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

Assignment 2018, Stage 1

Released: 16 March. Two deadlines: 22 March and 12 April

Objectives

To develop an understanding of a compiler's front-end, lexical and syntax analysis. To practice cooperative staged program development. To improve functional programming skills.

Introduction

Throughout the course of the semester, for your project work you will be asked to implement a compiler for a simple but capable compiled language called *Paz*. Paz is a subset of ISO 7185 Standard Pascal. The task is intended to be solved in groups of size 2 ± 1, and after a week you will be asked to commit to a team.

There will be three stages. In Stage 1 you will develop a parser and a pretty-printer for Paz. The Paz syntax is specified in Backus-Naur Form (BNF), and available on the LMS. The specification is a subset of that in the ISO 7185 document, but simplified in some places, and with the ordering of alternatives changed to make it easier to parse it using the tools we are learning about in this course. Later in this document you will find some rules for pretty-printing.

You parser must be written in Haskell. We recommend to use the *Parsec* parser combinator framework, which is part of the Haskell platform. As a starting point you are given a framework including a lexical analysis phase and a basic parser for Paz, which recognizes the overall program structure but not statements or expressions.

The framework given uses Parsec, but you are free to use tools such as *Alex* or *Happy* if you prefer. We have given you one Parsec parser which does the lexical analysis phase, and another Parsec parser which does (some of) the syntactic analysis phase. You may replace the former with Alex if you wish, and/or replace the latter with Happy, or use other Haskell tools.

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 Paz to an assembly language. We expect the main deadlines for Stages 2 and 3 to be 25 April and 23 May, respectively, but there will be some intermediate deadlines as well, for smaller tasks. The Stage 2 specification will be released by the end of March. The Stage 3 specification will be released in early April.

Backus-Naur Form

Backus-Naur Form is a popular way of giving formal or informal specifications for grammars (and also lexical structure). It is popular because it is very easy and intuitive for humans to

understand. For example, it is often used in manual pages to tell users the required parameters for a command.

To give you an idea of how BNF works, consider the following BNF fragment:

if_ statement : IF *expression* THEN *statement* [ELSE *statement*] ;

This states that an *if_ statement* consists of the keyword '`if`', followed by an *expression*, followed by the keyword '`then`' and so on. Here *if_ statement*, *expression* and *statement* are names for syntactic categories. The latter two, *expression* and *statement*, will be defined elsewhere in the specification.

In our usage we have to be a little more formal and explain each BNF construction explicitly. The basic format of the BNF rules you are given is:

nonterminal_ name : *bnf_ expression* ;

The *nonterminal_ name* is similar to an identifier in C. It gives a name to the syntactic category being defined. The *bnf_ expression* is the body of the rule, which the parser will match against the incoming text.

The *bnf_ expression* is generally a list of things to match in order, which could be either terminals (tokens like IF in the above example) or nonterminals (rules defined elsewhere in the specification). The *bnf_ expression* can have alternatives, repeats, or optional items, and may contain grouping.

The *bnf_ expression* has one of the following forms, described along with our own conventions for how we use this form (not relevant elsewhere):

- A terminal name, such as IF which represents the keyword '`if`'. If the terminal has no attached information and only its presence is significant (i.e., keywords of Paz), it is in uppercase. If the terminal has attached information, for instance it is a number or an identifier, it is in lowercase.
- A nonterminal name, which is defined by rules elsewhere in the specification.
- A sequence, *bnf_ expression bnf_ expression* ... *bnf_ expression*. It matches the given expressions sequentially in the order given.
- A group of alternatives, *bnf_ expression* | *bnf_ expression* | ... | *bnf_ expression*. It matches *one* of the given alternatives, trying them in the order given.
- A repeat, {*bnf_ expression*}. It matches zero or more of the given expression sequentially from the input, trying to match as many as possible.
- An option, [*bnf_ expression*]. It matches zero or one of the given expression, trying to match the expression if possible.
- A grouping by parentheses, (*bnf_ expression*). The parentheses themselves are not matched with the input. They are necessary because in BNF, sequentiality binds more tightly than alternation, so alternation may require parentheses.
- A single-quoted character 'c' or a double-quoted string "text" to match with the input.

Lexical vs syntactic analysis

In most compilers the parser is broken into a separate lexical analysis stage, followed by the actual syntax analysis stage. The stages can run sequentially or can be interleaved, from the logical point of view it does not matter (Haskell is lazy, so it interleaves computations as required in any case).

The lexical analysis stage takes a stream of characters and scans it to produce a stream of lexical tokens. Tokens can be operators of the language (such as '+', '-', '>=' etc), keywords of the language, identifiers, numbers and so on. This simplifies matters greatly, by identifying the token boundaries and by summarizing the information to be attached to each token (a number, a string, etc).

Syntax analysis takes the stream of tokens produced by the scanner and arranges them as a tree. The purpose of syntax analysis is to recognize nested constructs, e.g., '`1 + (2 * 3)`', and also to put each construct in context. For instance, a statement might only be recognized at certain points in the input, such as at the start of a procedure, and not inside an expression.

The project directory

You are provided with a distribution package stage1_framework.tar.gz, which you should unpack into a suitable directory in the `dimefox` or `nutmeg` or other student servers. You can also use your private PC if it is capable of running the `ghc` Haskell compiler. See `https://www.haskell.org/platform/` for installer packages.

In the project directory you will find the following files:

PazLexer.hs	importable module which scans a `String` to produce a list of type `[(SourcePos, LexicalToken)]`.
PazLexerTest.hs	driver to run PazLexer.hs on the standard input and 'show' the result to standard output.
PazLexerBNF.txt	text file containing a human readable description of the language that is parsed by PazLexer.hs.
PazParser.hs	importable module which scans a list of type `[(SourcePos, LexicalToken)]` to produce an abstract syntax tree. You are to implement your statement and expression parser by modifying this file.
PazParserTest.hs	driver to run PazLexer.hs and PazParser.hs on standard input and 'show' the result to standard output.
PazParserBNF.txt	text file containing a human readable description of the language to be parsed by PazParser.hs. A substantial part of this file describes the statement and expression grammar that you are to implement.
examples	directory containing a selection of *.paz programs which are written in the Paz subset of ISO Pascal.
example_ast	directory containing example.paz giving a very simple example of Paz syntax (not a valid program but syntactically correct), with example outputs showing what PazLexerTest and PazParserTest might produce after you implement your part of the grammar. We have indented these for better clarity.

The lexical tokens of Paz

We are fortunate that Pascal, and hence Paz, has quite a small set of operators and keywords. (Much functionality is provided by built-in procedures, which are outside the scope of lexical or syntax analysis, since they are invoked identically to user-written procedures).

We will consider all of the following to be reserved words in Paz: `and`, `begin`, `boolean`, `div`, `do`, `downto`, `else`, `end`, `false`, `for`, `if`, `integer`, `not`, `of`, `or`, `procedure`, `program`, `read`, `real`, `then`, `to`, `true`, `var`, `while`, `write`, `writeln`. The relational operators are '`<`', '`<=`', '`>`', '`>=`', '`=`', and '`<>`' (the latter is used for 'not equals'). The adding operators are '`+`', '`-`', and '`or`'. The multiplying operators are '`*`', '`/`', '`div`', and '`and`'.

We now take a tour of PazLexerBNF.txt, which defines the lexical structure of Paz. The detailed definitions are taken from the ISO 7185 Pascal specification, with some added structure provided by us to reflect the separation between the lexical analysis and syntax analysis phases.

From the lexical point of view, a Paz program is a sequence of lexical tokens separated by optional whitespace. This is captured by the first rule:

```
start_symbol
    : WHITESPACE {lexical_token WHITESPACE} EOF
    ;
```

`EOF` is a special symbol not defined in the BNF specification, but instead defined directly in Parsec. We will discuss Parsec in more detail later.

We will not discuss whitespace or comments, except to say they defined in the BNF specification like any other lexical token. See the file for details.

The most important definition in PazLexerBNF.txt is `lexical_token`:

```
lexical_token
    : LEFT_PARENTHESIS
    | RIGHT_PARENTHESIS
    | TIMES
    | PLUS
    | COMMA
    | MINUS
    | ELLIPSIS
    | DOT -- must be after ELLIPSIS
    | DIVIDE_BY
    | ASSIGN
    | COLON -- must be after ASSIGN
    | SEMICOLON
    | LESS_THAN_OR_EQUAL
    | NOT_EQUAL
    | LESS_THAN -- must be after LESS_THAN_OR_EQUAL and NOT_EQUAL
    | EQUAL
    | GREATER_THAN_OR_EQUAL
    | GREATER_THAN -- must be after GREATER_THAN_OR_EQUAL
    | LEFT_BRACKET
```

```
    | RIGHT_BRACKET
    -- commented tokens as follows are not recognized by the Parsec lexer,
    -- we get them by matching the "identifier" rule and post-processing it:
    --| AND
    --| ARRAY
    --| BEGIN
    --| BOOLEAN
    --| DIV
    --| DO
    --| DOWN_TO
    --| ELSE
    --| END
    --| FOR
    --| FUNCTION
    --| IF
    --| INTEGER
    --| NOT
    --| OF
    --| OR
    --| PROCEDURE
    --| PROGRAM
    --| REAL
    --| THEN
    --| TO
    --| VAR
    --| WHILE
    | character_string
    | identifier
    | unsigned_real
    | unsigned_integer -- must be after unsigned_real
    ;
```

Note how the BNF specification is filled with our comments, introduced by '--', explaining how have we implemented various parts of the lexical analyzer. By contrast PazLexer.hs does not contain significant comments, since we have generated it automatically from the BNF with only minor editing.

Each lexical token type has a definition further down in the file, for instance:

```
identifier
    : letter {letter | digit}
    ;
```

You should not need to be too concerned with the lexical analysis phase, since you only have to take the results of lexical analysis and do the syntax analysis. However, the provided lexical analyzer is a good example of how to use Parsec.

The syntax of Paz

We now look at the Paz syntax specification in PazParserBNF.txt, which will be your reference for implementing the statement and expression parser.

Appropriately enough, a Paz source file exactly matches the `program` rule:

```
start_symbol
    : program EOF
    ;
```

A Pascal or Paz `program` takes a fairly rigid format, as follows:

```
program
    : PROGRAM identifier SEMICOLON variable_declaration_part
            procedure_declaration_part compound_statement DOT
    ;
```

The declaration parts, `variable_declaration_part` and `procedure_declaration_part`, are already implemented for you.

You are to implement Parsec (or other) parsers for `compound_statement` and everything reachable from `compound_statement`, in particular, the `statement` and `expression` rules, and every kind of statement and expression.

The next section of the BNF specification that we examine, concerns statements, and you must implement everything in this section.

The corresponding part of PazParser.hs has only a dummy parser that skips past compound statements and anything looking like a nested compound statement. You are to delete this and add your own implementation.

The most important definition in the statements section is:

```
statement
    : assignment_statement
    | procedure_statement -- must go after assignment_statement
    | compound_statement
    | if_statement
    | while_statement
    | for_statement
    | empty_statement -- must go at the end
    ;
```

You are encouraged to follow the comments in the file in regards to the order of alternatives. Parsec is a greedy parser and will commit to the first matching alternative. Although Parsec also contains backtracking functionality, this will only be invoked if something fails to match. If something matches, it matches permanently.

For instance, suppose `empty_statement` matches. Then the only way it can be un-matched, is if the thing being parsed turns out to be not a statement at all, but something else altogether. This will cause backtracking, but it will not help Parsec to try other kinds of statements. Hence, the order is critical.

The next section of the BNF specification that we examine, defines `expression`. You are to implement everything in this section also.

In the syntax of Paz, expressions are defined by a cascading set of rules, taken directly from the ISO 7185 Pascal specification, as follows:

```
expression
    : simple_expression [relational_operator simple_expression]
    ;

simple_expression
    : [sign] term {adding_operator term}
    ;

term
    : factor {multiplying_operator factor}
    ;

factor
    : unsigned_constant
    | variable_access
    | LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
    | NOT factor
    ;
```

The reason for the cascading is that some operators bind more tightly than others. Specifically, `multiplying_operator` binds more tightly than `adding_operator`, which in turn binds more tightly than `relational_operator`.

All expressions ultimately resolve to either constants or variable accesses. See the respective definitions in the file for more detail.

Also note that although any number of adding or multiplying operators can be applied to an expression (see the use of { } above), only one relational operator is possible (see the use of [ ] above). This is intentional, and is in line with Pascal's philosophy of trying to limit non-sensical code as far as possible.

How to use Parsec

This section will be clearer in Week 4 when a lecture will be devoted to parsing with Haskell.

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 Haskell type signature

```
identifier :: Parser String
```

and contain the code that parses an identifier according to the rule given in PazLexerBNF.txt,

```
identifier
    : letter {letter | digit}
    ;
```

with the responsibility for composing the letters and digits together to make a Haskell `String` which is returned at the conclusion of parsing.

A parser combinator is a function that constructs a parser from other parsers. Parsec's parser combinators are operators that correspond, roughly, to BNF's operators, like sequencing, alternatives, options, and so on. They are used to compose simple parsers into more complex parsers for syntactic constructs such as `expression`, `statement`, `program`, etc.

The most fundamental operator is '`>>`' which joins together two parsers to parse two things in sequence. If there is a requirement to also combine the results of the parsers to create an abstract syntax tree containing information from both parsers, then the operator is '`>>=`'. We do not have to look at this in detail, because Haskell provides the '`do`' notation to simplify these operations.

Translating BNF to Parsec

The PazLexer.hs and PazParser.hs you have been given use a fairly small subset of Parsec's features, which map directly to the BNF.

For sequencing, if the BNF says `parse0 parse1 parse2`, we may write this in Parsec:

```
do
    x0 <- parse0
    x1 <- parse1
    x2 <- parse2
    return (x0, x1, x2)
```

where `parse0`, `parse1` and `parse2` are parsers that recognize the corresponding syntax fragments. The abstract syntax tree returned by the above is a triple of trees returned by the parsers.

For alternatives, if the BNF says `parse0 | parse1 | parse2`, we write this:

```
choice
    [
        try parse0,
        try parse1,
        parse2
        ]
```

All but the last alternative need the `try` wrapper, which allows backtracking if the alternative does not match. Note that binary alternatives can be done with '`<|>`' instead of `choice` if you prefer (there is no difference).

In the above fragment, the abstract syntax tree is simply that returned by the successful alternative, if any. This is not always appropriate, particularly if they are not type-compatible. We probably want the abstract syntax tree to say which alternative was chosen. If we have a data type such as the following,

```
data MyType =
    MyConstructor0 AST0 |
    MyConstructor1 AST1 |
    MyConstructor2 AST2
```

where the sub-parser `parse0` has type `Parser AST0` and so on, then we can translate the alternatives to Parsec in a more sophisticated way:

```
    choice
        [
            try (
                do
                    x <- parse0
                    return (MyConstructor0 x)
                ),
            try (
                do
                    x <- parse1
                    return (MyConstructor1 x)
                ),
            do
                x <- parse2
                return (MyConstructor2 x)
            ]
```

This is the basic way that abstract syntax trees are constructed with Parsec. You are encouraged to consult the Parsec documentation, or the provided sample code, to find the equivalents for the remaining BNF constructs.

We will just give one small warning: Instead of `many myParser` you probably want `many (try myParser)`, for the same reasons that `try` was added to the `choice` lists above.

As an optimization, you may wish to experiment with removing the `try` wrapper, since we added it blindly to everything, but in reality you do not need the `try` wrapper when the choices are uniquely distinguished by their first character (during lexical analysis) or first token (during syntax analysis).

Tracing the Parsec parser

When your parser fails to match the input as expected, it is almost impossible to know what is going on. Hence we developed a standard way to make the parser print out what it is doing as it proceeds. This has been done in the provided files, and you are encouraged to continue doing it in your own sections too. For tracing, a parser like the previous section's example:

```
myParser =
    do
        x0 <- parse0
        x1 <- parse1
        x2 <- parse2
        return (x0, x1, x2)
```

has to be instrumented by inserting some extra code after the '=' sign, and then placing the real definition of the parser in parentheses, as in:

```
myParser =
    trace
        "myParser"
        (
            do
                x0 <- parse0
                x1 <- parse1
                x2 <- parse2
                return (x0, x1, x2)
            )
```

This will cause a trace to come out on the *standard error* stream while your program runs. It does not interfere with the proper output of your program, which comes out on *standard output* rather than standard error. In your submitted project, we will ignore anything that comes out on standard error. You are highly encouraged to add tracing calls and leave them in.

Pretty-printing

As well as completing the provided sample code to implement your statement and expression parser, you are asked to provide a pretty-printing facility.

One objective of the pretty-printing is to have a platform for checking that your parser works correctly. We want a tool which can take a syntactically correct program, irrespective of how it is formatted, and format it uniformly, according to the following rules. The output must be converted to lowercase and stripped of comments. Consecutive sequences of white space should be compressed so that lexemes are separated by a single space, except as indicated below:

1. The program identification line '`program` *name*`;`' should not be broken. It should be followed by a single blank line.
2. There should be a single blank line before the program's body (that is, before the '`begin`' of the main program body).
3. The program's variable definitions section, and each individual procedure definition, should all be separated from each other by a single blank line.
4. There should be no other blank lines, except that it is acceptable to print an empty statement as a blank line.
5. The keywords '`program`' and '`procedure`' should begin at the start of a line—no indentation.
6. For each procedure (or program) declaration, the procedure's (or program's) variable declarations should be indented by four spaces; however, the keyword '`var`' which introduces the variable declarations section should not be indented, and neither should the '`begin`' and '`end`' that surround the procedure's (program's) body.
7. Each variable declaration (possibly involving several variables of the same type) should be on a separate line, irrespective of how long it is. Variable declaration lines start on the line

after the keyword '`var`'.

8. Each statement should start on a new line.

9. For a compound statement '`begin` *statement* `end`', the start of *statement* should be indented four spaces relative to the '`begin`', and the '`end`' should have the same indentation as the '`begin`'.

10. The statement inside a conditional, a while loop, or a for loop, should be indented as follows. If the statement is a compound statement, it should have the same indentation as the '`if`' (and possibly '`else`'), '`while`', or '`for`' (that is, its '`begin`' and '`end`' should be flush with the '`if`', '`else`' (if present), '`while`', or '`for`'). If the statement is not compound, it should be indented four spaces beyond the conditional, while loop or for loop it is part of.

11. 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`' and for '`for ... to ... do`'.

12. If a conditional has an '`else`' part then '`else`' should be printed on its own separate line, flush with the '`if`' that it corresponds to.

13. There should be no whitespace before a comma, colon or semicolon, and no whitespace after unary '`-`' or '`+`'. Square brackets or parentheses should have no extra space except where preceded by or followed by a binary operator. (Array declarations and array accesses do not have any space before the square brackets).

14. 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.

15. Uppercase and whitespace should be preserved inside strings, and if single quotation marks inside strings were un-doubled when reading the string, then they must be doubled again when pretty-printing the string.

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).
```

Figures 1 and 2 show before-and-after examples of pretty printing a typical source file.

Your 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. Rules for type correctness, static semantics and dynamic semantics, including parameter passing, will be provided in the Stage 3 specification.

```
program miscTest;

var
id: integer;
f: boolean;

procedure boo;
    var
        a: array [3..4] of integer;
        b: array [5..23] of integer;
begin
    while b[5] >= 0 do read(a[4])
end;

procedure foo (k: real);
  var ifigenia: integer;         t: boolean;
begin
  ifigenia := 4+ 5 - 9 *(-1 + 3)
end;

procedure goo (var q: boolean;
          var m: integer;
         n: real);
        var arr: array [9..2] of real;
        {ranges not in order, but we don't test that yet}
        matrix: array [0..4] of integer;
begin
if m - 2 < 3 *8 then matrix[2] := 7
end;

begin

id := 5 * 2 + 1;
f := not (false and true) or false;
read(f);
write('strings allowed in write');

        if f <> false then
        begin
        id:= (((9 - 1))) * 3;
        f := not false
        end
        else f:= true or true;

{some procedure calls}
boo ; foo(1.08) ; goo(true, 55, 90058.783)
end.
```

Figure 1: Original program

```
program misctest;

var
    id: integer;
    f: boolean;

procedure boo;
var
    a: array[3..5] of integer;
    b: array[5..23] of integer;
begin
    while b[5] >= 0 do
        read(a[4])
end;

procedure foo(k: real);
var
    ifigenia: integer;
    t: boolean;
begin
    ifigenia := 4 + 5 - 9 * (-1 + 3)
end;

procedure goo(var q: boolean; var m: integer; n: real);
var
    arr: array[9..2] of real;
    matrix: array[0..4] of integer;
begin
    if m - 2 < 3 * 8 then
        matrix[2] := 7
end;

begin
    id := 5 * 2 + 1;
    f := not (false and true) or false;
    read(f);
    write('strings allowed in write');
    if f <> false then
    begin
        id := (9 - 1) * 3;
        f := not false
    end
    else
        f := true or true;
    boo;
    foo(1.08);
    goo(true, 55, 90058.783)
end.
```

Figure 2: Pretty printed version

Abstract vs concrete syntax trees

The BNF specification captures the expected syntax of the input, but it is not good at describing the *meaning* of the input. If the syntax tree is produced directly from the BNF rules according to the scheme described earlier in this document, it can often be quite hard to read later. (And correspondingly, hard to process when you write your compiler in later parts).

For example, consider the rule:

```
identifier
    : letter {letter | digit}
    ;
```

The direct Parsec translation would be something like:

```
identifier :: Parser (Char, String)
identifier =
    do
        x0 <- letter
        x1 <- many (letter <|> digit)
        return (x0, x1)
```

(here we have used '`<|>`' instead of `choice` to save space, and we have also omitted `try` because we are only considering single characters).

However, having a pair of a character and a string in the abstract syntax tree is not very abstract. (We call this the 'concrete' syntax tree). Instead they should be merged into a string:

```
identifier :: Parser String
identifier =
    do
        x0 <- letter
        x1 <- many (letter <|> digit)
        return (x0 : x1)
```

We have done this for you in PazLexer.hs, but it would be possible to go further. For instance, the `unsigned_integer` rule could convert the given digit sequence into a Haskell `Int` rather than leaving it as a Haskell `String`.

We have not done this for you in PazParser.hs, and we recommend that you do so. For instance, the rule

```
formal_parameter_list
    : LEFT_PARENTHESIS formal_parameter_section
            {SEMICOLON formal_parameter_section}
            RIGHT_PARENTHESIS
    ;
```

has a similar problem to what was noted for `identifier` above. You can see the problems this causes, if you look in the file example_ast/example_parser.txt and try to annotate it with what all the various tuples and lists mean, with reference to PazParserBNF.txt.

A large part of the challenge you are being set, is to improve the syntax tree (produced by

the provided sample code and produced by your code), to make the pretty printer easier to implement. This will also make your code generator easier to implement, when you get to Stage 3 of the project.

Furthermore, the ISO Pascal syntax for expressions is really quite awkward when it comes to implementing your compiler. For instance, having an additive expression as a list of things to be added together, might be handy for doing certain types of optimizations, but it is terrible for ordinary code generation.

You are encouraged to define an abstract representation, consisting of a binary tree of operations to perform, each with a left operand and a right operand. You can do this either by changing the syntax recognized by the Parsec parser to something closer to what you need, or by post-processing the concrete syntax tree.

The compiler and abstract syntax trees

The main program that you need to create is `Paz.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

`Paz [-p]` *source_file*

where *source_file* is a Paz source file. If the '`-p`' option is given, output is a consistently formatted (pretty-printed) version of the source program, otherwise output is a program in the target (assembly) language. In either case, it is delivered through standard output. Standard error is ignored and may be used for tracing purposes.

For Stage 1, only '`Paz -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 is beyond the scope of this project.

Procedure

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 pairs ±1. By Thursday 22 March at 23:00, submit a file `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 COMP90045 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 Thursday 12 April at 23:00, submit any number of files, including a unix make file (called '`Makefile`'), Haskell code, Alex or Happy specifications (if needed), and whatever else is necessary for a '`make`' command to generate a 'compiler' called '`Paz`'. Submit these files using:

```
submit COMP90045 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 Paz 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. To do the pretty-printing, it must generate a suitable AST (Stage 3 will depend on that).

Collaboration and learning

On the LMS you will find scaffolding code to help get you started. If you prefer not to make use of that, that is fine, of course.

We encourage the use of lecture/tute time and, especially, the designated discussion forum on the LMS, 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.

It is important to pay attention, from the outset, to good project management. Plan carefully who, in your team, will be doing what, and put together a written plan with well-defined milestones. As an individual, report honestly to your team about your progress, and be prepared to share problems that you run into. Equally, be prepared to hold everybody to account, and to step in and help out where a sub-task looks like slipping.

Start early, and in your plan, leave time for unforeseen problems. A major task which can be started immediately, is to develop test data for the project—lots of examples of Paz programs with and without lexical or syntax errors.

Assessment

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 programmed/generated parser, 4 marks, and of pretty-printer, 3 marks), program 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.

Nick Downing and Harald Søndergaard
16 March 2018