UNIVERSITY OF ESSEX

ASSIGNMENT ONE - EXPRESSION ANALYSER

CE305 - Language and Compilers (CE305)

Author:
John PULFORD

Lecturer:
Dr. Chris Fox

February 15, 2014

Contents

1 Grammar

1.1 Source

1.1.1 Tokens

All possible tokens:

$$\begin{aligned} TOKEN &::= MUL \\ &\mid DIV \\ &\mid PLUS \\ &\mid MINUS \\ &\mid LEFTPAREN \\ &\mid RIGHTPAREN \\ &\mid FLOATEXPONENT \\ &\mid FLOAT \\ &\mid OPTIONALLYSIGNEDINT \\ &\mid INT \\ &\mid DIGIT \end{aligned}$$

Each individual token:

MUL	$::=$	$'*'$
DIV	$::=$	$'\backslash'$
$PLUS$	$::=$	$'+'$
$MINUS$	$::=$	$'-'$
$LEFTPAREN$	$::=$	$'('$
$RIGHTPAREN$	$::=$	$')'$
$FLOATEXPONENT$	$::=$	$'e'$
$FLOAT$	$::=$	$OPTIONALLYSIGNEDINT\ FLOATEXPONENT\ OPTIONALLYSIGNEDINT$
$OPTIONALLYSIGNEDINT$	$::=$	$(\ MINUS \mid PLUS\ )?\ INT$
INT	$::=$	$DIGIT\ DIGIT+$
$DIGIT$	$::=$	$[\ 0-9\ ]$

1.1.2 Expressions

$$
\begin{array}{lll}
expr & ::= & expr\ PLUS\ expr \\
 & \mid & expr\ DIV\ expr \\
 & \mid & expr\ MUL\ expr \\
 & \mid & expr\ MINUS\ expr \\
 & \mid & FLOAT \\
 & \mid & OPTIONALLYSIGNEDINT \\
 & \mid & PLUS?\ LEFTPAREN\ expr\ RIGHTPAREN \\
 & \mid & MINUS\ LEFTPAREN\ expr\ RIGHTPAREN
\end{array}
$$

1.2 Target Language

A format description of the target language is below note that extra language aspects such as loops and conditional statements are not mentioned as the compile will never convert any of the source language into such tokens / statements. Integers are only mentioned because the are part of the composite terminal float type. Integer arithmetic and pop symbols are also omited.

1.2.1 Tokens

$$
\begin{array}{lcl}
TOKEN & ::= & FLOATMUL \\
 & | & FLOATDIV \\
 & | & FLOATPLUS \\
 & | & FLOATMINUS \\
 & | & FLOATPOP \\
 & | & MUL \\
 & | & DIV \\
 & | & PLUS \\
 & | & MINUS \\
 & | & POP \\
 & | & FLOATEXPONENT \\
 & | & FLOAT \\
 & | & OPTIONALLYMINUSINT \\
 & | & OPTIONALLYSIGNEDINT \\
 & | & INT \\
 & | & DIGIT \\
 & | & SEPERATOR
\end{array}
$$

MUL	::=	$'*'$
DIV	::=	$'\backslash'$
$PLUS$	::=	$'+'$
$MINUS$	::=	$'-'$
POP	::=	$'.'$
$FLOATMUL$	::=	$'f*'$
$FLOATDIV$	::=	$'f\backslash'$
$FLOATPLUS$	::=	$'f+'$
$FLOATMINUS$	::=	$'f-'$
$FLOATPOP$	::=	$'f.'$
$FLOATEXPONENT$	::=	$'e'$
$FLOAT$	::=	$OPTIONALLYMINUSINT\ FLOATEXPONENT\ OPTIONALLYSIGNEDINT$
$OPTIONALLYMINUSINT$	::=	$MINUS?\ INT$
$OPTIONALLYSIGNEDINT$	::=	$(\ MINUS \mid PLUS\ )?\ INT$
INT	::=	$DIGIT\ DIGIT+$
$DIGIT$	::=	$[\ 0-9\ ]$
$SEPERATOR$	::=	$'\ '$

1.2.2 Expression

Whilst not strictly true I've modeled the grammar of Forth as one that will either except integer or float numbers. This is true from the perspective of this compiler but in reality there is the option of conversion to float within the forth language.

$$
\begin{array}{lcl}
start & ::= & expr \mid floatexpr \\
expr & ::= & expr\ expr\ PLUS \\
 & \mid & expr\ expr\ DIV \\
 & \mid & expr\ expr\ MUL \\
 & \mid & expr\ expr\ MINUS \\
 & \mid & OPTIONALLYMINUS\ INT \\
floatexpr & ::= & floatexpr\ floatexpr\ FLOATPLUS \\
 & \mid & floatexpr\ floatexpr\ FLOATDIV \\
 & \mid & floatexpr\ floatexpr\ FLOATMUL \\
 & \mid & floatexpr\ floatexpr\ FLOATMINUS \\
 & \mid & FLOAT
\end{array}
$$

2 Translation

2.1 Tokens

If a FLOAT or expr.OPTIONALLYSIGNEDINT has a plus sign prefix then the prefix is discarded in translation. This is implicit in the translation of terminals from source FLOAT to target FLOAT and from source OPTIONALLYSIGNEDINT to target OPTIONALLYMINUSINT.

2.2 Expression

The rule tables follow. The first column signifies the production rule. The second is how it is represented in the source language. The third is the semantic rule for that particular representation i.e the order the nodes are visited and the output of that production rule traversal which results in a Forth compatible grammar. The terminals in this column refer to forth terminals. Non-terminals refer to the source language.

There are also are two versions of the syntax directed translation. One when at least one floating number exists in the source and one when they do not.

2.2.1 Float Enabled Translation

Production		*Source*	*Target*
start	::=	$expr_1$	$expr_1$ *SEPERATOR FLOATPOP*
expr	::=	$expr_1$ *PLUS* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR FLOATPLUS*
	|	$expr_1$ *DIV* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR FLOATDIV*
	|	$expr_1$ *MUL* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR FLOATMUL*
	|	$expr_1$ *MINUS* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR FLOATMINUS*
	|	*FLOAT*	*FLOAT*
	|	*OPTIONALLYSIGNEDINT*	*FLOAT*
	|	*PLUS? LEFTPAREN expr RIGHTPAREN*	*expr*
	|	*MINUS LEFTPAREN expr RIGHTPAREN*	$'0e' - expr$

2.2.2 Float Disabled Translation

Production		*Source*	*Target*
start	::=	$expr_1$	$expr_1$ *SEPERATOR POP*
expr	::=	$expr_1$ *PLUS* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR PLUS*
	|	$expr_1$ *DIV* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR DIV*
	|	$expr_1$ *MUL* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR MUL*
	|	$expr_1$ *MINUS* $expr_2$	$expr_1$ *SEPERATOR* $expr_2$ *SEPERATOR MINUS*
	|	*OPTIONALLYSIGNEDINT*	*OPTIONALLYMINUSINT*
	|	*PLUS? LEFTPAREN expr RIGHTPAREN*	*expr*
	|	*MINUS LEFTPAREN expr RIGHTPAREN*	$'0' - expr$

3 Implementation

3.1 Running

To build, test and run with arguments use.

```
./gradlew clean build shadowJar run -Pargs="example.if"
```

Simply change "example.if" to another filepath to run with said file.

NOTE: Not applicable to university computers, unfortunately. Instead, to run the precompiled jar:

```
java -jar build/distributions/small-compiler-unspecified-shadow.jar example.if
```

The compiler will generate two files.

```
<filename>.fs and <filename>.ps
```

The former is the target code generated from the input file. The latter is a postscript parse tree representing the calls within the program. Note that ANTLR4 does not list the types of terminal characters and thus this is not technically a concrete syntax tree.

3.2 Tools

An ANTLR4 grammar was written and then compiled to generate a Parser and Lexer. Then the base visitor class was extended to generate the source string and control the order non-terminals were visited in. Graphing was also generated via ANTLR. Error handling was done by extending ANTLR's error handling classes and extending the Context-Free Grammar (CFG) with error catching. ANTLR4 grammar files have extensive enhancements over traditional grammars. Ignoring whitespace was achieved with one line.

Less significantly other tools were applied but mainly applied to the work flow.

- Junit was used for unit testing compiler output against target language output.
- Gradle was used for the build script, it compiles the ANTLR files, then my java files, then unit tests them, packages up the program as a jar and finally runs it with my specified arguments (details on how to do thi in README.txt).
- An awaiting-approval plugin for Gradle known as shadowJar which generates fat jars; jars that contain all of their dependencies.
- A crude attempt at an ANTLR4 plugin for Gradle from Github was used to facilitate this build scripts existence.

3.3 Extended Features

I have;

A visual reading of a parse tree Although as mentioned it is missing terminal types.

Support for floating-point expressions as well as integers Integers are converted into floats when floats are in the equation; the lexer is applied twice to achieve this.

Helpful and informative error messages Parenthesis control and error underlining.

Code has been divided, code inside:

src -> generated

Is generated by antlr during a build.

Code that is inside:

src -> main

src -> test

was written by me, this code also contains comments for further information.