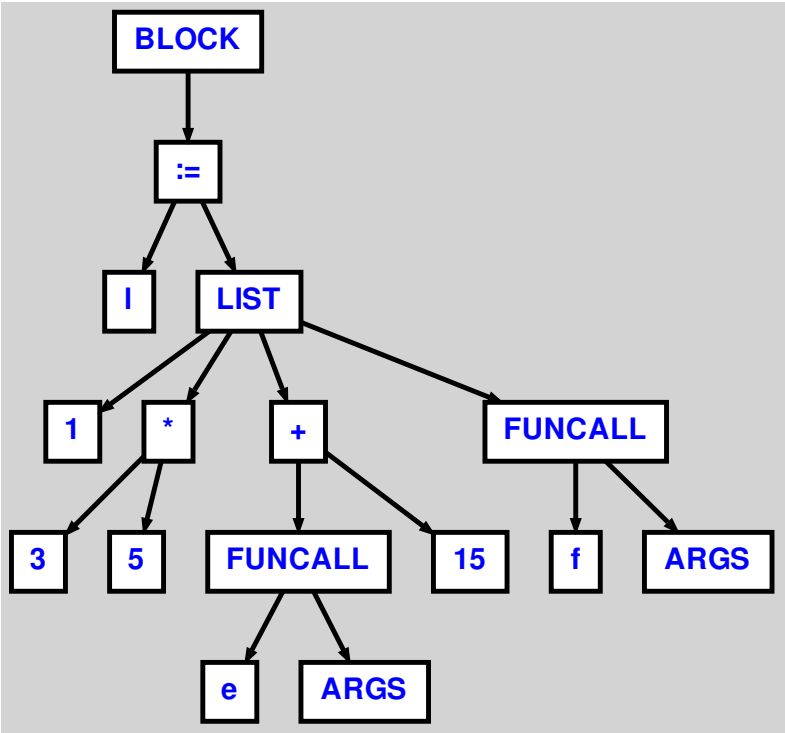

Figura 5: Llists. AST resultant

3 Breu descripció dels següents passos

El següent pas és, principalment, construir un evaluador de l'arbre sintàctic. En ell haurem de tenir en compte coses com els *scopes* de les variables i la taula de símbols per a les funcions. Ens podem basar una mica en l'`Asl`.

Una particularitat és que hem considerat que els identificadors pertanyin tots a crides de funcions, considerant les *variables* casos particulars de funcions amb 0 arguments, i que retornen el valor assignat. Creiem que el fet que tot és comporti com crides a funcions ens permetrà fer un evaluador més senzill, perquè serà més homogeni.

A `Hal` volem donar-li força importància a les funcions, per facilitar la generació de *DSL*s. Hi haurà altres tipus de dades bàsics (numèrics, llistes,...), volem centrar-nos en el tipus `Function`. Volem suportar *currying* i funcions de primer ordre, per tant, haurem de tenir una classe `Function` força potent.

També voldriem implementar classes. Per encapsular conjunts de funcions. Es cridarien amb la notació del `.` (p.e. `instancia.metode arg1, arg2`). Si arribéssim a suportar herència segurament seria només simple, per facilitar l'ordre de resolució de mètodes.

La idea és que a partir d'una base petita del llenguatge, amb un evaluador senzill, sigui possible construir la resta del `Hal` a sobre.