Programming Languages and Computation

Problem Sheet 5: The While Language

We consider the While language, use it to write some simple programs, and write a recursive-descent parser for it after doing a bit more work on its grammar. We then extend the set of tokens our lexer recognizes and extend our parser to treat these as *syntactic sugar*, which do not modify the structured type of *abstract syntax trees* we use to represent While programs.

You will first be working in while/src/WhilePG.hs to write some example While programs.

We'll then do a bit of work on grammars before opening up while/src/Parser.hs to write our parser. While doing so, we will rely on definitions from while/src/WhileAST.hs, the definitions of lexical categories in .

Tests will use the While interpreter in while/src/While.hs to execute the While programs you write and parse; it may be interesting to look at in the next couple of weeks—once we've defined the semantics of the language.

Writing While programs

* 1. Write the following While programs (a)–(e) as Haskell strings `progA` – `progE`. (We will later check that they are valid, syntactic While programs, and possibly evaluate some of them—although not the last one unless you *really* want to.) You may find it useful to refer to the example While program given in the lecture (shown also in Figure 1), which computes a factorial, taking its input from variable `n`, and placing its output in variable `r`. (Unlike what I say in the lecture, this program *does not* loop forever when the initial value of `n` is not positive.)

   (a) A program that increments the value stored in variable `n` by 2.

   (b) A program that stores in variable `r` the absolute value of the variable initially stored in variable `n`.

   (c) A program that swaps the values stored in variables `a` and `b`. (While programs operate over arbitrary integers in $\mathbb{Z}$ so you do not need to use a temporary variable, but you may do so.)

   (d) A program that stores in variable `r` the remainder of the initial value of variable `n` in the

```
r := 1
while (1 <= n) {
  r := n * r
  n := n - 1
}
```

Figure 1: The `factorial` program.

$$
\begin{array}{lll}
E \to E \texttt{ + } T & T \to T \texttt{ * } F & F \to \texttt{Id} \\
E \to E \texttt{ - } T & T \to F & F \to \texttt{IntLit} \\
E \to T & & F \to \texttt{ - IntLit} \\
 & & F \to \texttt{ (} E \texttt{)}
\end{array}
$$

Figure 2: A context-free grammar for arithmetic expressions in While.

division by 2. (If I were to write this as a C program: `r = n % 2`.) Assume that `n` is initially non-positive.

(e) A program that loops forever, constantly increasing by 1 the value of variable `n`.

** 2. Modify, as `progFact` the program from Figure 1 so that, if the value n of variable `n` is initially negative, the program terminates with the value of $-(|n|!)$ in variable `r`. (That is, the opposite of the factorial of the opposite of n.) Its behaviour on non-negative values of n should be unchanged.

Some Grammar Work

We want to build a predictive parser. (More precisely, we want to build a *recursive descent parser*, which is a specific predictive parser which only needs to look one token ahead in order to decide which rule to apply.) As we discussed in the lecture, doing this easily based on the grammar first requires us to put the grammar in a specific form. In this section, we consider the grammar for the While language and massage it so it can be used for predictive parsing.

Let's first focus on the (original, unambiguous and full) grammar for arithmetic expressions, shown in Figure 2, and recall that our goal is to be able to decide based on the first terminal (or lexical token) we read, which of the three rules for parsing E we should use.

** 3. Draw parse trees for derivations of the following expressions using the grammar from Figure 2. Recall that `a` and `x` are valid representatives of the terminal `Id`, and `12`, `54`, `145` and `42` are valid representatives of the terminal `IntLit`.

(a) `a + -12 - 54`

(b) `(a + 12) - -54`

(c) `x * 145 + 42`

(d) `x * (145 + 42)`

*** 4. This exercise considers the need to carefully define lexical categories to keep grammar definitions reasonable and unambiguous. For example, the production rules for sums and subtractions are very similar, and it could make sense to consider catching both + and - in a single terminal (or lexical class) `AddOp`. Argue that we cannot do so and express the same grammar as above. (That

$$
\begin{array}{lll}
E \to T\ E' & T \to F\ T' & F \to \texttt{Id} \\
 & & F \to \texttt{IntLit} \\
E' \to \texttt{+}\ T\ E' & T' \to \texttt{*}\ F & F \to \texttt{-}\ \texttt{IntLit} \\
E' \to \texttt{-}\ T\ E' & & F \to \texttt{(}\ E\ \texttt{)}
\end{array}
$$

Figure 3: A right recursive context-free grammar for arithmetic expressions in While.

$$
\begin{array}{lll}
B \to B \wedge C & C \to \texttt{!}\ A & A \to \texttt{BoolLit} \\
B \to C & C \to A & A \to E\ \texttt{CompOp}\ E \\
 & & A \to \texttt{(}\ A\ \texttt{)}
\end{array}
$$

Figure 4: A context-free grammar for boolean expressions in While.

is, show that any grammar over the amended set of terminals necessarily either: represents more concrete strings, or represents less concete strings.)

As discussed in the lecture, one of the issues with a grammar that prevents predictive parsing is left-recursion, where a non-terminal appears as the leftmost symbol in one of its own derivations. Indeed, if this happens, as it does in Figure 2 for E and T, then we cannot decide based on the first symbol alone whether we should instantiate the rule one more time. *Left-recursion elimination*, as seen in the lecture, rewrites offending left-recursive production rules to make use of right-recursion instead: if we have a rule $X \to X\ \gamma$ and another rule $X \to \alpha$ (where γ and α are strings of terminals and non-terminals, then we can represent the same grammar with rules $X \to \alpha\ X'$, $X' \to \gamma\ X'$ and $X' \to \epsilon$. The result of that process on the grammar from Figure 2 is shown in Figure 3.

** 5. Remove left recursion from the following grammars. If you're eager to get on with Haskell, do the first and skip over to the next section. Advanced work in the next section will rely on this having been done correctly, but the first couple of tasks do not rely on it.

(a) Over the set of terminals `noun`, `adj`, `adv`, and `,`.

$$
\begin{array}{ll}
G \to V\ A\ N & A \to A\ \texttt{,}\ \texttt{adj} \\
V \to \texttt{adv} & A \to \texttt{adj} \\
V \to \epsilon & N \to \texttt{noun}
\end{array}
$$

(b) The grammar of While boolean expressions. (See Figure 4.)

(c) The grammar of While statements. (See Figure 5.)

$$
\begin{array}{ll}
S \to I\ S & I \to \texttt{Id}\ \texttt{:=}\ E \\
S \to \epsilon & I \to \texttt{if}\ B\ \texttt{then}\ I\ \texttt{else}\ I \\
 & I \to \texttt{while}\ B\ I \\
 & I \to \texttt{\{}\ S\ \texttt{\}}
\end{array}
$$

Figure 5: A context-free grammar for statements in While.

A recursive descent parser

Open file while/src/Parser.hs. We will use it to write a recursive descent parser. A description of the lexical categories for the While language is provided in file while/src/Lexer.x, and transformed into a Haskell implementation of a lexer using a lexer generator called `alex`. We won't be touching the lexer much, but looking at the definitions can't hurt. (Can it?)

File while/src/WhileAST.hs contains the definition for our abstract representation of While programs: the Abstract Syntax Trees. Familiarise yourself with it now: our parser will be consuming a stream of tokens, and giving us one of those trees in exchange.

** 6. Let us first turn the grammar from Figure 3 into code. We will define one function for each non-terminal, and each of them will have a clause per rule. (Note that the functions will be mutually recursive, since the grammar rules are.) The functions take in as input a stream of tokens—the input that remains to parse—and outputs the AST of an expression along with a stream of tokens—containing the suffix of its input stream that wasn't consumed.
The key challenge is in figuring out which clause to activate based only on the next token the lexer sends us. In the following, "raise an error" simply means "return `Left s`" (with `s :: String` some string that will help you debug should it be required). We won't be doing anything fancy with error handling: if the input we're given isn't valid, we're just going to fail—using `Either` is just trying to make failures informative.

(a) Write the function `parseF` which parses factors (corresponding to non-terminal F). We describe it informally below, and invite you to compare it to the production rules for F in Figure 3.

- on input a token list `TId x : ts`, return the expression tree `EVar x` and the token string `ts`;
- on input a token list `TInt i : ts`, return the expression tree `EInt i` and the token string `ts`; (Note that the lexer turns the string of digits into an integer for us—how nice!)
- on input a token list `TMinus : TInt i : ts`, return the expression tree `EInt (-i)` and the token string `ts`;
- on input a token list `TLParen : ts`, call `parseE` on `ts` to obtain either an expression tree `t` and a token list `ts'` (injected on the `Right`), or some error message (injected on the `Left`), if parsing fails—in case of success, and if `ts' = TRParen : ts''` for some `ts''` (that is, if the next token to be parsed is a closing parenthesis), then return the expression tree `t` and the token string `ts''`; in case of failure, or if `ts'` does not start with a closing parenthesis, raise an error;
- in all other cases, raise an error.

Note in particular that the implementation follows exactly from the rules.

(b) Write the functions `parseT` and `parseT'` which, together, parse terms (corresponding to non-terminal T). There are two things that make `parseF` a lot easier than this one: 1. all the production rules for F *start* with a terminal; and 2. F does not have any empty derivations. The first point would normally require us to think ahead a bit more, but we are not trying to be fancy. The fact that T' has an empty derivation means—in essence—that it can't fail unless one of its sub-parsers fails. We describe the behaviour of both functions below, and invite you, once again, to compare the resulting program to the grammar in Figure 3. An additional difficulty arises from our left-recursion elimination: T' does not describe full terms. The parser for T' takes in an expression tree for the left-hand side of the multiplication as an additional argument, and constructs the full tree itself.

`parseT` on input any token string `ts`, calls `parseF` to obtain an expression tree `t` and a token string `ts'` (or raises an error on failure), then calls `parseT'` on `t` and `ts'` and returns its results (or raises an error on failure).

`parseT'`

- on input any expression tree `t` and a token string `TStar : ts`, calls `parseF` on `ts` to obtain an expression tree `t'` and a token string `ts'` (or raises an error on failure), and finally calls `parseT'` on `EMult t t'` (the left-hand side tree constructed from the input tree and the newly-parsed right-hand side) and `ts'`, returning its result;
- on any other input `t`, `ts`, return `t` and `ts`. (This captures the empty derivation.)

(c) Getting inspiration from the above, and following the rules of the grammar. Write the functions `parseE` and `parseE'` which, together, parse expressions (corresponding to non-terminal E).

You can now test your arithmetic expression parser.

*** 7. Write recursive descent parsers for the boolean expressions and statements of the While language—the latter might deserve an extra star. Don't forget to use your modified grammars where relevant.

At this point, you are able to test all your parsers, although we strongly recommend you test your boolean expression parser as soon as you have it ready. Once your parser passes the tests, you can also test that your answers to Questions 1 and 2 are valid programs.

Finally, you can use the evaluator provided to write and evaluate more While programs. The evaluator initializes all variables to 0 before running.

Syntactic sugar

*** 8. (optional) We provide an extended lexer, which also allows the use of symbols that are sometimes useful, like <, >, >=, `!=` (for inequality) and ∨ (for logical or).
Without modifying the type of abstract syntax trees, extend your parser to accept programs that use this extended language of expressions. You may need to recall de Morgan's laws, and you *will* need to extend the grammars to guide your implementation.

**** 9. (optional) Extend your parser to allow `if ... then ...` conditionals without an `else` clause. Extending the grammar naively will cause it to no longer be parsable using recursive descent. Can you see why? You will need to carry out a transformation called left-factoring, which factors out common prefixes from productions to delay rule selection.
For example, deciding which of the two productions $S \to \alpha\ \gamma$ and $S \to \alpha$ should be used to parse S cannot be decided based on the first symbol, since both will match the same prefix. However, it is possible to delay the time at which we need to decide which rule to use until the point where they differ by *factoring out* their prefix and defining the equivalent grammar $S \to \alpha\ X$, $X \to \gamma$, and $X \to \epsilon$.