Assignment #3: Parser

Due: February 5, 2010 11:59pm

Overview

The D$^\flat$ (D flat) programming language is similar to C, C++, Pascal, Java, Python, etc., but it differs syntactically and semantically, and is considerably simpler.

Over the remaining phases of this project you will write a program that interprets D$^\flat$ programs. Your interpreter will not be interactive; it will, instead, behave very much like a Java interpreter. The input to your interpreter is a D$^\flat$ program; the output from your program is the output generated from the execution of the D$^\flat$ program. The phases of this project should help you develop a better understanding of syntactic and semantic analysis. In addition, it should help to *demystify* the workings of interpreters and, to some extent, compilers.

Educational Goals

- Syntactic Analysis – parsing
- Abstract Syntax – generalization/abstraction of concrete syntax

The D♭ Language

The following grammar partially describes D♭'s syntax. In the EBNF below, non-terminals are typeset in **bold** font and terminals are typeset in `typewriter` font. Note that the following grammar, though much shorter than for "real" programming languages, contains many rules. This fact alone should indicate that the implementation of the parser will take time. **Do start early**.

program	→	**declarations functions statement**
declarations	→	{`var id` {, `id`}* ; }*
functions	→	{**function**}*
function	→	`fn id` (**parameters**) **declarations compound_statement**
parameters	→	{**parameter** {, **parameter**}*}$_{opt}$
parameter	→	`id`
statement	→	**compound_statement** | **assignment** | **write** | **conditional** | **loop** | **return**
compound_statement	→	{ {**statement**}* }
assignment	→	`id` := **expression** ;
write	→	`write` **expression** ; | `writeline` **expression** ;
conditional	→	`if` (**expression**) **compound_statement** {`else` **compound_statement**}$_{opt}$
loop	→	`while` (**expression**) **compound_statement**
return	→	`return` **expression** ;
expression	→	**boolterm** {**boolop boolterm**}*
boolterm	→	**simple** {**relop simple**}$_{opt}$
simple	→	**term** {**addop term**}*
term	→	**unary** {**multop unary**}*
unary	→	{**unaryop**}$_{opt}$ **factor**
factor	→	(**expression**) | `id` {(**arguments**)}$_{opt}$ | `number` | `true` | `false` | `unit`
arguments	→	{**expression** {, **expression**}*}$_{opt}$
boolop	→	& | |
relop	→	= | < | > | != | <= | >=
addop	→	+ | −
multop	→	* | /
unaryop	→	!

The following rules complete the syntactic definition of D♭.

- A D♭ program is followed by an end-of-file indicator; extra text is not legal.
- The terminal (token) "id" represents the set of identifiers defined in the lexical analysis assignment. Similarly, the terminal "number" represents the set of numbers.

Part 1: The Parser

First, if you find them to be helpful, rewrite the grammar using syntax graphs (these will not be turned in). Then, determine the *first* sets; this should be straightforward because the grammar is relatively simple (though there are a lot of productions, so there will be many such sets). Finally, translate the syntax graphs into a parser. If an error—e.g., missing "`=`" in an assignment statement—is encountered in parsing, print an appropriate error message (using "TextIO.output (TextIO.stdErr, ...)") and stop execution via "OS.Process.exit OS.Process.failure".

Your primary parsing function should be called `parse` and it should take the name of a file to parse.

From each of your auxiliary functions (those that parse each part of the grammar), you will want to return the current token. Since you will often want to return more than one value (see the next part), you can include this token in a tuple.

Test your parser to see that it recognizes syntactically legal D$^\flat$programs and complains about syntactically illegal programs.

Part 2: Abstract Syntax Tree

Define an abstract syntax for D$^\flat$. This will require the definition of a new datatype (or a number of datatypes). Each constructor should reflect a significant part of the grammar above. For example, you will want a constructor that represents an "if" statement. You do not, however, need constructors that represent punctuation, e.g., a '(' token need not be represented in the abstract syntax.

Using this abstract syntax, modify your parser so that it creates an *abstract syntax tree* for the source program during parsing. This abstract syntax tree will later be used in type-checking and evaluating the program.

This part is more conceptual. Allocate enough time to allow for questions.

Part 3: Echo

Write a function called `printAST` that takes an abstract syntax tree (the result of your parser) and prints the D$^\flat$ program represented by the given tree. The output must be a syntactically valid program equivalent to the original input program (e.g., it can differ in whitespace and parentheses).

Notes

- Strive for simplicity in your programming. Do not try to be fancy. For example, the syntax graphs translate directly into code. Learn and use that technique.
- Test data with "correct" output will also be given. Your output can differ in minor ways (e.g., syntax error message format) from this "correct" output yet still be correct; it is up to you to verify your code's correctness.
- Your program will be exercised by some shell scripts, which will be provided. These scripts depend on the exit status of your program. If your program detects a "serious" error, your program should use "OS.Process.exit" to terminate.
- Grading will be divided as follows.

Part	Percentage
1	70
2	20
3	10

- Get started **now** to avoid the last minute rush.