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

Assignment 1, 2019

Released: 21 March. Two deadlines: 27 March and 15 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.

Background and context

This assignment is the first stage of a larger task: to write a compiler for a procedural (C-like) language called Goat. An alex specification for a small subset, called Kid, is provided as a starting point, together with a rudimentary Kid 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 exceptionally 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 Goat programs to the assembly language of a target machine called Oz.

Stage 1 of the project requires you to write a parser and a pretty-printer for Goat. 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 Goat 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 Wednesday 27 March at 23:00. Individual: Submit all the names of your team members.

Stage 1b Monday 15 April at 23:00. Team effort, but single submission: Submit parser and pretty-printer.

Stage 2a Tuesday 16 April at 23:00. Team effort, but single submission: Re-submit, anonymised.

Stage 2b Monday 29 April at 23:00. Individual: Double-blind peer reviewing.

Stage 3a Friday 17 May at 23:00. Individual: Submit test data.

Stage 3b Monday 27 May at 23:00. Team effort, but single submission: Submit compiler.

The Stage 2 specification will be released in early April. The Stage 3 specification will be released around Easter.

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.

We now give a rough description of the source language, Goat. A Goat 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 base types, namely *int*, *float*, and *bool*. The first two are *numeric* types and allow for arithmetic and comparison operators. The six comparison operators can also be applied to Boolean values (the arithmetic operators can not). The Boolean values are considered to be ordered, so that $x \leq$ iff x is `false` or y is `true` (or both these hold). Goat also has static arrays and matrices (2-dimensional 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, type errors, parameter mismatches, or undeclared variables will still be pretty-printed.

Syntax

The following are reserved words: `begin`, `bool`, `do`, `else`, `end`, `false`, `fi`, `float`, `if`, `int`, `od`, `proc`, `read`, `ref`, `then`, `true`, `val`, `while`, `write`. The lexical rules are inherited from Kid whose lexical rules are given in an alex specification made available to you (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 int literal is a non-empty sequence of digits. 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 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).

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

1. the keyword `proc`,
2. a procedure header,
3. the keyword `begin`,
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 (possibly empty) comma-separated list of 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 shape indicator, which is either `[n]` or `[m,n]`, where `m` and `n` are integer literals (the former case indicates the declaration of an array of length `n`, the latter an `m` by `n` matrix). There may be any number of variable declarations, given in any order.

An atomic statement has one of the following forms:

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

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

A composite statement has 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> ]
<id> [ <expr> , <expr> ]
<const>
( <expr> )
<expr> <binop> <expr>
<unop> <expr>
```

The list of operators is

```
||
&&
!
= != < <= > >=
+ -
* /
-
```

Here `!` is a unary prefix operator (Boolean negation), 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, `x = y = z` is not well-formed). The six remaining binary operators are all left-associative. The relational operators yield Boolean values `true` or `false`, according as the relation is true or not. The Boolean operators `||` and `&&` are for disjunction ('or') and conjunction ('and'), respectively.

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 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 `Goat.hs` 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

```
Goat [-p] source_file
```

where `source_file` is a Goat 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 `Goat -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. You need not attempt to recover after syntax errors; it is fine to report a single syntax error at a time (as a Parsec parser does). 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 unfavourable 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, and there should be no other blank lines.
3. Each variable declaration should be on a separate line.
4. Each of the keywords `begin`, `end`, `else`, `fi`, and `od` should always appear alone on its line.
5. Each statement should start on a new line.
6. The keywords `proc`, `begin`, and `end` should begin at the start of a line—no indentation.
7. Within each procedure, declarations and top-level statements should be indented by four spaces.
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. Similarly, parentheses should have no extra space.
12. There should be no white space before a comma, but there should be one space after it.
13. There should be no white space after a unary operator.
14. Single spaces should surround the assignment operator `:=`. The same goes for each of the six binary operators, and for each of the six relational operators.
15. When printing expressions, you should not print any parentheses, except when an operand itself involves the application of a binary operator; in that case, put parentheses around that operand. For example, `(-(x) - (y + 4)*z)` is printed as `-x - ((y + 4) * z)`, and `(-(x*y) - y + 4*z)` is printed as `(-(x * y) - y) + (4 * z)` (recall that the operators are left-associative).
16. Upper/lower case, as well as 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 Goat program and what it looks like when pretty-printed according to these rules.

```
proc q ( val    float x
       , ref int      k
       )
int n; float y;
bool a[8];
begin
a[7] := true;
k := 42;
end

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



proc main ()
int m;
int n;
  begin
  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 call 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[8];
begin
    a[7] := true;
    k := 42;
end

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

proc main ()
    int m;
    int n;
begin
    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
                    call p(n);
                fi
            else
                m := n - m;
                m := m - 1;
            fi
        od
    od
end
```

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

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 `KidParser.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 to make sure you do not miss Lecture 8.

Of course there is no compulsion to use Parsec; the only restriction is that your parser and pretty-printer must be implemented in Haskell.

Procedure and assessment

The project is to be solved in groups of 3 ± 1 students. By 27 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` on Engineering student machines will be on the LMS. *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 15 April 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 `Goat`. Submit these files using '`submit COMP90045 1b`'. Each group should only submit once (under the name of one of the members).

In the first instance, the only service delivered by the compiler is an ability to pretty-print Goat 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 the LMS you will find an alex specification that defines the lexical aspects of Kid—the same rules apply to Goat. (Kid is a simple subset of Goat.) You don't have to make use of alex, or any other program generator, in your solution; the main role of `kid.x` is to lay down lexical rules. You will also find a rudimentary parser for Kid, 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.

This assignment 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, 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.

Harald Søndergaard
20 March 2019