COMP5570 Assignment 4
David Barnes
Version 2022.12.19

Please ensure that you are always working from the latest version of the assessment, which you can find on the Moodle page in the Assessments section.

You may notify me of errors or seek clarifications either via the anonymous question asking page available from the Assessments section on the Moodle page, or via email to d.j.barnes@kent.ac.uk

Introduction

This assignment is designed to provide you with a practical understanding of the task of writing the syntax analysis parts of a compiler for a high-level language called Dopl (David's own programming language). Dopl is a language designed by me for this particular assessment so you should not expect to find any additional information about it elsewhere on the Web/Internet. This document is the definitive definition/description of Dopl.

Date set: 12th December 2022.
Date due: 20th January 2023. Deadline 23:55.
Submission: Moodle page for COMP5570. Submit a single zip file of the source files of your complete implementation and the scripts required to compile and/or run it.

Requirements

The main requirement of this assessment is to write a lexical analyser and recursive-descent parser for the language defined in this document. Dopl source files are stored in a text file whose name ends in a `'.dopl'` suffix. Your program must analyse a single Dopl source file named as its only command-line argument. Your program will only be tested with files with the correct suffix.

The grammar to which valid programs must conform is given below. There is a further challenge element, concerned with type checking, worth 20% of the marks.

The program must not output anything other than a single diagnostic message on standard output (`System.out`) at the end of the parse. The message must be:

```
ok
```

for a successful parse or

```
error
```

for an unsuccessful parse.

A single error in the source file must cause the parser to output the error notification and stop the parse. No error recovery is required. Note that the message must be written to standard output (`System.out` in Java) regardless of whether the parse is successful or not.

You may implement the program in a language of your choice under the constraints that the language must be available to the marker on raptor and it must be possible for the marker both to compile (where required by the language) and execute your code without having to install a particular IDE or any other software, such as libraries or build tools. These constraints are important. Any libraries required must either be bundled with your submission or already pre-installed and accessible to the marker.

Plagiarism and Duplication of Material

The work you submit must be your own. We will run checks on all submitted work in an effort to identify possible plagiarism, and take disciplinary action against anyone found to have committed plagiarism.

Some guidelines on avoiding plagiarism:

- One of the most common reasons for programming plagiarism is leaving work until the last minute. Avoid this by making sure that you know what you have to do (that is not necessarily the same as how to do it) as soon as an assessment is set. Then decide what you will need to do in order to complete the assignment. This will typically involve doing some background reading and programming practice. If in doubt about what is required, ask a member of the course team.
- Another common reason is working too closely with one or more other students on the course. *Do not* program together with someone else, by which I mean do not work together at a single PC, or side by side, typing in more or less the same code or doing the equivalent in an online setting. By all means *discuss* parts of an assignment, but do not thereby end up submitting the same code.
- It is not acceptable to submit code that differs only in the comments and variable names, for instance. It is very easy for us to detect when this has been done and we will check for it.
- **Never** let someone else have a copy of your code, no matter how desperate they are. Always advise someone in this position to seek help from their class supervisor or lecturer. Otherwise, they will never properly learn for themselves.
- It is not acceptable to post assignments on sites such as Freelancer and we treat such actions as evidence of attempted plagiarism, regardless of whether or not work is payed for. In addition, posting anywhere other than kent.ac.uk would be an infringement of the author's copyright.
- It is not acceptable to post assignments on sites such as Chegg, Freelancer, etc. and we treat such actions as evidence of attempted plagiarism, regardless of whether or not work is paid for.

Further advice on plagiarism and collaboration is available from
https://www.cs.kent.ac.uk/teaching/student/assessment/plagiarism.local

You are reminded of the rules about plagiarism that can be found in the Stage Handbook. These rules apply to programming assignments. We reserve the right to apply checks to programs submitted for assignment in order to guard against plagiarism and to use programs submitted to test and refine our plagiarism detection methods both during the course and in the future.

Grammar for Dopl

Here is the grammar of the Dopl language. Note that this has been expressed in a slightly different form from that in the book/slides. You should use this grammar as the basis for structuring your parser.

Each rule of the grammar has a 'non-terminal' symbol on the left-hand side of the '::=' symbol. On the right-hand side of '::=' is a mix of 'terminal' symbols, non-terminal symbols and grammatical notation, terminated with a semicolon. The meaning of the grammatical notation is described below.

- The notation `::=` means 'is defined as'.
- A semicolon marks the end of each non-terminal 'rule'.
- Parentheses group terminal and non-terminal symbols as a single item.
- A `?` means the single preceding item is optional or the preceding parenthesized items as a group are optional. For instance:
  `a b ? c` means that both `ac` and `abc` are valid and
  `(a b) ? c` means that both `c` and `abc` are valid.
- One or more items enclosed between curly brackets means the enclosed items occur zero or more times. For instance:
  `{ a b }` means that ab may occur 0, 1, 2 or more times; e.g., `abab`, `abababab`, `abababababab` are all valid.
- A `|` separates alternatives. For instance:
  `{ a | b | c }` means that either `a` or `b` or `c` is valid.
- Items all in upper-case are terminal symbols: an identifier (`IDENTIFIER`), keyword (`START`, `FINISH`, etc.), integer constant (`INTEGER_CONSTANT`) or character constant (`CHARACTER_CONSTANT`).
- Items enclosed in single quotes are terminal symbols, e.g. `';'` and `'<-'`.

In the grammar be careful to distinguish between those characters not enclosed between single-quote characters (e.g., `::=` and `;`) and those that look similar but are enclosed (e.g., `';'`).

```
program ::=    START declarations statements FINISH ;

declarations ::=    { declaration ';' } ;

declaration ::=     dataType identifiers ;

dataType ::=   INTEGER | CHARACTER | LOGICAL;

identifiers ::= IDENTIFIER { ',' IDENTIFIER } ;
```

```
statements ::= { statement ';' } ;

statement ::=  assignment |
               conditional |
               print |
               loop ;

assignment ::= IDENTIFIER '<-' expression ;

conditional ::=     IF expression THEN statements
                    ( ELSE statements ) ? ENDIF ;

print ::= PRINT expression ;

loop ::=  LOOPIF expression DO statements ENDLOOP;

expression ::= term { binaryOp term } ;

binaryOp ::=   arithmeticOp | logicalOp | relationalOp ;

arithmeticOp ::=    '.plus.' | '.minus.' | '.mul.' | '.div.' ;

logicalOp ::=       .and. | .or. ;

relationalOp ::=    '.eq.' | '.ne.' |
                    '.lt.' | '.gt.' | '.le.' | '.ge.' ;

term ::=       INTEGER_CONSTANT |
               CHARACTER_CONSTANT |
               IDENTIFIER |
               '(' expression ')' |
               unaryOp term ;

unaryOp ::=    '.minus.' | '.not.' ;
```

Comments

Comments are not permitted in Dopl programs.

Example program

The following (nonsense) program illustrates the main syntactic features defined by the grammar.

```
start
    integer num, sum;
    character ch;
    sum <- 0;
    num <- 1;
    ch <- "a";
    loopif ch .lt. "z"
```

```
    do
        if (num .div. 3) .ne. 0 .and. (ch .ne. "y")
        then
            sum <- sum .plus. num .mul. 2;
            ch <- ch .plus. 1;
        else
            sum <- sum .minus. num;
            ch <- "m";
        endif;
        num <- num .plus. 1;
    endloop;
    print sum;
    print ch;
finish
```

Definitions of terminal symbols: lexical analysis

- The following are all keywords of the language: `character`, `do`, `else`, `endif`, `endloop`, `finish`, `if`, `integer`, `logical`, `loopif`, `print`, `start`, `then`. They are case-sensitive and must be all lower-case in the source. These are represented in the grammar by upper-case versions (e.g., `LOOPIF`).
- All symbols within a pair of single quote characters in the grammar are literals that will be found in the source code; e.g. , `'<-'`.
- IDENTIFIER: A sequence of one or more alphabetic, digit or underscore characters, of which the first must be alphabetic. E.g., `Ng1_f3` is valid but `1e4` is not. It is not permitted to define a variable that is the same as a keyword.
- INTEGER_CONSTANT: A sequence of one or more digits (0-9).
- CHARACTER_CONSTANT: A single character enclosed within a pair of *double-quote* characters. E.g., `"!"`, `"a"`, etc. Multi-character character constants are not permitted.
- All symbols between a pair of periods (full stops) are operators; e.g., `.plus.`

Limited type checking – challenge element (20%)

Type checking involves ensuring that variables have been defined and that data types are consistent in assignments and expressions. Identifiers and associated data type information are normally stored in a 'symbol table'. So, in order to complete this part, you will need to maintain a symbol table for identifiers.

Some type checking of expressions and assignments is required, so you will also need to associate a 'data type' (*character*, *integer, logical*) with each expression, according to the following rules:

- If the expression contains at least one `relationalOp`, `logicalOp` or unary logical negation operator (`.not.`) then its data type is *logical*.
- If the expression is not *logical* and it contains at least one `CHARACTER_CONSTANT` or an `IDENTIFIER` of data type `CHAR` then its data type is *character*.
- If the expression is neither *logical* nor *character* then its data type is *integer*.

Any of the following should result in a failed parse, with the error message given previous being printed and the parse stopping.

- Use of an `IDENTIFIER` in a statement that has not been declared within a `variable`.

- Assignment of an expression of one type to a variable of a different type in an `assignment`. It is only legal to assign a character expression to a variable of type character, an integer expression to a variable of type integer, and a logical expression to a variable of type logical.

- Use of an expression of type integer or character in the expression (condition) of either an `conditional` or `loop`. All condition expressions must be of type logical.

Suggested order of implementation

1. Start by writing a lexical analyser/tokenizer to handle all of the symbols of the Dopl language. Print out details of each token as it is recognised and test with small programs to start and then gradually build up to the example given above. Don't start on the parser until the tokenizer is working.

2. Start writing the parser to parse a `program` with no variables or statements. In other words, just the `start` and `finish` keywords. That requires just two tokens from the tokenizer: `START` and `FINISH`. That will give you confidence that the interaction between parser and tokenizer is working correctly.

3. Parse `variables`. As part of this process, start to build a symbol table that records each IDENTIFIER along with its data type. This will be required later for the type-checking parts.

4. Add parsing of `term` and `expression`. You can test the parsing of these alongside parsing `print`.

5. Add parsing of `loop` but don't worry yet about checking the type of the condition.

6. Add `conditional` and `assignment`. Again, don't worry about the type checking elements.

7. For the challenge part, you will need to create a simple 'symbol table' that holds information on variable identifiers. Identifiers will be added as each `variable` is parsed. You will also need to associate a type with each `expression` and `term` as it is parsed. You could either build an explicit 'parse tree' for an expression, in which each node of the tree stores the type of the tree below it, and the overall type of the expression will be at its root; or you could simply have the methods that parse expressions and terms return the type of that particular element and work out how to combine elements of different types according to the rules given above.

Parser testing

As you are developing the parser it will be best to test with small examples that gradually add code relating to the most recent feature you have added. For instance, you might start with a program with no variable declarations:

```
start finish
```

and check that it can be parsed successfully. Then systematically remove just one token from that code and check that each variation results in a parsing error.

If you then add parsing for a single variable declaration, you might test with the following code:

```
start
integer x;
finish
```

If that parses ok then you can switch `integer` for `character` and check that, and then perform error checking along the lines already described.

Next try declaring multiple variables in the same `declaration`.

And so on …

Updates

2022.12.19: `.times.` corrected to `.mul.` in the example code. Operators added to the section on lexical tokens.