LDTA 2011 Tool Challenge

at the
11th Workshop on Language Descriptions, Tools, and Applications (LDTA 2011)
in
Saarbrucken, Germany
March 26 & 27, 2011
www.ldta.info

The LDTA 2011 Tool Challenge is a community-building exercise to get tool developers working on the same problem, but with different tools and techniques. The goal is to create a discussion that fosters a better understanding, among tool developers and tool users, of relative strengths and weaknesses of different language processing tools, techniques, and formalisms.

LDTA 2011 Tool Challenge Problem Set

The LDTA 2011 Tool Challenge Problem Set is a collection of language processing tasks to be applied to a simple, but evolving, collection of related imperative programming languages. The individual problems in the Problem Set can be viewed as points in a two dimensional space: the first dimension specifying language processing tasks and the second dimension indicating the languages to which these tasks are to be applied. The task dimension consists of several traditional language processing tasks such as parsing, pretty printing, semantic analysis, optimization, and code generation. The language dimension consists of a simple, but evolving, set of imperative programming languages. These are all variations on Oberon0 (a small derivative of Pascal, Modula-2, and Oberon) used as an example language in Nicolas Wirth's book "Compiler Construction" (see URL below).

Note that not all aspects of most of the problems are fully and precisely defined; this is intentional. The purpose of the Tool Challenge to better understand how various tools, techniques, and formalisms can be applied to a variety of language processing tasks and this can be accomplished even if some variation to the problems exists.

Furthermore, tool developers should not feel that they must complete all of, or even a majority of, the tasks for all of the languages in order to participate in the challenge or to present their results at the workshop. Participants should look to demonstrate the unique qualities of their tool or technique and if this can be done on a subset of the tasks for the simple core languages then that is also encouraged. Thus, participants may attempt to fill in all points in the 2 dimensional problem space or focus on those that they find most interesting and applicable to their tools and techniques.

Tasks dimension

The tasks to be completed for the various languages are all traditional compiler tasks for imperative programming languages. These are discussed in most compiler construction texts, including Wirth's, and have been used in other problem sets such as the TIL Chairmarks created by James Cordy and Eelco Visser.

T1. Parsing and Pretty Printing.
Build a tool that reads in programs, constructs a parse tree (implicitly or explicitly), and then

pretty-prints the parse tree.

T2. Name binding.
Build a tool that reads in syntactically valid programs and binds all name uses to their declarations. If some names are not declared or declared more than once an appropriate error message should be displayed.

T3. Type checking.
Check that no type errors occur in the program; if any do, display appropriate error messages. Solutions should aim at providing informative, helpful error messages at an appropriate abstraction level.

T4. Source-to-source transformations.
There are two source-to-source transformation tasks:
T4.a. The first asks to what extent can language features introduced in extended versions of the language be seen as extensions that can be "de-sugared" or reduced to language constructs in a previously implemented version of the object language? For example, how can control flow constructs introduced in language L2 (see below) be transformed into other constructs already implemented in L1.
T4.b. The second task is to apply a number of traditional optimizations. These are constant propagation, dead code elimination, common sub-expression elimination, and strength reduction.

T5. Code generation.
There are two code generation tasks:
T5.a. The first is to translate Oberon0 to ANSI C; this should be straightforward as Oberon0 and C share many constructs.
T5.b. The second task is to translate to a lower-level language: Wirth's DLX, a simple RISC assembly language also described in his text. Participants may, alternatively, apply the optimizations in task T4.b to the generated DLX code.

Language dimension

The object languages to be implemented in this challenge are all based on Oberon0. This language is defined in Nicolas Wirth's textbook Compiler Construction (available at http://www-old.oberon.ethz.ch/WirthPubl/CBEAll.pdf). Chapter 6 (p. 30-31) defines the basic syntax of Oberon0 and subsequent chapters describe its semantics.

We define a series of languages, in most cases, each building on the previous one:

- **L1:** Oberon0 without procedures and with only primitive types (no arrays or records).
- **L2**: created by adding additional control structures to L1. Add a Pascal-style for-loop and a Pascal-style case statement.
- **L3**: created by adding Oberon0 procedures to L2.
- **L4**: created by adding Oberon0 arrays and records to L3.
- **L5:** created by adding a participant-defined notion of pointers to L4. This requires a new type expression as well as operators to take an address and to de-reference a pointer. The precise syntax of these can be based on Oberon, Pascal, C, or otherwise defined by the participant.

Choosing problems to solve

The task dimension specifies 5 tasks and the language dimension specifies 5 languages, but participants are not expected to complete 25 different software artifacts to solve each of these 25 problems. The languages are organized such that each is a subset, in terms of syntactic constructs, of the next language in the sequence. Thus, in many cases, a software artifact generated to solve a specific task for language L4 is also a solution for that task on languages L1, L2, and L3.

Furthermore, as stated above, participants need not solve all problems; they should choose problems in the grid that best exemplify the characteristics and features of their tool or technique.

By specifying simple languages such as L1 and L2 we aim to provide an easy path to participation in the tool challenge. While we hope that participants will provide task solutions to the more complex languages they should not feel that they need to solve all the tasks on all the languages in order to participate.

Another reason for specifying the object languages as a sequence of extended languages is to give participants the opportunity to show how language/task solutions can be developed in a modular manner in which extensions are added to an existing language/task solution to create another.

To enable some compatibility of software artifacts from the participants and to foster the sharing of sample programs and test cases, we provide the following list of suggested software artifacts to be completed:

- Artifact 1: Task T1 on language L3.
- Artifact 2: Task T1 on language L4.
- Artifact 3: Task T2 and T3 on language L3 or L4.
- Artifact 4: Task T4 on language L3 or L4.
- Artifact 5: Task T5 on language L3 or L4.

Participant defined contributions

Participants are encouraged to go beyond this problem set and include additional language features, processing tasks, or other components that specifically highlight interesting or special aspects of their tools and techniques.

For example, task 1 may be extended so that comments and layout are maintained in the pretty-printing of the input program or a second concrete syntax may be developed that has a more modern look-and-feel, perhaps something more akin to C, Java, or Haskell. Participants may also consider developing some sort of integrated development environment support for the languages. In keeping with the goal of tool generation, this support should be generated from the (possibly annotated) language specification.

The goal of these contributions should be to highlight special capabilities of the tool that may not be as visible on the predefined tasks.

Community Building

The community building envisioned by this Tool Challenge does not need to wait until the workshop begins. A "Google group" has been set up to support discussion of the challenge

problems and to encourage the sharing of test programs. If you have questions about the challenge or about specific problems please ask them there. The web page for this group is: http://groups.google.com/group/ldta-2011-tool-challenge.

Submission of abstracts and intention to participate

The Tool Challenge is open to developers of all kinds of grammarware tools and techniques. To participate, tool developers must submit the following by March 5, 2011:

- Names of participants and the name of their tool or technique.
- Presentation title and abstract. The short abstract should specify on what aspects of the problem set the tool was applied, where it excelled and where no solution was offered and/or the solution was considered less than optimal. We expect these to be only a few paragraphs in length.

This information is used for scheduling purposes and is not used for evaluation; as all tool developers interested in participating are welcome and will be given an opportunity to present their solution at the workshop. Submission of this information indicates a commitment to attend LDTA and to participate in the workshop. This information will be listed in the program.

Authors of submissions that appear to be outside of the scope of LDTA will be contacted to discuss the relevance of their work to the workshop. Tool developers who question whether their work falls with the scope of LDTA are encouraged to contact the PC chairs early on for clarification.

Joint Tool Challenge Paper

After the workshop a joint paper will be written by participants and submitted to a journal, most likely Science of Computer Programming. It is separate from the proceedings of the workshop and any special journal issue for the workshop.

LDTA Rubric

In preparing the presentation of one's solution for the workshop, participants are strongly encouraged to provide an analysis of their solutions based on the following criteria. The intent is to find common language for discussing the quality of the various solutions.

- ease of specification: Are the specifications declarative and at an appropriately high level of abstraction?
- analysis of specifications: Does the tool perform any domain specific analysis over the specifications to detect errors or improve performance?
- performance: Does the tool process the specifications in a reasonable amount of time and space? Does the generated language processing tool (*e.g.* a generated parser or code implementing an attribute grammar) run efficiently in time and space?
- flexibility, extensibility: How easy is it to modify and extend the language specification?
- modularity: Code reuse is critical in mainstream applications and languages - to what degree does the tool or technique support this in the evolving set of languages?
- quality of error messages: Do errors in the language specification result in error messages that are understandable and informative? Do errors in programs processed by generated tools result in good error messages (to the extent that the generating tools have some effect on this)?
- ease of use: Overall, is the tool easy to use?