TDDD55 Lesson 2 Top down parsing, Ambiguities, Left Recursion & Recursive descent

John Tinnerholm

LINKÖPING UNIVERSITY

Short recap of grammar

Agenda

- Hour I
  - Grammar
  - A short repetition of ambiguity
  - Left recursion
  - Top-down parsing
  - Hints for Lab 2
- Hour 2
  - Problem solving (See the exercises on the course homepage)

Context free grammar

- The context free grammar (CFG) for short consists of set of production rules
  - CFG:
    - <A> := <B>
    - <B> := <C>
  - Here our set contains two production rules
  - **:=** Is the same thing as **->** which you might see in other literature
    - Remember to clearly state and explain your notation and assumptions!
- The notation used in these slides will be the same as in the lab instructions

Some short notes related to left recursion

- Consider the following production rules and let's assume that it can be resolved somehow
  - <A> := <A><B> (1)
  - <B> := <C> (2)
  - …

```
//Production rule A
void A(void) {
A();
B();
}
```

```
//Production rule B
void B(void) {
C();
}
```

- **Intuition:** If we are going to determine the result of (1), we loop forever
- Left recursion might not always be obvious
  - Sometimes it can be indirect
  - See book for a general technique

Ambiguity & Elimination of direct left recursion

Hints Lab 2, the Grammar

```
<S> ::= <E> <end of line> <S> // Single Expression
   |   <end of line>        // No more input
<E> ::= <E> "+" <E>          // Addition
   |   <E> "-" <E>          // Subtraction
   |   <E> "*" <E>          // Multiplication
   |   <E> "/" <E>          // Division
   |   <E> "^" <E>          // Exponentiation
   |   "-" <E>              // Unary minus
   |   "(" <E> ")"          // Grouping
   |   id "(" <E> ")"       // Function call
   |   id                   // Symbolic constant
   |   num                  // Numeric value
```

Start by rewriting the grammar.
First eliminate ambiquity, once this is done eliminate left recursion! Keep your proposed grammar and send it along with your code.

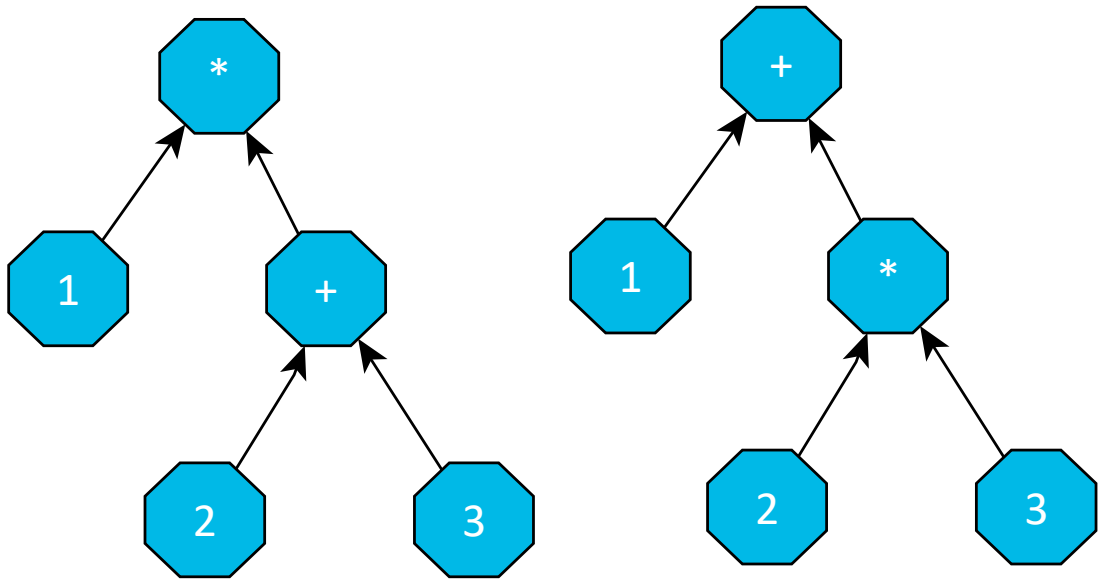
- ✓ Remember:
  - ✓ Rewrite the grammar, so that there is no ambiguity
  - ✓ Eliminate left recursion
  - ✓ Start coding (trial and error will not grant you luck)

Also it might be wise to consider a simpler grammar, and then try to generalize it

Hints Lab 2, A simplification of the grammar

<S> ::= <E> <end of line> <S> // Single Expression
| <end of line> // No more input
<E> ::= <E> "+" <E> // Addition
<E> ::= <E> "*" <E> // Multiplication
| num // Numeric value

<u>Consider 1+2*3</u>
The result should of course be **7**. How do we calculate the expression?
From the grammar to the left we can't really decide.
1 *(2+3) or 1 + (2 * 3) ☹



Hints Lab 2, Rewriting the simplified grammar

```
<S> ::= <E> <end of line> <S> // Single Expression
   |   <end of line>              // No more input
<E> ::= <E> "+" <E>             // Addition
<E> ::= <E> ”*" <E>             // Multiplication
   |   num                      // Numeric value
```

How to disambiguate?
Let’s say we want multiplication before addition here.
Solution:
We introduce a new nonterminal for each precedence level. **<F>** on the right also gives left associativity ☺ (no more additions can appear)
Note that we can not derive any more additions from **<F>. This grammar is for demonstration only!**

```
<S> ::= <E> <end of line> <S> // Single Expression
  |       <end of line>           // No more input
<E> ::= <E> "+" <F>             // Addition
  |       <F>
<F> ::= <F> "*" <G>             // Multiplication
  |       <G>
 <G> ::= num                    // Numeric value
```

An expression such as 1 + 2 * 3 will result in 1 + (2 * 3)

Hints for Lab 2, Eliminating left recursion

<exp>::= <term> | <exp> + <term>
| <exp> - <term>
<term>::= <factor> | <term> * <factor>
| <term> / <factor>
<factor>::= (<exp>) | <ident>
<ident>::= A | B | C ... | Z

Here we have only direct left recursion.
Thus we can use the general principle to eliminate direct left recursion (**X** and **Y** contains **NT's** and **T's**):

<A> **::=** <A>**<X>|<Y>**

For each of these replace them with two sets

<A> **::= <Y><A'>**
<A'> ::= <X> <A'> | ε

<exp> ::= <term> **<exp'>**
<exp'> ::= + <term>**<exp'>** | - <term>**<exp'>**
| ε
<term>::= <factor> **<term'>**
<term'> ::= * <factor>**<term'>** | / <factor>**<term'>**
| ε
<factor>::= (<exp>) | <ident>
<ident>::= A | B | C ... | Z

NT's = Nonterminals
T's = Terminals
Note this technique is not always applicable ☹

Hints for Lab 2, Recursive descent

- One method in the parser for each nonterminal

```
void S() {
...
}
```

```
void E() {
...
}
```

- Consider lookahead in the implementation!
- In the lab we are dealing with LL(1)

Predictive Parsing

- Like recursive descent. However, deterministic. No backtracking!
- Deterministic <=> Predictive
- Requires removal of common prefix, so we can decide upon what to do

NT/T	+	-	*
<B>	<C>	<error>	<error>
<C>	<error>	<D>	<error>
<D>	<error>	<error>	<empty>

The lookahead for the lab is 1. So **LL(1)**

Lab 2, Object-oriented overview

Hints for Lab 2

- Lab 2 consists of:
  - main.cc, you do not need to modify this
  - lex.hh, good to read, but you do not need to modify
  - lex.cc //See above
  - **lab2.hh**
  - **lab2.cc**
  - makefile, as usual

Lab 2, Class Overview

Scanner

int state;
int haveBuffered;
unsigned char bufferedChar;
char valueBuffer[kMaxTokenLength + 1];
long position;

void Reset(void);
void Accumulate(char);
void PutbackChar(unsigned char);
char *SymbolValue(void);
double NumberValue(void);
char GetChar(void);
Token Scan(void);

ScannerError

char errorCharacter;
char *message;
int state;

ParserEndOfFile

static int indent;
char *name;

ParserError

static int indent;
char *name;

Trace

void Recover(void);
double Parse(void);

Parser

void Recover(void);
double Parse(void);

There is currently no coupling between the different components. Think about what the Parser needs.
For instance, it might be wise to have an instance of the Scanner as a member variable in the Parser.

The Token Class

```
class Token
{
public:
   TokenType    type;
   double       numberValue;
   char         *symbolValue;

   char *Lookup(TokenType);
   char *Lookup(void);
   Token() : type(kUninitialized) {};
   Token(TokenType t) : type(t) {};
   Token(TokenType t, double x) : type(t), numberValue(x) {};
   Token(TokenType t, char *s) : type(t), symbolValue(s) {};
};
```

The Scanner Class

```
class Scanner {
   int             state;
   int             haveBuffered;
   unsigned char      bufferedChar;
   char            valueBuffer[kMaxTokenLength + 1];
   long            position;
   void Reset(void);
   void Accumulate(char);
   void PutbackChar(unsigned char);
   char *SymbolValue(void);
   double NumberValue(void);
   char GetChar(void);
public:
   Token Scan(void);
   Scanner() { haveBuffered = 0; Reset(); };
};
```

The TokenType enumeration

```
typedef enum
{
   kUninitialized = 0,
   kNumber = 1,
   kIdentifier = 2,
   kPlus = 3,
   kMinus = 4,
   kTimes = 5,
   kDivide = 6,
   kLeftParen = 7,
   kRightParen = 8,
   kAssign = 9,
   kEndMark = 10,
   kEndOfLine = 11,
   kPower
} TokenType;
```

The Trace class

```
class Trace {
   static int indent;
   char *name;
public:
   Trace(char *s) {
      name = s;
      cerr.width(indent);
      cerr << " ";
      cerr << "--> " << name << '\n' << flush;
      indent += 4;
   };
   ~Trace() {
      indent -= 4;
      cerr.width(indent);
      cerr << "";
      cerr << "<-- " << name << '\n' << flush;
   };
};
```

The Parser and Error Recovery

The parser class is unfished, make use of the available classes to finalize the construction of the parser. There are also two classes that are supposed to be used for error handling. **ParserError** and **ParserEndOfFile**

```
class Parser
{
public:
   void Recover(void);       // Error recovery routine
   double Parse(void);       // Main entry point
};
```

```
class ParserError {};
class ParserEndOfFile {};
```

To summarize

- Start small, and your design top-down
- Start coding when you have defined the grammar
- Remember to solve the exercises along with implementing the parser
- Play with your calculator ☺

Tutorial session

John Tinnerholm

www.liu.se

LINKÖPING UNIVERSITY

References

Hopcroft, J. E. (2008). *Introduction to automata theory, languages, and computation*. Pearson Education India.

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). Compilers, principles, techniques. *Addison wesley*, *7*(8), 9.