Software Programmable
Signal Processing Platform Analysis

Andrzej Wąsowski

Software Programmable DSP Platform Analysis
Episode 1, Tuesday 12 April 2005

Contents

- Structure of a compiler
- Architecture and instruction set of DSPs/VLIW
- Implementation of a compiler for DSPs
- Lexical analysis
- Parsing
- Diagnostics
- Register allocation
- Code selection
- Code optimization

Goals

You will

- undestand the C programming language much better.
- understand compilation error messages much better.
- know abilities and limitations of compilers.
- be able to program more efficiently by:
  - producing more efficient code.
  - using less time for development.
- be able read compiler documentation with understanding.
- be able to choose appropriate options for compiling your code.
- be able evaluate which compiler is more suitable for your application.
- learn various objective functions for code optimization.

Non objectives

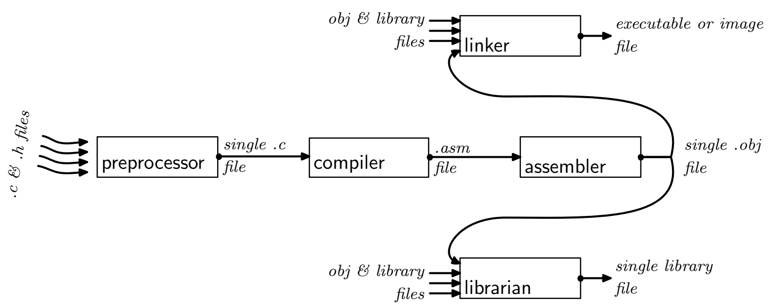
You will not

- be able to modify an existing compiler without excessive effort,
  - but you will be able to do so after someone introduces you to the implementation details of this particular compiler.
- be able to implement a compiler from scratch,
  - but you will know enough to succeed without greater obstacles when guided by a comprehensive textbook.
- know how to implement advanced language features of contemporary programming languages like:
  - objects, polymorphism, garbage-collectors, aspects, higher order functions, etc.
  - This topic belongs to CS traditionally, so shop for programming languages courses in nearby CS departments.
- learn programming languages theory (type systems, semantics, etc)
- learn mathematical linguistics (regular, context-free languages, etc)
  - but we will touch the issue at minimal required extent.

The course, teachers, lectures, exercises

- Teachers: Andrzej Wąsowski (compilers), Ole Olsen (architecture)
- The course home page is: `http://www.itu.dk/~wasowski/teach/dsp-compiler/`
- Contains: resources, schedule, exercise sheets and breaking news.
- Lecture slides are available from the course website (schedule)
- Please do ask questions during lectures.
- Reading: sections from Appel's *Modern Compiler Implementation in C* supplemented by other material distributed during the course.
- Dates: 12 April, 15 April (afternoon only), 19 April, 4 May
- Each module consists of approx. 90 minute lecture and 90 minute tutorial. A day contains two lectures and two tutorials.
- Exercises give practical experience with the content of the lecture. In depth understanding requires devoting more time to them than available at the tutorial session.

Compilation Environment



- Preprocessor expands macrodefinitions (`#define`'s), joins continued lines, removes comments (in C), includes files (`#include`).
- Compiler translates a single source file into assembly file
- Assembler translates .asm file to a binary .o file
- Linker consolidates bits and pieces into a single program.
- Modern linkers can perform global program optimizations, too.

Compilation Environment: Example

The following program:

hello.c

```
#define MSG "Hello, world!\n"
extern int printf(const char *format, ...);
/* A comment before the main function */
int main(int argc, const char * argv[])
{
    printf( MSG );
    return 0;
}
```

requires: preprocessing, compiling, assembling and linking with the startup code as well as with the standard C library.

Compilation Environment: Example (preprocessed)

hello.c

```
extern int printf(const char *format, ...);

int main(int argc, const char * argv[])
{
    printf( "Hello, world!\n" );
    return 0;
}
```

- Expanded macros
- Removed comments
- Included files (not in the example)

Compilation Environment: Example (compiled)

Hello.c compiled by the GNU C compiler, targeting x86, giving hello.s

```
        .file    "hello.c"
        .section .rodata
.LC0:   .string  "Hello, world!\n"
        .text
.globl main
        .type main, @function
main:   pushl %ebp
        movl  %esp, %ebp
        pushl $.LC0
        call  printf
        leave
        movl  $0, %eax
        ret
```

- This step is our main point of interest.
- The C program is translated into a flat list of simple instructions.
- Instructions and addresses are symbolic (mnemonics and labels).

Compilation Environment: Example (compiled II)

Hello.c compiled with TI's cl6x giving hello.asm (fragment):

```
SL1: .string "Hello, world!",10,0
     CALL .S1 _printf
     STW .D2T2 B3,*SP-(16)
     MVKL .S2 RL0,B3
     MVKL .S1 SL1+0,A3
     MVKH .S1 SL1+0,A3
     STW .D2T1 A3,*+SP(4)
     || MVKH .S2 RL0,B3 ;CALL OCCURS
RL0: LDW .D2T2 *++SP(16),B3
     ZERO .D1 A4
     NOP 3
     RET .S2 B3
```

- The assembly language of 67xx is different from x86.
- Compiler translates a portable code to a platform specific one.
- Some instructions are put in parallel (STW||MVKH).
- NOP (no operation) instructions are inserted.
- Seemingly nonlinear execution (call place and parameter passing).

Compilation Environment: Example (assembled)

Assembler resolves symbolic addresses and translates symbolic instructions to binary values. External symbols remain unresolved. Below statistics for the object file hello.o assembled from hello.s (GNU C/x86):

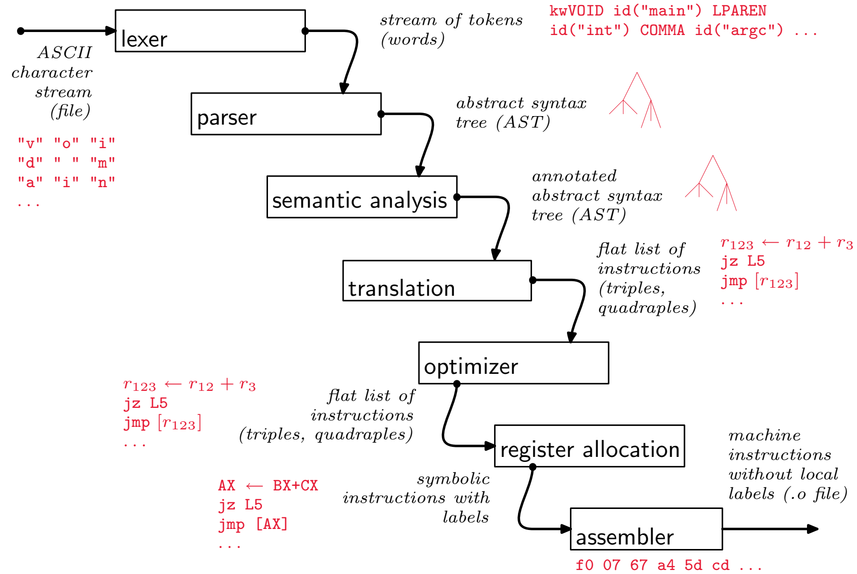
```
SYMBOL TABLE:
00000000 l df *ABS* 00000000 hello.cpp
00000000 l d .text 00000000
00000000 l d .data 00000000
00000000 l d .bss 00000000
00000000 l d .rodata 00000000
00000000 l d .eh_frame 00000000
00000000 l d .note.GNU-stack 00000000
00000000 l d .comment 00000000
00000000 g F .text 00000023 main
00000000 *UND* 00000000 printf
00000000 *UND* 00000000 __gxx_personality_v0
```

This object (.o) file needs to be linked with the C library or another .o file that provides the `printf` function.
In modern compilers the assembly is often incorporated in the compiler.

Architecture of a compiler

- Compilers are divided into layers, called *stages* or *passes*.
- Typically a stage inputs some kind of program representation, processes it and produces a different kind of program representation.
- The first stage typically inputs text files. The last stage typically outputs machine code, eg. an image that can be stored in EEPROM or a binary file that can be executed on your dekstop.
- The front parts of the compiler perform analyses, while the back parts of the compiler perform syntheses.

Architecture of a compiler (II)



Lexical analysis: Tokens

- A source program is represented as a sequence of characters
- A lexical analyzer (a lexer) breaks the sequence of characters into a sequence of corresponding tokens (like "words").

ID	`foo n14 last`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10. 1e67 5.5e-10`
IF	`if`
NOTEQ	`!=`
LPAR	`(`
RPAR	`)`

Lexical analysis: Tokens (continued)

The program

```
float match0 (char *s) /* find a zero */
{
    if (!strncmp(s,"0.0", 3))
    return 0.;
}
```

is translated to:

FLOAT ID(*match0*) LPAREN CHAR STAR ID(*s*) RPAREN LBRACE IF LPAREN BANG ID(*strncmp*) LPAREN ID(S) COMMA STRING(`0.0`) COMMA NUM(3) RPAREN RPAREN RETURN REAL(`0.0`) SEMI RBRACE EOF

- Lexer also removes comments (done by the preprocessor in C)
- Lexer removes white space from the code
- What are the words we need? How do we specify them?

Describing Tokens

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs newlines, and comments are ignored except as they serve to separate tokens. Some white space is requried to separate otherwise adjacent identifiers, keywords and constants.

- How do we write a program that detects identifiers?
- We need a precise way to describe them first.
- **Regular expresssions** offer such a way.

Regular Expressions

a	An ordinary character stands for itself
ϵ	The empty string.
$M \| N$	Alternation, chosing from M or N
$M \cdot N$	Concatenation, M followed by an N
M^*	Repetition zero or more times, Kleene's closure
M^+	Repetition one or more times
$M?$	Optional
$[a - zA - Z]$	Character set
.	Any single character except newline

The longest prefix of current input that can match any regular expression is taken as the next token.

Examples of Regular Expressions

`if`	an if keyword (IF)
$[a - z][a - z0 - 9]^*$	a simple identifier (ID), note: no capital letters
$[0 - 9]^+$	a decimal number (NUM)
$([0 - 9]^+"."[0 - 9]^*) \| ([0 - 9]^*"."[0 - 9]+)$	a real number (REAL)
$("//"[a - z]^*"\backslash n") \| (""\|"\backslash n"\|"\backslash t")^*)^*$	whitespace and one line comment

How can we describe the C identifier token? (2 slides ago)

Lexer Generators

- Lexer generators are programs that given a regular expression definitions for token types generate a program (lexer) able to translate a stream of characters to a stream of tokens.
- This is achieved by translating regular expressions to deterministic finite automata, similar to Mealy machines.
- The translation algorithm is standard and well known. More information in Appel, section 2.3–2.4.
- A popular free lexer generator targetting C is flex (see also lex in Appel, section 2.5).
- There exist such tools for any general purpose programming language.

Straight-Line Programs

Consider a simple (toy) language of straight-line programs.
The execution of the following program

```
a := 5+3;
b := (print (a, a+1), 10+a);
print(b)
```

produces

```
8 9
18
```


A Sample Straight-Line Programs

```
a := 5+3;
b := (print (a, a+1), 10+a);
print(b)
```

Token representation returned by the (hypothetical) lexer:

ID(a) ASSGN DEC(5) PLUS DEC(3) SEMI ID(b) ASSGN LPAR PRINT LPAR ID(a) COMMA ID(a) PLUS DEC(1) RPAR COMMA DEC(10) PLUS ID(a) RPAR SEMI . . .

- How we decide whether this token stream constitutes a legal program?
- How do we translate it to a tree?

Syntactical Analysis: Parsing

- A parser inputs the stream of tokens produced by the lexer.
- The tokens are analyzed and translated into an **Abstract Syntax Tree**
- This analysis is performed by finding a deriviation of the program with respect to a **context free grammar** of the source language.
- A context free grammar is a set of production rules describing the language's syntax.
- A production:

$$\textit{symbol} \rightarrow \textit{symbol}\ \textit{symbol}\ \ldots\ \textit{symbol}$$

- where *symbol* is either a token, called a **terminal** symbol now,
- or a **nonterminal** symbol.

A Grammar for Straight-Line Programs

Statement → Statement SEMI Statement
Statement → ID ASSGN Expression
Statement → PRINT LPAR List RPAR

Expression → ID
Expression → DEC
Expression → Expression PLUS Expression
Expression → LPAR Statement COMMA Expression RPAR

List → Expression
List → List COMMA Expression

- Terminals are capitalized
- Nonterminals: *Statement*, *Expression*, *List*
- *Statement* is the start symbol.
- See also Grammar 3.1, p. 41 in Appel.

It is often convenient to write character strings and symbols instead of tokens in the grammar:

$_1$ Statement $\rightarrow$ Statement ; Statement
$_2$ Statement $\rightarrow$ ID := Expression
$_3$ Statement $\rightarrow$ print (List)

$_4$ Expression $\rightarrow$ ID
$_5$ Expression $\rightarrow$ DEC
$_6$ Expression $\rightarrow$ Expression + Expression
$_7$ Expression $\rightarrow$ (Statement , Expression)

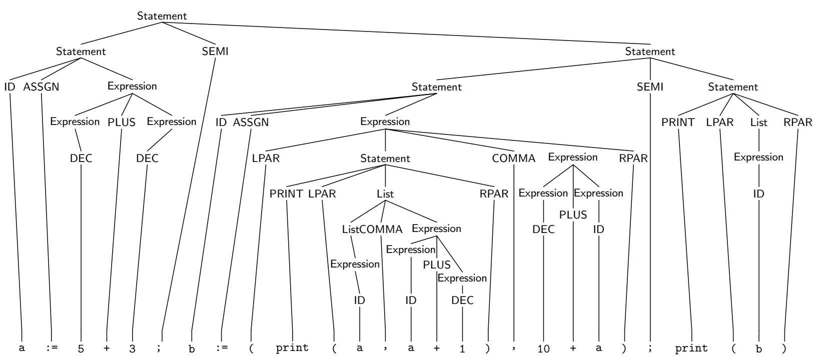
$_8$ List $\rightarrow$ Expression
$_9$ List $\rightarrow$ List , Expression

A stream of tokens constitutes a syntactically legal program in this language if it can be derived using the above rules.

The Rightmost Derivation for the Example

Statement $\rightarrow_1$ Statement ; Statement $\rightarrow_1$ Statement ; Statement ; Statement
$\rightarrow_3$ Statement ; Statement ; print(List)
$\rightarrow_8$ Statement ; Statement ; print(Expression)
$\rightarrow_4$ Statement ; Statement ; print(b)
$\rightarrow_2$ Statement ; b:=Expression ; print(b)
$\rightarrow_7$ Statement ; b:=(Statement, Expression); print(b)
$\rightarrow_6$ Statement ; b:=(Statement, Expression + Expression); print(b)
$\rightarrow_4$ Statement ; b:=(Statement, Expression + a); print(b)
$\rightarrow_5$ Statement ; b:=(Statement,10+a); print(b)
$\rightarrow_3$ Statement ; b:=(print(List),10+a); print(b)
$\rightarrow_9$ Statement ; b:=(print(List, Expression),10+a); print(b)
$\rightarrow_6$ Statement ; b:=(print(List,Expression+Expression),10+a); print(b)
$\rightarrow_5$ Statement ; b:=(print(List ,Expression +1),10+a); print(b)
$\rightarrow_4$ Statement ; b:=(print(List,a+1),10+a); print(b)
$\rightarrow_8$ Statement ; b:=(print(Expression,a+1),10+a); print(b)
$\rightarrow_4$ Statement ; b:=(print(a,a+1),10+a); print(b)
$\rightarrow_2$ a:=Expression ; b:=(print(a,a+1),10+a); print(b)
$\rightarrow_6$ a:=Expression+Expression ; b:=(print(a,a+1),10+a); print(b)
$\rightarrow_5$ a:=Expression+3; b:=(print(a,a+1),10+a); print(b)
$\rightarrow_5$ a:=5+3; b:=(print(a,a+1),10+a); print(b)

Parse Trees



- Sanitized parse tree (also called abstract syntax tree, or AST) is the first, and perhaps most important form of the program representation in the entire compilation process.

Parser Generators

- The process of parsing is a reverse of constructing a derivation.
- A parser is usually implemented as a push-down automaton (finite automaton with a stack).
- There exists several construction algorithms (and several parsing paradigms). See more in Appel, sections 3.2–3.3.
- Modern parsers are rarely hand-written.
- A parser generator translates a grammar description into a program that reads a stream of tokens and constructs a parse tree.
- Popular parser generators are yacc, bison, JavaCC, jjtree, ANTLR, . . .
- Such tools exist for all general purpose languages.