Kompilatorteknikk TDT4205 - Problem set 2

Ola Kristoffer Hoff

14th February, 2023

1 Top/down parsing tables

Consider the following grammar, which abstracts a switch statement:

$$
\begin{aligned}
S &\to sCxLDy \\
C &\to c \\
L &\to L;I|I \\
I &\to i \\
D &\to d|\epsilon
\end{aligned}
$$

1.1 Adapt the grammar for top/down parsing with an LL(1) parser.

My immediate concern about the grammar is the left recursion in the rule for L. This would cause trouble in the FIRST table with a double entry at (L, i). We can remove the recursion by switching the L-expression with:

$$
\begin{aligned}
L &\to IL' \\
L' &\to ;IL'|\epsilon
\end{aligned}
$$

This yields us the final adapted grammar:

$$
\begin{aligned}
S &\to sCxLDy \\
C &\to c \\
L &\to IL' \\
L' &\to ;IL'|\epsilon \\
I &\to i \\
D &\to d|\epsilon
\end{aligned}
$$

1.2 Tabulate the FIRST and FOLLOW sets for each non-terminal, as well as their nullability.

	FIRST	FOLLOW	nullable?
S	s	$	no
C	c	x	no
L	i	d, y	no
L′	;	d, y	yes
I	i	;, d, y	no
D	d	y	yes

Table 1: Table for FIRST, FOLLOW and nullable sets of grammar.

1.3 Construct the LL(1) parsing table for your grammar.

To create the parsing table we follow two rules:

- Enter the production $X \to \alpha$ at (X, t) where t is in $FIRST(\alpha)$
- When $\alpha \to *\epsilon$, enter the production $X \to \alpha$ at (X, t) where t is in $FOLLOW(X)$

After applying the first step we have table 2.

	s	x	y	c	i	d	;
S	$S \to sCxLDy$						
C				$C \to c$			
L					$L \to IL'$		
L′							$L' \to ;IL'$
I					$I \to i$		
D						$D \to d$	

Table 2: Parsing table for grammar after first step.

Now we apply the second step and end up with the finished parsing table, table 3.

	s	x	y	c	i	d	;
S	$S \to sCxLDy$						
C				$C \to c$			
L					$L \to IL'$		
L′			$L' \to \epsilon$			$L' \to \epsilon$	$L' \to ;IL'$
I					$I \to i$		
D			$D \to \epsilon$			$D \to d$	

Table 3: Parsing table for grammar, finished, after second step.

2 VSL specification

The directory in the code archive ps2_skeleton.tar.gz begins a compiler for a slightly modified 64-bit version of VSL ("Very Simple Language"), defined by Bennett (Introduction to Compiling Techniques, McGraw-Hill, 1990).

It's lexical structure is defined as follows:

- Whitespace consists of the characters '\t', '\n', '\r', '\v' and ' '. It is ignored after lexical analysis.
- *Comments* begin with the sequence '//', and last until the next '\n' character. They are ignored after lexical analysis.
- *Strings* are sequences of arbitrary characters other than '\n', enclosed in double quote characters '"'.
- *Reserved* words are **func**, **begin**, **end**, **return**, **print**, **break**, **if**, **then**, **else**, **for**, **in**, **while**, **do**, and **var**.
- *Operators* are the assignment operator ':=', the basic arithmetic operators '+', '−', '∗', '/', and relational operators '=', '! =', '<', '>'.
- *Numbers* are sequences of one or more decimal digits ('0' through '9').
- *Identifiers* are sequences of at least one letter followed by an arbitrary sequence of letters and digits. Letters are the upper- and lower-case English alphabet ('A' through 'Z' and 'a' through 'z'), as well as underscore ('_'). Digits are the decimal digits, as above.

The syntactic structure is given in the context-free grammar on the last page of this document. (not in this document).

Building the program supplied in the archive **ps2_skeleton.tar.gz** combines the contents of the "**src/**"-subdirectory into a binary "**src/vslc**" which reads standard input, and produces a parse tree.

The structure in the "**vslc/**" directory will be similar throughout subsequent problem sets, as the compiler takes shape. See the slide set from the PS2 recitation for an explanation of its construction, and notes on writing Lex/Yacc specifications.

2.1 Scanner

*Complete the Lex scanner specification in "**src/scanner.l**", so that it properly tokenizes VSL programs.*

In figure 1 you can see the code that I added in the **scanner.l**-file. I did not add anything for the operators, since I did not see the use for that, at least not yet, they will be interpreted through strings.

```
%{
#include <vslc.h>
// The tokens defined in parser.y
#include "y.tab.h"
%}
%option noyywrap
%option array
%option yylineno

WHITESPACE [\ \t\v\r\n]
COMMENT \/\/[^\n]+
QUOTED \"([^\"\n]|\\\")*\"

STRINGS \"^\n*\"
NUMBERS [0-9]+
IDENTIFIERS [a-zA-Z][a-zA-Z0-9_]*

%%
func                    {return FUNC;}
print                   {return PRINT;}
return                  {return RETURN;}
break                   {return BREAK;}
if                      {return IF;}
then                    {return THEN;}
else                    {return ELSE;}
while                   {return WHILE;}
for                     {return FOR;}
in                      {return IN;}
do                      {return DO;}
begin                   {return OPENBLOCK;}
end                     {return CLOSEBLOCK;}
var                     {return VAR;}

{WHITESPACE}+           { /* Eliminate whitespace */ }
{COMMENT}               { /* Eliminate comments */ }
{QUOTED}                { return STRING; }
  /*
   TODO:

   Add the rest of the translation rules here.
   See the lexical structure definition of the modifie
   Also see the `%token` directives in parser.y for all

   Hint to get you started:
   The scanner returns STRING when matching the QUOTED
   When should the scanner return a NUMBER, IDENTIFIER,
   In which specific scenarios should the scanner retur
  */

  /* Unknown chars get returned as single char tokens *

{STRINGS}               { return STRING; }
{NUMBERS}               { return NUMBER; }
{IDENTIFIERS}           { return IDENTIFIER; }

.                       { return yytext[0]; }
%%
```

Figure 1: Code for the scanner.

2.2 Tree construction

*A **node_t** structure is defined in **include/tree.h**. In **tree.c**, complete the auxiliary functions **node_init** and **node_finalize** so that they can initialize/free "**node_t**"-sized memory areas passed to them by their first argument. The function **destroy_subtree** should recursively remove the subtree below a given node, while **node_finalize** should only remove the memory associated with a single node.*

See the code for what I have done. This was fairly straightforward, only concern I had was with the destroying of sub-trees. Since we destroy and free the node we call destroy on as well, I was a bit worried about the parent still having a pointer to it, and having a free-after-free issue. However, per now, it's only the root whose called with this, so it's fine for now.

2.3 Parser

*Complete the Yacc parser specification to include the VSL grammar, with semantic actions to construct the program's parse tree using the functions implemented above. The top-level production should assign the root node to the globally accessible **node_t** pointer root (declared in **src/tree.c**).*

This took a long time to do, lots of repetition, and a single type that sent me to oblivion for a handful of hours. Other than that it was fairly straightforward when being systematic and meticulous.

2.4 Testing your program

*Compile using make in the "**vslc/**"-folder, and run your compiled program using **src/vslc**. To get the created syntax tree printed by the **node_print** function, pass the **-t** option when running. The program takes input from standard in, and writes its output to standard out. In other words you can start the program and start typing VSL code into the terminal. Once a syntax error occurs, it will be printed, and execution will stop. Entering tokens one by one can be a useful way of discovering where the parser has bugs. Press **Ctrl+D** to indicate the end of input. Alternatively you can pass in input as a file like so: "**src/vslc -t < vsl_programs/ps2-parser/helloworld.vsl**" The folder **vslc/vsl_programs** contains a few example programs in VSL, which you can pass through your compiler. First run make in the "**vslc/**"-folder to build your compiler, and then run make in **vslc/vsl_programs** to parse all the **.vsl** files, and output **.ast** files. Running make check will compare the outputed files against the output of our suggested reference compiler. You don't need to match it completely, but you should understand why your output differs from ours. Keep in mind that simplifying the parse tree into an abstract syntax tree is part of the next problem set, so this exercise should stay true to the grammar.*

I had some trouble with the "make check" not working properly. When I checked the files manually one by one, they were correct, but the auto checking stated that there was some differences.