TDD55: Compilers and Interpreters

Lesson 1

Filip Strömbäck (filip.stromback@liu.se)
Department of Computer and Information Science
Linköping University

Purpose of lessons

- Practice theory
- Introduce the laboratory assignments
- Prepare for the final examination

Prepare by reading the lab instructions, the course book and lecture notes.

All instructions and material available in the course directory (~TDDD55/lab/) or on the course homepage.

Schedule

1. Formal languages and automata theory
2. Formal languages and automata theory, Flex
3. Bison and intermediate code generation
4. Exam preparation
5. Exam preparation

Laboratory assignments

- Goal: Get some practical experience in compiler construction
- 4 assignments to complete in 4x2 sessions → non-scheduled time required
  1. Attribute Grammars and Top-Down parsing
  2. Scanner Specification
  3. Parser Generators
  4. Intermediate Code Generation

Handing in and deadline

- Demonstrate during scheduled sessions
- Then, hand in code and answers to any questions in the assignment via e-mail. One e-mail per group, subject: TDDD55: Lab n. From your LiU-email.
- Deadline: 21st December
- Sign up in Webreg!

Skeleton

Copy to your home directory using:

```
cp -r ~TDDD55/lab .
```

- ~TDDD55
  - /lab
    - /doc
      - Documentation for the assignments.
    - /lab1
      - Contains all the necessary files to complete the first assignment
    - /lab2
      - Contains all the necessary files to complete the second assignment
    - /lab3-4
      - Contains all the necessary files to complete assignment three and four

Lab 1

- Some grammar rules given:
- Rewrite the grammar to LL(1)
- Add attribute rules to the grammar
- Implement the LL(1) grammar and the attributes in a C++-class named Parser. Parser shall contain a method Parse() which returns the value of a single statement in the language.

Lab 2

- Finish a scanner specification in Flex (scanner.l) by adding rules for comments, identifiers, integers and reals.
- More details in lesson 2.

Lab 3

- Finish a parser specification in Bison (parser.y) by adding rules for expressions, conditions, function definitions, etc.
- You also need to add error productions.
- More details in lesson 3.

Lab 4

- Generate intermediate code from a parse tree.
- Finish a generator for intermediate code by adding rules for some language statements.
- More details in lesson 3.

Lab 1: Problems in the given grammar

- Ambiguous
- Contains left recursion
- No operator precedence
- No operator associativity

Not suitable to a top-down approach

Lab 1: Rewriting the grammar

- Use one non-terminal for each precedence level:

  E -> E + E | E – E | T

  T -> T * T | T / T

- Associativity (left):

  E -> E + T | E – T | T

- See for example: *http://www.lix.polytechnique.fr/~catuscia/teaching/cg428/02Spring/lecture_notes/L03.html*

Lab 1: Rewriting the grammar

- The grammar so far is left-recursive and therefore not suitable for a top-down parser.
- Transform the grammar:

  A -> Aα | β (where b may not be preceded by A)

  Rewritten to

  A -> βA’

  A’-> αA’ | ε
- See *Lecture 5* for details

Lab 1: Attribute Grammars

- Define attributes for the productions
- Example:

```
S -> E            { display(E.val); }
E -> E1 + T       { E.val = E1.val + T.val; }
T -> T1 * F       { T.val = T1.val * F.val; }
F -> ( E )        { F.val = E.val; }
   |  num         { F.val = num.val; }
```

- See the course book and *Lecture 8* for details.

Lab 1: Implementation

- Given main function:

```
int main(void) {
    Parser parser; double val;
    while (1) {

        try {
            cout << "Expression: " << flush;
            val = parser.Parse();
            cout << "Result:     " << val << '\n' << flush;
        } catch (ScannerError& e) {
            cerr << e << '\n' << flush;
            parser.Recover();
        } catch (ParserError) {
            parser.Recover();
        } catch (ParserEndOfFile) {
            cerr << "End of file\n" << flush; exit(0);
        }
    }
}
```

Lab 1: Implementation

- lex.cc and lex.hh implement the lexer.
- The lexer reads from standard input
- No need to change anything in these files

Lab 1: Implementation

- lab1.cc and lab1.hh shall contain the Parser class
- The function Parse() is used to start parsing an expression.

```
double Parser::Parse() {
    Trace x(“Parse”);
    double val = 0;
    current_token = scanner.Scan();
    switch (current_token.type) {
    case kIdentifier:
    case kNumber:
    case kLeftParen:
    case kMinus:
        val = ParseP();
        if (current_token.type != kEndOfLine)
            throw ParserError();
        break;
    default:
        throw ParserError();
    }
    return val;
}
```

Lab 1: Implementation

- One function for each non-terminal in the grammar.
- Implement some simple error recovery in your Parser class.
- See *lecture 5* for details.