Norwegian University of Science and Technology

Compiler Construction

Lecture 10: Context-sensitive analysis

2020-02-11

Michael Engel

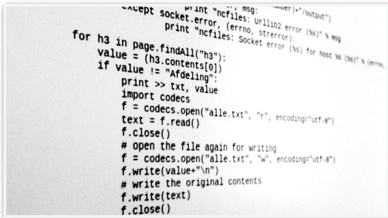
Overview

- Where are we standing now?
- There's more to languages than context-free grammars can describe…
  - From syntax to semantics
- Syntax-directed translation
  - Ad-hoc approach
  - Examples
  - A tiny (very imperfect) arithmetical expression to ARM assembly compiler

Where are we standing now?

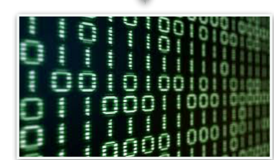
Semantic analysis

Source code



Lexical analysis → **Syntax analysis** → Semantic analysis → Code optimization → Code generation

token sequence *syntax tree*

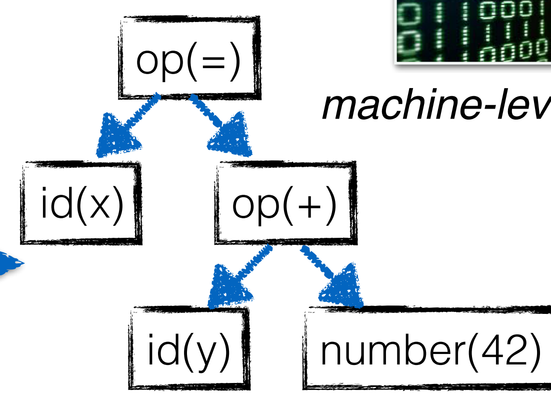


machine-level program

Syntax analysis (parsing)

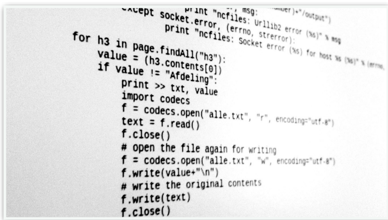
- Uses *grammar* of the source language
- Decides if input *token sequence* can be derived from the grammar

```
expression → term { (+|-) term }
term →       factor { (*|/) factor }
factor →     '(' expression ')'
             | id | number
```



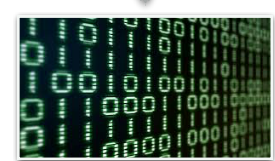
What is missing?

Source code



Lexical analysis → Syntax analysis → **Semantic analysis** → Code optimization → Code generation

syntax tree *syntax tree*



machine-level program

Semantic analysis

- *Name analysis* (check def. & scope of symbols)
- *Type analysis* (check correct type of expressions)
- Creation of *symbol tables* (map identifiers to their types and positions in the source code)

Beyond syntax: Example

- Consider this C program
  - Which errors can you detect?
  - Which of these can be detected using a context-free grammar?

```
bar(int a, int b, int c, int d) {
    …
}

foo() {
    int f[3], g[0], h, i, j, k;
    char *p;
    bar(h,i,“ab”,j, k);
    k = f * i + j;
    h = g[17];
    printf(“<%s,%s>.\n”,p,q);
    p = 10;
}
```

Wrong number of arguments to bar()

Declared g[0], used g[17]

"ab" is not an int

wrong dimension when using f

undeclared variable q

10 is not a character string

Beyond syntax

Semantic analysis

- All of these errors are "deeper than syntax"
  - There is a level of correctness that is ***deeper than grammar***
  - To generate code, we need to ***understand its meaning***!
- To generate code, the compiler needs to answer many questions, such as:
  - Is "`x`" a scalar, an array, or a function? Is "`x`" declared?
  - Are there names that are not declared? Declared but not used?
  - Which declaration of "`x`" does a given use reference?
  - Is the expression "`x * y + z`" type-consistent?
  - In "`a[i,j,k]`", does `a` have three dimensions?
  - Where can "`z`" be stored? (*register, local, global, heap, static*)
  - In "`f = 15`", how should 15 be represented?
  - How many arguments does "`bar()`" take? What about "`printf()`"?
  - Does "`*p`" reference the result of a "`malloc()`"?
  - Do "`p`" and "`q`" refer to the same memory location?
  - Is "`x`" defined before it is used?

All these are beyond the expressive power of a context-free grammar!

Context-sensitive analysis

Semantic analysis

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars?
  - Attribute grammars? (attributed grammars?)
- Use ad-hoc techniques
  - Symbol tables
  - Ad-hoc code (action routines)

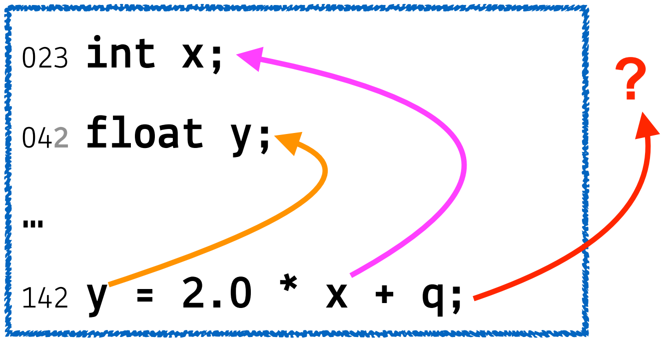
For parsing and scanning, formal approaches won

In context-sensitive analysis, ad-hoc techniques are often used in practice

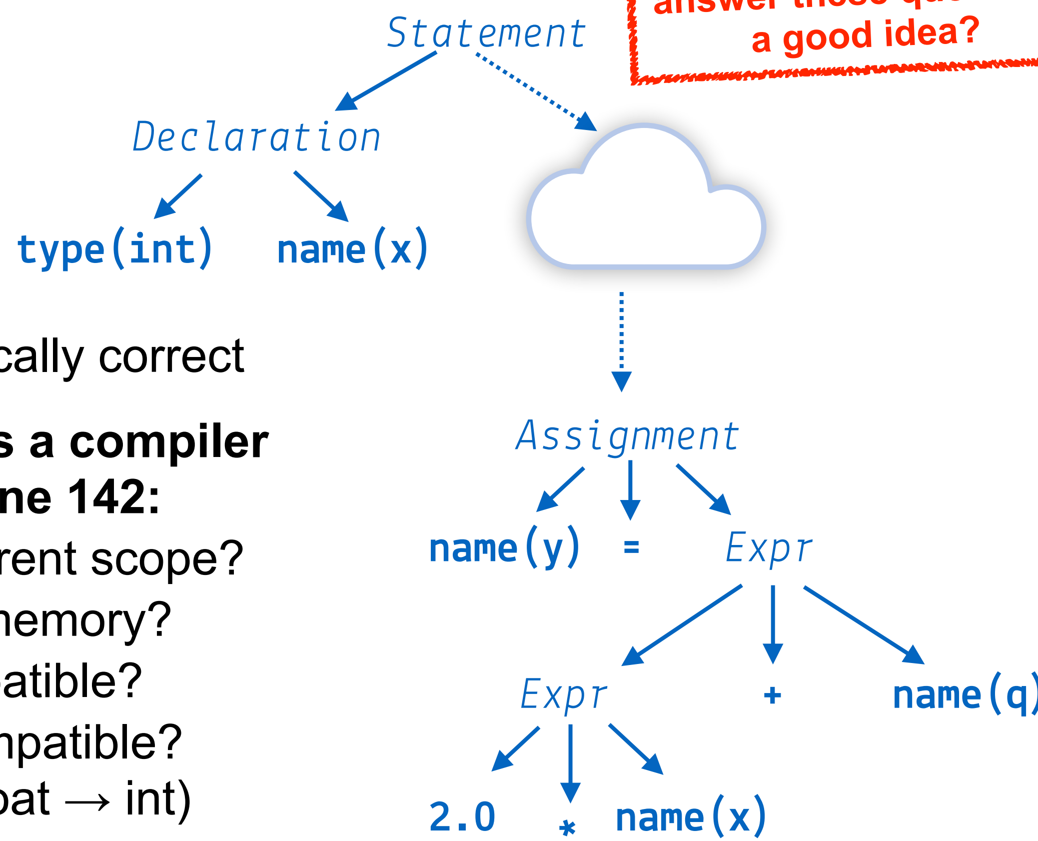
Non-syntactical information

Semantic analysis

Idea: Track the definitions of symbols in a global structure



Excerpt from simplified AST:



Is traversing the AST to answer these questions a good idea?

This program (excerpt) is syntactically correct

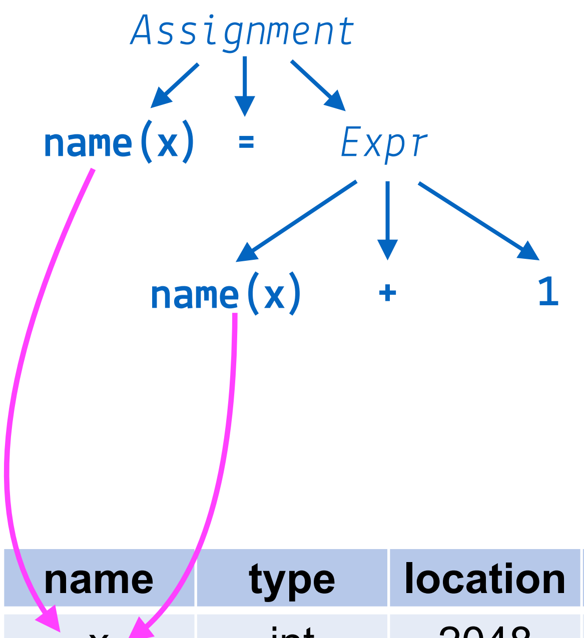
Some non-syntactical questions a compiler has to consider when parsing line 142:

- Are x, y and q defined in the current scope?
- Where are x, y and q stored in memory?
- Are the types of x, y and z compatible?
  - If not, can they be made compatible? (by implicit typecasts, e.g. float → int)

Symbol tables

Which information is required to compile an instruction?

```
023 int x;
…
099 x = x + 1;
```



name	type	location	…etc…
x	int	2048	…
…	…	…	…

Line 99 might be translated to:

1. Read value from **memory location** of x
2. Add **integer** value 1 to this
3. Store value to **memory location** of x

It is convenient to store all this information in a table and link the nodes of the AST to this information

Implementing symbol tables

This linking requires finding the table entry of x every time that name is used

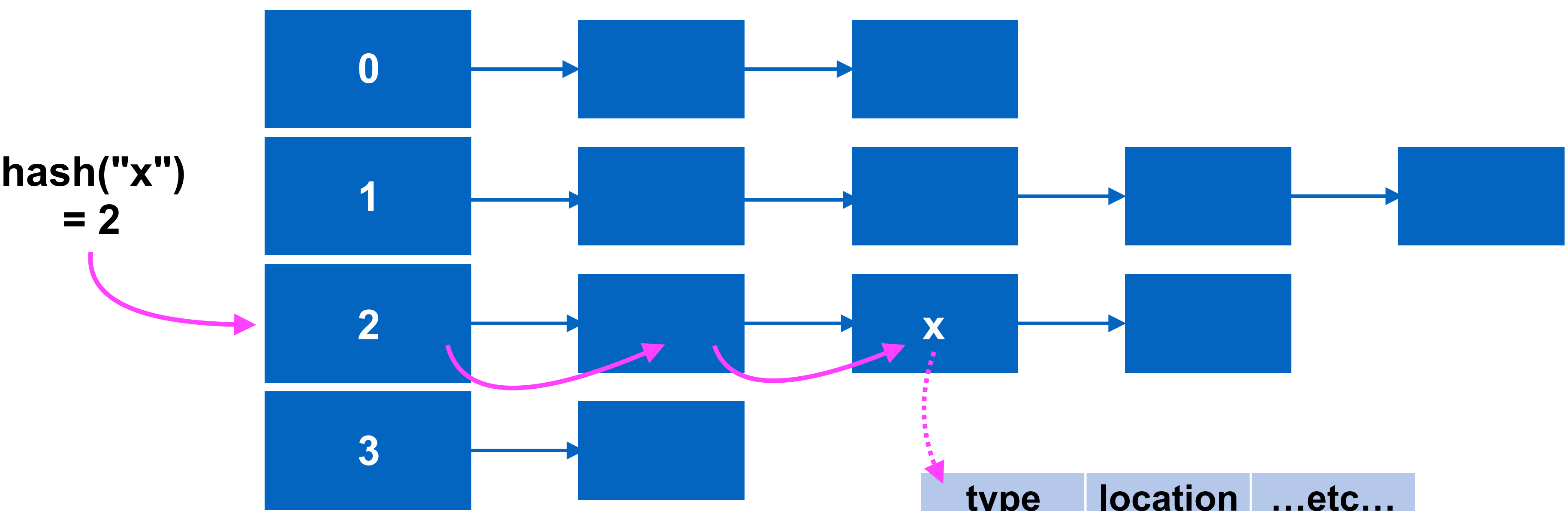
- We only get the name (→ scanner), so this is a text search problem
- We potentially have thousands of names when compiling a program

Possible approaches:

- *Direct indexing*: keep table where the index is a function of the text → limits number of identifiers to size of symbol table
- *Linked list*: keep a dynamic list, go through it and compare → expensive searches for identifiers in the back of the list
- *Hash table*

Symbol tables as hash tables

- An unpredictable, fixed-length code (*hash value*) can be computed from any length of identifier
- Elements stored in fixed-length array of linked lists
  - Search and compare only in the list where hash value matches



type	location	…etc…
int	2048	…

Advantage of hash tables

Hash tables are a good compromise

- Can dynamically grow with number of stored elements
- Constant time to find the right list to search
- If the hashing function distributes elements evenly, search time is divided by the number of lists
- Balance between static size limitation and list length can be adjusted depending on the data stored

However…

- No implementation of hash tables directly available in C 😕

Ad-hoc syntax-directed translation

Semantic analysis

Similar ideas work for top-down parsers

Build on bottom-up, shift-reduce parser

- Associate a snippet of code with each production
- At each reduction, the corresponding snippet runs
- Allowing arbitrary code provides complete flexibility
  - Includes ability to do *tasteless and bad things*

To make this work

- Need names for attributes of each symbol on LHS & RHS
- Typically, one attribute passed through parser + arbitrary code (structures, globals, statics, …)
- Yacc introduced `$$`, `$1`, `$2`, … `$n`, left to right
- Need an evaluation scheme
- Fits nicely into LR(1) parsing algorithm

Example: expression grammar

Semantic analysis

Introduce the *cost of expressions* to grammar

```
Block  → Block Assign
       | Assign
Assign → ident = Expr          { cost = cost + COST(store); }
Expr   → Expr + Term           { cost = cost + COST(add); }
       | Expr - Term           { cost = cost + COST(sub); }
       | Term
Term   → Term × Factor         { cost = cost + COST(mult); }
       | Term ÷ Factor         { cost = cost + COST(div); }
       | Factor
Factor → "(" Expr ")"
       | number                { cost = cost + COST(loadImm); }
       | ident                 { i = hash(ident);
                                 if (table[i].loaded == false) {
                                     cost = cost + COST(load);
                                     table[i].loaded = true; }}
```

One thing was missing…

Semantic analysis

```
0  Start  → Init Block
.5 Init   → ε                  { cost = 0; }
1  Block  → Block Assign
2         | Assign
3  Assign → ident = Expr       { cost = cost + COST(store); }
…
```

Initialize variable "cost"

Before parser can reach *Block*, it must reduce *Init*

- Reduction by Init sets cost to zero

- We split the production to create a reduction in the middle — for the sole purpose of hanging an action there
  - This trick has many uses

That wasn't ~~chicken~~ yacc...

Semantic analysis

Complete yacc+lex code is online

```
Start : Block                   { printf("Cost: %d\n", $$); }
Block : Block Assign            { $$ = $1 + $2; }
      | Assign                  { $$ = $1; }
Assign: ident '=' Expr          { $$ = cost(STORE) + $3; }
Expr  : Expr '+' Term           { $$ = $1 + cost(ADD) + $3; }
      | Expr '-' Term           { $$ = $1 + cost(SUB) + $3; }
      | Term                    { $$ = $1; }
Term  : Term '*' Factor         { $$ = $1 + cost(MULT) + $3; }
      | Term '/' Factor         { $$ = $1 + cost(DIV) + $3; }
      | Factor                  { $$ = $1; }
Factor: '(' Expr ')'            { $$ = $2; }
      | number                  { $$ = cost(LOADIMM); }
      | ident                   { int i = hash(ident);
                                  if (table[i].loaded == 0) {
                                    $$ = $$ + cost(LOAD);
                                    table[i].loaded = 1;
                                  }
                                  else $$ = 0;
                                }
```

Use case example: timing, energy

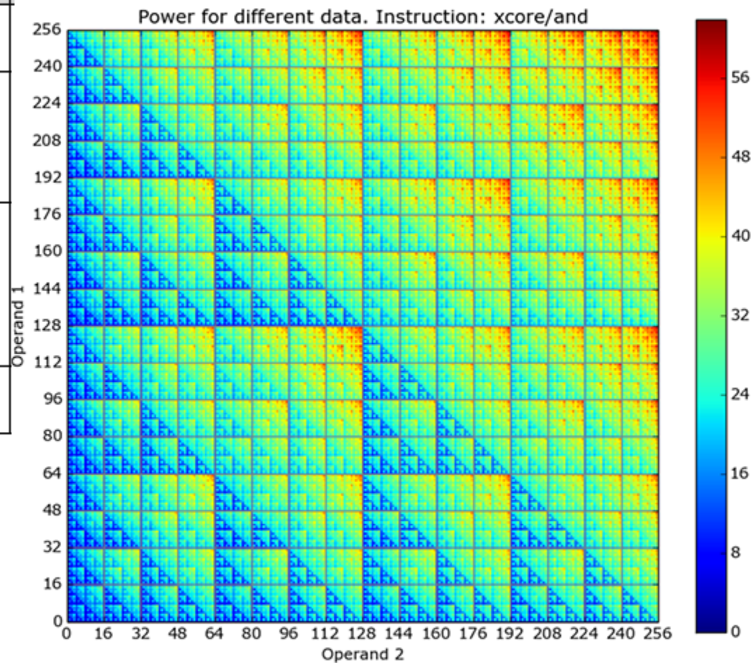
Semantic analysis

- How long does a piece of code take to execute?
- How much energy will the code consume?

Much more complex to assess for modern high-end CPUs (due to superscalarity, pipelines, caches, …)

3.5 Divide and Multiply Instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines
Divide	SDIV, UDIV	4 - 20	1/20 - 1/4	M
Multiply	MUL, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, SMULWT, SMMUL{R}, SMUAD{X}, SMUSD{X}	3	1	M
Multiply accumulate	MLA, MLS, SMLABB, SMLABT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMLAD{X}, SMLSD{X}, SMMLA{R}, SMMLS{R}	3 (1)	1	M
Multiply accumulate long	SMLAL, SMLALBB, SMLALBT, SMLALTB, SMLALTT,	4 (2)	1/2	M



Far more complex analyses required due to loops and conditional branches

Example: building an AST

So far, our syntax tree was only implicit – we need to operate on it

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume yacc-like syntax

```
Start : Expr                    { $$ = $1; }
Expr  : Expr '+' Term           { $$ = MakeAddNode($1, $3); }
      | Expr '-' Term           { $$ = MakeSubNode($1, $3); }
      | Term                    { $$ = $1; }
Term  : Term '*' Factor         { $$ = MakeMultNode($1, $3); }
      | Term '/' Factor         { $$ = MakeDivNode($1, $3); }
      | Factor                  { $$ = $1; }
Factor: '(' Expr ')'            { $$ = $2; }
      | number                  { $$ = MakeNumberNode(token); }
      | ident                   { $$ = MakeIdentNode(token); }
```

Example: emitting ARM assembly

Semantic analysis

Early simple compilers derived machine code directly from AST

- We won't do it this way later – need more optimization opportunities
- Still a nice example (*if the CPU instructions fit this scheme*)
- Assume that `NxReg()` returns a CPU register number

We omit symbol table handling here…

```
Start : Expr                  { $$ = $1; }
Expr  : Expr '+' Term         { $$=NxReg(); Emit("add", $$, $1, $3); }
      | Expr '-' Term         { $$=NxReg(); Emit("sub", $$, $1, $3); }
      | Term                  { $$ = $1; }
Term  : Term '*' Factor       { $$=NxReg(); Emit("mul", $$, $1, $3); }
      | Term '/' Factor       { $$=NxReg(); Emit("div", $$, $1, $3); }
      | Factor                { $$ = $1; }
Factor: '(' Expr ')'          { $$ = $2; }
      | number                { $$=NxReg(); EmitLI("mov", $$, yylval); }
      | ident                 { $$=NxReg(); EmitLD("ldr", $$, yytext); }
```

Example: emitting ARM assembly

Semantic analysis

Emit, EmitLI and EmitLD print assembler instructions

- NxReg should return ***free*** (*unused*) register number

We will *run out of registers* for complex expressions!

```
int NxReg(void) {
  static int reg = 0;
  if (reg > 11) { reg = 0; return reg; }    // wraparound if > 12 registers used!
  return reg++;
}

void EmitLD(char *op, int rd, char *adr) {  // emit memory load from address "adr"
  printf("\tldr r%d, =%s\n", rd, adr);
  printf("\t%s r%d, [r%d]\n", op, rd, rd);
}

void EmitLI(char *op, int rd, int val) {    // emit load of constant value "val"
  printf("\t%s r%d, #%d\n", op, rd, val);
}

void Emit(char *op, int rd, int rs1, int rs2) { // emit given arithmetic instrn.
  printf("\t%s r%d, r%d, r%d\n", op, rd, rs1, rs2);
}
```

Example: compiler output

Semantic analysis

Input: (z-3)*x+5

```
$ echo "(z-3)*x+5" | ./compile
    ldr r0, =z
    ldr r0, [r0]     // r0 = z
    mov r1, #3       // r1 = 3
    sub r2, r0, r1   // r2 = z-3
    ldr r3, =x
    ldr r3, [r3]     // r3 = x
    mul r4, r2, r3   // r4 = (z-3)*x
    mov r5, #5       // r5 = 5
    add r6, r4, r5   // r6 = (z-3)*x+5
```

Input: (z-3)*x)+5

```
$ echo "(z-3)*x)+5" | ./compile
    ldr r0, =z
    ldr r0, [r0]
    mov r1, #3
    sub r2, r0, r1
    ldr r3, =x
    ldr r3, [r3]
    mul r4, r2, r3
syntax error: )
```

Directly generating code during parsing → partial assembler code is being emitted!

ARM instruction overview:

`ldr rd, =z` -------------------- load address of memory location `z` into reg. `rd`

`ldr rd, [rs]` ------------------- load contents of memory at addr. `rs` into `rd`

`mov rd, #val` ------------------- copy numerical value `val` into register `rd`

`(add|sub|mul|div) rd, rs1, rs2` – execute `rd` = `rs1 (+|-|*|/) rs2`

Example: register wraparound

Semantic analysis

Number of registers in NxReg() reduced to 5 here to make example shorter!

Input: `(a+(b+(c+(d+e))))*x`

```
$ echo "(a+(b+(c+(d+e))))*x" | ./compile
    ldr r0, =a
    ldr r0, [r0]      // r0 = a
    ldr r1, =b
    ldr r1, [r1]      // r1 = b
    ldr r2, =c
    ldr r2, [r2]      // r2 = c
    ldr r3, =d
    ldr r3, [r3]      // r3 = d
    ldr r4, =e
    ldr r4, [r4]      // r4 = e
    add r5, r3, r4    // r5 = d+e
    add r0, r2, r5    // r0 = (d+e)+c
    add r0, r1, r0
    add r1, r0, r0
    ldr r2, =x
    ldr r2, [r2]
    mul r3, r1, r2
```

No more unused registers: wraparound!
r0 is overwritten here
Value of "a" is lost
→ incorrect result!

A real compiler needs a method for *register allocation*

- assign values to ***free*** registers
- when running out of registers, ***spill*** (save to memory) register contents and ***restore*** them when needed later
- efficient register allocation is complex – as we will see later

What’s next?

Semantic analysis

- A quick look at attribute grammars
- Some insight into type systems and type analysis

References

[1] ARM Cortex-A57 Software Optimization Guide
http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/Cortex_A57_Software_Optimization_Guide_external.pdf

[2] Kerstin Eder and John P. Gallagher, Energy-Aware Software Engineering,
DOI: 10.5772/65985
https://www.intechopen.com/books/ict-energy-concepts-for-energy-efficiency-and-sustainability/energy-aware-software-engineering

[3] Peter Marwedel, slide set on Embedded System Evaluation and Validation: WCET analysis (sl. 14 ff.)
https://ls12-www.cs.tu-dortmund.de/daes/media/documents/staff/marwedel/es-book/slides11/es-marw-5.1-evaluation.pdf

[4] ARM Instruction Set reference guide
https://static.docs.arm.com/100076/0100/arm_instruction_set_reference_guide_100076_0100_00_en.pdf