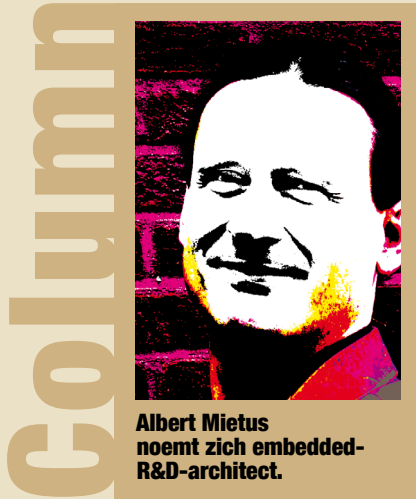


Begrijpt u wat ik bedoel?

De meest gebruikte compiler, ook in de embedded-wereld, is waarschijnlijk GCC. We hebben zo'n compiler nodig om C te vertalen naar machinetaal en daar is GCC goed in. Deze opensource code kent diverse releases en wordt goed onderhouden. Maar de software stamt toch al weer



**Albert Mietus
noemt zich embedded-
R&D-architect.**

uit de jaren tachtig en dat is te merken. Niet als we GCC gebruiken als compiler, maar wel als we kijken naar de architectuur.

Een compiler bestaat uit diverse onderdelen zoals de preprocessor, de lexer en de parser. Dat zijn de onderdelen die de code lezen en als taal begrijpen: wat zijn variabelen en wat zijn functies? Maar ook: hoeveel

argumenten heeft elke functie en wat is de scope van elke variabele? Ook de moderne editor wil dat begrijpen, om de programmeur bij te staan met *syntax highlighting*, automatische aanvullingen en *refactoring*.

Het lijkt logisch dat je editor en de compiler exact hetzelfde naar de code moeten kijken. Bijvoorbeeld om te voorkomen dat ze net een ander C-dialect gebruiken. Zo begrijpt een moderne C-compiler ook complexe getallen, inclusief een *i*-suffix voor het imaginaire deel. Maar herkent je editor dat ook? Er zijn meer verschillen in de praktijk, al is het maar door een andere parserimplementatie. Je zou dus willen dat je editor de parser van je compiler (her)gebruikt.

Dat geldt ook voor codeanalysetools. Ze doen een uitspraak over de kwaliteit van de code, terwijl ze deze waarschijnlijk anders uitleggen dan je compiler zal doen. En wat te denken van refactor tools? Mag een tool je code veranderen, terwijl hij die anders begrijpt dan je editor of compiler? Dat kan zelfs fouten introduceren. Deze problemen zijn te voorkomen door altijd dezelfde parser te gebruiken. Waarom gebruiken we die van GCC dan niet vaker?

Een mogelijke verklaring is de oude architectuur van GCC. Want al bevat deze compiler een parser en al is die opensource, het is vrijwel onmogelijk om die te hergebruiken. Toen GCC ontstond, waren een hoop moderne architectuurprincipes nog niet bedacht en waren modernismen als dynamisch laadbare bibliotheken nog niet uitgevonden. Bovendien was er destijds helemaal geen behoefte aan deze toepassing; dit soort editors c.q. hulpmiddelen zijn relatief nieuw. Er is dus geen herbruikbare parsermodule, laat staan een parserplug-in.

Zouden we vandaag een compiler ontwikkelen, dan zouden we een andere architectuur kiezen. In plaats van een vrij monolithische opzet, met slechts een grove indeling in compileerfasen, zouden we waarschijnlijk kiezen voor nette, stabiele, interne Api's en een verdere opdeling in functionele modules. Dat is exact een van de doelstellingen van Clang. Sinds enkele jaren wordt er hard gewerkt aan deze nieuwe GCC-compatibele compiler die inmiddels diverse dialecten en standaarden van C begrijpt. Zo worden C89 en C99 (met

complexe getallen) ondersteund evenals Objective-C en ook het grootste deel van C++. Clang gebruikt bovendien de LLVM-compilerinfrastructuur als basis, die ook deze flexibele, modulaire uitgangspunten kent. Elk onderdeel is een zelfstandige module met een duidelijke interface zodat elke deel kan worden hergebruikt. Mix en match onderdelen die je nodig hebt.

Zo ontstaat een compiler die net als andere compilers C-code kan vertalen in machinetaal, maar waarbij de onderdelen ook anders kunnen worden gebruikt. LLVM/Clang bevat bijvoorbeeld ook een Jit-compiler, iets dat we tot nu toe vooral uit Java kennen. Dus je kunt een tool bouwen die C-code kan lezen en vertalen in machine-instructies en die direct uitvoert. Ik denk dat zo'n tool best handig kan zijn. Een andere optie is om de tool als een soort van interpreter te gebruiken. Handig als de echte codegenerator nog niet af is, zeker in een R&D-omgeving.

Ook de C-parser is een bibliotheek. Die leest en begrijpt de code dankzij een ingebouwde semantische analyse. De library heeft een Api, die is gebaseerd op de *syntax tree* (AST) van C en daardoor vrij eenvoudig te gebruiken. Technisch maakt dat het mogelijk om deze parser op te nemen in een ander programma zoals je editor of in elk denkbaar hulpmiddel dat de programmeur werk uit handen kan nemen. Vrijwel automatisch begrijpt dat tool dan het juiste C-dialect. Clang kent er al meer dan tien, van C89 tot C++0x, met of zonder Gnu-extensies.

Clang geeft nog een groot voordeel, dat bovenstaande ook realistisch maakt. Het gebruikt namelijk niet de Gnu-alles-moet-vrij-zijn-GPL. Om ervoor te zorgen dat de modules echt kunnen worden gebruikt in andere tools, is er gekozen voor BSD-stijl licentievoorwaarden. Hiermee mag je verbeteringen aanbrengen en publiceren, maar mag je het ook opnemen in een commercieel product, zonder ook maar iets weg te geven.

Architectuur gaat niet alleen over techniek, maar ook over bruikbaarheid, kwaliteit, efficiency en zelfs over kosten. Al deze zaken moeten op elkaar worden afgestemd. Dat geldt voor de opensourcewereld, maar zeker binnen een R&D-afdeling. Willen we software beter maken, dan moeten we niet alleen technische oplossingen bedenken. Een R&D-afdeling is als een goede compiler: ze vertaalt requirements naar een werkend product. Dat resultaat moet goed en snel zijn, maar ook die vertaling moet zo snel en goedkoop mogelijk zijn. Clang claimt meer dan twee keer sneller te zijn dan GCC en veel minder resources te gebruiken. Welke R&D-afdeling durft die vergelijking aan?