Project in Compiler Construction

1 Introduction

The project in Compiler Construction is normally carried out jointly by two students. Below, a "standard" project is described. Special projects are also possible, e.g., 3-person projects, use of other compiler tools, generation of other machine code, or implementation of another kind of language. Such special projects must be discussed and cleared with your supervisor in advance.

The standard project is to design and implement a compiler for a small procedural language. The compiler translates programs from source text to Intel assembly code.

The project is divided into three sub-tasks:

- Task 1: Design
- Task 2: Front end
- Task 3: Back end

There is a final deadline in June when all parts of the project must have been checked and approved by your project supervisor. Projects not approved by this deadline will have to be completed at the next instance of the course.

In the standard project, the same tools will be used as in the programming assignments (JavaCC, JJTree, JastAdd). If you wish to use some other tools this must be cleared with the supervisor beforehand. The supervisor might not be able to help you if you run into tool problems with nonstandard tools.

2 The language

You should implement a simple procedural language with the following features:

- Variables and literals of integer and Boolean types.
- Expressions: arithmetic expressions with addition, subtraction, multiplication, division, and unary negation. Relations with the usual operators (`==`, `!=`, `<`, `<=`, `>`, `>=`). Boolean expressions with the operators *and*, *or*, and *not*. Usual rules for associativity and precedence. Sub-expressions within parentheses. Function calls with zero or more parameters, and with and without return value.
- Strong typing: Variables, parameters, and functions with return values should have declarations stating their types (integer or boolean). Expressions should not allow mixing of types, e.g., adding an integer to a boolean should be a type error.
- Statements: assignments, some kind of loop (while, for, repeat, or similar), conditional statement.
- Built-in support for reading and writing integer values and for writing newlines. Your language could support this either by including predefined procedures, e.g.,

```
x = read(); writeint(x); writeln();
```

or by supplying special statements/expressions for this, e.g.,

```
x = readint; writeint x; writeln;
```

In your generated assembly code, these constructs will be implemented as calls to provided library procedures or functions in the C library.

- Procedure declarations with an arbitrary number of parameters. Procedures should be able to return a result value. Recursive calls should be allowed.
- Block structure with the possibility to nest procedures inside procedures, and the possibility for inner procedures to access variables in outer procedures and to call other visible procedures.
- The possibility to write comments in the programs.
- The compiler should report compile-time errors from name analysis and type checking.

In addition to this you may add more constructs. In that case, you should organize your work so that you implement the mandatory constructs first, and implement any additional constructs only if you find that you have time for that.

3 Test programs

You should test systematically with most test cases testing a single construct of the language. Your tests should include programs equivalent to those in the Appendix.

4 Tasks

Each task should be submitted to your supervisor by email. Supervisors and deadlines are specified on the project home page. You should suggest some times for an appointment. Attach a zip file with the full project. The project catalog should contain an index.html file with links to the different parts of the project. There is a template index file on the project home page.

The supervisor will make an appointment if it is required.

Task 1: Design

Define an abstract grammar for your language in an .ast file. The grammar should be designed in a good object oriented manner with simple and clear concepts and names.

Design the run time system for the language. You have to decide on the structure of activation records on the target machine, and how registers are going to be used. Test your design by translating the example programs `printx`, `printByRecursion`, `printNested`, and `printTwoNested` in the Appendix into Intel assembler the way you intend your compiler to do it. This means that all variables should reside in activation records and that non-local variables should be accessed via static links. Take some time to get used to the debugger graphical interface *ddd*.

Use *as* and *ld* or *gcc* to create executables and run them.

Update the index.html file before submitting.

Task 2: Front end

Decide on a concrete grammar and implement a parser. There should be a main program to print the abstract syntax tree. The parser may halt after reporting the location and type of the first error.

Do name analysis and report all missing and multiple declarations.

The compiler should report all type errors including any argument and return mismatch or missing return statements. All errors should be printed in a single list sorted by line number.

Implement intermediate code generation. You need to extend the ICode grammar. It should be possible to print generated code.

Update the index.html file before submitting.

Task 3: Back end

Generate assembly code.

Focus on doing code generation simple. If you are interested in optimizations it is recommended that you take the subsequent course on optimizing compilers.

If there are platform dependent tests they should be performed using a special `ant` target, e.g. `testintel`.

Update the index.html file before submitting.

Appendix: Assembly code generation tests

When generating Intel code it is convenient to be able to print values early, so your first tests should include programs equivalent to:

```
void empty() {
}
```

```
void print1() {
  writeint(1);
}
```

```
void printx() {
  integer x;
  x=1;
  writeint(x);
}
```

Proceed with arithmetic and boolean expressions, other statements, procedures with parameters and return values. Your tests for recursive procedures and block structure should include test cases equivalent to

```
void printByRecursion() {
  void printR(integer n); {
    if (n == 0) return;
    writeint(n);
    printR(n-1);
  }
  printR(5);
}
```

```
void printNested() {
  integer x;
  void innerProc() {
    writeint(x);
  }
  x = 1;
  innerProc();
}
```

```
void printTwoNested() {
  integer x;
  void p1() {
    p2();
  }
  void p2() {
    writeint(x);
  }
  x = 1;
  p1();
}
```

```
void printComplexNested() {
  integer x;
  void p1() {
    integer y;
    void p2() {
      x = x+1;
      p1();
    }
    y = 2;
    x = x+y;
    if (x==y) p2();
  }
  x = 0;
  p1();
  writeint(x-4);
}
```

```
void computeFactorial() {
  integer n, result;
  integer factorial(integer n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
  }
  n = readint();
  while (n >= 0) {
    result = factorial(n);
    writeint(result);
    writeln();
    n = readint();
  }
}
```

Your final tests should include some meaningful programs exercising different constructs in a more unsystematic way like `computeFactorial`.