COCOM tool set

Vladimir Makarov, vmakarov@users.sourceforge.net Sep. 10, 2004

This document describes COCOM tool set oriented towards the creation of compilers, cross-compilers, interpreters, and other language processors. COCOM tool set is oriented towards the creation of compilers, cross-compilers, interpreters, and other language processors. Now COCOM tool set consists of the following components:

- Ammunition (reusable packages)
- Sprut (internal representation description translator)
- Nona (code selector description translator)
- Msta (syntax description translator)
- Oka (pipeline hazards description translator)
- Shilka (keywords description translator)

All of these components are written in ANSI C and have common style input languages (a la YACC). All code generated by the components is in also strict ANSI C and in standard C++. All documentation exists in ASCII, TeX dvi, Postsrcipt, HTML, and GNU info formats.

1. Ammunition (reusable packages)

Currently there are the following packages:

allocate

Allocating and freeing memory with automatic fixing some allocation errors.

vlobject

Work with variable length objects (VLO). Any number of bytes may be added to and removed from the end of VLO. If it is needed the memory allocated for storing variable length object may be expanded possibly with changing the object place. But between any additions of the bytes (or tailoring) the object place is not changed. To decrease number of changes of the object place the memory being allocated for the object is longer than the current object length.

objstack

Work with stacks of objects (OS). Work with the object on the stack top is analogous to one with a variable length object. One motivation for the package is the problem of growing char strings in symbol tables. Memory for OS is allocated by segments. A segment may contain more one objects. The most recently allocated segment contains object on the top of OS. If there is not sufficient free memory for the top object than new segment is created and the top object is transferred into the new segment, i.e. there is not any memory reallocation. Therefore the top object may change its address. But other objects never change address.

hashtab

Work with hash tables. The package permits to work simultaneously with several expandable hash tables. Besides insertion and search of elements the elements from the hash tables can be also removed.

The table element can be only a pointer. The size of hash tables is not fixed. The hash table will be automatically expanded when its occupancy will became big.

position

Work with source code positions. The package serves to support information about source positions of compiled files taking all included files into account.

errors

Output of compiler messages. The package serves output one-pass or multi-pass compiler messages of various modes (errors, warnings, fatal, system errors and appended messages) in Unix style or for traditional listing. The package also permits adequate error reporting for included files.

commline

Work with command line. The package implements features analogous to ones of public domain function 'getopt'. The goal of the package creation is to use more readable language of command line description and to use command line description as help output of program.

ticker

Simultaneous work with several tickers (timers).

bits

Work with bit strings (copying, moving, setting, testing, comparison).

arithm

Implementing host machine-independently arbitrary precision integer numbers arithmetic. The implementation of the package functions are not sufficiently efficient in order to use for run-time. The package functions are oriented to implement constant-folding in compilers, cross-compilers.

IEEE

Implementing host machine-independently IEEE floating point arithmetic. The implementation of the package functions are not sufficiently efficient in order to use for run-time. The package functions are oriented to implement constant-folding in compilers, cross-compilers.

earley

The package 'earley' implements earley parser. The earley parser implementation has the following features:

- It is sufficiently fast and does not require much memory. This is the fastest implementation of Earley parser which I know. The main design goal is to achieve speed and memory requirements which are necessary to use it in prototype compilers and language processors. It parses 30K lines of C program per second on 500 MHz Pentium III and allocates about 5Mb memory for 10K line C program.
- It makes simple syntax directed translation. So an abstract tree is already the output of Earley parser.
- It can parse input described by an ambiguous grammar. In this case the parse result can be an abstract tree or all possible abstract trees. Moreover it produces the compact representation of all possible parse trees by using DAG instead of real trees. This feature can be used to parse natural language sentences.

- It can parse input described by an ambiguous grammar according to the abstract node costs. In this case the parse result can be an minimal cost abstract tree or all possible minimal cost abstract trees. This feature can be used to code selection task in compilers.
- It can make syntax error recovery. Moreover its error recovery algorithms finds error recovery with minimal number of ignored tokens. It permits to implement parsers with very good error recovery and reporting.
- It has fast startup. There is no practically delay between processing grammar and start of parsing.
- It has flexible interface. The input grammar can be given by YACC-like description or providing functions returning terminals and rules.
- It has good debugging features. It can print huge amount of information about grammar, parsing, error recovery, translation. You can even output the result translation in form for a graphic visualization program.

Current state: Implemented, documented, and tested. All these packages have been used in several products.

Under development: Design of some reusable packages for compilers.

2. SPRUT (internal representation description translator)

SPRUT is a translator of a compiler internal representation description (IRD) into Standard Procedural Interface (SPI). The most convenient form of the internal representation is a directed graph. IRD defines structure of the graph. SPI provides general graph manipulating functions. The defined graph nodes can be decorated with attributes of arbitrary types.

IRD declares types of nodes of the graph. Nodes contains fields, part of them represents links between nodes, and another part of them stores attributes of arbitrary types. To make easy describing internal representation the IRD supports explicitly multiple inheritance in node types. There can be several levels of internal representation description in separate files. The nodes of one level refer to the nodes of previous levels. Therefore each next level enriches source program internal representation. For example, the zero level representation may be internal representation for scanner, the first level may be internal representation for parser, and so on.

SPI can contains functions to construct and destroy graphs and graph nodes, to copy graphs or graph nodes, to read and write graphs or graph nodes from (to) files, to print graphs or graph nodes, to check up constraints on graph, to traverse graphs, and to transform acyclic graphs in some commonly used manners. SPI can also check up the most important constraints on internal representation during work with node fields. SPI can automatically maintain back links between internal representation nodes.

Using SPRUT has the following advantages:

1. brief and concise notation for internal representation
2. improving maintainability and as consequence reliability of the compiler
3. user is freed from the task of writing large amounts of relatively simple code

Current state: Implemented, documented, and tested. SPRUT has been used in several products (the biggest one is extended Pascal cross-compiler with moderate optimizations and with 3 different internal representations).

3. NONA (code selector description translator)

NONA is a translator of a machine description (MD) into code for solving code selection and possibly other back-end tasks. The machine description is mainly intended for describing code selection task solution, i.e. for determining by machine-independent way a transformation of a low level internal representation of source program into machine instruction level internal representation. But the machine description can be used also to locate machine dependent code for solving other back-end task, e.g. register allocation. To describe machine description a special language is used.

An machine description describes mainly tree patterns of low level internal representation with associated costs and semantic actions. NONA generates the tree matcher which builds cover of low level internal representation by the tree patterns with minimal cost on the first bottom up pass and fulfills actions associated with the chosen tree patterns on the second bottom up pass. Usually the actions contain code to output assembler instruction.

Analogous approach for solving code selection task is used by modern generator generators such as BEG, Twig, Burg and Iburg. The tree matcher generated by NONA uses algorithm similar to one of BEG and Iburg, i.e. the algorithm is based on dynamic programming during fulfilling code selection.

Although the algorithm used by BURG and based on dynamic programming during tree pattern matcher generation time is considerably more fast, it is not acceptable for us. Its main drawback which is to need usage of less powerful machine description results in necessity of usage of more machine-dependent low level internal representation. For example, the special internal representation node types for 8-bits, 16-bits constants besides 32-bits constants would be needed. Also the algorithm used by BURG is considerably more complex.

Tree pattern matchers generated by NONA also can work with directed acyclic graphs besides trees. This feature is useful when target machine instruction is generated from the internal representation which is result of some optimizations such as common sub-expression elimination.

Current state: Implemented, documented (only plain text), and tested. NONA has been used in several products (the biggest is extended Pascal cross-compiler for superscalar RISC processor 'AMD 29500' with moderate optimizations).

Under development: Additional generation of the tree pattern matcher based on dynamic programming during generation of the tree pattern matcher. Pascal implementation experience shows that time of the tree pattern matcher work is practically the same as the time of all front-end work.

4. MSTA (syntax description translator)

The MSTA can emulate YACC (Posix standard or System V Yacc). The MSTA have the following additional features:

- Fast LR(k) and LALR(k) grammars (with possibility resolution of conflicts). Look ahead of only necessary depth (not necessary given k). Originally LALR(k) parsers are generated by modified fast DeRemer's algorithm. Parsers generated by MSTA are up to 50% faster than ones generated by BISON and BYACC but usually have bigger size.
- Extended Backus-Naur Form (EBNF), and constructions for more convenient description of the scanners. More convenient naming attributes.
- Optimizations (extracting LALR- and regular parts of grammars and implementing parsing them by adequate methods) which permit to use MSTA for generation of effective lexical analyzers. As consequence MSTA permits to describe easily (by CFG) scanners which can not be described by regular expressions (i.e. nested comments).

- More safe error recovery and reporting (the 1st additional error recovery method besides error recovery method of YACC).
- A minimal error recovery and reporting (the 2nd additional error recovery method besides error recovery method of YACC).
- Fast generation of fast parsers.

Current state: Implemented, documented, and tested. Now MSTA is stable. More verbose documentation is needed.

MSTA uses several methods (parser optimizations) nowhere described.

5. OKA (pipeline hazards description translator)

OKA is a translator of a processor pipeline hazards description (PHD) into code for fast recognition of pipeline hazards. A pipeline hazards description describes mainly reservations of processor functional units by an instruction during its execution. The instruction reservations are given by regular expression describing nondeterministic finite state automaton (NDFA). All analogous tools are based only on deterministic finite state automaton (DFA).

OKA is accompanied with the scheduler on C and C++ for scheduling basic blocks.

Current state: Implemented, documented, and tested. OKA has been used in experimental C/C++ compiler for Alpha.

6. SHILKA (keywords description translator)

SHILKA is oriented to fast recognition of keywords and standard identifiers in compilers. SHILKA is analogous to GNU package 'gperf' but based on minimal pruned O-trie which can take into account the frequency of keyword occurrences in the program. Gperf can not make it. SHILKA is up to 50% faster than Gperf. SHILKA is also simpler than Gperf in the usage.

Current state: Implemented, documented, and tested.

7. DINO interpreter

DINO is high level scripting dynamic-typed language. DINO is oriented on the same domain of applications as famous scripting languages perl, tcl, python. The most of programmers know C language. Therefore Dino aims to look like C language where it is possible. Dino is an object oriented languages with garbage collection. Dino has possibilities of parallelism description and exceptions handling. Dino is an extensible language with possibility of dynamic load of libraries written on other languages. The high level structures of Dino are

- heterogenous extensible arrays
- extensible associative tables with possibilities of deleting table elements
- objects

Originally, Dino was used in a russian graphics company ANIMATEK for description of movement of dinosaurs in an project. It has been considerably redesigned and implemented with the aid of COCOM tool set.

In future, I am going to implement debugger and port a GUI library into DINO. Any help in this will be appreciated. I am also going to use DINO for compiler prototyping.

Current state: Implemented, documented, and tested. Experimental status.