The University of Melbourne
Department of Computing and Information Systems
COMP90045 Programming Language Implementation

Assignment 1, 2017

Released: 12 March. Two deadlines: 20 March and 10 April at 23:00

Objectives

To provide a better understanding of a compiler's front-end, lexical analysis, and syntax analysis. To practice cooperative, staged software development, using ocamllex, ocamlyacc and OCaml.

Background and context

This stage is part of a larger task: to write a compiler for a procedural language, Snick. ocamllex and ocamlyacc specifications for a small subset, Snack, of the language is provided as a starting point.

The overall task is intended to be solved by teams of 3, and after one week you will be asked to commit to a team. If it helps allocations of members to teams, we may allow teams of size 2. Eventually your task is to write a complete compiler which translates Snick programs to the assembly language of a target machine Brill.

Stage 1 of the project requires you to write a scanner, a parser and a pretty-printer for Snick. Stage 2 is student peer reviewing of the software submitted for Stage 1. Each student will review two randomly allocated project submissions (none of which will be their own). Stage 3 is to write a semantic analyser and a code generator that translates abstract syntax trees for Snick to the assembly language of Brill. The generated programs can then be run on a (provided) Brill emulator. The three stages have *seven* deadlines (add these to your diary):

Stage 1a Monday 20 March at 23:00. Individual: Submit all the names of your team members.

Stage 1b Monday 10 April at 23:00. Team effort: Submit parser and pretty-printer.

Stage 2a Tuesday 11 April at 23:00. Team effort: Re-submit, anonymised, to PRAZE.

Stage 2b Friday 21 April at 23:00. Individual: Double-blind peer reviewing (via PRAZE).

Stage 2c Friday 28 April at 23:00. Team effort (optional): Feedback to reviewers.

Stage 3a Friday 5 May at 23:00. Team effort: Submit test data.

Stage 3b Friday 19 May at 23:00. Team effort: Submit compiler.

The Stage 2 specification will be released by the end of March. The Stage 3 specification will be released close to the deadline for Stage 1.

The languages involved

The implementation language must be OCaml, and ocamllex and ocamlyacc should be used to implement the lexical and syntax analyses. We now describe the syntax of Snick.

A Snick program lives in a single file and consists of a number of procedure definitions. There are no global variables. One (parameter-less) procedure must be named "main". Procedure parameters can be passed by value or by reference.

The language has three types, namely *int*, *float*, and *bool*. (The first two are *numeric* types and allow for arithmetic and comparison operators. We do not consider the Boolean values to be ordered, but Boolean values can still be compared for equality `=` and `!=`.) It also has static arrays. The "write" command can print integers, floating point numbers, booleans and, additionally, strings.

Rules for type correctness, static semantics and dynamic semantics, including parameter passing, will be provided in the Stage 3 specification. For now, let us just mention that a variable must be declared (exactly once) before used, that numeric variables are initialised to 0, and Boolean variables are initialised to *false*. Stage 1's parser and pretty-printer are not assumed to perform semantic analysis—a program which is syntactically well-formed but has, say, parameter mismatches or undeclared variables will still be pretty-printed.

Syntax

The following are reserved words: `and`, `bool`, `do`, `else`, `end`, `false`, `fi`, `float`, `if`, `int`, `not`, `od`, `or`, `proc`, `read`, `ref`, `then`, `true`, `val`, `while`, `write`. The lexical rules are inherited from Snack whose syntax is defined in the example ocamllex/ocamlyacc specification made available (see below). An identifier is a non-empty sequence of alphanumeric characters, digits, underscore and apostrophe (`'`), which does not start with a digit or apostrophe. An int literal is a sequence of digits, possibly preceded by a minus sign. A float literal is a sequence of one or more digits, followed by a decimal point, and another sequence of one or more digits. A float literal may also be preceded by a minus sign. A literal which starts with a minus sign can not include any whitespace: '`-5`' is the int literal -5, while '`- 5`' is an expression applying the unary minus operator to the literal 5 (and both have the same value). We shortly discuss some consequences of this rule.

A Boolean constant is `false` or `true`. A string constant (as can be used by "`write`") is a sequence of characters between double quotes. The sequence itself cannot contain double quotes or newline/tab characters. However, it may contain "`\n`" to represent a newline character.

The arithmetic binary operators associate to the left, and unary operators have higher precedence (so for example, '`-5+6`' and '`4 - 2 - 1`' both evaluate to 1). An expression such as '`n-1`' is syntactically incorrect, but will be interpreted as the lexeme '`n`' followed by the lexeme '`-1`'. Hence Snick programmers need to use white space around binary minus, as in '`n - 1`'. Note that precedence rules mean that '`-5+6`' is a sum of -5 and 6. If we write '`- 5+6`', that does not change, as the minus sign is unary and still binds tighter than the binary plus.

A Snick program consists of one or more procedure definitions. Each definition consists of

1. the keyword `proc`,
2. a procedure header,
3. a procedure body,
4. the keyword `end`,

in that order. The header has two components (in this order):

1. An identifier—the procedure's name.
2. A comma-separated list of zero or more formal parameters within a pair of parentheses (so the parentheses are always present).

Each formal parameter has three components:

1. A parameter passing indicator (`val` or `ref`).
2. A type (`bool`, `float` or `int`).
3. An identifier.

The procedure body consists of zero or more local variable declarations, followed by a non-empty sequence of statements. A variable declaration consists of one of the keywords `bool`, `float` or `int`, followed by an identifier, terminated with a semicolon. Before the semicolon may be a non-empty comma-separated list of integer intervals, the list enclosed in square brackets, to indicate that the identifier names an array. An integer interval is of the form `m .. n`, where `m` and `n` are integer constants. The number of intervals corresponds to the array's dimension. For example, `float point[0..4,2..3,0..1]` declares a three-dimensional array `point` consisting of 20 floats. There may be any number of variable declarations, given in any order.

Atomic statements have one of the following forms:

```
<id> := <expr> ;
<id> [ <expr_list> ] := <expr> ;
read <id> ;
read <id> [ <expr_list> ] ;
write <expr> ;
<id> ( <expr-list> ) ;
```

where `<expr-list>` is a (possibly empty) comma-separated list of expressions. *An* `<expr-list>` *appearing in an array access (between [and], must be non-empty.* Composite statements have one of the following forms:

```
if <expr> then <stmt-list> fi
if <expr> then <stmt-list> else <stmt-list> fi
while <expr> do <stmt-list> od
```

where `<stmt-list>` is a non-empty sequence of statements, atomic or composite. (Note that semicolons are used to *terminate* atomic statements, so that a sequence of statements—atomic or composite—does not require any punctuation to *separate* the components.)

An expression has one of the following forms:

```
<id>
<id> [ <expr_list> ]
<const>
( <expr> )
<expr> <binop> <expr>
<unop> <expr>
```

The list of operators is

```
or
and
not
= != < <= > >=
+ -
* /
-
```

Here `not` is a unary prefix operator, and the bottom "`-`" is a unary prefix operator (unary minus). All other operators are binary and infix. All the operators on the same line have the same precedence, and the ones on later lines have higher precedence; the highest precedence being given to unary minus. The six relational operators are not associative. The six remaining binary operators are left-associative. The relational operators yield Boolean values `true` or `false`, according as the relation is true or not.

The language supports comments, which start at a # character and continue to the end of the line. White space is not significant, but you will need to keep track of line numbers for use in error messages.

The compiler and abstract syntax trees

The main source file that you need to create is `snick.ml` which will eventually be developed into a full compiler. For now, it will make use of a bison-generated parser to construct an abstract syntax tree (ast) which is suitable as a starting point for both pretty-printing and compiling. The compiler is invoked with

```
snick [-p] source_file
```

where `source_file` is a Snick source file. If the `-p` option is given, output is a consistently formatted (pretty-printed) version of the source program, otherwise output is a Brill program. In either case, it is delivered through standard output.

For Stage 1, only `snick -p source_file` is required to work. If the `-p` option is omitted, a message such as "Sorry, cannot generate code yet" should be printed.

For syntactically incorrect programs, your program should print a suitable error message, and not try to pretty-print any part of the program. Syntax error handling is not a priority in this project, so the precise text of this error message does not matter. Also, you need not attempt to recover after syntax errors. Syntax error recovery is very important in practice, but it also happens to be very difficult to do well, so including it in the project would have an unfavorable cost/benefit ratio.

Pretty-printing

One objective of the pretty-printing is to have a platform for checking that your parser works correctly. Syntactically correct programs, irrespective of formatting, need to be formatted uniformly, according to the following rules. The output must be stripped of comments, and consecutive sequences of white space should be compressed so that lexemes are separated by a single space, except as indicated below:

1. The pretty-printer should output the formatted procedures in the order they appeared in the input.

2. Two consecutive procedure definitions should be separated by a single blank line.
3. The keywords `proc` and `end` should begin at the start of a line—no indentation.
4. Within each procedure, declarations and top-level statements should be indented by four spaces.
5. Each variable declaration should be on a separate line.
6. Each statement should start on a new line.
7. The declarations and the statements should be separated by a single blank line.
8. The statements inside a conditional or while loop should be indented four spaces beyond the conditional or while loop it is part of.
9. In a while statement, "`while` ... `do`" should be printed on one line, irrespective of the size of the intervening expression. The same goes for "`if ... then`".
10. The terminating `od` should be indented exactly as the corresponding `while`. Similarly, the terminating `fi` (and a possible `else`) should be indented exactly as the corresponding `if`.
11. There should be no white space before a semicolon, and no white space before an opening square bracket.
12. When printing expressions, you should not print any parentheses, except when omission of parentheses would change the meaning of the expression. For example, `((x) - (y - (4*z)))` should be printed as `x - (y - 4 * z)`.
13. White space should be preserved inside strings.

Pretty-printers usually ensure that no output line exceeds some limit (typically 80 characters), but your pretty printer has no such responsibility.

A program output by the pretty-printer should be faithful to the source program's structure. Figure 1 gives an example of a Snick program and what it looks like when pretty-printed according to these rules.

Procedure and assessment

The project is to be solved in groups of 3 students (in cases where it helps allocation, 2 will be allowed). By 20 March at 23:00, submit a file called `Members.txt`, containing a well-chosen name for your team, as well as the names and usernames of all members of your team. (Please restrict team names to ASCII characters.) *Every* student should do this, using the unix command '`submit COMP90045 1a Members.txt`'. Instructions on using `submit` will be on the LMS.

By 10 April at 23:00, submit any number of files, including a unix make file (called `Makefile`), ocamllex and ocamlyacc specifications and whatever else is needed for a `make` command to generate a "compiler" called `snick`. Submit these files using '`submit COMP90045 1b`'. Each group should only submit once (under one of the members' name).

In the first instance, the only service delivered by the compiler is an ability to pretty-print Snick programs. As described above, the compiler takes the name of a source file on the command line. It should write (a formatted source or target program) to standard output, and send error messages to standard error.

```
proc q ( val    float x
       , ref int      k
       )
int n; float y;
bool a[1..8];
a[8] := true;
k := 42;
end

proc p (ref int i)
i:=(6*i) + 4; end

proc main ()
int m;
int n;
  read n;
  while n>1 do
    m := n;
    while m>0 do
          if m>0 then
              n := n - 1;
              m := m - 1;
              if m=0 then p(n); fi
       else   m := n - m; m := m - 1  ;
fi od od
end
```

```
proc q (val float x, ref int k)
    int n;
    float y;
    bool a[1..8];

    a[8] := true;
    k := 42;
end

proc p (ref int i)

    i := 6 * i + 4;
end


proc main ()
    int m;
    int n;

    read n;
    while n > 1 do
        m := n;
        while m > 0 do
            if m > 0 then
                n := n - 1;
                m := m - 1;
                if m = 0 then
                    p(n);
                fi
            else
                m := n - m;
                m := m - 1;
            fi
        od
    od
end
```

Figure 1: A Snick program(left) and its pretty-printed version (right)

On the LMS you will find ocamllex and ocamlyacc specifications for an Snack parser (Snack is a simple subset of Snick). This should help you get started.

This project counts for 10 of the 30 marks allocated to project work in this subject. Members of a group will receive the same mark, unless the group collectively sign a letter to me, specifying how the workload was distributed. Marks for Stage 1 will be awarded on the basis of correctness (of generated parser, 3 marks, and of pretty-printer, 3 marks), programming structure, style and readability (2 marks), and presentation (commenting and layout) (2 marks). A bonus mark may be given for some exceptional aspect, such as solid error recovery or reporting.

We encourage the use of lecture/tute time and, especially, the LMS's discussion board, for discussions about the project. Within the class we should be supportive of each others' learning. However, soliciting help from outside the class will be considered cheating. While working on the project, groups should not share any part of their code with other groups, but the exchange of ideas is strongly encouraged. The code review stage will facilitate learning-from-peers and at the end of that stage, we will endeavour to make a model Stage 1 solution available for all.

Graeme Gange
12 March 2017, revised 2 April