Tooling support

Tooling for in-browser execution is often of uneven quality. WebAssembly aims at making it possible to support truly great tooling by exposing low-level capabilities instead of prescribing which tooling should be built. This enables:

- Porting of existing and familiar tooling to WebAssembly;
- Building new tooling that's particularly well suited to WebAssembly.

WebAssembly development should be self-hosting, and not just as a cute hack but as enjoyable platform that developers actively seek out because the tools they want and need *just work*. Developers have high expectations, and meeting these expectations on tooling means WebAssembly has the features required to build rich applications for non-developers.

The tooling we expect to support includes:

- Editors:
  - Editors such as vim and emacs should *just work*.
- Compilers and language virtual machines:
  - Compilers for languages which can target WebAssembly (C/C++, Rust, Go, C#) should be able to run in WebAssembly themselves, emit a WebAssembly module that can then be executed.
  - Virtual machines for languages such as bash, Python, Ruby should work.
  - Virtual machines which use a just-in-time compiler (JavaScript VMs, luajit, pypy) should be able to support a new just-in-time backend for WebAssembly.
- Debuggers:
  - Basic browser integration can be done through source map support.
  - Full integration for languages like C++ require more standardization effort on debugging information format as well as permissions for interrupting programs, inspecting their state, modifying their state.
  - Debug information is better delivered on-demand instead of built-in to a WebAssembly module.
- Sanitizers for non-memory-safe languages: asan, tsan, msan, ubsan. Efficient support of sanitizers may require improving:
  - Trapping support;
  - Shadow stack techniques (often implemented through `mmap`'s `MAP_FIXED`).
- Opt-in security enhancements for developers' own code: developers targeting WebAssembly may want their own code to be sandboxed further than what WebAssembly implementations require to protect users.
- Profilers:
  - Sample-based;
  - Instrumentation-based.
- Process dump: local variables, call stack, heap, global variables, list of threads.
- JavaScript+WebAssembly size optimization tool: huge WebAssembly+JavaScript mixed applications, WebAssembly calling to JavaScript libraries which communicate with the rest of the Web platform, need tooling to perform dead code stripping and global optimization across the API boundary.

In many cases, the tooling will be pure WebAssembly without any tool-specific support from WebAssembly. This won't be possible for debugging, but should be entirely possible for sanitizers.