CS3470 Lab Session 3

To explore intermediate code generation using the parser we look at `miniloop` (slightly simplified below).

```
TREE
PARSER(program)
USES("ml_aux.h")
TREE
OUTPUT_FILE("minilpSimple.mvm")

SYMBOL_TABLE(mini 101 31
     symbol_compare_string
     symbol_hash_string
     symbol_print_string
     [* char* id; *]
    )

check_declared ::= [* if (symbol_lookup_key(mini, &dst, NULL) == NULL)
   { text_message(TEXT_ERROR, "Undeclared variable '%s'\n", dst); }
           *].

program   ::=  [* emit_open(rdp_sourcefilename, rdp_outputfilename); *]
               { [var_dec | statement] ';'}
               [* emit_close(); *].

var_dec ::= 'int' ID:dst
              [* emitf(" \n DATA\n%s: WORD 1\n\n CODE\n",dst); *]
              ['=' e0:left [* emit("CPY", "", dst, left, NULL); *] ]
              [* symbol_insert_key(mini, &dst, sizeof(char*), sizeof(mini_data));
              *].

statement ::= ID:dst check_declared
              '=' e0:left [* emit("CPY", "", dst, left, NULL); *] |

              [* integer label = new_label(); *]
              [* emitf("__IF_%lu:\n", label); *]
              'if' e0:left
              [* emitf(" BEQ  %s,__ELSE_%lu\n",left,label); *]
              'then' statement
              [* emitf(" BRA  __FI_%lu\n__ELSE_%lu:\n", label, label); *]
              'else' statement
              [* emitf("__FI_%lu:\n", label); *] |

              [* integer label = new_label(); *]
              [* emitf("__DO_%lu:\n", label); *]
              'while' e0:left
              [* emitf(" BEQ  %s,__OD_%lu\n",left,label); *]
              'do' statement
              [* emitf(" BRA  __DO_%lu__OD_%lu:\n", label, label); *] |
```

```
             'print' '(' String:left [* emit_print('S', left); *] [',']
                         [ e0:left [* emit_print('I', left); *] ] ')' |

             'begin' statement {';' statement } 'end'.

e0:char* ::= [* char* dst; *] e1:left [ [* dst = new_temporary(); *]
               ('>' e1:right [* emit("GT ", ">", dst, left, right); *]  |
                '<' e1:right [* emit("LT ", "<", dst, left, right); *]  |
                '>=' e1:right [* emit("GE ", ">=", dst, left, right); *]|
                '<=' e1:right [* emit("LE ", ">=", dst, left, right); *]|
                '==' e1:right [* emit("EQ ", "==", dst, left, right); *]|
                '!=' e1:right [* emit("NE ", "!=", dst, left, right); *]
               ) [* left = dst; *]
             ] [* result = left; *].

e1:char* ::= [* char* dst; *] e2:left { [* dst = new_temporary(); *]
               ( '+' e2:right [* emit("ADD", "+", dst, left, right); *] |
                 '-' e2:right [* emit("SUB", "-", dst, left, right); *]
               )
               [* left = dst; *]
             } [* result = left; *].

e2:char* ::= [* char* dst; *] e3:left { [* dst = new_temporary(); *]
               ( '*' e3:right [* emit("MUL", "*", dst, left, right); *] |
                 '/' e3:right [* emit("DIV", "/", dst, left, right); *]
               )
               [* left = dst; *]
             } [* result = left; *].

e3:char* ::= [* int negate = 1; char* dst;*]

             {'-' [* negate *= -1; *]} e4:result (* negate *)
             [* if (negate==-1) {dst = new_temporary();
                emit("SUB", "-", dst, "0", result); result = dst; } *].


e4:char* ::= ID:dst check_declared [* result = dst; *] |
             INTEGER:val [* result = (char*) mem_malloc(12);
                            sprintf(result, "#%lu", val); *] |
           '(' e1:result ')'.

comment ::= COMMENT_NEST('(*' '*)').
String:char*  ::= STRING_ESC('"' '\\'):result.
```

To understand this we can begin begin by making a copy of this file, call it say my_loop.bnf, and strip out all the sematic actions so that you can see what the underlying grammar is.

```
program   ::=  { [var_dec | statement] ';'} .

var_dec ::= 'int' ( ID ['=' e0:left ] )@',' .

statement ::= ID '=' e0 |
              'if' e0 'then' statement [ 'else' statement ] |
              'while' e0 'do' statement |
              'print' '(' String [','] [ e0 ] ')' |
              'begin' statement {';' statement } 'end'.

e0 ::= e1 [ ('>' e1 | '<' e1 | '>=' e1 | '<=' e1 | '==' e1 | '!=' e1 ) ] .

e1 ::= e2 { ( '+' e2 | '-' e2 ) } .

e2 ::= e3 { ( '*' e3 | '/' e3 ) } .

e3 ::= { '-' } e4 .

e4 ::= e5 [ '**' e4 ] .

e5  ::= ID | INTEGER | '(' e1 ')'.

comment ::= COMMENT_NEST('(*' '*)').
String  ::= STRING_ESC('"' '\\').
```

Run a web browser and go to the departmental web pages. Select **Research**, then **Centre for Software Language Engineering**, then **Lanugage engineering and generalised parsing**, then **Tools**. This contains the **rdp** web site. Now select **The RDP LL(1) parser generator**. Open up the language development case study manual. The MVM instructions are on page 67.

The miniloop translator takes statements written in the mini language and generates MVM assembley code.

```
/* mini code */
int a;
a = 3+4;

/* corresponding assembler */
/* equivalent three address is a comment in the assembler code */
a: WORD 1

CODE
ADD  t0, #3, #4         ; t0 := #3 + #4
CPY  a, t0              ;  a := t0
```

We can run miniloop the miniloop compiler to generate MVM assembler.

```
cim-ts-node-01$ cd CS3470/rdp
cim-ts-node-01$ make miniloop
```

```
cim-ts-node-01$ make mvmasm
cim-ts-node-01$ make mvmsim
cim-ts-node-01$ cp /CS/courses/CS3470/mini1.m .
```

The file `mini1.m` contains the following lines:

```
int a;
a = 3+4;
int b;
b = 2*a +1;
if b == 15 then print ("b equals 15") else b=a;
```

Use miniloop to generate an MVM assembley version of this file as follows:

```
cim-ts-node-01$ ./miniloop -omini1.mvm mini1
cim-ts-node-01$ emacs mini1.mvm&
```

rdp creates a compiler `miniloop` from the attributed grammar `miniloop.bnf`. Miniloop is described beginning on page 67 of the manual, and the grammar is on page 78. Look at page 78.
We implement an `if` construct in three address code using labels.

```
    ifn B goto L0
    statement1
    goto L1
L0: statement2
L1:
```

Miniloop outputs MVM assembler with the corresponding three address code as a comment.

```
if 1 then fred=4 else john=2

  __IF_0:
         BEQ #1, __ELSE_0     ; ifn 1 go to __ELSE_0
         CPY fred, #4         ; fred := 4
         BRA __FI_0           ; go to __FI_0
__ELSE_0:
         CPY john, #2         ; john := 2
  __FI_0:
```

The semantic actions which cause the MVM assembler to be written to the output *.m file are the `emit()` functions.

```
cim-ts-node-01$ emacs ml_aux.h&
cim-ts-node-01$ emacs ml_aux.c&
```

```
TREE
PARSER(program)
USES("ml_aux.h")
OUTPUT_FILE("miniloop.mvm")
```

```
statement ::= ID:dst check_declared
              '=' e0:left [* emit("CPY", "", dst, left, NULL); *] |

              [* integer label = new_label(); *]
              [* emitf("__IF_%lu:\n", label); *]
              'if' e0:left
              [* emitf(" BEQ  %s,__ELSE_%lu\t;ifn %s go to __ELSE_%lu \n",
                           left,label,left, label); *]
              'then' statement
              [* emitf(" BRA  __FI_%lu\t;go to __FI_%lu\n__ELSE_%lu:\n",
                             label, label, label); *]
              [ 'else' statement ]
              [* emitf("__FI_%lu:\n", label); *] |
```

We can generate 'machine' code from the assembler and then run it through the simulator.

```
cim-ts-node-01$ ./mvmasm -omini1.sim mini1
cim-ts-node-01$ more mini1.sim
cim-ts-node-01$ ./mvmsim mini1.sim
```

You can run miniloop on other test files

```
cim-ts-node-01$ more testloop.m
cim-ts-node-01$ ./miniloop -otestlp.mvm testloop.m
cim-ts-node-01$ ./mvmasm -otestlp.sim testlp.mvm
cim-ts-node-01$ ./mvmsim testlp.sim
```

To use **rdp** directly you need to include the standard library files which are in the directory `/usr/local/rdp`.

```
cim-ts-node-01$ cp miniloop.bnf minilp2.bnf
cim-ts-node-01$ ./rdp -F -ominilp2 minilp2.bnf
cim-ts-node-01$ gcc -Irdp_supp/ -P -w -c minilp2.c
cim-ts-node-01$ gcc -o minilp2 minilp2.o ml_aux.o arg.o symbol.o graph.o
memalloc.o scan.o scanner.o set.o textio.o
```

The object files must all be included without line breaks, and clearly this is quite a lot of typing, which is why we have been using a makefile. You can avoid typing the above commands each time by adding them as a new target to the makefile.

```
cim-ts-node-01$ emacs makefile&
```

Below is the makefile target for miniloop adapted to a new target called minilp2.

```
minilp2: ./minilp2.bnf ./rdp ml_aux.o $(RDP_SUPP)
   ./rdp -F -ominilp2 minilp2.bnf
   gcc -Irdp_supp/ -P -w -c minilp2.c
   gcc -o minilp2 minilp2.o ml_aux.o arg.o symbol.o graph.o memalloc.o
   scan.o scanner.o set.o textio.o
```

Then use the makefile to generate the minilp2 compiler.

```
cim-ts-node-01$ rm minilp2
cim-ts-node-01$ rm minilp2.c
cim-ts-node-01$ rm minilp2.o
cim-ts-node-01$ make minilp2
cim-ts-node-01$ rm mini1.mvm
cim-ts-node-01$ rm mini1.sim
cim-ts-node-01$ ./minilp2 -omini1.mvm mini1.m
cim-ts-node-01$ ./mvmasm -omini1.sim mini1.mvm
cim-ts-node-01$ ./mvmsim -t mini1.sim
```