Introduction

What is a parser?

A parser for a programming language is a program that analyses a sequence of symbols, determines whether the sequence is a syntactically legal program in the language and, if it is legal, builds some kind of data structure representing the essential features of the program.

Concrete and abstract syntax

The *concrete syntax* of a programming language is a definition of which sequences represent legal programs in that language. An *abstract syntax* for a programming lanuage is a definition of a data structure for storing the essential information from the text of programs in that language. Grammars on EBNF-notation are useful for defining either kind of syntax. In section 3.4.8 of V&H the term *input language* is used for the language described by the concrete syntax, and the term *output language* is used for the language described by the abstract syntax.

Parsing by recursive descent

The parsing method that you will use in this exercise is called recursive descent. A recursive descent parser has separate program components (functions, in our case) for the different non-terminal symbols in the grammar. Some of the components may be recursive, and the parsing "descends" the derivation tree from top to bottom. In the following we will assume that the program components are functions.

The parsing begins by calling the program function for the start symbol in the grammar. When a function is called, it looks at the beginning of the list of tokens it is given to decide which grammar rule to apply. It then checks whether the tokens are valid according to that rule and then returns the correct abstract syntax structure along with what is left of the tokens. If the rule contains non-terminal symbols, the function for the rule will call the functions for these symbols.

The parser that you will write in this exercise is a *left-to-right* recursive descent parser with a *lookahead* of one token. *Left-to-right* means that the parser starts from the leftmost token in a rule. A *Lookahead* of one token means that the parser is only allowed to look at the first token before deciding which grammar rule it will follow.

Concrete syntax of DKL

Tokenizer

In the file `TokenizeDKL.oz` you are given a pre-made tokenizer that creates a list of tokens from a string containing a program of DKL. The types of tokens are as follows:

- Tokens representing reserved words: `'if'`, `'then'`, `'else'`, `'case'`, `'of'`, `'end'`, `'local'`, `'in'`, `'proc'`, `'skip'`, `true`, `false`. Since these are also reserved words in the language you will use to implement the parser, we treat them as atoms by enclosing them in single quotes.

  The words are turned into atoms by enclosing them in '-s.

- Tokens representing special characters: `'('`, `')'`, `'{'`, `'}'`, `'='`, `'$'` and `':'`. The characters are treated as atoms by enclosing them in single quotes.

- Tokens representing variable identifiers: These are tuples with the name `ident` and contain the atom equivalent of the identifier. For example, the identifier `X` would be represented by the token `ident(`X`)`.

- Tokens representing atoms, integers, or floating point numbers: These are tuples with the names `atom`, `int` or `float`, and contain the atom, integer or floating point value. Examples: `atom(leaf)`, `int(13)`, `float(3.14159)`.

The set of rules for how characters are combined into tokens is called the *lexical syntax* of a programming language. You can read more about the lexical syntax of Oz in Appendix C.6 if V&H.

Grammar

The following is a grammar for programs in DKL.

The grammar is the same as the one on pages 49-50 in V&H except for differences in notation and the inclusion of the last four rules that show how identifiers, integers, floating point numbes and atoms are represented as tokens. The grammar contains the following kinds of symbols:

1 ::=, |, {, }, [and] are part of the EBNF notation.

2 Symbols enclosed by ⟨ and ⟩ are non-terminals.

3 Words and characters enclosed by '-s correspondt to tokens that are identical to their textual representation. For example, the symbol `'skip'` corresponds to an atom written as `'skip'` in Oz.

4 The last four rules are special in that the left sides each contain a single token that holds a value. For example, (*atom*) is the Oz representation of an atom value. *ident*(and) are the remaining parts of the Oz record holding that value.

Transforming the grammar

The grammar will be difficult to use as the direct basis for a a left-to-right recursive-descent parser with one token lookahead. One problem is that the rule for sequential composition is left-recursive (it is also right-recursive, but that is not a problem). Another problem concerns statements that start with an identifier. In those cases it is not possible to determine which rule that should be used just by looking at the first token.

To implement a left-to-right recursive-descent parser with one token lookahead, the grammar must be left-factored and free of left recursion. Algorithm 1 and Algorithm 2, as taken from Aho, Sethi and Ullman (1986), define the necessary transformations.

Example: Algorithm 1 transforms the following left-recursive grammar

$$\begin{array}{lcl} S & \rightarrow & Aa \mid b \\ A & \rightarrow & Ac \mid Sd \mid \epsilon \end{array}$$

into this grammar

$$\begin{array}{lcl} S & \rightarrow & Aa \mid b \\ A & \rightarrow & bdA' \mid A' \\ A' & \rightarrow & cA' \mid adA' \mid \epsilon \end{array}$$

Elimination of Immediate Left Recursion

To resolve *immediate* left recursion, rewrite rules of the form

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

into two rules

$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array}$$

Apply Algorithm 1 to remove general left-recursion.

In our case, there is one instance of left-recursion. First, divide the rule for `<stat>` into two new rules:

```
<stat>        ::= <stat> <stat> | <simple_stat>
<simple_stat> ::= 'skip' | <local> ...,
```

Algorithm 1: Eliminating left recursion

Data : Grammar G with no cycles or ϵ-productions.
Result : An equivalent grammar with no left recursion.
begin
  Arrange the non-terminals in some order A_1, A_2, ..., A_n
  for $i := 1$ *to* n **do**
    for $j := 1$ *to* $i - 1$ **do**
      replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$, where $A_j \rightarrow \delta_1|\delta_2|\cdots|\delta_k$ are all the current A_j-productions.
    end
    eliminate the immediate left-recursion among the A_i-productions
  end
end

Algorithm 2: Left factoring a grammar.

Data : Grammar G.
Result : An equivalent left-factored grammar.
begin
  For each nonterminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$, i.e. there is nontrivial common prefix, replace all the A productions $A \rightarrow \alpha\beta_1|\alpha\beta_2|\cdots|\alpha\beta_n|\gamma$ where γ represents all alternatives that do not begin with α by

$$\begin{array}{lcl} A & \rightarrow & \alpha A' \mid \gamma \\ A' & \rightarrow & \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array}$$

  Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.
end

This immediate left-recursion is eliminated as follows. The rule

$$\underbrace{\langle \texttt{stat} \rangle}_{A} \;::=\; \underbrace{\langle \texttt{stat} \rangle}_{A}\underbrace{\langle \texttt{stat} \rangle}_{\alpha_1} \;\mid\; \underbrace{\langle \texttt{single\_stat} \rangle}_{\beta_1}$$

is rewritten into

$$\underbrace{\langle \texttt{stat} \rangle}_{A} \;::=\; \underbrace{\langle \texttt{single\_stat} \rangle}_{\beta_1}\underbrace{\langle \texttt{stat\_seq} \rangle}_{A'}$$
$$\underbrace{\langle \texttt{stat\_seq} \rangle}_{A'} \;::=\; \underbrace{\langle \texttt{stat} \rangle}_{\alpha_1}\underbrace{\langle \texttt{stat\_seq} \rangle}_{A'} \;\mid\; \epsilon.$$

Left-factoring

We now apply Algorithm 2. In our case, we have two rules with nontrivial common prefix. The rule `<stat>` has the prefix `<ident> '='`.

```
<stat> ::= ...
         | <ident> '=' <ident> <stat>
         | <ident> '=' <var> <stat>
         | ...
```

The rule(s) `<rec/pat>` with the optional production suffix `'(' { <feat> ':' <ident> } ')'` has implicitly two productions with a nontrivial common prefix `<lit>`.

```
<rec/pat> ::= <lit>
            | <lit> '(' { <feat> ':' <ident> } ')'
```

We rewrite the grammar by modifying the rules `<stat>` and `<rec/pat>` as follows:

```
<stat>     ::= ... | <ident> '=' <ideqn> | ...
<ideqn>    ::= <ident> | <val>

<rec/pat>  ::= <lit> <rec/pat2>
<rec/pat2> ::= '(' { <feat> ':' <ident> } ')' | epsilon
```

Final grammar used in Recursive-descent parsing

This is the grammar that you should base the parser on.

Operations in DKL

Where are operations such as addition and tests for equality in the concrete syntax of DKL? It's not in the grammar and not in the kernel language, but we need it to make programs of the type we are used to. Table 2.3 in V&H contains examples of basic operations. As far as this exercise is concerned, you can think that these operations are procedures bound to identifiers equal to their names in a context outside the programs we deal with here. For example, use of the addition operator would look like this: {`Plus A B Result`}.

Example program

This program can also be found in the file `Square.dkl`.

Another example program can be found in the file `Search.dkl`.

Abstract syntax of DKL

The following *type description* defines a set of trees which will be used to store the information we want to keep from the programs we parse. Such trees are called *abstract syntax trees*. The token sequences that are recognized by the grammar can be turned into Oz records by removing the '-s that enclose tokens and then concatenating the tokens.

The grammar follows sections 2.4.3 and 2.4.4 in V&H pretty closely.

It is important to understand that this type specification describes something fundamentally different from the grammar in the previous section.

The description contains the following kinds of symbols:

1 ::=, |, {, }, [and] are part of the EBNF notation.

2 Symbols enclosed by ⟨ and ⟩ are non-terminals.

3 Symbols enclosed by parentheses correspond to the Oz representation of the value of the name enclosed.

4 The rest of the symbols correspond to pieces of text that can be concatenated literally together with the value representations of 3 to make the oz record.

Example AST

This AST can be found in the file `Square.ast`.

The syntax tree of the program in the file `Search.dkl` can be found in the file `Search.ast`.