LIRK
Lisp AVR with assembly Kernel

Łukasz Dubiel

24 lutego 2012

1 Wprowadzenie

Na platformie AVR istnieje już kilka języków mających ugruntowaną pozycję oraz zestaw narzędzi. Kilka głównych z nich to C, Basecom czy Pascal. Większość z nich to języki niezależne od platformy. Jednak mając takie zabawki jakimi są mikrokontrolery z rodziny AVR operacje języki które operują na pełnych słowach czy wielokrotnościach stają się częściowo niewygodne. Dlatego postanowiłem spróbować innego podejścia do problemu.

1.1 Lisp

W roku 1958 John McCarty na MIT stworzył język programowania który został nazwany Lispem. Był to drugi, zaraz po FORTRANie język wysokiego poziomu. Przez ten czas Lisp ewoluował, jak również pojawiały się różne dialekty (oczywiście nie kompatybilne z sobą) aż stworzyły całą rodzinę języków. Obecnie gdy używa się słowa Lisp to mówi się o całej rodzinie języków lub o Common Lispie, dialekcie ustandaryzowanym przez ANSI w 1994 z późniejszymi modyfikacjami.

1.2 Motywacje

Z racji mojego dużego zainteresowania językami z rodziny Lispów oraz problemów jakie napotyka C w momencie kiedy trzeba operować na pojedynczych bitach a nie słowach zdecydowałem się na stworzenie "dialektu" Lispa na rodzinę AVR.

Początkowo miało to być ćwiczenie w celu lepszego poznania Common Lispa, lecz potem zdecydowałem się, że to właśnie jego będę wykorzystywał w moich hobbystycznych projektach.

2 Zamierzenia i cele

Lirk nie ma (na razie) ambicji stania się nowym językiem programowania. Ma jedynie udostępniać funkcje które zminimalizuje ilość "nie potrzebnego" kodu w konstrukcjach stricte assemblerowych. W dalszej części referatu przedstawię cele.

2.1 Ewaluowanie parametrów instrukcji

Do działania Lispa faktycznie potrzebne są dwa byty. Reader i Evaluator. Pierwszy z nich ma za zadanie sparsować dane oraz rozwinąć tzw reader-macra (proste aliasy na najczęściej spotykane

konstrukty językowe takie jak ′ na quote czy #′ na function) oraz wszystkie nazwy zmieni na duże litery (lisp jest caseinsensitive). Drugi z nich evaluator za jedno zadanie wyznaczyć wartość wyrażenia które stworzył reader.

Daje to dość duże możliwości (przykładem jest system makr lispowych), które możemy wykorzystać by wzmocnić ekspresywność assembly.

[Prosty listing dający możliwość agregowania kodu]

Rozwiązanie jest podobne do zagnieżdżaniu funkcji w C jednak tam się to dzieje w momencie wykonania, a Likr używa mechanizmu podczas generowania pliku wykonywalnego. Oczywistym jest, że mamy dostępny dowolny ficzer Common Lispa.

2.2 Definicje oraz wołanie procedur

Jednym z pierwszych problemów jakie nękały programistów była duplikacja kodu. Były badania wskazujące na to, że średnia ilość błędów na jednostkę kodu (KLOC czyli tysiąc linii) jest stała i zależy do wielkości projektu [Potrzebne źródło]. Z racji tego, że assembler jest najmniej ekspresywnym językiem (ilość kodu potrzebna do wykonania zamierzonej akcji jest dość pokaźna) jest podatny na duplikacje.

Oprócz niskopoziomowych aspektów programowania interesuję się bardziej ogołnymi technikami jak i metodykami. Bardzo spodobał się mi manifest programowania zwinnego (ang. agile) oraz późniejszy, chociaż bardziej ważny dla programistów manifest Software Craftsmanship, którego jednej z głównych haseł (obok programowych) jest "Don't repeat yourself." Hasło bardzo ważne, gdyż powtarzanie kodu jest jedną z głownych przyczyn bugów.

Jak widać "klasyczne" assembly, ze względu na swoją ekspresywność, stoi dość mocnym kontraście względem programów tych manifestów.

Na tak małej platformnie jaką jest AVR bardzo dobrze sprawuje się programowanie proceduralne. Na podobnych platformach działało C 30 lat temu. Lekko problematyczne w C jest to iż, każda procedura działa na zasadzie skoku i powrotu (standard C99 zaczerpnął z C++ słowo kluczowe inline, jednak sprawia on, że funkcja nie zawsze jest rozwijana w miejscu i zależy to od wielu czynników).

Lirk z zamierzeniu ma oferować dla programisty obie te metody. Procedura może być wywołana przez skok z odłożeniem adresu powrotu na stos, jak i być rozwinięta w miejscu.

funkcję można wywołać bezpośrednio w celu rozwinięciu jej w miejscu.

[listing z rozwinięcie w miejscu]

albo wywołać jak wykonując skok

[listing z wykonaniem w programie]

Oczywiście w tym przypadku kompilator automatycznie zarządza etykietami oraz dodanie odpowiedni fragment kodu z instrukcjami powrotu.

2.2.1 Przerwania

Na przerwania można popatrzyć jak na specyficznie definiowane procedury. Procedury które wyzwalane są eventami generowanymi przez procesor. Wiadome jest że na samym początku przestrzeni programu znajduje się tablica wektorów przerwań.

Skoro jest to specyficzna funkcja możemy sobie zdefiniować makro które ułatwi nam jej definiowanie

[definicja macra]

Dzięki niemu możemy definiować przerwania w następujący sposób.
[definicja kilku przerwań]

2.2.2 Dalsze plany

Dodatkowo powinien w planach jest dodanie możliwości, że raz wołana funkcja jest rozwijana w miejscu, bez dodania niepotrzebnych instrukcji i etykiet. Lecz jeżeli programista wykona wołanie dwukrotnie funkcję, w assembly pojawiają się wszystkie potrzebne elementy by wykonać odpowiednie przejście.

Oferuje to większą granulację kodu, oraz ułatwia testowanie jednostkowe. Nawet można się pokusić o użycie TDD [http://www.agiledata.org/essays/tdd.html] jednak wymaga to napisanie odpowiednich narzędzi (emulatorów platformy,narzędzi do budowania), które nie byłyby aż tak problematyczne (przynajmniej w podstawowej formie).

2.3 Tworzenie rozgałęzień sterowania

Asembler oferuje kilkanaście instrukcji warunkowych. Są one w postaci predykatów. Język predykatowy jest naturalnym sposobem tworzenia warunków w językach funkcyjnych.

Widać, że kod w Common Lispie (którego składnie przyejmuje Lirk) wygląda inaczej niż analogiczny kod w C. Różniece nie są duże i bardzo łatwo je przeskoczyć. Dodatkowo podobnie jak w Pythonie a nawet bardziej kod programu przypomina (z drobnymi modyfikacjami) tekst pisany w jęyzyku naturalnym opisujący sposób działania.

Utrzymywanie bardzo rozgałaziąnego kodu w assembly wymaga bardzo dużego skupienia oraz bardzo rozległego modelu mentalnego. W przypadku assembly dużo więcej jest w głowie programisty niż jest napisane. Lirk ma na celu lekkie odzielenie oraz osłabienie wymagań skupienia programisty oraz większe zgrupowania akcji powiązanych z sobą w logiczną całość.

Rozwiązania oparte na predykatach (które w większości stosuje programowanie funkcyjne) bardzo dobrze mapuje się na instrukcje rozgałęzienia sterowania w kodzie.