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

Assignment 2016, Stage 1

Released: 11 March. Two deadlines: 18 March and 11 April

Objectives

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

Background and context

This stage is part of a larger task: to write a compiler for a procedural language, Bean. OCamllex and OCamlyacc specifications for a small subset, Sprout, of the language are 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, in an exceptional case, allow a team of size 3 ± 1. Eventually your task is to write a complete compiler which translates Bean programs to the assembly language of a target machine Oz.

Stage 1 of the project requires you to write a scanner, a parser and a pretty-printer for Bean. 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 Bean to the assembly language of Oz. The generated programs can then be run on a (provided) Oz emulator. The three stages have *seven* deadlines (add these to your diary):

Stage 1a Friday 18 March at 23:00. Individual: **Each student** to submit **all** the names of that student's team members.

Stage 1b Monday 11 April at 23:00. Team effort: Submit parser and pretty-printer.

Stage 2a Tuesday 12 April at 23:00. Team effort: Re-submit, anonymised, to PRAZE.

Stage 2b Friday 22 April at 23:00. Individual: Double-blind peer reviewing (via PRAZE).

Stage 2c Friday 29 April at 23:00. Team effort (optional): Feedback to reviewers.

Stage 3a Friday 6 May at 23:00. Team effort: Submit test data.

Stage 3b Friday 20 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 in early April.

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 Bean.

A Bean 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 two primitive types, namely *int*, and *bool*. (*int* is a *numeric* type and allows 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 (non-recursive) compound types and type aliases. The "write" command can print integers, 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 use, 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. If the input program has lexical/syntax errors, it should not be pretty-printed; instead suitable error messages should be produced.

Syntax

The following are reserved words: `and`, `bool`, `do`, `else`, `end`, `false`, `fi`, `if`, `int`, `not`, `od`, `or`, `proc`, `read`, `ref`, `then`, `true`, `typedef`, `val`, `while`, `write`. The lexical rules are inherited from Sprout whose syntax is defined in the example ocamllex/ocamlyacc specification made available (see below). An int literal is a sequence of digits, possibly 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 Bean 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 Bean program consists of zero or more type definitions, followed by one or more procedure definitions.

Each type definition consists of

1. the keyword `typedef`,
2. a type specification,
3. an identifier

in that order.

A type specification is any one of:

- the keywords `bool` or `int`,
- a comma-separated list of field definitions surrounded by `{` and `}`,
- an identifier

where each field definition is an identifier and a type specification, separated by a colon.

Each procedure consists of

1. the keyword `proc`,
2. a procedure header,
3. a sequence of variable declarations,
4. a procedure body,
5. 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 specification.
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 a type specification followed by an identifier, terminated with a semicolon. There may be any number of variable declarations, given in any order.

Atomic statements have one of the following forms:

```
<lvalue> := <rvalue> ;
read <lvalue> ;
write <expr> ;
<id> ( <expr-list> ) ;
```

where `<expr-list>` is a (possibly empty) comma-separated list of expressions. The left-hand side of an assignment (`lvalue`) is either a variable identifier, or an lvalue followed by a period, then a field name.

The right-hand side of an assignment (`rvalue`) is either an expression, or a structure initialization which consists of `{`, a comma separated list of field initializers of the form `<ident> = <rvalue>`, followed by `}`.

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.)

The right hand side of an has one of the following forms:

```
<expr>
{ <ident> = <rvalue>, ..., <ident> = <rvalue> }
```

An expression has one of the following forms:

```
<lvalue>
<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 seven 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 program that you need to create is `bean.ml` which will eventually be developed to a full compiler. For now, it will make use of an ocamlyacc-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

```
bean [-p] source_file
```

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

For Stage 1, only `bean -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. A type definition should be printed on one line, no matter how many field definitions it makes use of.
2. 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.
12. When printing expressions, you should not print any parentheses, except when an operand itself involves the application of a binary operator; and then only when omission of parentheses would change the meaning of the expression.
13. White space should be preserved inside strings.

Pretty-printers usually ensure that no output line exceeds some limit (typically 80 characters), but this does not apply to your pretty-printer.

A program output by the pretty-printer should be faithful to the source program's structure. However, it should not use more parentheses than necessary. For example, you should reproduce the input expression '`(6 * (((3*2)) + 4*5))`' as '`6 * (3 * 2 + 4 * 5)`'. Figure 1 gives an example of a Bean program and what it looks like when pretty-printed according to these rules.

```
typedef {
  f1 : int, f2 :
{ g1 : bool, g2 : bool }
} a_record
typedef {
  used : bool,
  rec : a_record
} b_record

proc q ( val       bool x
       , ref b_record k
       )
int n; bool y;
a_record z;
z.f1 := 42;
z.f2 := { g2 = true,
g1 = false };
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
```

```
typedef {f1 : int, f2 : {g1 : bool, g2 : bool}} a_record
typedef {used : bool, rec : a_record} b_record

proc q(val bool x, ref b_record k)
    int n;
    bool y;
    a_record z;

    z.f1 := 42;
    z.f2 := {g2 = true, g1 = false};
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 well-formed but silly Bean program (left) and its pretty-printed version (right)

Procedure and assessment

Please read and follow these instructions carefully—the automated submission/feedback system depends on everybody following the submission instructions to the letter.

The project is to be completed in groups of 3 (possibly ±1). By 18 March at 23:00, submit a file called `Members.txt`, containing a well-chosen name for your team, as well as the

names of all members of your team. *Every* student should do this, using the unix command '`submit 90045 1a Members.txt`'. The file is to be submitted to one of the Engineering student servers; instructions on using `submit` for this will be posted on the LMS. (You are encouraged to use the same machine for testing, well before submission, so that you don't run into machine-dependent surprises close to the deadline.)

By 11 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 `bean`. Submit these files using '`submit 90045 1b`'. Only one team member should submit, on behalf of his/her team.

In the first stage, the only service delivered by the compiler is an ability to pretty-print Bean 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. In order to do the pretty-printing, it must generate a suitable AST (Stage 3 will depend on that).

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

Stage 1 counts for 12 of the 30 marks allocated to continuous assessment in this subject. Each member of a group will be expected to take on a fair share of the work, and each will receive the same mark, unless the group collectively sign a letter to us, specifying how the workload was distributed. Marks for Stage 1 will be awarded on the basis of correctness (of generated parser, 4 marks, and of pretty-printer, 3 marks), programming structure, style and readability (3 marks), and presentation (commenting and layout) (2 marks). A bonus mark may be given for some exceptional aspect, such as unusually solid error recovery or error 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 acceptable, and 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 and Harald Søndergaard
11 March 2016