INFO-F403: First assignment

INTRODUCTION TO LANGUAGE THEORY AND COMPILATION

TYPY

Markus LINDSTRÖM

Academic Year 2010-2011

1 Introduction

Great news! The computer science department has just acquired a revolutionary new Color Screen System™ which offers an incredible 64×64 pixel screen along with a dashing 16 colour palette, which it hopes to use to participate at the TAT '11 (Terrible Art Tournament). However, the screen is only compatible with the somewhat primitive GPMACHINE and its low-level language, which makes using the screen somewhat painful.

Since the department would really like to show off its skills at making incredible graphics and animations, it has devised a higher-level language called TYPY to allow users to make use more easily of the above mentioned Color Screen System™ (did we mention it really *is* quite smashing[1], not to mention prohibitively expensive?). Unfortunately, there is currently no working compiler for the language, and this is obviously where you come in: we need you to craft a powerful tool that may allow us to someday earn research funds by becoming invited speakers at avant-garde conferences on modern art!

2 TYPY

TYPY (short for TYPED PYTHON) is a programming language that can best be described as a mixture of PYTHON and C. Its syntax is mainly derived from the former, but its typing system is mostly inspired by the latter. The basic structure of a TYPY program is the following, in order:

1. Global variable declarations
2. Function definitions
3. Main program code (execution of a TYPY program starts here)

2.1 Typing system

TYPY is an *explicitly* and *strongly* typed language where variables and expressions can have only two different types: integer (`int`) or boolean (`bool`). The language supports functions which may have three different return types: `int`, `bool` or none at all (much like `void` functions in C). A `return` statement can be used to return a value from a function, but obviously makes no sense if the function has no return type.

Boolean literals are either `True` or `False`. Identifiers have the same format as in C: they must begin with a letter or an underscore but can be followed by an arbitrary number of letters, numbers or underscores. TYPY is case sensitive. Like PYTHON, the language delimits its blocks by using

[1] We're not selling out, honest!

indentation instead of using curly braces as in C. Like PASCAL, the language is designed to be processed by a single-pass compiler (which means it can only process the input file once) by satisfying the following constraints:

1. Variables must be declared before use (in addition, all local variables used must be declared at the beginning of their function or the main block).
2. Functions must be defined before use. This trivially prevents writing mutually recursive functions. However, directly recursive functions are allowed.

Comments in TYPY programs come in two possible shapes:

1. Single line comments start with a `'#'` character
2. Multiline comments start and end with three double quotes: `"""`

Comments and blank lines (i.e. lines with only whitespace and possibly a comment) are to be filtered at the lexical analysis level, which means the scanner must not return a token to the parser but simply continue reading the rest of the input until the next token is found.

An example program is given in Figure 1.

2.2 Formal grammar and example

TYPY's formal grammar can be found on p. 4. The † symbol represents the newline character (`'\n'`).

2.3 Recognising INDENT and DEDENT tokens

Being based on PYTHON, TYPY defines its blocks based on indentation levels; this is done transparently by the grammar on the condition that the scanner detects the INDENT and DEDENT tokens correctly. In TYPY, we'll only allow tabulations (the `'\t'` character) to define indentation (PYTHON also allows spaces) and we'll consider that the number of tabs at the beginning of each input line is its *indentation level*. In essence, the scanner must produce an INDENT token if it detects an increase in the indentation level, whereas DEDENT tokens must be produced when indentation decreases. TYPY only allows indentation levels to increase by at most one every time, but they can decrease by as many levels as possible in one go. Consider the following code excerpt :

```
if x == 3:
  while i < N:
    print(i+j)
print(N)
```

The scanner would produce a single INDENT token on lines 2 and 3, but would produce two successive DEDENT tokens on line 4. Algorithm 1 (p. 5) outlines a method for recognising INDENT and DEDENT tokens which should be used by your scanner. It assumes two counters, *indentc* and *dedentc* are created at the beginning of the program and initialised with the value 0.

```
# Global variables
int size, term

# Example recursive function
def int silly_add(int n, int m):
  """ This function illustrates
  how adding should *not* be done.
  """
  int result
  if m = 0:
    result = n
  else:
    result = 1 + silly_add(n,m-1)
  return result

# Another function
def fancy_square(int line, int col):
  int i,j
  # This function obviously can't use local
  # variables from the previous one.
  i = line
  while i < line+size:
    j = col
    while j < col+size:
      putr(i,j)
      j = j + 1
    i = i + 1

# Program entry point
read(size)
fancy_square(size,size)
read(term)
print silly_add(term,term)
```

Figure 1: An example TYPY program.

[1] <PROGRAM> → <VARDECL> <FUNCTIONLIST> <MAIN> $
[2] <FUNCTIONLIST> → <FUNCTION> † <FUNCTIONLIST>
[3] → ε
[4] <FUNCTION> → def <FTYPE> ID(<ARGS>): <BODY>
[5] <FTYPE> → <TYPE>
[6] → ε
[7] <ARGS> → <ARGLIST>
[8] → ε
[9] <ARGLIST> → <TYPE> ID, <ARGLIST>
[10] → <TYPE> ID
[11] <TYPE> → int
[12] → bool
[13] <BODY> → † INDENT <VARDECL> <STATEMENTLIST> DEDENT
[14] <VARDECL> → <VARLINE> <VARDECL>
[15] → ε
[16] <VARLINE> → <TYPE> <IDLIST> †
[17] <IDLIST> → ID, <IDLIST>
[18] → ID
[19] <MAIN> → <VARDECL> <STATEMENTLIST>
[20] <STATEMENTLIST> → <STATEMENT> <STATEMENTLIST>
[21] → <STATEMENT>
[22] <STATEMENT> → <SIMPLESTMT> †
[23] → <COMPOUNDSTMT>
[24] <SIMPLESTMT> → ID = <EXPRESSION>
[25] → <CALL>
[26] → read(ID)
[27] → print(<EXPRESSION>)
[28] → return <EXPRESSION>
[29] → pass
[30] → <SCREENCOMMAND>
[31] <COMPOUNDSTMT> → <IF>
[32] → <WHILE>
[33] <EXPRESSION> → <EXPRESSION> <OP> <EXPRESSION>
[34] → (<EXPRESSION>)
[35] → − <EXPRESSION>
[36] → not <EXPRESSION>
[37] → <CALL>
[38] → ID
[39] → INTEGER_LITERAL
[40] → BOOLEAN_LITERAL
[41] <OP> → +
[42] → -
[43] → *
[44] → /
[45] → and
[46] → or
[47] → ==
[48] → <
[49] → >
[50] → <=
[51] → >=
[52] → !=
[53] <IF> → if <EXPRESSION>: <BODY>
[54] → if <EXPRESSION>: <BODY> else: <BODY>
[55] <WHILE> → while <EXPRESSION>: <BODY>
[56] <SCREENCOMMAND> → put(<EXPRESSION>, <EXPRESSION>, <EXPRESSION>)
[57] → putr(<EXPRESSION>, <EXPRESSION>)
[58] → get(<EXPRESSION>, <EXPRESSION>)
[59] → hold
[60] → release
[61] → reset
[62] <CALL> → ID(<CALLARGS>)
[63] <CALLARGS> → <EXPRLIST>
[64] → ε
[65] <EXPRLIST> → <EXPRESSION>
[66] → <EXPRESSION>, <EXPRLIST>

Algorithm 1: Sketch for detection of INDENT and DEDENT tokens during lexical analysis

```
if dedentc > 0 then
    dedentc ← dedentc − 1;
    token ← DEDENT;
else
    n ← number of tabs at beginning of line (removes them from input);
    if n = indentc + 1 then
        indentc ← indentc + 1;
        token ← INDENT;
    else if n < indentc then
        dedentc ← indentc − n − 1;
        token ← DEDENT;
    else if n = indentc then
        handle rest of input line as usual (using your DFA) to get next token;
    else
        indentation error (n > indentc + 1);
    end
end
return token;
```

3 GPMACHINE/CSS

The objective of a compiler is to generate useful code. We ask that your compiler generates code for a stack-based GPMACHINE-like environment. Specifically, we'll ask that you use an extended version, called GPMACHINE/CSS, that also simulates[2] the Color Screen System™. This version can be downloaded from the exercises and assignments Web site:

`http://www.ulb.ac.be/di/info-f403/`

The screen is in essence a square matrix of pixels. It also implements a buffer memory. This allows the user to "hold" the current screen contents (the *display*), change the buffer memory (thus without changing the display), then finally "release" the buffer which then refreshes the display with the buffer contents. This allows the display to be changed in one go rather than by individual pixels, which allows for smoother transitions.

GPMACHINE/CSS supports the same instructions as the original GPMACHINE and adds six new ones pertaining to the screen:

1. `put`: pops three values off the stack (*color*, *column* then *line* respectively), then puts the given colour into the screen at the given position. Will ignore invalid coordinates.

2. `putr`: pops two values off the stack (*column* then *line* respectively), then puts a random colour into the screen at the given position. Will ignore invalid coordinates.

3. `get`: pops two values off the stack (*column* then *line* respectively), then pushes the colour currently stored at the given position. Will return -1 on invalid coordinates.

[2] We prefer you use a simulator instead of the real hardware so you don't break our shiny toy ;)

4. `hold`: puts the display on hold. Modifying pixels in this state will have no *visible* effect but will still modify the screen's buffer memory. This instruction does not change the stack and has no effect if the display is already on hold.

5. `rls`: releases the screen hold. This forces the buffer memory to become the new screen contents. This instruction does not change the stack and has no effect if the display is not on hold.

6. `rst`: releases the screen hold (if set) then resets the screen, making it black. This instruction does not change the stack.

The instructions obviously only work with integer values on the stack; other types will cause errors. As seen in the TYPY grammar, these instructions appear as predefined statements in the language. Note that colour values given to `put` will be truncated to values between 0 and 15; this is also the only values possibly returned by `get` (with the addition of -1 in case of invalid coordinates).

4 Implementing function calls in GPMACHINE

The instructions given during the exercise sessions are not sufficient to implement real function calls in a language, let alone recursive ones. The special session on this assignment, given February 7, explains how this can be handled in GPMACHINE.

5 *Desiderata*

Given the grammar of TYPY and its semantics, we ask that you build a fully operational single-pass compiler for the language. It can be implemented in C++ or Java (if you wish to use another language, please check with the assistant first). We expect the following from your assignment:

1. Identify the *lexical units* (or tokens) of TYPY and their corresponding regular expressions;

2. Give a deterministic finite automaton that recognises them correctly (note that the INDENT and DEDENT tokens can be left out of the DFA since they require special processing, as seen in Section 2.3);

3. Program a *scanner* that recognises the tokens of TYPY. It should be implemented as a function that, at each call, returns the next token read on input. A symbol table (which contains global, local variables, function types, etc.) must also be implemented.

4. Modify the given grammar:

   (a) In order for it to become LL(1);

   (b) And to account for operator precedence and associativity, which are given in Table 1.

5. Give the LL(1) parsing table for your new grammar as well as the needed *First()* and *Follow()* calculations you needed to derive it;

6. Decorate the grammar to produce adequate low-level code for GPMACHINE/CSS;

7. Program a recursive descent LL(1) parser that outputs code for GPMACHINE/CSS.

Operators	Associativity
`()`	
`not`, unary − (rule 35)	right
`*`, `/`	left
+, binary − (rule 42)	left
>, <, >=, <=, `!=`, ==	left
`and`	left
`or`	left

Table 1: TYPY operator precedence (top to bottom) and associativity

8. The compiler must support *systematic type checking*. Each expression can be either of integer or boolean type, and the compiler must signal an error if some odd mixture arises. For example, `a < 3 and 2`, wrong `return` types, wrong assignment types and wrong parameters passed to function calls must be rejected at compilation time. The compiler must also check that function calls make sense by checking the number and types of its parameters.

9. The compiler does not need to support *variable shadowing*. If a local variable happens to have the same name as a global variable, an error can be thrown. However, global variables must obviously be accessible anywhere in the program.

We require that your work be presented in a report (written in either English or French) that contains all the various information we asked for above and that explicits and justifies any and all choices or hypotheses made during your work. Your report should also include instructions on how to use your compiler.

We expect you to deliver an archive (`.zip` or `.tar.gz`) containing an electronic copy of your report and the source code of your compiler (preferably with a working *Makefile* if it needs to be compiled) as well as any other files (test files written in TYPY, etc.) you'd find useful for our evaluation by way of e-mail to `mlindstr@ulb.ac.be`. The deadline for sending the archive is **Monday 7 March 2011 at 23:59**. We also expect you to hand in a printed copy of your report the same day **before 15:30** to the students' secretary, Maryka Peetroons (room N8.104). Deadline misses will *not* be tolerated.

The assignment can be made individually or in pairs (which we recommend). Evaluation of the assignment will be based on analysis of your work as well as an *individual* ten minute oral defence. We thus expect all students to be able to explain the *entirety* of their assignments, even if made in pairs.

As a final note, any changes to the assignment will be signalled on the exercises and assignments Web site and will also be sent to the student delegates (Pierre Gewelt for engineers and Simon De Baets for computer scientists).

Good luck!