Programming Languages and Computation

Week 3–5: Parsing the While Language

In this small project you will implement a parser for the While programming language. You may use write the parser in any programming language of your choice, however the instructions are most easily followed in a non-pure functional programming language such as OCaml, or an imperative/object-oriented programming language.

Using OCaml (Optional)

OCaml is available on the lab machines, but you can also follow the instructions at https://ocaml.org/install to install it on your own machine. Make sure to set up a complete development environment including the Dune build system.

If you are on the lab machines, you need to execute the following command in the terminal, each session, to setup the OCaml enviroment:

```
> module load ocaml
```

If you installed it on your own machines, the instructions will likely tell you to execute a different command to setup the environment. For ease, one can put these commands in the shell rc script (e.g. ~/.bashrc or ~/.zshrc) so that they are run automatically.

Create a new directory for the parser project and add two files for configuring the build. The first should be called *dune*, and contain a description of the build object:

```
(library
  (name while_parser)
  (wrapped false)
  (modules wlexer wparser))
```

We are setting the parser up as a library initially, because it will be easier to test it with the OCaml toplevel. Later, the library can be turned into an executable.

The second configuration file, which should be called *dune-project*, specifies data about the project as a whole, but all we need is to stipulate an appropriate version of Dune (the build system):

```
(lang dune 3.7)
```

Finally, create two empty source code files *wlexer.ml* and *wparser.ml*. Their names should match the names of the modules in the *dune* file. Once you have some OCaml code in these files, it will automatically appear in the modules *Wlexer* and *Wparser* respectively.

To check that your installation works, insert some boring code into *wlexer.ml*, such as:

```
let square x = x * x
```

Build the library, using, from the terminal:

```
> dune build
```

and enter the toplevel:

```
$ dune utop
```

and test the function. The command *open* brings all of the values and functions defined in a module into scope.

```
utop # open Wlexer;;
utop # square 6;;
- : int = 36
```

The toplevel allows for multi-line input by default, so the double semicolon must be used explicitly to terminate the expression that you wish to execute.

When editing code, a good choice is VSCode with the OCaml Platform extension. For those of you who remember your Haskell well, a useful comparison is: http://blog.ezyang.com/2010/10/ocaml-for-haskellers/.

Task 1: Lexer

1. Recall that a *token* is a pair consisting of a terminal symbol and (possibly) an associated lexeme (string). Create a data type to represent tokens of each of the following kinds:

Terminal Symbol	if	then	else	while	do	skip	id	+	−	∗	true	false
Store Associated Lexeme?							✓					

Terminal Symbol	<	=	&&	‖	!	num	(	)	{	}	;	←	$
Store Associated Lexeme?						✓							

An empty cell in the second row indicates that the associated lexeme can be discarded.

For example, in a functional programming language, you will want to create a new algebraic datatype with one constructor for each row of the table, each of which is nullary (has no arguments), except for the constructors corresponding to id and num which will each take a string argument. In an imperative language, you can simulate this by first creating an enumeration with one constant for each possible terminal symbol, then the token datatype can be a record or class which has a field that stores a value from this enumeration (that says what kind of token an object of the class represents) along with the associated lexeme (which will be null/empty in most cases).

2. Set up the state of the lexer. Create a variable, say *input*, to hold the current state of the input (a string/list of characters). Implement the following API:

peek() returns the next character of the input without consuming it

eat(c) given character *c*, if *c* is the next character of the input, consume it (remove it from *input*), otherwise fail

is_more() returns true if *input* is non-empty and false otherwise.

init(s) given a string *s*, sets *input* to contain exactly *s*.

If you are implementing this in a pure functional programming language like Haskell, then the first three of these functions will need to depend on the state of the input, i.e. rather than having a mutable variable *input*, each must be passed the current state as an argument. In this case, there is no need to implement an *init* function.

$$
\begin{array}{rcl}
\mathit{Prog} & \longrightarrow & \mathit{Stmt}\ \mathit{Stmts}\ \$ \\
\\
\mathit{Stmt} & \longrightarrow & \mathsf{if}\ \mathit{BExp}\ \mathsf{then}\ \mathit{Stmt}\ \mathsf{else}\ \mathit{Stmt} \\
& | & \mathsf{while}\ \mathit{BExp}\ \mathsf{do}\ \mathit{Stmt} \\
& | & \mathsf{skip} \\
& | & \mathsf{id} \leftarrow \mathit{AExp} \\
& | & \{\mathit{Stmt}\ \mathit{Stmts}\} \\
\\
\mathit{Stmts} & \longrightarrow & ;\ \mathit{Stmt}\ \mathit{Stmts} \\
& | & \epsilon \\
\\
\mathit{BExp} & \longrightarrow & \mathit{BFac}\ \mathit{BExps} \\
\\
\mathit{BExps} & \longrightarrow & ||\ \mathit{BFac}\ \mathit{BExps} \\
& | & \epsilon \\
\\
\mathit{BFac} & \longrightarrow & \mathit{BNeg}\ \mathit{BFacs} \\
\\
\mathit{BFacs} & \longrightarrow & \&\&\ \mathit{BNeg}\ \mathit{BFacs} \\
& | & \epsilon \\
\\
\mathit{BNeg} & \longrightarrow & !\ \mathit{BNeg} \\
& | & \mathit{BRel} \\
\\
\mathit{BRel} & \longrightarrow & \mathit{AExp}\ \mathit{BRels} \\
\\
\mathit{BRels} & \longrightarrow & < \mathit{AExp} \\
& | & = \mathit{AExp} \\
& | & \epsilon \\
\\
\mathit{AExp} & \longrightarrow & \mathit{AFac}\ \mathit{AExps} \\
\\
\mathit{AExps} & \longrightarrow & + \mathit{AFac}\ \mathit{AExps} \\
& | & - \mathit{AFac}\ \mathit{AExps} \\
& | & \epsilon \\
\\
\mathit{AFac} & \longrightarrow & \mathit{Atom}\ \mathit{AFacs} \\
\\
\mathit{AFacs} & \longrightarrow & * \mathit{Atom}\ \mathit{AFacs} \\
& | & \epsilon \\
\\
\mathit{Atom} & \longrightarrow & \mathsf{num} \\
& | & \mathsf{true} \\
& | & \mathsf{false} \\
& | & \mathsf{id} \\
& | & (\ \mathit{BExp}\ )
\end{array}
$$

Figure 1: LL(1) Grammar for While Programs

3. Define a function *lex_number()* which takes no arguments and returns a token. Its purpose is to consume the largest prefix of digits from the input and return the corresponding num token. This can be implemented by the following algorithm:

```
lex_number() : token
  lexeme := ""
  while peek() is a digit do
    c := peek()
    eat(c)
    add c onto the end of lexeme
  return a new num token with lexeme stored
```

Note, if you are implementing this in a pure functional programming language like Haskell, then you will need to thread through the state of the input (i.e. the variable *input* above) because mutable variables are awkward. This can either be done explicitly, by giving this variable to the function as an argument and returning it along with the token, or implicitly, using a state monad (this is what happens in parser combinators).

4. Define a function *lex_kw_or_id()* which takes no arguments and returns a token. Its purpose is to read the largest prefix of identifier characters (letters, numbers and the prime character ') from the input, check if it corresponds to one of the keywords of the language (if, then, else, while, do, true, false, skip) and returns the corresponding token. The algorithm can be sketched as follows:

```
lex_kw_or_id() : returns a token
  lexeme := ""
  while peek() is a letter or number or prime do
    c := peek()
    eat(c)
    add c onto the end of lexeme
  if lexeme is a keyword (if, then, else, while...) then
    return a new token of the correct kind
  else
    return a new id token with lexeme stored
```

If you are implementing this in a pure functional programming language, you will need to adopt the same strategy as above to simulate mutable state.

5. Define a function *lex(s)* which takes a string as input and returns a sequence of tokens. Its purpose is to lex the input string *s*, that is, to transform it into the corresponding sequence of tokens. The algorithm can be sketched as follows:

```
lex(s) : returns a sequence of tokens
  init(s)
  tokens := empty sequence
  while is_more() do
    c := peek()
    - if c is '=' :
        eat(=)
        add new = token onto the end of tokens
    - if c is '!' :
        eat(!)
        add new ! token onto the end of tokens
```

```
    - if c is '+' :
        eat(+)
        add new + token onto the end of tokens
    ...
    - if c is '&' :
        eat(&)
        eat(&)
        add new && token onto the end of tokens
    - if c is '<' :
        eat(<)
        if peek() = '-' then
          eat(-)
          add new <- token onto the end of tokens
        else
          add new < token onto the end of tokens
    ...
    - if c is a digit :
        add lex_number() onto the end of tokens
    - if c is a lowercase letter :
        add lex_kw_or_id onto the end of tokens
    - if c is whitespace : eat(c)
  add new $ (end-of-input) token onto the end of tokens
  return tokens
```

6. Test that your lexer works correctly. For example:

   - lex("myVar <- x * (foo + bar)") should return the sequence of tokens corresponding to:

     id, ←, id, ∗, (, id, +, id,), $

   - lex("if x = 0 then x <- x * y else skip") should return the sequence of tokens corresponding to:

     if, id, =, num, then, id, ←, id, ∗, id, else, skip, $

   - lex("while foo < 32 do { x <- x + 1; y <- y - 1 }") should return the sequence of tokens corresponding to:

     while, id, <, num, do, {, id, ←, id, +, num, ;, id, ←, y, −, 1, $

Task 2: Recogniser

In this task you will implement a recogniser for the grammar of Figure 1, that is: a parser that returns only a yes/no answer rather than a representation of the parsed program (e.g. an abstract syntax tree). A good approach is to create a new module and import the lex function and token datatype from the previous task.

7. Construct the parse table for the grammar in Figure 1. This will be laborious to do by hand, so please do make use of one of the many online LL(1) parse table generators, such as:

   https://jsmachines.sourceforge.net/machines/ll1.html

8. Set up the state of the recogniser, a variable *tokens* which stores a sequence of tokens. Implement the following API:

   peek() returns the next token from *tokens* without consuming it.

   eat(tk) given a token *tk*, if *tk* is the next token in *tokens*, then consume it (remove it from *tokens*), otherwise fail.

   init(tks) given a token sequence *tks*, sets *tokens* to contain exactly *tks*.

9. Implement one parsing function for each nonterminal symbol of the grammar, according to the parse table. Suppose the parse table T contains entries for a non-terminal X in $T[a_1]$, $T[a_2]$, ..., $T[a_n]$ (i.e. in columns corresponding to some terminal symbols a_1, ..., a_n). Then the function corresponding to X should have the following shape:

   ```
   parse_X () : returns void
     c := peek()
     - if c is a_1 then proceed according to rule T[a_1]
     - if c is a_2 then proceed according to rule T[a_2]
     ...
     - if c is a_n then proceed according to rule T[a_n]
     - otherwise fail
   ```

   Here proceeding "according to rule" means carrying out an action corresponding to each symbol in the sentential form of the given rule in the given order. For a non-terminal, this means calling the corresponding parsing function, and for a terminal this means consuming it from the input. For example, implementing a rule $X \longrightarrow aYbX$ can be achieved by:

   ```
   eat(a); parse_Y(); eat(b); parse_X()
   ```

   A rule whose right hand side is empty can be implemented by simply returning (void).

10. Implement a top-level *recognise(s)* function, which takes a string as input and returns true if the string is a While program and false otherwise. Test your function to check that it works correctly:

    - recognise("while true do skip") should return true.
    - recognise("if x <= 3 then x <- x - 1 else y <- y + 1") should return false.
    - recognise("while y + 3 < 2 do y <- y + 1; x <- 0") should return true.
    - recognise("y <- y + 1; x <- 0;") should return false.
    - recognise("whiley <- (iff + sskip) * doo - thenn") should return true.

Operator	Associativity
*	Left
+, -	Left
!	-
&&	Right
||	Right

Table 1: Associativity of While language operators, higher precedence at top.

Task 3: Parser

In this task you will modify the recogniser to produce an abstract syntax tree representation of a recognised program. You can find the associativity and precedence of operators in Table 1.

11. Create a data type *exp* for the abstract syntax trees of expressions, and another data type *stmt* for the abstract syntax trees of statements. These should support the following nodes, listed with the types of their immediate children:

Exp Node Label	Child 1?	Child 2?
True		
False		
Less	exp	exp
Eq	exp	exp
Not	exp	
And	exp	exp
Or	exp	exp
Var	string	
Num	integer	
Plus	exp	exp
Minus	exp	exp
Times	exp	exp

Stmt Node Label	Child 1?	Child 2?	Child 3?
Skip			
Assn	string	exp	
Seq	stmt	stmt	
Cond	exp	stmt	stmt
While	exp	stmt	

If you are writing in a functional programming language, then it is very straightforward to represent these as algebraic data types. In imperative/object languages, you will need to describe a tree structure explicitly using pointers/references. In this case, each child will be field.

12. The parsing functions of the recogniser currently return void.

    (a) Modify the parsing functions so that:

    - The parsing functions corresponding to non-terminals *BExp*, *BExps*, *BFac*, *BFacs*, *BNeg*, *BRel*, *BRels*, *AExp*, *AExps*, *AFac*, *AFacs*, *Atom* each return an abstract syntax tree of type *exp*.
    - The parsing functions corresponding to non-terminals *Stmts*, *Stmt* and *Prog*, each return an abstract syntax tree of type *stmt*.
    - The parsing functions whose name ends in 's', *BExps*, *BFacs*, *BRels*, *AExps*, *AFacs* and *Stmts* each take an AST (of type *exp* or *stmt* as appropriate) as input.

    The idea is that, in addition to consuming tokens and calling other parsing functions, each parsing function also constructs an abstract syntax tree corresponding to the portion of the string that it is responsible for parsing. For example, we can sketch out the behaviour

of the atom parsing function as follows:

```
atom () : returns exp
  c := peek()
  - if c is a "true" token:
      eat the true token
      return new True AST node
  - if c is a "false" token:
      eat the false token
      return new False AST node
  - if c is a "num" token with associated string s:
      eat the num token
      n := string_to_int(s)
      return new Num AST node with child n
  - if c is an "id" token with associated string s:
      eat the id token
      return new Var AST node with child s
  - if c is a "(" token:
      eat the ( token
      e := bexp()
      eat the ) token
      return e
```

For example, in the case when the next token is the opening parenthesis, we consume this token, then call the parsing function corresponding to *BExp*, and (once the implementation is complete) this returns an AST describing the expression inside the parentheses. Then we consume the trailing right parenthesis and return the AST (parentheses are not part of the abstract syntax tree - the tree structure itself describes nesting).
For the nonterminals whose name ends in 's', e.g. *BRels*, *AExps* etc, creating a AST is a bit tricky because the strings that they parse do not correspond to complete expressions, or complete statements. For example, *BFacs* generates strings such as: "&& true", && false && 1 < 2 and the empty string. None of these is a complete expression, the first two are missing the first operand of && and the last is empty. Thus, we need to supply the missing piece as an input to the parsing functions corresponding to these nonterminals and then combine it with any ASTs generated by the parsing function itself.