Code Generation Project 2016: *Mini-Pascal compiler*

Implement a compiler for the Mini-Pascal programming language. The work has two parts: firstly, a front end making a full syntactic and semantic analysis of Mini-Pascal; secondly, a back end that generates target code. The language analyzer must correctly recognize and process all valid (and invalid) Mini-Pascal programs. It should report all syntactic and semantic errors, and then continue analyzing the rest of the source program. It must construct an AST and do a semantic analysis on this internal representation. If the given program was found free from errors, the back-end generates target code that can be subsequently executed. Note that you cannot use any automatic tools to generate target code: the task of this project is to *implement* the algorithms for such a tool.

Implementation requirements and grading criteria

You are expected to properly use and apply the compiler techniques taught and discussed in the course lecture materials and exercises. C# is used as the implementation language by default. Generating C as the target language is recommended as a reliable and probably the easiest choice (the option 1 below). However, you can choose an alternative target language among the following list of options, but these may need extra study and considerably more design work.

1. Simplified C, using only the lowest-level parts of C, as a sort of portable assembler. The restrictions are the following. (1) No structured control statements (such as, **if**, **while**, **for**) are allowed. Instead, use only **goto** statements or simple **if-goto** statements for altering the sequence of execution to other parts of the program. (2) Expressions must be in a very simplified form. Only one call operation **or** one primitive operation is allowed per expression. Expressions may not contain parentheses (but for a call to enclose its list of arguments) nor the conditional-expression operator ("**?:**").

2. The .NET *System.Reflection.Emit* namespace contains built-in API classes that allow a C# program to emit metadata and Microsoft common intermediate language (CIL) and optionally generate a PE file (.exe) on disk. These ready-made classes are generally useful for script engines and compilers.

3. Design and build your own software tools to generate CIL, either in symbolic or in binary form.

4. JBC (Java Byte Code), in binary format (i.e., a Java class file). JVM includes no official symbolic assembly code.

5. A target platform and target language of your choice. This may be a low-level target language (such as the LLVM intermediate representation) or a high-level programming language (such as *JavaScript*), or some other appropriate generally available target language.

The emphasis of the grading is the quality of the implementation: its overall architecture, clarity, and modularity. Pay attention to programming style and commenting. Grading of the assignment will consider (undocumented) bugs, level of completion, and its overall success (solves the problem correctly). The evaluation will particularly cover technical advice and techniques given by the course. You must make sure that your compiler system can be run and tested on the development tools available at the CS department.

Documentation

Write a report on the assignment, as a document in PDF format. The title page of the document must show appropriate identifications: the name of the student, the name of the course, the name of the project, and the date and time of delivery.

Describe the overall architecture of your language processor with UML diagrams. Explain the diagrams. Clearly describe the testing process, and the design of test data. Tell about possible shortcomings of your program (if well documented and explained they may be partly forgiven). Give instructions how to build and run your compiler. The report must include the following parts:

1. The Mini-Pascal token patterns as *regular expressions* or, alternatively, as *regular definitions*.
2. A *modified context-free grammar* that is more suitable for recursive-descent parsing, and techniques (backtracking or otherwise) used to resolve any remaining syntactic problems. These modifications must not affect the language that is accepted.
3. Specify *abstract syntax trees* (AST), i.e. the internal representation for Mini-Pascal programs. You can use UML diagrams or alternatively give a syntax-based definition of the abstract syntax.
4. *Language implementation-level decisions*. Many programming languages have left items (e.g. evaluation orders, or data representations for values) to an implementation. A language may also allow but does not require for a specific feature. For example, C and Java do not specify how numbers should be represented (can use efficient native machine-based values), or in what order some list of expressions are evaluated. Usually evaluation proceeds in left-to-right order, but an implementation may have the freedom to use whatever ordering it may prefer for optimizations. Identify any such relevant issues as related to Mini-Pascal and its definition, and specify the decicions made for your own implementation. Explain your choises.
5. *Semantic analysis*. Make a comprehensive list of all the semantics rules and checks needed for Mini-Pascal programs. You can use this list when you design your implementation and inputs for testing.
6. The major problems concerning *code generation*, and their solutions. What were the most problematic or demanding issues (language constructs, features, behaviour) when translating from the source language to the target language. Explain your solutions and discuss how well they worked out.
7. *Error handling* strategies and solutions used in your Mini-Pascal implementation (in its scanner, parser, semantic analyzer, and code generator).

For completeness, include this original project definition and the Mini-Pascal specification as appendices of your document. You can refer to them when explaining your solutions.

Delivery of the work

The final delivery is due at 23 o'clock (11 p.m.) on Sunday 22 of May, 2016. After the appointed deadline, the maximum points to be gained for a delivered work diminishes linearly, decreasing two (2) points per each hour late.

The project must be stored in a zip form. This zip should include all relevant files (including comprehensive sample of source and targets programs), contained in a directory that is named according to your unique department user name. The deliverable zip file must contain (at least) the following subfolders.

```
<username_ . . >
   ./doc
   ./src
```

Compress (zip) the whole folder and deliver its *address* to the exercise assistant.

When naming your project (.zip) and document (.pdf) files, always include your CS user name and the packaging date. These constitute nice unique names that help to identify the files later. Names would be then something like:

project zip: username_proj_2016_05_22.zip
document: username_doc_2016_05_22.pdf

When delivering by e-mail, send (cc) extra copies of the mail message to (1) yourself and (2) juha.vihavainen (at) helsinki.fi. You can then easily confirm that your mail message was really transferred and contained valid data. These extra copies also serve as backups and affirmations in case of any mix ups or failures in delivery. More detailed instructions and the requirements for the assignment are given in the exercise group sessions. If you have questions about the folder structure and the ways of delivery, or in case you have questions about the whole project or its requirements, please contact the teaching assistant Jiri Hamberg.