Real Compilers, Tools, and Systems

Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

8. Februar 2008
“Standard” Structure

source(text) → lexical analysis(7) → tokenized-program → syntax analysis(8) → syntax-tree → semantic-analysis(9) → decorated syntax-tree → optimizations(10) → intermediate representation → code-generation(11, 12) → machine-program

- finite automata
- pushdown automata
- attribute grammar evaluators
- abstract interpretation + transformations
- tree automata + dynamic programming + ···
Issues

▶ Used tools
▶ Compiler systems
▶ Development goals:
  ▶ Compiler speed
  ▶ Target code efficiency
  ▶ Portability
  ▶ Retargetability
▶ Where does the development go?
Lexical and Syntactical Analysis

- Lexical analyser generators:
  - lex (lexical analyzer generator, by M.E. Lesk and E. Schmidt, AT&T)
  - flex (fast lexical analyzer generator, by Vern Paxon, ~1987)
  - other derivatives: aflex, quex, ...

- Syntactical analyser generators:
  - yacc (yet another compiler compiler, by S. C. Johnson, AT&T)
  - bison (GNU version of yacc)
  - others: Berkeley yacc, MKS yacc, Abraxas pcyacc, ...
Semantic Analysis

▶ Attribute grammar systems:
  ▶ GAG (Karlsruhe, Paderborn),
  ▶ OPTRAN (UdS), incl. tree pattern matching, transformation, attribute reevaluation
  ▶ Cornell Program Synthesizer Generator (commercial), incl. reevaluation
  ▶ FNC-2 (INRIA)
  ▶ ELI (Boulder, Paderborn)

▶ Natural semantics based systems:
  ▶ CENTAUR (INRIA, CWI),
  ▶ RML (Linköping),
  ▶ TwoBig (UdS).
Static Program Analysis

- **Abstract Interpretation-based systems:**
  - Program Analyzer Generator (PAG) (UdS), integrated into many products of AbsInt, see http://www.program-analysis.com/ and http://www.absint.com
  - Spare (Madison), in use?
  - Sharlit, part of SUIF.

- **Constraint-based systems:**
  - BANE, The Berkeley ANalysis Engine, see http://www.eecs.berkeley.edu/Research/Aiken/bane.html
Code Generators

Code selector generators, (mostly) with on-the-fly register allocator:

- BEG (developed in Karlsruhe, integrated into CoSy), a code-selector generator based on regular tree grammars, see http://www.hei.biz/beg/index.html
- 1burg, iburg (Bell Labs, U. Arizona), implementations of code selectors based on bottom-up tree rewriting system, see http://www.cs.princeton.edu/software/iburg/
Toolkits

Set of (generator) tools with defined interfaces:

- ELI (Boulder, Paderborn), last publication 1998
- Cocktail (Karlsruhe) (commercial), last update 2005
- Many tools sold as “Compiler Generators” in fact only offer Lexer and Parser generators, e.g., Coco/R (Linz), Jack (FH Bielefeld)

List of compiler construction tools:
http://catalog.compilertools.net/.
Compiler Systems

Modular design with predefined or definable data structures and interfaces, often together with a set of generators for compiler modules:

- CoSy (ACE)
- SUIF (Stanford University)
CoSy

(ACE), developed by the Compare Consortium, see http://www.ace.nl/compiler/cosy.html.

*Based upon its highly modular design, surrounding a generic, extensible intermediate representation (IR) and extensive use of generators, the CoSy environment enables construction of production-quality performance compilers in a highly efficient manner, reducing time-to-market, time-to-performance, as well as development and maintenance costs.*
SUIF

(Stanford University), see http://suif.stanford.edu/, most recent update 2005, formerly part of the National Compiler Infrastructure (US).

**Scanning, Parsing, and Rewriting Kit: a “little language framework” that supports the construction of language processors in Python.** The SUIF compiler is an infrastructure designed to support collaborative research in optimizing and parallelizing compilers. Independently developed compilation passes work together by using a common intermediate format to represent programs.
EDG Frontends


The front end does complete syntax and semantic analysis, including complete error checking. ... The front end translates source programs into a high-level, tree-structured, in-memory intermediate language. The intermediate language preserves a great deal of source information (e.g., line numbers, column numbers, original types, original names), which is helpful in generating symbolic debugging information as well as in source analysis and transformation applications. Implicit and overloaded operations in the source program are made explicit in the intermediate language, but constructs are not otherwise added, removed, or reordered. The intermediate language is not machine dependent (e.g., it does not specify registers or dictate the layout of stack frames).
lcc: A Retargetable Compiler for ANSI C

- Production C compiler by Christopher Fraser, David Hanson,
- Goal: A C compiler for teaching purposes, portability, retargetability, simple design, moderate optimizations (only local not global), fast execution
- Respects calling conventions for the targets
- Not using some tools to “avoid the dependence on them”, seen as a problem when the design started
Compiler Structure and Interfaces

- Imported preprocessor
- Lexing with `lex`-generated scanner
- Parsing with a recursive descent parser, combined with construction and decoration of abstract syntax trees, type checking, introduction of conversions
- Conversion to DAGs, eliminating common subexpressions
- Control flow representation in a “code list”, the interface between front end and back end
- Instruction selection done by an `1burg` generated code selector (code as annotations on the DAGs in the code list)
- Three pass register allocator using a simple heuristics for spills
- Generated code is assembled and linked by UNIX tools
gcc: GNU C Compiler

- Production quality compiler, in widespread use
- Open source software project
- Developed by many (voluntary) developers (Free Software Foundation, FSF)
- Frontends for various programming languages
- Generates code for a multitude of architectures
- Many optimizations: Copy propagation, Constant propagation, Dead code elimination, ...
- Difficult maintenance (has improved with recent releases)
Supported Programming Languages

The GNU Compiler Collection supports the following programming languages:

- C (gcc)
- C++ (g++)
- Java (gcj)
- Ada (gnat)
- Objective-C
- Objective-C++
- Fortran (GFortran)

Available, but not included in the standard are: Modula-2, Modula-3, Pascal, PL/I, D (dgcc), Mercury, VHDL (GHDL).
## Supported Architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Vendor</th>
<th>Processor</th>
<th>Technology</th>
<th>CPU Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>68HC11</td>
<td>ETRAX</td>
<td>MCORE</td>
<td>PowerPC</td>
<td></td>
</tr>
<tr>
<td>A29K</td>
<td>FR-30</td>
<td>MIPS</td>
<td>R8C/M16C/M32C</td>
<td></td>
</tr>
<tr>
<td>Alpha</td>
<td>FR-V</td>
<td>MMIX</td>
<td>ROMP</td>
<td></td>
</tr>
<tr>
<td>ARC</td>
<td>H8/300</td>
<td>MN10200</td>
<td>SPARC</td>
<td></td>
</tr>
<tr>
<td>ARM</td>
<td>HC12</td>
<td>MN10300</td>
<td>Stormy16</td>
<td></td>
</tr>
<tr>
<td>AVR32</td>
<td>IA-32 (x86)</td>
<td>Motorola 68000</td>
<td>SuperH</td>
<td></td>
</tr>
<tr>
<td>Blackfin</td>
<td>IA-64</td>
<td>Motorola 88000</td>
<td>System/390/zSeries</td>
<td>V850</td>
</tr>
<tr>
<td>C4x</td>
<td>Intel i960</td>
<td>NS32K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D30V</td>
<td>IP2000</td>
<td>PA-RISC</td>
<td>x86-64</td>
<td></td>
</tr>
<tr>
<td>DSP16xx</td>
<td>M32R</td>
<td>PDP-11</td>
<td>Xtensa</td>
<td></td>
</tr>
</tbody>
</table>

...
Compiler Structure

- Separate preprocessor (cpp)
- Hand-written lexer
- yacc-generated parser that constructs parse tree
- Parse tree is converted into an intermediate representation called register transfer language (RTL)
- RTL is a LISP-like generic assembly language (see http://gcc.gnu.org/onlinedocs/gccint/RTL.html)
  \[
  (\text{set}:\text{SI} (\text{reg}:\text{SI} 140) (\text{plus}:\text{SI} (\text{reg}:\text{SI} 138) (\text{reg}:\text{SI} 139)))
  \]
- RTL description is generated out of (complex) machine description
- The idea behind RTL was first described in: Davidson and Fraser, “The Design and Application of a Retargetable Peephole Optimizer”, ToPLaS v2(2), 191-202, April 1980
Compiler Structure (pre 4.x)

1. **Source Code** → **Parser** → **Syntax Tree**
2. **Syntax Tree** → **RTL Generator** → **RTL** → **Optimizer** → **Optimized RTL** → **Code Generator** → **Assembler**
3. **Optimized RTL** → **Assembler**
Compiler Structure (recent)
Compiler Structure (cont’d)

- GENERIC allows language-independent representation of functions in trees
- GIMPLE is a reduced form of GENERIC that is suitable for language- and target-independent optimizations
  - Expressions are broken down into 3-address form
  - Control structured are lowered to gotos
  - Optimizations: inlining, constant propagation, tail call elimination, redundancy elimination
- RTL is used (as before) to perform machine-dependent optimizations and to steer code selection
GIMPLE: Example

```c
struct A { A(); ~A();

int i;
int g();

void f()
{
    A a;
    int j = (--i, i ? 0 : 1);
    for (int x = 42; x > 0; --x)
    {
        i += g()*4 + 32;
    }
}
```
void f() {
    int i.0;
    int T.1;
    int iftmp.2;
    int T.3;
    int T.4;
    int T.5;
    int T.6;
    {
        struct A a;
        int j;
        __comp_ctor (&a);
        try {
            i.0 = i;
            T.1 = i.0 - 1;
            i = T.1;
            i.0 = i;
            if (i.0 == 0)
                iftmp.2 = 1;
            else
                iftmp.2 = 0;
            j = iftmp.2;
            {
                int x;
                x = 42;
                goto test;
                loop:;

                T.3 = g ();
                T.4 = T.3 * 4;
                i.0 = i;
                T.5 = T.4 + i.0;
                T.6 = T.5 + 32;
                i = T.6;
                x = x - 1;
                test:;
                if (x > 0)
                    goto loop;
                else
                    goto break_;
                break_;;
            }
        } finally {
            __comp_dtor (&a);
        }
    }
}