March 16, 2014

Jani Vihervä̈s
jani.vihervas@cs.helsinki.fi

Mini-PL Interpreter

58144 Compilers Project

http://www.cs.helsinki.fi/u/vihavain/k14/compilers/project/course_project_2014.html

1 Overview

The interpreter I developed has four main class components: Scanner, Parser, Statements and Interpreter, which is the only executable class. In it's main program the Interpreter reads the source code file via FileReader class, tokenizes the source code via Scanner, parses the tokens via Parser and finally executes the code via Statements which represents the abstract syntax tree.

In the src folder you can find a batch file named `compile.bat`, which compiles the source code. If you want to run the NUnit unit tests, download the NUnit class library file `nunit.framework.dll` from NUnit's web page `http://nunit.org/?p=download`, put it in src folder and compile the source code with the `compile_with_tests.bat`. Once the source code is compiled, you can run the interpreter with the command `Interpreter.exe filename.mpl`, where `filename.mpl` is the source code file or the path to it.

2 Grammar

The grammar was modified so that there are no LL(1) violations. The following is the modified context-free grammar:

$$
\begin{aligned}
< prog > &\rightarrow < stmts > \\
< stmts > &\rightarrow < stmt > \ ; \ < stmts' > \\
< stmts' > &\rightarrow \epsilon \mid < stmts > \\
< stmt > &\rightarrow \textbf{var} \ < ident' > \ : \ < type >< stmt' > \\
&\mid \ < ident > \ := \ < expr > \\
&\mid \textbf{for} \ < ident > \ \textbf{in} \ < expr > \ .. \ < expr > \ \textbf{do} \ < stmts > \ \textbf{end for} \\
&\mid \textbf{read} \ < ident > \\
&\mid \textbf{print} \ < expr > \\
&\mid \textbf{assert} \ (< expr > \ ) \\
< stmt' > &\rightarrow \epsilon \mid := \ < expr > \\
< expr > &\rightarrow \ < opnd >< expr' > \\
< expr' > &\rightarrow \epsilon \mid < op >< opnd > \\
< opnd > &\rightarrow \ < int > \\
&\mid \ < string > \\
&\mid \ < opnd' >< bool > \\
&\mid \ < ident > \\
&\mid \ (\ < expr > \ ) \\
< opnd' > &\rightarrow \epsilon \mid < unary > \\
< type > &\rightarrow \textbf{int} \mid \textbf{string} \mid \textbf{bool} \\
< reserved \ \ keyword > &\rightarrow \textbf{var} \mid \textbf{for} \mid \textbf{end} \mid \textbf{in} \mid \textbf{do} \mid \textbf{read} \mid \textbf{print} \mid \textbf{assert} \\
&\mid \textbf{int} \mid \textbf{string} \mid \textbf{bool} \mid \texttt{true} \mid \texttt{false} \\
< unary > &\rightarrow \textbf{!} \\
< op > &\rightarrow + \mid \textbf{-} \mid \textbf{*} \mid / \mid < \mid > \mid <= \mid >= \mid \textbf{!}= \mid = \mid \&
\end{aligned}
$$

$< ident' >$ adds identifier to symbol table, where as $< ident >$ looks the identifier from the symbol table. Operators >, <=, >= and != were added, because they are very easy to implement.

Predict sets:

Production	Predict set
$< prog >$	**var**, $< ident >$, **for**, **read**, **print**, **assert**
$< stmts >$	**var**, $< ident >$, **for**, **read**, **print**, **assert**
$< stmts' >$	$$
	var, $< ident >$, **for**, **read**, **print**, **assert**
$< stmt >$	**var**
	$< ident >$
	for
	read
	print
	assert
$< stmt' >$	;
	:=
$< expr >$	`i`, `s`, **!**, `b`, $< ident >$, (
$< expr' >$	;, .., **do**
	`i`, `s`, **!**, `b`, $< ident >$, (
$< opnd >$	`i`
	`s`
	!, `b`
	$< ident >$
	(
$< opnd' >$	`b`
	!
$< type >$	**int**
	string
	bool

3 Scanner

Scanner class tokenizes the source code. Tokenizing was implemented in a simple way, where the current lexeme is matched as whole into every reserved keyword, types etc. This is not the most efficient way, because there is a lot of backtracking, but it simplifies the code making it easier to read. If lexeme was matched a token is created and added to the list of scanned tokens. Every scan of a certain lexeme can be presented as a regular expression `lexeme`, where `lexeme` is the lexeme to be scanned, f.g. `for`. If the lexeme was not any reserved keyword or identifier, an error token is produced. Error lexeme consists of any character but white space, letters or numbers and if consecutive error tokens are produced, they are combined together. Scanner scans the whole source code and returns a list of all the tokens.

Different token types are Token, TokenError, TokenIdentifier and TokenTerminal<T>. Every token class is inherited from Token class and have attributes Line, StartColumn and Lexeme, which are self explanatory. Extended methods and properties include combining TokenErrors and holding a scanned value of the type T, which can be `int`, `bool`, or `string`.

4 Parser

Parser parses the tokens and produces the abstract syntax tree. Parser is implemented as recursive descent parser, where each production of the grammar is implemented as a method. $< stmt' >$ production is not implemented, as it would have been harder to produce the syntax tree with as little nesting as possible.

Parser accesses the SymbolTable singleton, where each found variable identifier is stored. Note that the grammar is modified in a way that variable identifier can only be added to the symbol table in the $< ident' >$ production. This was an easy way to handle errors produced by uninitialized variables. Also there is no need to check if the variable identifier is reserved keyword, because the Scanner doesn't ever produce a TokenIdentifier object with a reserved keyword as a lexeme.

If the lexeme of the current token cannot be matched to the predic set of the production rule, an Error object is added to the error list and tokens are skipped to the first occurence of a lexeme in the productions follow set. If there is a syntax error in the middle of a production, f.g. `var i : int = 1;`, which has no assignment symbol, tokens are skipped until the first set of the next nonterminal or the follow set of the production. When all of the tokens are parsed, a constructed abstract syntax tree is returned if there were no syntax errors found. If there were syntax errors, a ParserException is thrown and the execution of the program stops.

Each Error object holds the information of on what line and column the error was and what is the symbol that was expected. Error object also shows the error producing line and a cursor at the starting column of the error, f.g.

```
Line 1, column 13: Was expecting :=.
var i : int = 1;
            ^
```

5 Abstract syntax tree

The root node of the abstract syntax tree is a Statements object, which can hold multiple Statement objects. These include StatementAssert, StatementFor, StatementRead, StatementPrint, Statements, StatementVarAssignment and StatementVarInitialize. Each of the objects are inherited from the Statement base class and each overrides the `Execute()` method. The `Execute()` method in the Statements object calls every child Statement object's overrided `Execute()` method. Other statement objects have their own unique straight-forward actions for execution which are all easy to understand from the source code.

Error handling is implemented in a similar way with the Parser with the exception that if an error is found, the execution of the statement stops and Statements object executes the next Statement object in the list. When there are no more Statement objects to execute, the Statements object checks if there were errors found, and will throw an AbstractSyntaxTreeException stopping the programs execution if there were any. As with the Parser, every Statement adds an Error object to the list of semantic errors, which is passed on with the exception.

As there were no specific instructions on how to handle expression evaluating, I implement the evaluating in a way, where one can apply operators to different types of operands. This means that an expression `("s" + true) + 1` produces a string with a value `"strue1"`. I thought of making changes to the grammar allowing expression two have infinite number of operands, but there were no time. In it's current state for example an expression `1 + 2 + 3` has to be typed as `(1 + 2) + 3` or `1 + (2 + 3)`. Another problem is that for example an expression `("t" + "t") != ("tt" + "t")` cannot be evaluated, as the expressions in the parentheses are not of boolean type.

6 Testing

The whole development process followed the Test Driven Development guide line, and the test coverage is quite good. Tests consist of white-box unit tests using NUnit testing framework but the tests follow the use cases and as such have an acceptance test feel to them.