Compilers (Sheet #1)

Marius Gavrilescu

1. The NFA is essentially a trie containing the keywords. Each node is a state, and the characters on the edges are exactly the symbols that cause a transition. Instead of having values in nodes we mark nodes that represent keywords as accepting states. The root of the trie is the starting state. This produces a NFA that accepts all keywords and is compact.

   Nondeterminism in this NFA comes from multiple keywords sharing a common prefix. If we read an "e" we don't know if it stands for "else" or "end", so we will have two outgoing transitions.

   To additionally accept other identifiers, have an epsilon transition from the root to a new state X, and transitions from X to X for any symbol that is a letter of the Latin alphabet. Anything finishing in this state (and no accepting state) is a non-keyword identifier. (we can skip this part if we've restricted our alphabet before to only letters of the Latin alphabet, and if we ensured the token is nonempty).

   To convert this to a DFA we can use the standard construction: each state of the DFA is a set of states of the NFA.

2. The regex in standard leaning toothpick style can be written as `/\/\*.*?\*\//` (which looks even more beautiful in the LaTeXsource). A more sensible example (where we change the delimiters) would be `m,/\*.*?\*/,`.

   Here we must escape the stars to avoid confusing them with repetition operators. We also want to use non-greedy repetition (i.e. `x*?` instead of `x*`) to avoid parsing the whole string as a single comment: `/* comment1 */ code; /* comment2 */`.

   The advantage of using a regex for comments is that it simplifies our parser. However this makes our lexer harder to understand and it requires a lexer that can cope with non-greedy repetition (to avoid the error mentioned above).

3. With a powerful regex engine we can build a regex that matches any keyword (by taking the individual regexes and joining them with the disjunction operator) and we can wrap the entire regex in a negative lookahead assertion. For the example in exercise 1.1, our regex will look like:

   ```
   /^(?!(?:while|do|if|then|else|end)$)/
   ```

   Here we also needed to anchor the negative lookahead to the very beginning of the string (i.e. the entire string is none of these keywords), and to anchor the body of the lookahead to the very end of the string (to avoid matching words like "done").

   Without such a regex engine, we could write a program that takes a list of keywords and produces a regex that matches everything else. For example with keywords "if" and "else" we can build the regex:

   ```
   /[^i]|[^e]|i[^f]|e[^l]|el[^s]|els[^e]|if.|else./
   ```

   If we anchor it to the beginning of the string, it can be read as: "does not start with i, or does not start with e, or starts with i but not if, or starts with e but not el, or starts with el but not els, or starts with els but not else, or starts with if and any other character, or starts with else and any other character". The last two rules match all words that have a keyword as a prefix, the other rules match all words that do not start with a keyword.

4. This convention allows us to wrap any block of code in (* and *) to comment it even if it already contains comments.

   In ocamllex this looks like:

   ```
   rule token =
     parse
       ...
       | "(*"  { comment 0 lexbuf; token lexbuf }
       ...
   and comment level =
     parse
       | "(*" { comment (level + 1) lexbuf }
       | "*)" {
         if level = 0 then
           ()
         else
           comment (level - 1) lexbuf
         }
       | _    { comment lexbuf }
       | eof  { () }
   ```

5. Suppose "1" is a valid expr and "nop" is a basic-stmt. Then the following string can be interpreted in two ways:

   `if 1 then if 1 then nop else nop`

   Specifically, does the else belong to the first if or to the second one?

   A shift-reduce parser will eventually get to the state

   `if expr then if expr then stmt` with remaining input `else nop`.

   Here we can either reduce to `if expr then stmt` or shift.

   The unambiguous grammmar is:

```
stmt -> basic-stmt
      | if expr then stmt
      | if expr then then-stmt else stmt

then-stmt -> basic-stmt
           | if expr then then-stmt
```

6. Instead of:

```
  | IF expr THEN stmts END              { IfStmt ($2, $4, Skip) }
  | IF expr THEN stmts ELSE stmts END   { IfStmt ($2, $4, $6) }
```

   we write:

```
  | IF expr THEN stmts elsifs END
      { IfStmt ($2, $4, $5) }
```

   where `elsifs` is defined as

```
elsif :
  | ELSIF expr THEN stmts { ($2, $4) }
  | ELSE stmts            { (Constant 1, $2) }

elsifs :
  | elsif { IfStmt (fst $1, snd $1, Skip) }
  | elsif elsifs { IfStmt(fst $1, snd $1, $2) }
```

7. There are 199 tokens (100 identifiers and 99 commas). With a left-recursive grammar, we can start by shifting, reducing, and then we can repeatedly do two shifts followed by a reduce. This uses at most three elements on the stack.

   With a right-recursive grammar, we need to shift everything onto the stack before we can start reducing, which requires 199 stack elements.

   For cons the right-recursive grammar is

```
list -> nil
list -> elt ":" list
```

   and the left-recursive grammar is

```
list -> nil
list -> elts ":" nil

elts -> elt
elts -> elts ":" elt
```

   The right-recursive grammar is significantly easier to understand.

8.
```
expr -> ident
expr -> binop expr expr
expr -> monop expr

binop -> '+' | '-' | '*' | '/'

monop -> '~'
```

Suppose "-" was both unary and binary. Then the (infix) expressions `((1 - (-2)) - 3)` and `(-1) - (2 - 3)` would both become in Polish notation `- - 1 - 2 3`.

```
type binop = PLUS | MINUS | MULT | DIV
type monop = MINUS
type token =
  | NUMBER of int
  | VAR of string
  | BINOP of binop
  | MONOP of monop
type tree =
  | Number of int
  | Variable of string
  | Bin of binop * tree * tree
  | Mon of monop * tree

%token<monop> MONOP
%token<binop> BINOP
%token<int> NUMBER
%token<string> VAR

%type<tree> expr
%start      expr

%%

expr :
  | NUMBER n              { Number $2 }
  | VAR v                 { Variable $2 }
  | BINOP binop expr expr { Binop ($2, $3, $4) }
  | MONOP monop expr      { Monop ($2, $3) }
```

9. Ambiguity arises because the parser does not know after a YEAR has been read whether to reduce it to a "years" or to keep going (because a COMMA and another year could follow it!).

   Fixed grammar gets rid of years:

```
%%

file:
 | record { [$1] }
 | record file { $1 :: $2 };

record:
    YEAR COMMA ACTOR  { ([$1], $3) }
  | YEAR COMMA record { ($1 :: fst $3, snd $3) }
```

   If years are "Y", actors are "A", and commas are ",", the regex is $((Y,) + A)*$. This is a sequence of records, each record being a nonempty sequence of YEAR COMMA followed by a single ACTOR.