UNIVERSITÉ TOULOUSE III PAUL SABATIER

Labwork 2 – Improving Autocell

These labworks are automatically assessed based on an archive you have to deliver on time on the Moodle webpage. To make the archive, you have to type the command:

```
> make archive
```

This produces a file named `archive.tgz` that you have to deposit. As the compilation labworks are held each week, deposit deadline is set one day before your next session (to perform the assessment). This deadline is not strict but delay will have a negative impact on your mark and you will not benefit from the comments of your teacher.

This labwork aims to set up the procedure to extend our Cellang compiler and to add the support for variable and for arithmetic expressions.

1 The big picture

As exposed in the course, the compiler is divided into the front-end (that parse the source program) and the back-end (that generates the machine code). In turn, the front-end is divided in three phases:

1. the lexical analysis generated by `ocamllex` from file `lexer.mll`,
2. the syntactic analysis generated by `ocamlyacc` from file `parser.mly`,
3. the semantic analysis written in *OCAML* and inserted as actions in `parser.mly`.

Notice that you do not have to call the different tools yourself but just to use the provided `Makefile`:

```
> make
```

1.1 The parser (syntactic analysis)

The main program is the syntactic analyzer. The programming language is described as a grammar. The example below show two rules for an `expression` and for designing a `cell` (found in `parser.mly`):

```
expression:
        cell
                { CELL (0, fst $1, snd $1) }
|       INT
                { CST $1 }
;

cell:
        LBRACKET INT COMMA INT RBRACKET
                { ($2, 4) }
;
```

The rule starts by the non-terminal name, ":", followed by several production separated by "|" and ended by ";". Each production is a sequence of symbols (terminal in uppercase, non-terminal in lowercase) completed by an action between "{...}". The action is in OCAML and *is triggered when the production is accepted during the $LALR(1)$ analysis*. Except for the action, the syntax is very close to what have been presented in the course and hence is processed in the same way.

The terminals (also called *tokens*) are declared inside `ocamlyacc` but are generated by the lexical analyzer that process the source program. Inside `parser.mly`, the declaration of tokens is performed by:

```
%token DIMENSIONS
%token OF
%token ASSIGN
%token COMMA
```

The token identifier must be in uppercase and must meet the usual identifier rules of OCAML. They are turned into an union type by `ocamlyacc`:

```
type token =
  | DIMENSIONS
  | OF
  | ASSIGN
  | COMMA
  | ...
```

To Do Examine the file `parser.mly` to locate the elements presented above.

1.2 The lexer (lexical analysis)

Now, the work of the lexer is to scan the source text in order to recognize the tokens and to return the corresponding value to the syntactic analyzer. You have to notice that there is no straigtht link between the token name and the corresponding word. If `DIMENSIONS` scan the keyword `dimensions`, `ASSIGN` matches the word ":=". Therefore, `ocamlyacc` does not know the actual shape of these tokens.

These tokens inside `ocamllex` are scanned using the following rules (from `lexer.mll`):

```
rule token = parse
          ...
|         "dimensions"      { DIMENSIONS }
|         "of"                      { OF }
|         ":="                      { ASSIGN }
|          ','                              { COMMA }
...
```

Each rule starts with a "|", followed by a *regular expression* (RE) and completed by an OCAML action between "{...}". The role of this action is to return the token corresponding to the scanned word but it may be any OCAML code returning a token.

The token `INT` is specially interesting: it does not match only one word but any word representing an integer value (in decimal). From the point of view of grammar, it is sufficient to know that we get an integer. But at some point (typically in the back-end), the compiler will need the actual value of the integer. Such a value is called a *semantic value* and is declared in the token with the syntax (the type of integer semantic value is `int`):

```
%token<int> INT
```

In turn, `ocamllex` has to compute and pass this value:

```
|         dec       as n                { INT (int_of_string n) }
```

`dec` is a named RE scanning decimal integer. This results in a word string called `n` that is converted to integer and passed as parameter to `INT` token.

`dec` RE is declared by:

```
let digit = ['0'-'9']
let sign = ['+' '-']
let dec = sign? digit+
```

That must be read as:

- `dec` possibly starts with a `sign`,
- then it is composed of non-empty sequence of `digit`,
- a `sign` is one of '+' or '-' characters (single quotes are mandatory),
- a digit is one character between the character '0' and the character '9' (decimal digit).

To Do Examine the file `lexer.mll` to locate the elements presented so far.

Note: the RE used in ocamlyacc can be:

- "c_1c_2..." to scan the sequence of characters C_1, c_2, ...,
- [*chars*] with *chars* a sequence of single character 'c' or range of characters 'c_1'-'c_2',
- E* to repeat E zero or several times,
- E+ to repeat E one or several times,
- E_1 E_2 ... scans the sequence E_1 then E_2 then...,
- E_1 | E_2 scans E_1 or E_2,
- (E) to manage priorities inside RE.
- _ (underscore) represents any character $c \in \Sigma$.

1.3 Extending our compiler

To summarize,

1. The Lexical Analysis get the character for a token, for example "123".
2. This text match the DFA corresponding to `dec`.
3. The corresponding action is invoked with $n =$ "123".
4. This build a token `INT 123` that is passed to the Syntactic Analysis.
5. The syntactic analysis uses this token first to perform a *action*.
6. When enough tokens has been accumulated in the stack, the corresponding action is invoked. For example, `cell` action with `LBRACK INT COMMA INT RBRACKET`.

Hence, to extend our language, we have to:

1. declare missing tokens in `parser.mly`
2. write the added grammar rules in `parser.mly`
3. check if this compiles (`make` command)
4. add the missing – but previously declared – tokens in `lexer.mll`
5. check if everything compiles together
6. test the new rules with a program using them.

For example, to test if the current compiler compiles the AutoCell used in the first labwork session, we have typed:

```
> ./autocc autos/shift.auto
Assembly saved to autos/shift.s
```

But, I get a syntax error (not already supported) with:

```
> ./autocc autos/vars.auto
ERROR:3:1: illegal char 'x'
```

To Do Test the commands above.

2 First steps

In its current state, `autocc` only supports programs of the form:

```
2 dimensions of n..m end
[0, 0] := E
```

With E being a cell or an integer and made of only one assignment. In this exercise, we will allow:

- variables (used or assigned)
- multiple assignments

Variables in Autocell: A variable is defined by its identifier that may be assigned or used:

```
x := [1, 1]
y := x
[0, 0] := y
```

In Autocell, a variable does not need to be declared: it is created the first time it is assigned. Yet, a variable is identified by its name, an identifier starting by a letter and followed any number of letters, digits or '_'.

1 We have to define a token to support the scanning of identifiers, let name it `ID`. Does it need a semantic value? Yes, because its value is important to identify the corresponding variable. What is its type? `string` for sure! So, we add the token definition to `parser.mly` in the token declaration part.

```
%token <string> ID
```

2 Where a variable can be used in our language? Two places: in an expression, or as the target of an assignment. Let start with an expression. Add the following line in the productions of `expressions`:

```
| ID
        { NONE }
```

The action is set to `NONE` because the action has to return a representation of the expression. For now, we just do not care but we have to be sure that we still compile.

Perform the compilation and fix the possible errors:

```
> make
```

3 The work is done at the syntactic analysis level but not for the lexical level. Open the file `lexer.mll` and the scanning of an identifier:

- write the RE for an identifier,
- write the action that returns an `ID`.

Check that all compiles.

4 Test your compiler with the file `autos/var1.auto`, `autos/var2.auto`, `autos/var3.auto` and `autos/var4.auto`[1].

5 Now, we want to add the assignment of a variable to our language. Observe in the `statement` non-terminal how it is done for a `cell`. Notice the difference between a `statement` and an `expression`: a statement can change memory or the execution flow while an expression is in charge of computing a value.

Add a production to implement the assignment of variables. Its action will be `NOP` (representation of the statement that does nothing).

Test your compiler with `autos/varassign.auto`.

6 The next issue with our language is that it only supports one assignment: check the compilation fails with `autos/vars.auto`.

Modify the semantic analyzer (`parser.mly`) in order to have programs with several statements. Test it with `autos/vars.auto`.

Summary To extend the Autocell language handled by the compiler, we have to (a) create new tokens in `parser.mly` and in `lexer.mll`, (b) to write new grammar rules in `parser.mly` (with null action first) and (c) to test them using Autocell source files containing these new structures. We will go on this way in the next exercises.

Debugging If you need it, you can use output functions of OCAML to help you to find bugs: `print_string`, `print_int`, `printf`, etc. But recall the way the $LALR(1)$ parser works to avoid misunderstanding the result of output.

[1] This one should fail.

3 Adding arithmetic expressions

The goal of this exercise is to extend the `expressions` of our Autocell implementation with arithmetic operators:

- addition, subtraction
- parentheses
- multiplication, division, modulo

The following proposes an order for implementing these different syntactic constructions and corresponding `.auto` test files. You can follow or not this order but we believe it may help beginners with `ocamllex` and `ocamlyacc`. Whatever your choice, set the actions of the added production to { `NONE` } for now. We will put more constructive actions in the next labwork.

The work plan is listed below:

1. Add to `expressions` the addition operator in its simple form "$A_1 + A_2$" with A_i being a variable, a cell or an integer (spaces are optional). Test it with `autos/add1.auto`.

2. Now we extend the addition to support the associativity "$A_1 + A_2 + A_3 + ...A_n$" that, in fact, can be viewed as a combination of binary additions: "$(...((A_1 + A_2) + A_3) + ... + A_n)$". Observe that, in Autocell (and in most programming languages), the addition is left-associative. Implement it and test it with `autos/add2.auto`.

   Hint: it is not easy to test if you really implement the left associativity but you can check it using displays in the action of the expressions. Add the following display actions to the expressions:

   - `printf "%d\n" $1` for `INT`,
   - `printf "%s\n" $1` for `ID`,
   - `printf "[%d, %d]\n" (fst $1) (snd $1)` for `ID`,
   - `printf "+\n"` for the addition.

   As the analysis performed by `ocamlyacc` is bottom-up, the output is expressed in reverse-polish notation (RPN). For example, the expression `1 + 2 + 3` will produce the output `1 2 + 3 +`: first we sum 1 and 2, then we sum the result with 3 and this really implements left associativity.

   If you get `1 2 3 + +`, you will sum 2 and 3 and then we 1 add to the result and the produced expression is in fact right-associative.

   Once you get the expected result, you can comment out the `printf` calls.

3. Extend the `expressions` with the subtraction "$A_1 - A_2$". The subtraction is also left-associative and has the same priority than the addition:

   - $A_1 - A_2 - A_3 \iff (A_1 - A_2) - A_3$

- $A_1 + A_2 - A_3 \iff (A_1 + A_2) - A_3$
- $A_1 - A_2 + A_3 \iff (A_1 - A_2) + A_3$

Test it with `autos/addsub.auto`.

4. We can add the parenthesis `expression` "(any expression)". This may be useful to change the associativity effects: "$A_1 - (A_2 + A_3) \iff A_1 - A_2 - A_3$". Test it with `autos/parent.auto`.

5. Now it is time to add the multiplication "$A_1 * A_2$". The multiplication is left-associative and has a priority higher than the addition or the subtraction. This means that $A_1 + A_2 * A_3 \iff A_1 + (A_2 * A_3)$. Test it with `autos/mult.auto`.

6. Add the division "A_1/A_2" and the modulo "$A_1\%A_2$" that both are left-associative and have the same priority as the multiplication. Test it with `autos/divmod.auto`.

7. Finally add the unary "+" and "-". Notice that these operations are not associative and have the highest priority: they directly apply to atomic expressions (cell, variable, integer, ...) following them. Test it with `autos/neg.auto`.