Compilers 2014-2015
Assignment 2: A C-Compiler

Naomi Christis

Office G0.28

February 11, 2015

For the project of the Compilers-course you will develop, in groups of 2 students (or alone, though this will make the assignment obviously more challenging), a compiler capable of translating a program written in a subset of C towards instructions for the P stack machine. The compiler must be written in Java. From the large gamma of parser/AST generators[6, 18] you will use the Java-based ANTLR tool. This tool converts a declarative lexer and parser specification (e.g. [9] and [10]) into Java code capable of constructing an explicit abstract syntax tree (AST) from a given C source file. The declarative parser specification consists of a grammar of the source language (C). The tree, which should be serialisable, should be traversed a number of times and Java code should be added in order to check input programs for semantic validity, apply optimizations and generate P code.

1 Assignment

Invoking the "mvn" command in your base dir should result in the generation of an executable named c2p[4]. This executable, when invoked with c2p c_prog p_prog, will start the compilation of an input C program c_prog into a P code file named p_prog.

1.1 The Source Language: a C subset

Since support for the full ANSI C [3] specification is unrealistic for the project, the assignment is limited to a subset of the language, based on Small-C[5], a language slightly smaller than what we require, but a useful starting point. In this section, we enumerate the features your compiler should support in order to pass the course and possibly earn extra grades. You are allowed to suggest extra features yourself. Inspiration can be found in Stroustrup's appendices [14].

1. Types (mandatory)
   Minimally, there should be support for the primitive data types *char, int*, and *pointer*.

2. Import (mandatory)
   The import of *stdio* should be supported (#include <stdio.h>). Furthermore, only the functions `int printf(char *format, ...)` and `int scanf(const char *format, ...)` must be supported, as defined in [12]: the "format" string allows interpretation of sequences of the form *%[width][code]*(width only in case of output). Provide support for at least the type codes *d, i, s* and *c*. You may consider the *char** types to be char arrays. *Flags* and *modifiers*, as described in [15], do not need to be supported. The behavior of *scanf* is well documented in the man pages [16, 11].

3. Reserved words (mandatory)
   The list of keywords for which compiler support is mandatory is shown in the feature matrix (BlackBoard). Based on Small-C, minimally *break, continue, else, if, return, while* and *for* must be supported. Add to these *const, char, int* and *void*. If you want to go beyond these, a full list can be found in [2, 17].

4. Constants, local and global variables (mandatory)

5. Comments (mandatory)
   Support for single line comments.

6. Functions
   This feature is mandatory and entails the definition and calling of functions and procedures, including support for passing parameters of basic types by value as well as by reference. Moreover, consistency of a return statement with the type of the enclosing function must be checked. You should also verify consistency between forward declarations and function definition signatures. As an optional feature, you can check whether all paths in a function body end with a return statement (not required for procedures that return void).

7. Arrays
   Array variables should be supported, as well as operations on individual array elements. Mind the correct use of dimensions and indices. Support for 1-dimensional static arrays is a mandatory feature; support for multi-dimensional arrays is optional, as well as dynamic arrays and assignments of complete arrays or array rows (in case of multi-dimensional arrays). An optional feature is the use of arrays as function and procedure parameters: it should be possible to pass these by reference (default approach in C) as well as by value. In order to discriminate between either mechanism, assume an array is passed by value when the variable name is used as actual argument (different from plain C!). To indicate that it should be passed by reference, use the & operator.

8. Conversions (optional)
   As a first extension you can support implicit conversions. The following order holds: `int isRicherThan char`.
   Implicit conversions of a richer to a poorer type (e.g. *int* to *char*) should cause a warning indicating possible loss of information. Another extension could be support for explicit casts (i.e. the cast operator). This enables the programmer to indicate he is aware of possible information loss. Hence the compiler should not yield a warning anymore.

9. Final Remark
   Use the Feature Matrix (cf. BlackBoard) as the reference for mandatory and optional features!

1.2 Error Analysis

The compiler is allowed to stop when it encounters a syntax error. An indication of the location of the syntax error should be displayed, but diagnostic information about the type of error is optional (and non-trivial). For semantical errors, it is necessary to output more specific information about the encountered error. For example, for usage of a variable of the wrong type, you might output: "[Error] line 54, position 13: variable x has type y while it should be z". When in doubt, the Gnu C Compiler [7] with options *ansi* and *pedantic* is the reference (except for the assumption made Sect. 1.1 on array parameter passing).

1.3 The Target Language: P

This language is the machine language of the virtual P stack machine from the course material, augmented with input, output and halt instructions. Documentation on this stack machine, as well as executables, can be found on the compilers website [1].

>>**Remarks for code generation from C to P:**

- **Initialisation of variables without initializer** By default, initialize variables without initializer to 0. This is also possible for char. Obviously this has a negative effect on performance, especially with arrays, which is why initialization is usually performed in a loop.
  Hence, it is possible, as optional optimization, not to implement default initialization of array elements and generate a warning when an array is read from before its elements have been initialized.
  Note that a warning should be generated for variables which are initialized with themselves (e.g. `int a = a`).

- **scanf and strings** For a `scanf("%s", ...)` statement, generate a loop of *in c* instructions. Exit the loop upon reading the escape character (ascii code 27).

1.4 Optimisations

Apart from correctness, some attention can be paid to the performance of your compiler, e.g. the runtime performance of the generated stack machine program, or the size of your target code in terms of primitive P instructions. Through static evaluation of constants, you can already obtain a large speedup. Apart from that, you might implement diverse "peephole optimisations". Such optimisations are optional; if you add any, do remember to carefully document which optimisations you developed and elaborate on the rationale.

1.5 Reference

When you want to compare certain properties (output, performance, ...) of your compiler to an existing compiler, the reference is the Gnu C Compiler [7] with options *ansi* and *pedantic*. When in doubt over the behavior of a piece of code (syntax error, semantical error, correct code, ...), GCC 4.6.2 is the reference. Apart from that, you can consult the ISO and IEC standards [8], although only with regards to the basic requirements and bearing in mind the difference in array parameter passing.

2 Tools

The framework of your compiler is generated by specialized tools:

- In order to convert your grammar to parsing code, you use ANTLR [13]. ANTLR has got several advantages compared to the more classic Lex/Yacc tools. On the one hand, your grammar specifications are shorter. On the other hand, the generated Java code is relatively readable.
- CAUTION: you should construct an **explicit** AST (not the one generated by ANTLR), and it should be serialisable such that it can be saved to file using the *c2p* option *—saveast*.

Make sure your compiler is platform independent. In other words, take care to avoid absolute file paths in your source code. Moreover, your compilation and test process should be controlled by the Apache Maven tool. Your compiler will be tested on the Radix student server. Make sure you regularly test your compiler on that platform.

3 Deadlines and Evaluation

Evaluation:

To fully test each group member's understanding of their compiler, evaluations will take place **individually**.

Deadlines:

The following deadlines are strict:

- By **February 20** you should send an e-mail with the members of your group (usually 2 people, recommended).
- By **March 29** (before the Easter holiday), you should be able to demonstrate that your compiler is capable of detecting lexical and syntactic errors, as well as constructing an AST from a valid input program. This corresponds to completing the parsing rules in the .g (grammar) file. Moreover, the AST and symbol table should be in use, at least partially.
  - Make sure your compiler has been thoroughly tested on a number of C files. Describe briefly (in the README file) which input files test which constructions and which Maven test targets launch your tests.
  - You should be able to demonstrate that you understand the relations between the different rules.
  - You should understand the role of a symbol table. Make sure you can indicate which data structure you will use and how this relates to the AST structure.
- By **April 24**, your compiler will have to detect and report semantical errors. By now, the grammar file, AST classes and symbol tables should be completed and in use.
  - Show that every rule instantiates an AST class.
  - Show which rules fill the symbol table and which rules read from it.
  - Show that the code generation phase will only require one more pass over your AST and the symbol table is superfluous in that phase.
- The final version of your project should be submitted **before start of exams**, **May 25**. The semantical analysis should be complete now, and code generation should be working. Indicate, in the README file, which optimizations you chose to implement.

No solutions will be accepted via e-mail; only timely submissions posted on BlackBoard will be accepted and assessed; no extensions of the deadline will be granted.

3.1 Groups

The members of your group should be reported by e-mail. Mention the names and student numbers of the group members. Students are graded based on the intermediary presentations as well as the exam.
Each student should demonstrate **individually** that he/she fully understands the software of the group and is able to identify :

- the cause of possible errors.
- which classes are involved in the implementation of new functionality.

3.2 Intermediary Evaluation: before Easter

You will have to give a small demonstration (individually). This will only take about 10 minutes of your time and a power-point is therefore not needed.
Cover the following points:

- Demo:
  - Demonstrate, by means of a couple of C examples, that the required syntactic and semantical errors can be detected and reported.
  - Explain which extras your compiler is capable of.
  - Run the C examples from blackboard (will be provided) and explain the results
- Architecture:
  - Which are the most important components of your compiler? Show some fragments of your grammar, show the documentation
  - How does everything work together in order to compile a program? Discuss the use of the symbol table.
  - Demonstrate that you master your compiler sufficiently in order to fix bugs and add new features. Recall that work submitted for grading must ultimately be your **own work**, reflecting your personal learning curve and performance. Cheating is a serious academic offense; we do not tolerate cheating, nor assisting others to do so.

3.3 Reporting

At each evaluation point, a version of your compiler should be submitted. Upload a zip file on blackboard which contains the following (if applicable):

- **Java Sources of the compiler**
- **ANTLR grammar**
- **Maven build file:**
  A single, valid *pom.xml* file will serve as the driver of your compiler. Any external dependencies used, should be retrieved automatically from publicly accessible Maven repositories. Include specific test cases for sample input programs, and include clear documentation in the README file (more instructions below).

- **C sample sources:**
  Demonstrate, using some test files, that faulty source files can be handled appropriately (describe the errors in specific statements using comments), and code can be generated form correct input files.
- **Features.html:**
  A filled out version of the feature matrix. Indicate which features are supported by your compiler by checking those cells. Note that the feature matrix is an indication for your grades, but the final marks are assigned based on your insights and understanding.
- **Readme.html:** Overview of appendices and instructions for testing the compiler. Indicate in a separate section which optional features were implemented.

3.4 Exam

The schedule for the final presentations will be available on blackboard and discussed with all groups. You can use the intermediary presentation as a starting point for the final presentation. In case you wish to report on the progress of your compiler at an earlier date than indicated, please let us know.

3.5 Questions

Questions will be answered via e-mail or Blackboard, and additional remarks may appear there. Keep in mind that a certain amount of time is needed to read and answer emails.
>> **Check BlackBoard on a regular basis!** <<

References

[1] http://ansymore.uantwerpen.be/courses/compilers.

[2] Peter Aitken and Bradley L. Jones. Teach yourself C in 21 days – appendix b. http://www.phy.hr/ matko/C21/apb/apb.htm.

[3] ANSI. ISO/IEC 9899:1999, Programming Languages – C. http://webstore.ansi.org.

[4] Apache. Maven. maven.apache.org/.

[5] Futch H. Egdares. Small-C. http://maestros.unitec.edu/ efutch/small-c.html.

[6] Fraunhofer Institute for Computer Architecture and Software Technology. Lexer and parser generators. http://catalog.compilertools.net/lexparse.html.

[7] Free Software Foundation. GCC home page. http://gcc.gnu.org/.

[8] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 9899:201x. http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1539.pdf.

[9] Jeff Lee and Ma Xiao. ANSI C grammar, Lex specification. http://www.lysator.liu.se/c/ANSI-C-grammar-l.html.

[10] Jeff Lee and Ma Xiao. ANSI C Yacc grammar. http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

[11] Linux Programmer's Manual and Panagiotis Christias. scanf (3). http://unixhelp.ed.ac.uk/CGI/man-cgi?scanf+3.

[12] A. D. Marshall. I/O: stdio.h. http://www.cs.cf.ac.uk/Dave/C/node18.html.

[13] T. Parr et al. Another tool for language recognition (antlr), 2005. http://www.antlr.org/.

[14] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[15] The C++ Resources Network. printf. http://www.cplusplus.com/ref/cstdio/printf.html.

[16] The C++ Resources Network. printf. http://www.cplusplus.com/reference/clibrary/cstdio/scanf/.

[17] WikiPedia. C syntax. http://en.wikipedia.org/wiki/C_syntax#Reserved_keywords.

[18] WikiPedia. Comparison of parser generators. http://en.wikipedia.org/wiki/Comparison_of_parser_generators.