INFO-F-403 – Project

Vadim Baele – Quentin Stievenart

March 17, 2013

The purpose of this project is to develop a compiler for a simplified version of Perl that compiles to ARM assembler, in order to be able to execute compiled Perl code on Android devices. The simplified version of Perl to be implemented is described in the project statement.

This report describes the implementation of the compiler, written in OCaml. Note that most of the documentation of the code is written in the interface files (`.mli`).

1 Lexer

1.1 Lexical Units

The lexical units of this simplified version of Perl are given here, with their regular expression and the corresponding token used in the implementation. Some of the tokens are grouped into categories for a matter of clarity.

- **Symbols**:
  - `{` : `LBRACE`
  - `}` : `RBRACE`
  - `(` : `LPAR`
  - `)` : `RPAR`
  - `;` : `SEMICOLON`
  - `,` : `COMMA`
  - `&` : `CALL_MARK`
  - `=` : `ASSIGN`
- **Operators**:
  - `+` : `PLUS`
  - `-` : `MINUS`
  - `*` : `TIMES`
  - `/` : `DIVIDE`

- `||` : LAZY_OR
- `&&` : LAZY_AND
- `==` : EQUALS
- `!=` : DIFFERENT
- `>` : GREATER
- `<` : LOWER
- `>=` : GREATER_EQUALS
- `<=` : LOWER_EQUALS
- `eq` : STRING_EQUALS
- `ne` : STRING_DIFFERENT
- `gt` : STRING_GREATER
- `ge` : STRING_GREATER_EQUALS
- `lt` : STRING_LOWER
- `le` : STRING_LOWER_EQUALS
- `not` : NOT_WORD
- `!` : NOT

- **Keywords**:
  - `if` : IF
  - `unless` : UNLESS
  - `else` : ELSE
  - `elsif` : ELSEIF
  - `return` : RETURN
  - `sub` : SUB
- **Integer**: `[0-9]+` : INTEGER
- **String**: `("[^"]"|'[^']')` : STRING
- **Identifier**: `[a-zA-Z][a-zA-Z0-9]*` : IDENTIFIER
- **Variable**: `$[a-zA-Z0-9]+` : VARIABLE

1.2 Deterministic Finite Automaton

The DFA is divided in multiple figures for a matter of clarity. All the following figures have the same start state, and some refer to the special state `to identifier`, which *redirects* the DFA into the corresponding figure. The accepting states are represented with dashed lines.

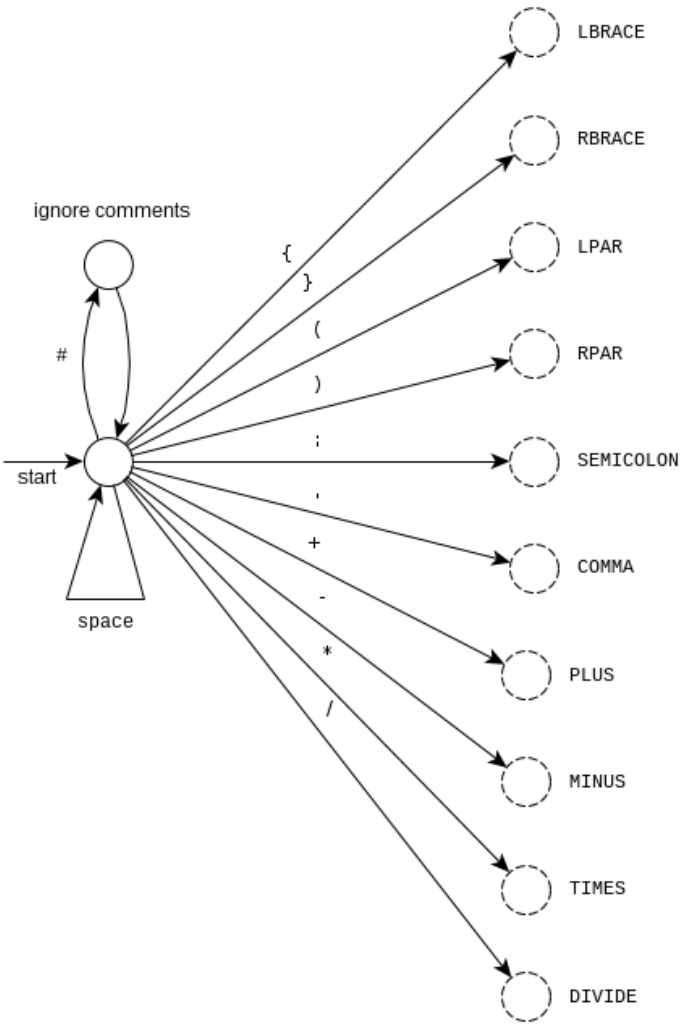

Figure 1: DFA for single character symbols, spaces and comments

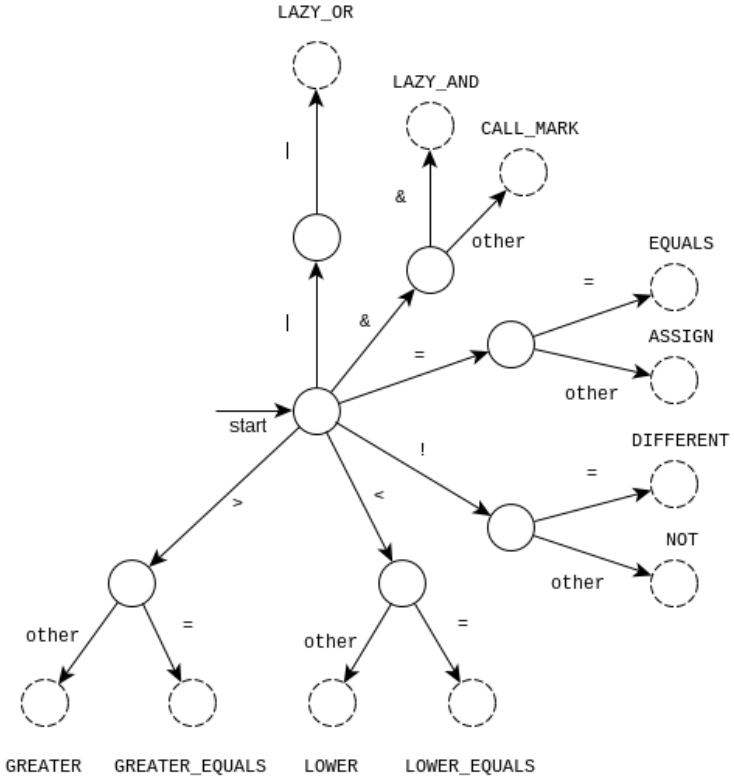

Figure 2: DFA for multi-character symbols

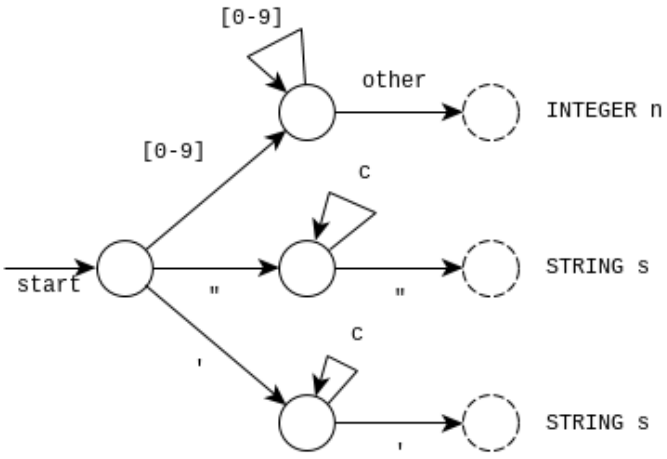

Figure 3: DFA for integer and strings

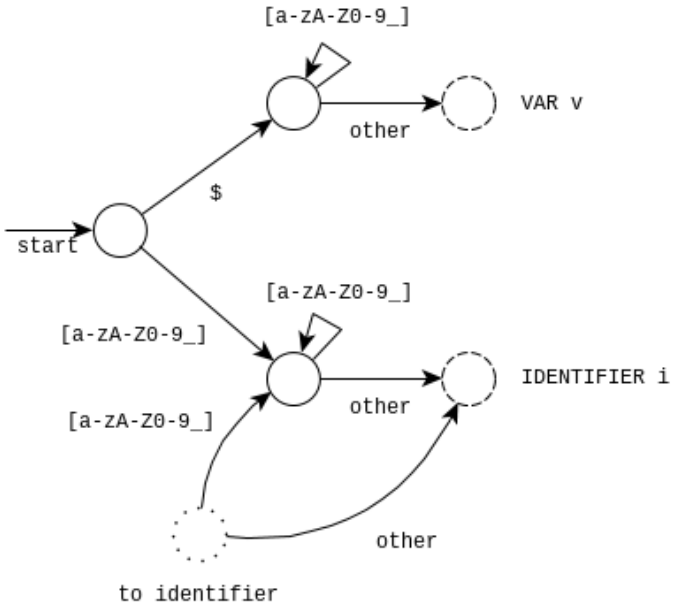

Figure 4: DFA for variables and identifiers

1.3 Implementation

The different tokens are defined as a type (in `token.ml`). The lexer (or scanner) is implemented in `hand_lexer.ml` and its implementation is a simple translation from the DFA to OCaml code, which looks at the next character from the input stream and goes to the corresponding state. The main function of the lexer has the following type signature:

```
val lex : in_channel -> (string, token) either Stream.t
```

Which means that the lexer reads its input from an input channel (`stdin` or a file), and it outputs a stream of `(string, token) either`, meaning that the stream can contain either a token, or a string (which describes a lexing error).

2 Parser

2.1 Modified grammar

The grammar used is a modification of the first grammar given (in the project statement). This modified grammar is given below.

$$
\begin{array}{lll}
\langle \textit{program} \rangle & \rightarrow & \langle \textit{function list} \rangle \ \langle \textit{instr list} \rangle \\
& | & \langle \textit{function list} \rangle \\
& | & \langle \textit{instr list} \rangle
\end{array}
$$

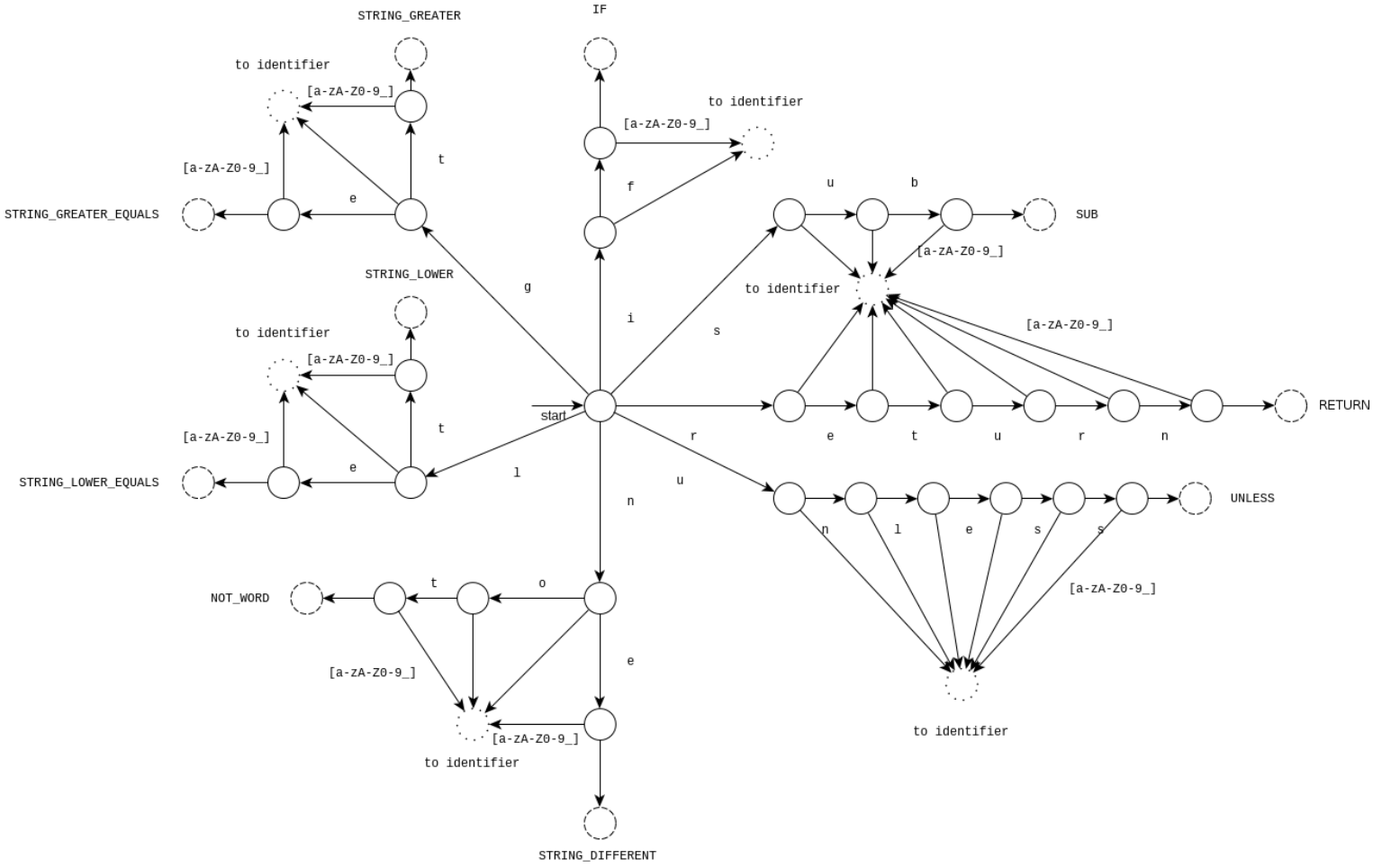

Figure 5: DFA for keywords

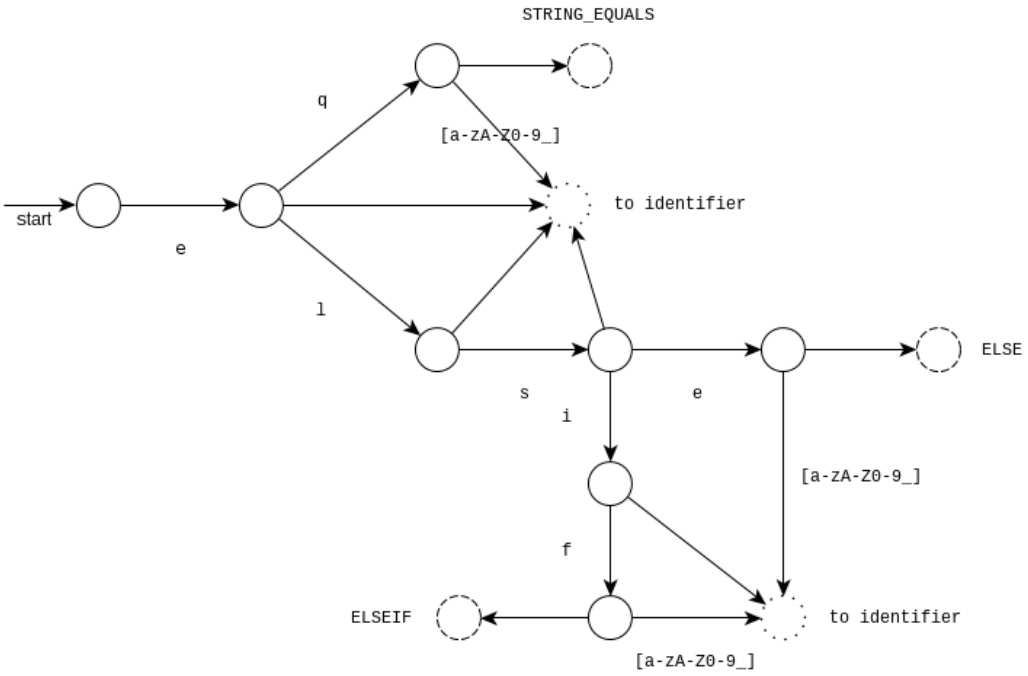

Figure 6: DFA for keywords starting with `e`

$\langle function\ list \rangle \rightarrow \langle function\ list \rangle\ \langle function \rangle$
$\mid\ \langle function \rangle$

$\langle function \rangle \rightarrow$ '`sub`' identifier $\langle function\ args \rangle$ '`{`' $\langle instr\ list \rangle$ '`}`'

$\langle function\ args \rangle \rightarrow$ '`(`' $\langle arg\ list \rangle$ '`)`'
$\mid\ \epsilon$

$\langle arg\ list \rangle \rightarrow \langle arg\ list \rangle$ '`,`' var
$\mid$ var
$\mid\ \epsilon$

$\langle instr\ list \rangle \rightarrow \langle instr\ list \rangle\ \langle instr \rangle$ '`;`'
$\mid$ '`{`' $\langle instr\ list \rangle$ '`}`'
$\mid\ \langle instr \rangle$ '`;`'

⟨*funcall*⟩ → '`&`' identifier ⟨*funcall args*⟩
| identifier ⟨*funcall args*⟩

⟨*funcall args*⟩ → '`(`' ⟨*args call list*⟩ '`)`'

⟨*args call list*⟩ → ⟨*args call list*⟩ '`,`' ⟨*instr*⟩
| ⟨*instr*⟩
| ϵ

⟨*instr*⟩ → ⟨*cond*⟩
| ⟨*expr*⟩
| ⟨*expr*⟩ '`=`' ⟨*expr*⟩
| ⟨*expr*⟩ '`if`' ⟨*expr*⟩
| ⟨*expr*⟩ '`unless`' ⟨*expr*⟩
| '`return`' ⟨*expr*⟩

⟨*cond*⟩ → '`if`' ⟨*expr*⟩ '`{`' ⟨*instr list*⟩ '`}`' ⟨*cond end*⟩
| '`unless`' ⟨*expr*⟩ '`{`' ⟨*instr list*⟩ '`}`' ⟨*cond end*⟩

⟨*cond end*⟩ → '`else`' '`{`' ⟨*instr list*⟩ '`}`'
| '`elsif`' ⟨*expr*⟩ '`{`' ⟨*instr list*⟩ '`}`' ⟨*cond end*⟩
| ϵ

⟨*simple expr*⟩ → ⟨*funcall*⟩
| var
| integer
| string

⟨*expr*⟩ → '`not`' ⟨*expr*⟩
| ⟨*expr-or*⟩

⟨*expr-or*⟩ → ⟨*expr-or*⟩ '`||`' ⟨*expr-and*⟩
| ⟨*expr-and*⟩

⟨*expr-and*⟩ → ⟨*expr-and*⟩ '`&&`' ⟨*expr-eq*⟩
| ⟨*expr-eq*⟩

⟨*expr-eq*⟩ → ⟨*comp*⟩ '`==`' ⟨*comp*⟩
| ⟨*comp*⟩ '`!=`' ⟨*comp*⟩
| ⟨*comp*⟩ '`eq`' ⟨*comp*⟩
| ⟨*comp*⟩ '`ne`' ⟨*comp*⟩
| ⟨*comp*⟩

⟨*comp*⟩ → ⟨*calc*⟩ '`>`' ⟨*calc*⟩
| ⟨*calc*⟩ '`<`' ⟨*calc*⟩
| ⟨*calc*⟩ '`>=`' ⟨*calc*⟩
| ⟨*calc*⟩ '`<=`' ⟨*calc*⟩
| ⟨*calc*⟩ '`lt`' ⟨*calc*⟩
| ⟨*calc*⟩ '`gt`' ⟨*calc*⟩
| ⟨*calc*⟩ '`le`' ⟨*calc*⟩
| ⟨*calc*⟩ '`ge`' ⟨*calc*⟩
| ⟨*calc*⟩

⟨*calc*⟩ → ⟨*calc*⟩ '`+`' ⟨*term*⟩
| ⟨*calc*⟩ '`-`' ⟨*term*⟩
| ⟨*calc*⟩ '`.`' ⟨*term*⟩
| ⟨*term*⟩

⟨*term*⟩ → ⟨*term*⟩ '`*`' ⟨*factor*⟩
| ⟨*term*⟩ '`/`' ⟨*factor*⟩
| ⟨*factor*⟩

⟨*factor*⟩ → ⟨*simple expr*⟩
| '`!`' ⟨*factor*⟩
| '`+`' ⟨*factor*⟩
| '`-`' ⟨*factor*⟩
| '`(`' ⟨*expr*⟩ '`)`'

The differences between this grammar and the grammar given in the project's statement are the following:

- `<program>`: a program should consist first of the function definition and then of the program instructions, else it would cause conflicts in the parsing table

- `<instr list>`: every instruction has to end with a semicolon, else the first and follow sets of every non-terminals would be really big and the chances of having a conflict would grow. This implies a minor difference of syntax with Perl: conditions have to end with a semicolon, for example: `if ... { ... };`
- `<funcall>`: functions can now be called without being prefixed by a `&`
- `<funcall args>`: function calls now have mandatory parentheses, to avoid conflicts (those conflicts should however not arise, due to the mandatory semicolon, but this change was made before having the mandatory semicolon)
- `<instr>` and `<cond>`: post-condition forms are now in the non-terminal `<instr>`, to avoid conflicts (since now `<cond>` cannot start with an `<expr>`, which conflicted with the fact that `<instr>` can also start with an `<expr>`)
- `<instr>`: an assignation is now an instruction instead of an expression, which makes more sense
- `<cond end>`: the `else` branch was incorrect
- `<expr>`: the old `<expr>` non-terminal has been splitted in multiple non-terminals in order to respect the operators' precedences

2.2 LL(1) grammar

The grammar has been converted to a LL(1) grammar by eliminating left-recursion, and the result is given below. A non-terminal symbol `<S>` has been added to handle the `EOF` character, needed to properly compute the follow sets and thus implement the grammar.

$$\langle S \rangle \rightarrow \langle program \rangle \texttt{ EOF} \quad (1)$$

$$\langle program \rangle \rightarrow \langle function\ list \rangle\ \langle program' \rangle \quad (2)$$
$$\mid \langle instr\ list \rangle \quad (3)$$

$$\langle program' \rangle \rightarrow \langle instr\ list \rangle \quad (4)$$
$$\mid \epsilon \quad (5)$$

$$\langle function\ list \rangle \rightarrow \langle function \rangle\ \langle function\ list' \rangle \quad (6)$$

$$\langle function\ list' \rangle \rightarrow \langle function \rangle\ \langle function\ list' \rangle \quad (7)$$
$$\mid \epsilon \quad (8)$$

⟨*function*⟩	→ ‘`sub`’ identifier ⟨*function args*⟩ ‘`{`’ ⟨*instr list*⟩ ‘`}`’	(9)
⟨*function args*⟩	→ ‘`(`’ ⟨*arg list*⟩ ‘`)`’	(10)
	| ϵ	(11)
⟨*arg list*⟩	→ var ⟨*arg list'*⟩	(12)
	| ϵ	(13)
⟨*arg list'*⟩	→ ‘`,`’ var ⟨*arg list'*⟩	(14)
	| ϵ	(15)
⟨*instr list*⟩	→ ⟨*instr*⟩ ‘`;`’ ⟨*instr list'*⟩	(16)
	| ‘`{`’ ⟨*instr list*⟩ ‘`}`’	(17)
⟨*instr list'*⟩	→ ⟨*instr*⟩ ‘`;`’ ⟨*instr list'*⟩	(18)
	| ϵ	(19)
⟨*funcall*⟩	→ ‘`&`’ identifier ⟨*funcall args*⟩	(20)
	| identifier ⟨*funcall args*⟩	(21)
⟨*funcall args*⟩	→ ‘`(`’ ⟨*args call list*⟩ ‘`)`’	(22)
⟨*args call list*⟩	→ ⟨*instr*⟩ ⟨*args call list'*⟩	(23)
	| ϵ	(24)
⟨*args call list'*⟩	→ ‘`,`’ ⟨*instr*⟩ ⟨*args call list'*⟩	(25)
	| ϵ	(26)
⟨*instr*⟩	→ ⟨*cond*⟩	(27)
	| ⟨*expr*⟩ ⟨*instr'*⟩	(28)
	| ‘`return`’ ⟨*expr*⟩	(29)
⟨*instr'*⟩	→ ‘`=`’ ⟨*expr*⟩	(30)
	| ‘`if`’ ⟨*expr*⟩	(31)
	| ‘`unless`’ ⟨*expr*⟩	(32)
	| ϵ	(33)

⟨*cond*⟩	→ ‘`if`’ ⟨*expr*⟩ ‘`{`’ ⟨*instr list*⟩ ‘`}`’ ⟨*cond end*⟩	(34)
	| ‘`unless`’ ⟨*expr*⟩ ‘`{`’ ⟨*instr list*⟩ ‘`}`’ ⟨*cond end*⟩	(35)
⟨*cond end*⟩	→ ‘`else`’ ‘`{`’ ⟨*instr list*⟩ ‘`}`’	(36)
	| ‘`elsif`’ ⟨*expr*⟩ ‘`{`’ ⟨*instr list*⟩ ‘`}`’ ⟨*cond end*⟩	(37)
	| ϵ	(38)
⟨*simple expr*⟩	→ ⟨*funcall*⟩	(39)
	| var	(40)
	| integer	(41)
	| string	(42)
⟨*expr*⟩	→ ‘`not`’ ⟨*expr*⟩	(43)
	| ⟨*expr-or*⟩	(44)
⟨*expr-or*⟩	→ ⟨*expr-and*⟩ ⟨*expr-or'*⟩	(45)
⟨*expr-or'*⟩	→ ‘`||`’ ⟨*expr-and*⟩ ⟨*expr-or'*⟩	(46)
	| ϵ	(47)
⟨*expr-and*⟩	→ ⟨*expr-eq*⟩ ⟨*expr-and'*⟩	(48)
⟨*expr-and'*⟩	→ ‘`&&`’ ⟨*expr-eq*⟩ ⟨*expr-and'*⟩	(49)
	| ϵ	(50)
⟨*expr-eq*⟩	→ ⟨*comp*⟩ ⟨*expr-eq'*⟩	(51)
⟨*expr-eq'*⟩	→ ‘`==`’ ⟨*comp*⟩	(52)
	| ‘`!=`’ ⟨*comp*⟩	(53)
	| ‘`eq`’ ⟨*comp*⟩	(54)
	| ‘`ne`’ ⟨*comp*⟩	(55)
	| ϵ	(56)
⟨*comp*⟩	→ ⟨*calc*⟩ ⟨*comp'*⟩	(57)

$\langle comp' \rangle \rightarrow$ '>' $\langle calc \rangle$ (58)
| '<' $\langle calc \rangle$ (59)
| '>=' $\langle calc \rangle$ (60)
| '<=' $\langle calc \rangle$ (61)
| 'lt' $\langle calc \rangle$ (62)
| 'gt' $\langle calc \rangle$ (63)
| 'le' $\langle calc \rangle$ (64)
| 'ge' $\langle calc \rangle$ (65)
| ϵ (66)

$\langle calc \rangle \rightarrow \langle term \rangle \langle calc' \rangle$ (67)

$\langle calc' \rangle \rightarrow$ '+' $\langle term \rangle \langle calc' \rangle$ (68)
| '-' $\langle term \rangle \langle calc' \rangle$ (69)
| '.' $\langle term \rangle \langle calc' \rangle$ (70)
| ϵ (71)

$\langle term \rangle \rightarrow \langle factor \rangle \langle term' \rangle$ (72)

$\langle term' \rangle \rightarrow$ '*' $\langle factor \rangle \langle term' \rangle$ (73)
| '/' $\langle factor \rangle \langle term' \rangle$ (74)
| ϵ (75)

$\langle factor \rangle \rightarrow \langle simple\ expr \rangle$ (76)
| '!' $\langle factor \rangle$ (77)
| '+' $\langle factor \rangle$ (78)
| '-' $\langle factor \rangle$ (79)
| '(' $\langle expr \rangle$ ')' (80)

2.3 First and follows

The *first* and *follow* sets of the non-terminal symbols are given below.

$\langle First(factor) \rangle$ = {'(', '!', '+', '-', '&', var, integer, string, identifier}

$\langle First(term') \rangle$ = {'*', '/'}

⟨*First(term)*⟩ = {‘(’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(calc')*⟩ = {‘`+`’, ‘`-`’, ‘`.`’}

⟨*First(calc)*⟩ = {‘(’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(comp')*⟩ = {‘`>`’, ‘`<`’, ‘`>=`’, ‘`<=`’, ‘`lt`’, ‘`gt`’, ‘`le`’, ‘`ge`’}

⟨*First(comp)*⟩ = {‘(’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(expr-eq')*⟩ = {‘`==`’, ‘`!=`’, ‘`eq`’, ‘`ne`’}

⟨*First(expr-eq)*⟩ = {‘(’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(expr-and')*⟩ = {‘`&&`’}

⟨*First(expr-and)*⟩ = {‘(’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(expr-or')*⟩ = {‘`||`’}

⟨*First(expr-or)*⟩ = {‘(’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(expr)*⟩ = {‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(simple expr)*⟩ = {var, integer, string, ‘`&`’, identifier}

⟨*First(cond end)*⟩ = {‘`else`’, ‘`elsif`’}

⟨*First(cond)*⟩ = {‘`if`’, ‘`unless`’}

⟨*First(instr)*⟩ = {‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(instr')*⟩ = {‘=’, ‘`if`’, ‘`unless`’}

⟨*First(args call list')*⟩ = {‘`,`’}

⟨*First(args call list)*⟩ = {‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(funcall args)*⟩ = {‘`(`’}

⟨*First(funcall)*⟩ = {‘`&`’, identifier}

⟨*First(instr list')*⟩ = {‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(instr list)*⟩ = {‘`{`’, ‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(arg list')*⟩ = {‘`,`’}

⟨*First(arg list)*⟩ = {var}

⟨*First(function args)*⟩ = {‘`(`’}

⟨*First(function)*⟩ = {‘`sub`’}

⟨*First(function list')*⟩ = {‘`sub`’}

⟨*First(function list)*⟩ = {‘`sub`’}

⟨*First(program)*⟩ = {‘`sub`’, ‘`{`’, ‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(program')*⟩ = {‘`{`’, ‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*First(S)*⟩ = {‘`sub`’, ‘`{`’, ‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier}

⟨*Follow(program)*⟩ = {`EOF`}

⟨*Follow(program')*⟩ = {`EOF`}

⟨*Follow(function list)*⟩ = {‘`{`’, ‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier, `EOF`}

⟨*Follow(function)*⟩ = {‘`sub`’, ‘`{`’, ‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier, `EOF`}

⟨*Follow(function list')*⟩ = {‘`{`’, ‘`return`’, ‘`if`’, ‘`unless`’, ‘`not`’, ‘`(`’, ‘`!`’, ‘`+`’, ‘`-`’, ‘`&`’, var, integer, string, identifier, `EOF`}

⟨*Follow(function args)*⟩ = {‘`{`’}

⟨*Follow(instr list)*⟩ = {‘`}`’, `EOF`}

⟨*Follow(arg list)*⟩ = {‘`)`’}

⟨*Follow(arg list')*⟩ = {‘`)`’}

⟨*Follow(instr)*⟩ = {‘`;`’, ‘`,`’, ‘`)`’}

⟨*Follow(instr')*⟩ = {‘`;`’, ‘`,`’, ‘`)`’}

⟨*Follow(instr list')*⟩ = {‘`}`’, `EOF`}

⟨*Follow(funcall args)*⟩ = {‘`*`’, ‘`/`’, ‘`+`’, ‘`-`’, ‘`.`’, ‘`>`’, ‘`<`’, ‘`<=`’, ‘`>=`’, ‘`lt`’, ‘`gt`’, ‘`le`’, ‘`ge`’, ‘`==`’, ‘`!=`’, ‘`eq`’, ‘`ne`’, ‘`&&`’, ‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(args call list)*⟩ = {‘`)`’}

⟨*Follow(args call list')*⟩ = {‘`)`’}

⟨*Follow(cond)*⟩ = {‘`;`’, ‘`,`’, ‘`)`’}

⟨*Follow(expr)*⟩ = {‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`=`’, ‘`if`’, ‘`unless`’}

⟨*Follow(cond end)*⟩ = {‘`;`’, ‘`,`’, ‘`)`’}

⟨*Follow(funcall)*⟩ = {‘`*`’, ‘`/`’, ‘`+`’, ‘`-`’, ‘`.`’, ‘`>`’, ‘`<`’, ‘`<=`’, ‘`>=`’, ‘`lt`’, ‘`gt`’, ‘`le`’, ‘`ge`’, ‘`==`’, ‘`!=`’, ‘`eq`’, ‘`ne`’, ‘`&&`’, ‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(simple expr)*⟩ = {‘`*`’, ‘`/`’, ‘`+`’, ‘`-`’, ‘`.`’, ‘`>`’, ‘`<`’, ‘`<=`’, ‘`>=`’, ‘`lt`’, ‘`gt`’, ‘`le`’, ‘`ge`’, ‘`==`’, ‘`!=`’, ‘`eq`’, ‘`ne`’, ‘`&&`’, ‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(expr-or)*⟩ = {‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(expr-and)*⟩ = {‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(expr-or')*⟩ = {‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(expr-eq)*⟩ = {‘`&&`’, ‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(expr-and')*⟩ = {‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(comp)*⟩ = {‘`==`’, ‘`!=`’, ‘`eq`’, ‘`ne`’, ‘`&&`’, ‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(expr-eq')*⟩ = {‘`&&`’, ‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(calc)*⟩ = {‘`>`’, ‘`<`’, ‘`<=`’, ‘`>=`’, ‘`lt`’, ‘`gt`’, ‘`le`’, ‘`ge`’, ‘`==`’, ‘`!=`’, ‘`eq`’, ‘`ne`’, ‘`&&`’, ‘`||`’, ‘`=`’, ‘`{`’, ‘`)`’, ‘`;`’, ‘`,`’, ‘`if`’, ‘`unless`’}

⟨*Follow(comp')*⟩ = {'==', '!=', 'eq', 'ne', '&&', '||', '=', '{', ')', ';', ',', 'if', 'unless'}

⟨*Follow(term)*⟩ = {'+', '-', '.', '>', '<', '<=', '>=', 'lt', 'gt', 'le', 'ge', '==', '!=', 'eq', 'ne', '&&', '||', '=', '{', ')', ';', ',', 'if', 'unless'}

⟨*Follow(calc')*⟩ = {'>', '<', '<=', '>=', 'lt', 'gt', 'le', 'ge', '==', '!=', 'eq', 'ne', '&&', '||', '=', '{', ')', ';', ',', 'if', 'unless'}

⟨*Follow(factor)*⟩ = {'*', '/', '+', '-', '.', '>', '<', '<=', '>=', 'lt', 'gt', 'le', 'ge', '==', '!=', 'eq', 'ne', '&&', '||', '=', '{', ')', ';', ',', 'if', 'unless'}

⟨*Follow(term')*⟩ = {'+', '-', '.', '>', '<', '<=', '>=', 'lt', 'gt', 'le', 'ge', '==', '!=', 'eq', 'ne', '&&', '||', '=', '{', ')', ';', ',', 'if', 'unless'}

2.4 Parsing Table

The parsing table for the LL(1) grammar is given in tables 1, 2 and 3. The numbers refers to the rules of the grammar given in section 2.2.

Table 1: Parsing table (1)

	var	integer	string	identifier	sub	return	&	{	}	(	)	;	,
<S>		1	1	1	1	1	1	1		1			
<program>	3	3	3	3	2	3	3	3		3			
<program'>	4	4	4	4		4	4	4		4			
<function list>					6								
<function list'>	8	8	8	8	7	8	8	8		8			
<function>					9								
<function args>								11		10			
<arg list>	12										13		
<arg list'>											15		14
<instr list>	16	16	16	16		16	16	17		16			
<instr list'>	18	18	18	18		18	18		19	18			
<funcall>				21			20						
<funcall args>										22			
<args call list>	23	23	23	23		23	23			23	24		
<args call list'>											26		25
<instr>	28	28	28	28		29	28			28			
<instr'>											33	33	33
<cond>													
<cond end>											38	38	38
<simple expr>	40	41	42	39			39						
<expr>	44	44	44	44			44			44			

Continued on next page

Table 1: Parsing table (1)

	var	integer	string	identifier	sub	return	&	{	}	(	)	;	,
<expr-or>	45	45	45	45			45			45			
<expr-or'>								47			47	47	47
<expr-and>	48	48	48	48			48			48			
<expr-and'>								50			50	50	50
<expr-eq>	51	51	51	51			51			51			
<expr-eq'>								56			56	56	56
<comp>	57	57	57	57			57			57			
<comp'>								66			66	66	66
<calc>	67	67	67	67			67			67			
<calc'>								71			71	71	71
<term>	72	72	72	72			72			72			
<term'>								75			75	75	75
<factor>	76	76	76	76			76			80			

Table 2: Parsing table (2)

	if	unless	else	elsif	not	!	+	-	*	/	=	.	or	&&
<S>	1	1			1	1	1							
<program>	3	3			3	3	3	3						
<program'>	4	4			4	4	4	4						
<function list>														
<function list'>	8	8			8	8	8	8						
<function>														
<function args>														
<arg list>														
<arg list'>														
<instr list>	16	16			16	16	16	16						
<instr list'>	18	18			18	18	18	18						
<funcall>														
<funcall args>														
<args call list>	23	23			23	23	23	23						
<args call list'>														
<instr>	27	27			28	28	28	28						
<instr'>	31	32									30			
<cond>	34	35												
<cond end>			36	37										
<simple expr>														
<expr>					43	44	44	44						
<expr-or>						45	45	45						
<expr-or'>	47	47									47		46	
<expr-and>						48	48	48						
<expr-and'>	50	50									50		50	49
<expr-eq>						51	51	51						
<expr-eq'>	56	56									56		56	56
<comp>						57	57	57						
<comp'>	66	66									66		66	66
<calc>						67	67	67						

Continued on next page

Table 2: Parsing table (2)

	if	unless	else	elsif	not	!	+	-	*	/	=	.	or	&&
<calc'>	71	71					68	69			71	70	71	71
<term>						72	72	72						
<term'>	75	75					75	75	73	74	75	75	75	75
<factor>						77	78	79						

Table 3: Parsing table (3)

	==	!=	>	<	>=	<=	eq	ne	gt	lt	ge	le	EOF
<S>													
<program>													
<program'>													5
<function list>													
<function list'>													8
<function>													
<function args>													
<arg list>													
<arg list'>													
<instr list>													
<instr list'>													19
<funcall>													
<funcall args>													
<args call list>													
<args call list'>													
<instr>													
<instr'>													
<cond>													
<cond end>													
<simple expr>													
<expr>													
<expr-or>													
<expr-or'>													
<expr-and>													
<expr-and'>													
<expr-eq>													
<expr-eq'>	52	53					54	55					
<comp>													
<comp'>	66	66	58	59	60	61	66	66	63	62	65	64	
<calc>													
<calc'>	71	71	71	71	71	71	71	71	71	71	71	71	
<term>													
<term'>	75	75	75	75	75	75	75	75	75	75	75	75	
<factor>													

2.5 Grammar decoration

The grammar has been decorated to produce an abstract syntax tree. The decoration of the grammar is given below (the numbers corresponds to the rules of the grammar given

in section 2.2), where `[]` corresponds to the empty list, and `::` is the *cons* operation (which creates a list from a head and a tail). `inh` is an *inherited* attribute, while `node` is a *synthesized* attribute. When a non-terminal has the same name on the left-hand side and right-hand side of a rule, those in the right-hand side are numbered left-to-right, while the one on the left-hand side remains the same.

The resulting abstract syntax tree can then be converted to assembly code (this process is described in section 3). The advantage of using an abstract syntax tree is that both the decoration of the grammar and the code generation are simplified. The decoration of the grammar already does some simplifications, which avoid having to take lots of different cases into account during code generation (eg. a `unless x` condition is translated in the same way as would be an `if (not x)` condition).

1. `<S>.node = <program>.node`
2. `<program>.node = (<function list>.node, <program'>.node)`
3. `<program>.node = ([], <instr list>.node)`
4. `<program'>.node = <instr list>.node`
5. `<program'>.node = []`
6. `<function list>.node = <function>.node :: <function list'>.node`
7. `<function list'>.node = <function>.node :: <function list'>.node`
8. `<function list'>.node = []`
9. `<function>.node = Fundef(identifier.entry, <function args>.node, <instr list>.node`
10. `<function args>.node = <arg list>.node`
11. `<function args>.node = []`
12. `<arg list>.node = var.entry :: <arg list'>.node`
13. `<arg list>.node = []`
14. `<arg list'>.node = var.entry :: <arg list'>.node`
15. `<arg list'>.node = []`
16. `<instr list>.node = <instr>.node :: <instr list'>.node`
17. `<instr list>.node = <instr list>.node`
18. `<instr list'>.node = <instr>.node :: <instr list'>.node`
19. `<instr list>.node = []`
20. `<funcall>.node = Funcall(identifier.entry, <funcall args>.node)`
21. `<funcall>.node = Funcall(identifier.entry, <funcall args>.node)`
22. `<funcall args>.node = <args call list>.node`
23. `<args call list>.node = <instr>.node :: <args call list'>.node`
24. `<args call list>.node = []`
25. `<args call list'>.node = <instr>.node :: <args call list'>.node`
26. `<args call list'>.node = []`
27. `<instr>.node = <cond>.node`
28. `<instr'>.inh = <expr.node> <instr>.node = <instr'>.node`
29. `<instr>.node = Return(<expr>.node)`
30. `<instr'>.node = Assign (<instr'>.inh.variable, <expr>.node)`

31. <instr'>.node = Cond(<expr>.node, [<instr'>.inh], CondEnd)
32. <instr'>.node = Cond(UnOp(Not, <expr>.node), [<instr'>.inh], CondEnd)
33. <instr'>.node = <instr'>.inh
34. <cond>.node = Cond(<expr>.node, <instr list>.node, <cond end>.node
35. <cond>.node = Cond(UnOp(Not, <expr>.node), <instr list>.node, <cond end>.node
36. <cond end>.node = Cond(Value True, <instr list>.node, CondEnd)
37. <cond end>.node = Cond(<expr>.node, <instr list>.node, <cond end>.node)
38. <cond end>.node = CondEnd
39. <simple expr>.node = <funcall>.node
40. <simple expr>.node = Variable var.entry
41. <simple expr>.node = Value integer.entry
42. <simple expr>.node = Value string.entry
43. <expr>.node = UnOp(Not, <expr>.node)
44. <expr>.node = <expr-or>.node
45. <expr-or'>.inh = <expr-and>.node
    <expr-or>.node = <expr-or'>.node
46. <expr-or'> $_1$ .inh = Or(<expr-or'>.inh <expr-and>.node)
    <expr-or'>.node = <expr-or'> $_1$ .node
47. <expr-or'>.node = <expr-or'>.inh
48. <expr-and'>.inh = <expr-eq>.node
    <expr-and>.node = <expr-and'>.node
49. <expr-and'> $_1$ .inh = And(<expr-and'>.inh, <expr-eq>.node>)
    <expr-and'>.node = <expr-and'> $_1$ .node
50. <expr-and'>.node = <expr-and'>.inh
51. <expr-eq'>.inh = <comp>.node
    <expr-eq>.node= = <expr-eq'>.node
52. <expr-eq'>.node = BinOp(Equals, <expr-eq'>.inh, <expr-eq'> $_1$ .node)
53. <expr-eq'>.node = BinOp(Different, <expr-eq'>.inh, <expr-eq'> $_1$ .node)
54. <expr-eq'>.node = BinOp(StrEquals, <expr-eq'>.inh, <expr-eq'> $_1$ .node)
55. <expr-eq'>.node = BinOp(StrDifferent, <expr-eq'>.inh, <expr-eq'> $_1$ .node)
56. <expr-eq'>.node = <expr-eq'>.inh
57. <comp'>.inh = <calc>.node
    <comp>.node = <comp'>.node
58. <comp'> $_1$ .inh = BinOp(Greater, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node
59. <comp'> $_1$ .inh = BinOp(Lower, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node
60. <comp'> $_1$ .inh = BinOp(GreaterEquals, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node
61. <comp'> $_1$ .inh = BinOp(LowerEquals, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node
62. <comp'> $_1$ .inh = BinOp(StrLower, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node

63. <comp'> $_1$ .inh = BinOp(StrGreater, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node
64. <comp'> $_1$ .inh = BinOp(StrLowerEquals, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node
65. <comp'> $_1$ .inh = BinOp(StrGreaterEquals, <comp'>.inh, <calc>.node)
    <comp'>.node = <comp'> $_1$ .node
66. <comp'>.node = <comp'>.inh
67. <calc'>.inh = <term>.node
    <calc>.node = <calc'>.node
68. <calc'> $_1$ .inh = BinOp(Plus, <calc'>.inh, <term>.node)
    <calc'>.node = <calc'> $_1$ .node
69. <calc'> $_1$ .inh = BinOp(Minus, <calc'>.inh, <term>.node)
    <calc'>.node = <calc'> $_1$ .node
70. <calc'> $_1$ .inh = BinOp(Concat, <calc'>.inh, <term>.node)
    <calc'>.node = <calc'> $_1$ .node
71. <calc'>.node = <calc'>.inh
72. <term'>.inh = <factor>.node
    <term>.node = <term'>.node
73. <term'> $_1$ .inh = BinOp(Times, <term'>.inh, <factor>)
    <term'>.node = <term'> $_1$ .node
74. <term'> $_1$ .inh = BinOp(Divide, <term'>.inh, <factor>)
    <term'>.node = <term'> $_1$ .node
75. <term'>.node = <term'>.inh
76. <factor>.node = <simple expr>.node
77. <factor>.node = UnOp(Not, <factor> $_1$ .node)
78. <factor>.node = UnOp(UnaryPlus, <factor> $_1$ .node)
79. <factor>.node = UnOp(UnaryMinus, <factor> $_1$ .node)
80. <factor>.node = <expr>.node

2.6 Implementation

The implementation of the parser simply consisted of transcripting the combination of the rules of the action table with the decorated grammar into OCaml code. The parser is implemented in `parser.ml`, and each non-terminal is represented by a function that parses the non-terminal from a stream.

The type signature of the parser is:

```
val parse : token Stream.t -> (expr list * expr list) * Symtable.t
```

Thus, the parser reads its input from a stream of token (which is the stream emitted by the parser, with the errors removed after being displayed to the user), and returns three things: a list of function definitions, a list of instructions, and a symbol table. The symbol table contains the list of global variables and the arity of the functions, and will be used by the code generator.

3 Code generation

The code generation step consists of converting the abstract syntax tree to ARM assembly code.

The code generator is implemented in `codegen.ml`, and its type signature is:

```
val gen : out_channel -> expr list * expr list -> Symtable.t -> unit
```

Thus, it takes an output channel, a list of function definitions, a list of instructions, and an already filled symbol table, writes the generated code to the output channel and returns nothing.

The following conventions are used by the code generator:

- every computed value is stored into the `r4` register
- global variables are stored with their name prefixed by `perl_global_`, and functions are stored with their name prefixed by `perl_fun_`, to allow having the same name for a variable and a function

3.1 Values

The different values are represented as follows:

- a string is represented by its pointer, which is a multiple of 4, since we store each string word-aligned. Thus, the last two bits of a string pointer always are `00`
- integers are stored left-shifted by one bit, and added to 1. So, integers are 31 bits, with the last bit always being set to 1, which allows to distinguish an integer from a string at run time
- `undef` is stored as 2, so that it is distinguishable from integers and from strings
- `true` is stored as integer value 1, and `false` as `undef`

During the generation of the code, all the encountered strings are stored in a string table. After the code has been generated, it is prepended by the declaration of all those strings. This allow to easily manage strings, and to only have one instance of each string, even if it appears multiple times in the source code.

3.2 Variables and assignments

Global variables are gathered during parsing, and have their value set to `undef` by default. Their value is declared at the beginning of the assembly file, and they are refered in the code as a certain offset from the `Lglobals` label.

Local variables (function arguments) are simply stored on the stack, and are referred as their address on the stack.

Assignments simply consists of storing a new value into the address of the destination variable.

3.3 Binary operations

A binary operation `BinOp(op, e1, e2)` is computed as follows:

- Compute `e1`, and push it on the stack
- Compute `e2`, and store it in the `r1` register
- Restore the computed value of `e1` into the `r0` register
- Call the native function corresponding to `op`
- Move the return value of the function (which is in `r0`) into `r4`

3.4 Unary operations

There are three possible unary operations:

- **Unary plus**: it doesn't do anything, so the code generated is the same as the code without this unary operation
- **Unary minus**: the code for `-x` is generated as the code for `0-x`, using the binary operation code generation
- **Not**: the code for the expression is first computed, then the result is passed to the function `perl_not`, which does the negation of its argument.

3.5 Conditions

Each condition branch uses two labels: an *alternative* label, and an *end* label. First, the code for the condition is generated. If the result of the condition is false, we jump to the alternative label. Then, the code for the *consequent* is generated (and is thus executed if the condition is true), followed by a jump to the end label, the alternative label, the code of the alternative, and finally of the end label.

The special operators `&&` and `||` are generated as conditions, since the last result of a condition remains in the register `r4`:

- `e1 && e2` is equivalent to `if (e1) { e2; } else { false; };`
- `e1 || e2` is equivalent to `if (e1) { true; } else { e2; };`

3.6 Function definitions

A function definition is generated as follows:

- a *return* label is generated and stored in the code generator state
- the names of the function arguments are also stored in the code generator state, in order to distinguish local and global variables

- the body of the function is generated in a separate buffer, in order to know by how much bytes the stack should grow (this information is returned by each code generation function)
- the function is declared and stores the `fp` and `lr` registers on the stack
- the stack pointer is decremented by the space needed by the body, plus the space needed to store the function arguments
- the first function arguments (up to the first four) are stored onto the stack
- the code for the body of the function is merged with the current code
- the value `undef` is stored in `r0`
- the return label is printed
- the stack is restored as it was before, and the function terminates, returning the value stored in `r0` (either `undef`, or something else if the function used a `return` instruction)

3.7 Function calls

Two types of functions can be called: native functions, and user-defined functions. Native functions handle their arguments in a special way (eg. `print` can take any number of arguments), while user-defined functions all behave in the same way.

The native functions acts as follows:

- `print` concatenates all its arguments, by transforming a call from `print(e1,e2,e3...)` to `print(e1.e2.e3)`
- `substr` passes `undef` as the last argument if it is called with only two arguments. The native function will thus compute the correct value of the last argument when its value is `undef`
- In all the other cases, the number of arguments have to be checked, but the function call is generated exactly as a normal function call

Natives functions are implemented in C in the file `perl.c`, which should be linked with the assembler file produced by the compiler, in order to have a working executable.

For user-defined functions, first there is a check to test whether the function is called with the correct number of arguments (if this in not the case, the code generator reports an error). Then:

- the arguments are evaluated right-to-left, and pushed on the stack one by one
- the first four arguments (or less, if there is less than four arguments) are popped from the stack and stored in the registers `r0` to `r3`
- the function is called (`bl` instruction)
- once the function has returned, its return value (which is in `r0`) is copied into `r4`

3.8 Returns

A `return` instruction simply consists of computing the return value, and then jumping to the current *return* label. If there is no such label (which means that the code generated is not inside a function body) an error is displayed, because return instructions outside function definitions are not allowed.

4 Miscellaneous

The architecture of the program is such that it is easy to extend it with another lexer, parser, or code generator. For example a second lexer, that uses Ocamllex, is already implemented and can be used using the `-l ocamllex` flag.

Also, a Perl evaluator has been implemented, and can be called (instead of the code generator) with the `-e` flag. This allowed us to have a working program without needing a code generator, and thus to refine the grammar and the abstract syntax tree early in the project. It could be used to automate tests: if the output of the evaluated program and the execution of a compiled program is the same, then the chances that the generated assembly code is correct are high.

The compiler has been tested on multiple examples (in the directory `tests/`), and produces the expected result for all of them.

The command `make` can be used to compile the compiler (it should work with any modern version of OCaml installed, as long as it supplies `ocamlbuild`). To compile a Perl file to an assembler file, use the following command:

```
ocamlrun -b ./main.byte -i tests/lib.pl -o foo.s
```

And to compile the assembler file to an executable and run it on an android device (connected phone or virtual machine):

```
make lib && make obj && make try
```