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

Assignment 1, 2020

Released: 20 August. Two deadlines: 26 August and 16 September

Objectives

To provide a better understanding of a compiler's front-end, lexical analysis, and syntax analysis. To practice cooperative, staged software development. To improve Haskell programming skills.

Background and context

This assignment is the first stage of a larger task: to write a compiler for a procedural (C-like) language called Roo. An alex specification for a small subset, called Joey, is provided as a starting point, together with a rudimentary Joey parser written using Haskell's Parsec library.

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 can allow teams of size 2 or 4, and in exceptional circumstances, 1. The team size has no impact on the required tasks, or the assessment; it is the same project, irrespective of team size. Eventually your task is to write a complete compiler which translates Roo programs to the assembly language of a target machine Oz.

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

Stage 1a Monday 26 Aug at 23:00. Individual: Submit all the names of your team members.

Stage 1b Wednesday 16 Sep at 23:00. Team effort, but single submission: Submit parser and pretty-printer.

Stage 2a Thursday 17 Sep at 23:00. Team effort, but single submission: Re-submit, anonymised.

Stage 2b Monday 28 Sep at 23:00. Individual: Double-blind peer reviewing.

Stage 3a Monday 19 Oct at 23:00. Individual: Submit test data.

Stage 3b Wednesday 28 Oct at 23:00. Team effort, but single submission: Submit compiler.

The Stage 2 specification will be released in early September. The Stage 3 specification will be released in late September.

The languages involved

The implementation language must be Haskell. You are free to use scanner and parser generators such as alex and happy, and equally, you are free not to. The parser generator happy is a useful tool, but be warned that it is very difficult to understand happy's error messages without a solid understanding of LR parsing principles, and we won't get to cover those in lectures until Week 6.

The following is a rough description of the source language, Roo. A Roo program lives in a single file and consists of a number of procedure definitions. Roo has global type definitions for aggregate types (records and arrays), but there are no global variables. One (parameterless) procedure must be named "main". Procedure parameters of aggregate type are passed by reference. Other parameters can be passed by value or by reference.

The language has three base types, namely *integer*, *boolean*, and *string*. The integer type allows a number of arithmetic and comparison operators. The boolean type has two inhabitants, `false` and `true`. These values are ordered, so that $x \leq y$ iff x is `false` or y is `true` (or both). So the six comparison operators can also be applied to Boolean values (the arithmetic operators can not). No variables can have string type, and no operations are available on strings. But string literals are available, so that meaningful messages can be printed from a Roo program. The "write" and "writeln" commands can print inhabitants of all three base types.

There are two aggregate types, records and arrays. However, their use is restricted as follows. All arrays are one-dimensional, and are static in the sense that the size of an array is determined at compile time. An array of size n uses indices 0 to $n-1$. Arrays are homogeneous—all elements of a Roo array must have the same type. The element type can be integer, boolean, or a record type. Records cannot be nested; in fact, record fields can only have type integer or boolean.

Rules for type correctness, static semantics and dynamic semantics, including parameter passing, will be provided in the Stage 3 specification. 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. Sometimes there are aspects of well-formedness that can be captured in a context-free grammar (and thus checked by the parser), but which are more conveniently handled by semantic analysis. In those case we leave it to you to exercise your judgement as to the best approach.

NB: 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`, `array`, `boolean`, `call`, `do`, `else`, `false`, `fi`, `if`, `integer`, `not`, `od`, `or`, `procedure`, `read`, `record`, `then`, `true`, `val`, `while`, `write`, `writeln`. The lexical rules are inherited from Joey whose syntax is defined in the example alex specification made available (see below). An identifier is a non-empty sequence of alphanumeric characters, underscore and apostrophe (`'`), and it must start with a (lower or upper case) letter.

An integer literal is a sequence of digits, possibly preceded by a minus sign. A boolean literal is `false` or `true`. A string literal is a sequence of characters between double quotes. The sequence itself cannot contain double quotes or newline/tab characters. It may, however, contain '`\"`', '`\n`', and '`\t`', respectively, to represent those characters.

The arithmetic binary operators associate to the left, and unary operators have higher precedence. It follows, for example, that '`-5+6`' and '`4-2-1`' both evaluate to 1.

A Roo program consists of zero or more record type definitions, followed by zero or more array type definitions, followed by one or more procedure definitions.

Each record type definition consists of (in the given order):

1. the keyword `record`,
2. a non-empty list of *field declarations*, separated by semicolons, the whole list enclosed in braces,
3. an identifier, and
4. a semicolon.

A field declaration is of `boolean` or `integer`, followed by an identifier (the field name).

Each array type definition consists of (in the given order):

1. the keyword `array`,
2. a (positive) integer literal enclosed in square brackets,
3. a type name which is either an identifier (a type alias) or one of `boolean` and `integer`,
4. an identifier (giving a name to the array type), and
5. a semicolon.

Each procedure consists of (in the given order):

1. the keyword `procedure`,
2. a procedure header, and
3. a procedure body.

The header has two components (in this order):

1. an identifier—the procedure's name, and
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 two components (in the given order):

1. a parameter type/mode indicator, which is one of these five: a type alias, `boolean`, `integer`, `boolean val`, or `integer val`, and
2. an identifier.

The procedure body consists of zero or more local variable declarations, followed by a non-empty sequence of statements, the statements enclosed in braces. A variable declaration consists of a type name (`boolean`, `integer`, or a type alias), followed by a non-empty comma-separated list of identifiers, the list terminated with a semicolon. There may be any number of variable declarations, in any order.

An atomic statement has one of the following forms:

```
<lvalue> <- <exp> ;
read <lvalue> ;
write <exp> ;
writeln <exp> ;
call <id> ( <exp-list> ) ;
```

where `<exp-list>` is a (possibly empty) comma-separated list of expressions.

A composite statement has one of the following forms:

```
if <exp> then <stmt-list> fi
if <exp> then <stmt-list> else <stmt-list> fi
while <exp> 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 lvalue (`<lvalue>`) has four (and only four) possible forms:

```
<id>
<id> . <id>
<id> [ <exp> ]
<id> [ <exp> ] . <id>
```

An example lvalue is `point[0].xCoord`. An expression (`<exp>`) has one of the following forms:

```
<lvalue>
<const>
( <exp> )
<exp> <binop> <exp>
<unop> <exp>
```

where `<const>` is the syntactic category of boolean, integer, and string literals. The list of operators (`<binop>` and `<unop>`) 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 non-associative (so, for example, `a = b = c` is not a well-formed expression). The six remaining binary operators are left-associative.

The language supports comments, which start at a # character and continue to the end of the line. White space is not significant, but you may want to keep track of line numbers (and even column positions) for use in error messages.

The compiler and abstract syntax trees

The main program that you need to create is `Roo.hs` which will eventually be developed to a full compiler. For now, it will simply construct an abstract syntax tree (AST) which is suitable as a starting point for both pretty-printing and compiling. The compiler is invoked with one of these commands:

```
Roo -p source_file
Roo -a source_file
```

where `source_file` is a Roo source file. If the `-p` option is given, output is should be a consistently formatted (pretty-printed) version of the source program. If the `-a` option is given, output should be the result of parsing, that is, an AST. (Later, when your full compiler has been constructed, a call to `Roo` with no options should yield target code as output.) If the parser/compiler processes input without finding any issues, it should return an exit code of 0; if it does encounter errors, it should return a non-zero exit code. The parser for Joey shows how functions from Haskell's `System.Exit` library can be used to achieve this.

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. You need not attempt to recover after syntax errors; it is okay to report a single syntax error at a time (as Parsec generated parsers tend to do). Syntax error recovery is very important in practice, but it is also very time consuming to do well, so including it in the project would have an unfavourable cost/benefit ratio.

Pretty-printing

One objective of the pretty-printing is to have a platform for checking that your parser works correctly. But pretty-printing snippets of code can also be useful for a code generator—if the target language allows for comments, the code generator could include chunks of source language expressions as comments, to help a human reader of the target program.

A program output by the pretty-printer should be faithful to the source program's structure. 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 in the printing rules below. (When we say "indented", we mean indented by four spaces. In the pretty-printed program, there should be no tabs.)

The pretty-printer should take a syntactically correct program and, irrespective of how it is laid out, format it uniformly according to the following rules.

1. The pretty-printer should output the formatted record type definitions, array type definitions, and procedures in the order they appeared in the input.

2. If there are no record and array type definitions, the first procedure should start on line 1. Otherwise there should be a single blank line between the type definitions and the first procedure.

3. A record type definition involving n fields should be printed on $n + 2$ lines, as follows: The first line contains the word `record`. The remaining lines should be indented, with the first n containing one field declaration each (the first preceded by a left brace and a single space, the rest preceded by a semicolon and a single space), and with the last line containing the record name, preceded by a right brace and a single space, and followed by a semicolon; see the example in Appendix B.

4. An array type definition should be printed on a single line. It contains the word `array`, followed by a positive integer in square brackets—all without intervening white space. That string, the type, and the type alias, should be separated by single spaces, and the whole line terminated by a semicolon.

5. Each type definition should start on a new line, and there should be no blank lines between type definitions.

6. Consecutive procedure definitions should be separated by a single blank line.

7. The keyword `procedure` should begin at the start of a line—no indentation. The procedure head (that is, the keyword, procedure name, and list of formal parameters) should be on a single line.

8. The `{` and `}` that surround a procedure body should begin at the start of a line—no indentation. Moreover, these delimiters should appear alone, each making up a single line.

9. Within each procedure, declarations and top-level statements should be indented.

10. Each variable declaration should be on a separate line.

11. In a procedure body, each statement should start on a new line.

12. Each of the keywords `else`, `fi`, and `od` should always appear alone on its line.

13. The statements inside a conditional or while loop should be indented beyond the conditional or while loop it is part of.

14. 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`".

15. 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`.

16. White space, and upper/lower case, should be preserved inside strings.

17. There should be no white space after an opening parenthesis, and no space after unary minus.

18. There should be no white space before these: a comma, a semicolon, an opening square bracket, a closing square bracket, and a closing parenthesis.

19. There should be a single space after a comma, and also after `not`.

20. Single spaces should surround the assignment operator `<-`, as well as each of the 12 binary operators.

21. Ideally, expressions should be printed with only the absolutely necessary parentheses. This means taking precedence and associativity of the different operators into account.

The last rule is not easy to implement, so we will accept solutions that come close to the ideal, even if they don't meet it fully. But simply wrapping all sub-expressions in parentheses is not acceptable. In all cases, the original and the pretty-printed expression must be semantically equivalent. As a tricky example, `((13+(3*5)*7)-(2*7))-(11/(20/(2*5)))` should ideally be printed as `13 + 3 * 5 * 7 - 2 * 7 - 11 / (20 / (2 * 5))`.

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

Appendix A gives an example of a Roo program and Appendix B shows what it looks like when pretty-printed according to the rules above.

How to use the Parsec library

Lecture 8 will be devoted to parsing with Haskell, and will include a brief overview of Parsec.

Parsec is a parser combinator library written in Haskell. In the Parsec view, a parser p is an opaque object (but essentially a function) of type `Parser` t, where t is some other Haskell type. The parser p parses a specific syntactic construct, returning an abstract syntax tree of type t. For instance, a parser called `identifier` could have the type signature

```
identifier :: Parser String
```

and contain the code that parses an identifier according to the lexical rules for identifiers in the source language.

Parser combinators are functions that construct parsers from other parsers. They are used to compose simple parsers into more complex parsers for syntactic constructs such as expressions and statements. Parsec's parser combinators are higher-order functions that correspond, roughly, to the operators in Backus-Naur form for context-free grammars, such as sequencing, alternatives, and options. In fact, Parsec's combinators go well beyond that, and properly used, they can make the code for a parser both simple and elegant.

The file `JoeyParser.hs` (see below) contains an example of the use of Parsec. Sequencing of parsers is simple. Suppose we want a parser for a well-designed business letter which has a header and a salutation (the opening), followed by a body, followed by a greeting and details about the sender (the closing), a parser for a letter may be defined like so:

```
parseLetter :: Parser Letter
parseLetter
  = do
      opening <- parseOpening
      body <- parseBody
      closing <- parseClosing
      return (Letter opening body closing)
```

Here `parseOpening` is a parser for the opening part of the letter, and it is most likely defined as a combination of even simpler parsers. The abstract syntax tree returned by the above represents the triple of trees (produced by the sub-parsers), corresponding to the three parts of the business letter.

For alternation, Parsec offers the combinator `<|>`. The idea is that `p <|> q` is a parser that wants to behave like the parser `p`, but if application of `p` fails, it will behave instead like `q`. Similarly there are combinators corresponding to Kleene star, and in fact many more sophisticated combinators. There are also a few pitfalls in the use of these combinators, so you are encouraged not to miss Lecture 8.

Using Parsec is not mandatory; the only restriction is that your parser and pretty-printer must be implemented in Haskell.

Procedure and assessment

The project is to be completed in groups of 3 (possibly ±1). By 26 August 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 (restrict team names to use ASCII printable characters.) *Every* student should do this, using the unix command '`submit COMP90045 1a Members.txt`'. We will use the Engineering student servers '`dimefox2`' and '`nutmeg2`' for all submissions. Instructions for the use of '`submit`' on these servers will be published on Canvas. (You are encouraged to use the student servers for testing, well before submission, so that you don't run into machine-dependent surprises close to the deadline.) *With all submissions, we try to automate the processing as much as possible; hence it is important that you follow the instructions exactly. For example, file names matter.* `Members.txt` *should be written exactly like that, with an initial capital* `M`*, and so on.*

By 16 September at 23:00, submit any number of files, including a unix make file (called `Makefile`), and whatever else is needed for a `make` command to generate a "compiler" called `Roo`. Submit these files using '`submit COMP90045 1b`'. Each group should only submit once (under the name of one of the members).

In the first stage, the only service delivered by the compiler is an ability to pretty-print Roo 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.

On Canvas you will find an alex specification that defines the lexical aspects of Joey—the same rules apply to Roo. (Joey is a simple subset of Roo.) You don't have to make use of alex, or any other program generator, in your solution; the main role of `joey.x` is to lay down lexical rules. You will also find a rudimentary parser for Joey, constructed with the help of the parser library Parsec. Again, you don't have to make use of any of this, but you may appreciate having an example parser to start from. If you find any errors with the provided parser, it is part of your task to correct those errors.

Stage 1 counts for 12 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, 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 solid error recovery or reporting.

We encourage the use of lecture/tute time and, especially, Piazza, 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.

Harald Søndergaard
19 August 2020, updated 21 August 2020

Appendix A: An example Roo program

```
# Roo program used to exemplify pretty-printing

record {integer client_number;integer balance;
                                    boolean approved}
        account;

array [10] integer
        tax;
array [10] account    # Do we even have 10 clients?
        account_array;

  procedure main ()

      integer i,
              n,
              bal;
      account_array a;
      integer to_pay;    tax t;
    {
        write "Number of clients (up to 10): ";
        read n;  write     "\n";

        i<-0;
        while (i< n)
          do write "Client number: ";
          read a[i].client_number;
          write "Client's balance: " ;
            read bal;
            a[i].balance <- bal ;
            if bal<=18200 then    to_pay<-0 ;  else
            if bal<=37000 then to_pay <- ( bal-18200 )/5;
            else to_pay <- (3640 + ( (bal-37000) / 3 ) );
            fi fi
            t [i] <- to_pay;
            i <- i+1;
        od
        i<-0;
        while i < n do writeln t[i]; od
  }
```

Appendix B: Pretty-printed version of the example

```
record
    { integer client_number
    ; integer balance
    ; boolean approved
    } account;
array[10] integer tax;
array[10] account account_array;

procedure main ()
    integer i, n, bal;
    account_array a;
    integer to_pay;
    tax t;
{
    write "Number of clients (up to 10): ";
    read n;
    write "\n";
    i <- 0;
    while i < n do
        write "Client number: ";
        read a[i].client_number;
        write "Client's balance: ";
        read bal;
        a[i].balance <- bal;
        if bal <= 18200 then
            to_pay <- 0;
        else
            if bal <= 37000 then
                to_pay <- (bal - 18200) / 5;
            else
                to_pay <- 3640 + (bal - 37000) / 3;
            fi
        fi
        t[i] <- to_pay;
        i <- i + 1;
    od
    i <- 0;
    while i < n do
        writeln t[i];
    od
}
```