IN2009 Tutorial 3

In this tutorial you will develop, first a parser, and then a compiler, for very simple programming language. On the next page you will find the grammar for the language. It is *extremely* limited. The only expressions in this language are integer literals (no arithmetic). It has no variables or loops. It does have if-statements. The condition of an if-statement is an integer (because this is the only data type that the language has). Zero is treated as false (selects the else-branch) and any other integer, whether positive or negative, is treated as true. The **`printint`** command outputs the decimal representation of an integer (compile to syscall 3); the **`printchar`** command outputs the char with a given code (compile to syscall 2).

Task 0

Draw parse trees for the following two small programs.

```
begin
    printint 789;
end
```

```
begin
    if (0)
        printint 10;
    else
        printint 20;
    printchar 13;
    printchar 10;
end
```

Task 1

Write a recursive-descent parser for this language. You have been provided with a prototype implementation. Test your parser on all the test inputs. For those in the negative-tests folder, your parser should throw a ParseException. Tests in the positive-tests folder should terminate normally and print the "Parse Succeeded" message.

Do *not* try to implement all the grammar rules at once. Look at the test files. You will see that some of them only use parts of the grammar, so you can implement those parts, test, and debug, before you move on to the next part. The structure of a recursive descent parser lends itself to incremental development in this way (for `Stm()`, you can add, test and debug one new case at a time in the switch statement).

You may find it useful to borrow ideas and/or code from the Compiler2 class in the Tutorial2Solutions project (ignoring the code generation parts for now).

Task 2

Write a compiler for this language. Make a copy of your parser and add code so that it outputs SSM assembly code. Test your compiler on all the test inputs. The negative tests should still fail with ParseExceptions. The positive tests should output SSM code which produces the expected outputs when assembled and executed.

Note: the code that you generate for if-statements will include jump instructions, so you will need to generate label names for the jump targets. Since a source program may contain arbitrarily many if-statements, you will need to maintain some kind of counter in your compiler and use it to ensure that all your label names are distinct.

The Teeny-Weeny Language Grammar

$$
\begin{aligned}
\textit{Prog} &\rightarrow \text{BEGIN } \textit{Stm}^* \text{ END} \\[1em]
\textit{Stm} &\rightarrow \text{IF LBR } \textit{Exp} \text{ RBR } \textit{Stm} \text{ ELSE } \textit{Stm} \\
\textit{Stm} &\rightarrow \text{PRINTINT } \textit{Exp} \text{ SEMIC} \\
\textit{Stm} &\rightarrow \text{PRINTCHAR } \textit{Exp} \text{ SEMIC} \\
\textit{Stm} &\rightarrow \text{LCBR } \textit{Stm}^* \text{ RCBR} \\
\textit{Exp} &\rightarrow \text{INT}
\end{aligned}
$$

The Teeny-Weeny Language Tokens

- INT: signed integer literals (as used in the calc language)
- SEMIC: semicolon `;`
- LBR, RBR: left bracket `(` and right bracket `)`
- LCBR, RCBR: left curly bracket `{` and right curly bracket `}`
- All other tokens stand for the corresponding lower-case keyword (**`begin`**, **`end`**, etc).

Task 3 (*Challenge*)

Extend the grammar, parser and compiler to support arithmetic and variables. A possible expression grammar, suitable for direct implementation by recursive descent parsing, is provided on page 3. You will need to invent your own test inputs. If you manage all this, you’re on a roll, so why not go ahead and add while-loops to the language as well?

A simple language of arithmetic-expressions

$$
\begin{array}{lll}
\textit{Exp} & \rightarrow & \textit{BasicExp}\ \ \textit{ExpRest} \\
\\
\textit{BasicExp} & \rightarrow & \text{INT} \\
\textit{BasicExp} & \rightarrow & \text{ID} \\
\textit{BasicExp} & \rightarrow & \text{LBR}\ \ \textit{Exp}\ \ \text{RBR} \\
\\
\textit{ExpRest} & \rightarrow & \text{ADD}\ \ \textit{BasicExp} \\
\textit{ExpRest} & \rightarrow & \text{MUL}\ \ \textit{BasicExp} \\
\textit{ExpRest} & \rightarrow & \text{SUB}\ \ \textit{BasicExp} \\
\textit{ExpRest} & \rightarrow & \text{DIV}\ \ \textit{BasicExp} \\
\textit{ExpRest} & \rightarrow &
\end{array}
$$

This is easy to parse. It sidesteps operator precedence issues by forcing expressions to be fully-bracketed. The ID token should be implemented as some appropriate regular expression for variable names (such as non-empty sequences of letters). ADD, MUL, SUB, DIV are tokens for the obvious integer arithmetic operators.

To get a feel for how the grammar works, write down derivation trees for the following expressions:

```
2 - fred

123

7 + 9

(1 + 2) * 3
```

Note that the final empty rule for *ExpRest* is *not* a mistake (try building a derivation tree for `123` without this rule!).

Note: you will find that you must leave at least one space after the minus sign in expressions like `5-2`. If you don't, the `-2` part will be lexed as an INT, rather than as separate SUB and INT tokens. This is annoying, but it requires a slightly smarter lexer to get this right, so we will just live with it.