TDT4205 Compilers
Exercise 2

Stian Hvatum (hvatum)
MTDT

February 9, 2012

Part 1

Task 1.1

NT	FIRST(NT)	FOLLOW(NT)
E	i, n, p, m	$, a, m
S	i, n, p, m	$, r, a, m
T	i, n	$, r, a, m
F	i,n	$, r, a, m
I	p,m	i, n

Task 1.2

1.2.1

A top-down parser creates the parse tree from the root node and down. It has to predict the right outline of the tree before it can replace nodes with their productions. This is usualy implemented with guessing and back-tracking. A bottom-up parser generates the tree from the leaves, and the brances connects where they are supposed to on the way to the top. This is much easier to get right, and thereby a more powerfull method.

1.2.2

GNU Bison generates a bottom-up parser [1].

1.2.3

Writing compilers include a lot of complex both theory and practice. Writing it by hand is error-prone, and will most likely NOT be better than writing it by hand. Of course, if you are very well experienced in the field, hand writing may

[1] http://alumni.cs.ucr.edu/~lgao/teaching/bison.html

be an option. Known programs as `bash` and `Zend`[2] uses Bison-generated parser by today, and also GCC used to be backed up by a parser generated by GNU Bison, but changed in 2000[3].

Task 1.3

1.3.1

A shift-reduce parser is a bottom-up parser that keeps a buffer of input and a stack of grammar symbols. The shift-reduce parser has primarily two internal functions[4], shift and reduce. Shift takes one token from the input buffer and puts it on the stack. Reduce takes the content on the stack, and tries to reduce it using productions backwards. For each reduce, one grammatical step is done. Usually, we reduce until there is no more to reduce, and then we shift, though exceptions may occur if we use lookahead, and find it better to shift more in order to gain some better reductions on our input.

1.3.2

Shift-reduce conflicts are conflicts where we must decide wherever the correct outcome comes from a shift or a reduce, while reduce-reduce conflicts are conflicts where there are more than one possible reduction on the given input, and we have to decide wich one is the correct to use.

Figure 1: Example of a shift-reduce conflict
Given the following grammar:

```
stmt → if expr then stmt
     |  if expr then stmt else stmt
     |  other
```

If we now have

```
if expr then stmt
```

on the stack, we cannot know if there is best to reduce as normal, or if we should shift some more to see if there is an **else** comming, making it an `if expr then stmt else stmt` instead of a `if expr then stmt`.

[2]The PHP-parser
[3]http://gcc.gnu.org/ml/gcc/2000-10/msg00573.html
[4]acctually four, if you count “accept” and “error” as functions

Figure 2: Example of a reduce-reduce conflict

Given the following grammar:

```
stmt           → id ( parameter_list )
stmt           → expr :← expr
parameter_list → parameter_list , parameter_list
parameter_list → parameter
parameter      → id
expr           → id ( expr_list )
expr           → id
expr_list      → expr_list , expr_list
expr_list      → expr
```

If we now have

```
id ( id
```

on the stack, we have multiple allowed reductions. The difference to the last example, where we got a different valid reduction, is that we now have multiple valid reductions with the given data allready on the stack. Here, we can reduce the right most **id** to either *parameter* or to *expr*. We have to know if **id** is the entire struct is a *stmt* or a *expr* to decide this correctly. One solution to this problem, is to have a grammar where this cannot happen, and use a more sophisticated lexer that can tell the difference between a **id** that ends up as a *stmt* and a **id** that ends up as an *expr*.