Compiler Construction

Project and Exam

Lennart Andersson

Revision 2012-02-29

2012

Standard project

In teams of 2 persons. Prerequisites:

- Approved assignments
- Assignment supervisor may grant postponement

Design a small procedural language

- integer and boolean types
- variables, constants, expressions, statements, ...
- block structure with nested procedures
- parameters, return values, recursion
- name analysis
- type analysis
- intermediate code generation
- assembly code generation

Nonstandard project

Design a language of your choice. Must be accepted by project supervisor in advance. Typical requirements:

- non-trivial grammar
- non-trivial name analysis
- significant semantic computations
- translation to some intermediate code
- translation to native code

Project administration

- Estimated work load: 40 hours (20-80)
- Report to Lennart by email if you are changing groups.
- Task report deadlines on project home page

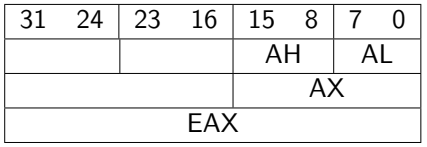
Project supervisors

- Emma Söderberg

Intel 386/486/Pentium processor architecture

registers	purpose
EAX, EBX, ECX, EDX	32 bit general registers
ESI, EDI	general
ESP	stack pointer
EBP	base pointer (frame pointer)
EIP	instruction pointer
C, O, Z, S, P, ...	several one bit registers
CS, DS, ES, SS	16 bit segment registers

EAX



EBX, ECX, EDX have the same structure

Memory

- every *byte* (b, 8 bits) has an address, 0, 1, ...
- *word* (w, 16 bits)
- *long* (l, 32 bits)

The Linux loader (ld) allocates a stack (2Mb)

- grows downword
- ESP points to the topmost byte in the stack

Assembler languages

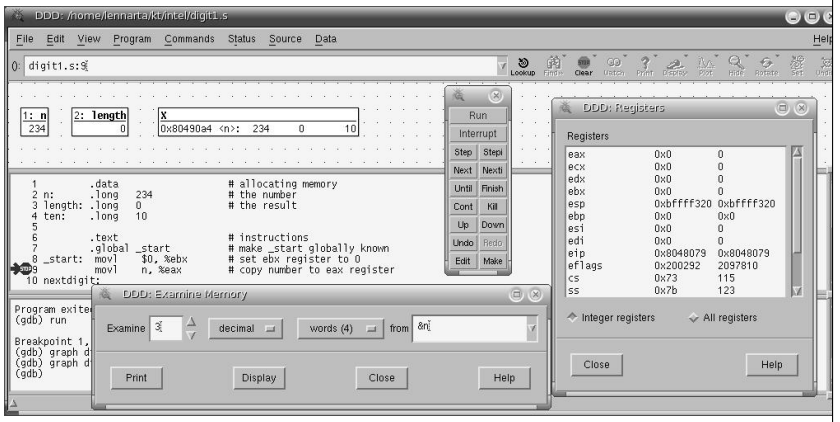
- Intel style assemblers. The main assembler of this kind is *nasm*.
- AT&T style assemblers, mainly the Gnu assembler, *as* or *gas*.

Program example

```
        .data                   # allocating memory
n:      .long   234             # the number
length: .long   0               # the result
ten:    .long   10              # the divisor

        .text                   # instructions
        .global _start          # make _start globally known
_start: movl    $0, %ebx        # use ebx as counter
        movl    n, %eax         # copy number to eax
nextdigit:
        movl    $0, %edx        # prepare for long division
        idivl   ten             # divide combined edx:eax by 10
                                # quotient to eax
        addl    $1, %ebx        # add 1 to counter
        cmpl    $0, %eax        # compare eax to 0
        jg      nextdigit       # jump if eax>0
        movl    %ebx, length    # copy counter to memory
```


ddd debugger (gdb)



Memory allocation

In the previous example variables have predetermined locations in memory an can be referred to by names.
In the project

- all variables reside on the stack.
- memory for the stack is allocated by `ld`.
- you will not need a .data segment!

Useful operand forms

operand	refers to
`$1448`	constant 1448 (base 10)
`nextdigit`	label address
`%eax`	value in eax
`(%ebp)`	value at address contained in ebp
`4(%ebp)`	value at 4 bytes after address in ebp
`(%ebp,%eax,4)`	value at ebp+4*eax

The last three types refer to values in main memory.

Useful instructions

instruction	operands	effect
`movl`	rmc32, rm32	rm32 ← rmc32
`addl`	rmc32, rm32	rm32 ← rm32+rmc32
`subl`	rmc32, rm32	rm32 ← rm32-rmc32
`negl`	rm32	rm32 ← -rm32
`idivl`	rm32	eax ← edx:eax/rm32, edx ← remainder
`notl`	rm32	rm32 ← ! rm32, bitwise, false = 0
`andl`	rmc32, rm32	rm32 ← rm32 & rmc32, bitwise
`orl`	rmc32, rm32	rm32 ← rm32 | rmc32, bitwise
`cmpl`	rmc32_1, rmc32_2	compare by computing rmc32_2-rmc32_1
`leal`	m32, r32	r32 ← address denoted by m32

Operand types
- r register
- m memory
- c constant

An instruction can have at most one memory (m) operand.

Conditional and jump instructions

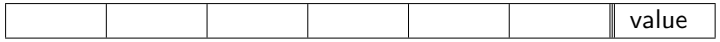
Condition codes (*cc*) set by the compl instruction:

`l`	`le`	`e`	`ne`	`g`	`ge`
$<$	$\leq$	$=$	$\neq$	$>$	$\geq$

`set`*cc* rm8	rm8 = *cc* ? 1 : 0
`cmov`*cc* rm32, r32	r32 = rm32 if *cc*

`jmp` dest	jump unconditionally
`j`*cc* dest	jump if *cc*

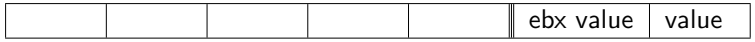
Stack instructions



← towards address 0

instruction	operand	effect
`pushl`	rac32	push value in rac32
`popl`	ra32	pop to ra32

```
pushl %ebx
```



Procedure calls

instruction	operands	effect
`call`	c32	push return address and jump
`ret`		pop return address and jump
`int`	c32	interrupt to kernel

```
        call p   # will push address of next instruction
        ...

p:
        ...
        ret      # will pop address and jump
```

C compiler conventions

- arguments are pushed on the stack in reverse order in the caller's activation record
- caller pops arguments after return
- callee must restore EBX, ESI, EDI, ESP, and EBP before returning
- EAX is used for return values

The exam

Regular exam: Wednesday March 7, 8-13, Victoria 2C.
Next exam: Friday April 13, E:3315. One week advance registration is required.

Allowed material at the exam

- Manual page on JastAdd Abstract Syntax (.ast files)
- Appendix about ICode
- Dictionary between English and your native language

Bonus points from the seminar exercises

Are counted at both the above examination dates, but not next year.

Prerequisites for writing the exam

Approved assignments

Assignment supervisor may grant postponement

Old exams

See the course web site but note that ...

- from 2008 a slightly different intermediate code is used
- in 2003 and earlier, a slightly different JastAdd notation was used

New topics this year

- LR parsing. You should be able to construct an NFA and an equivalent DFA for a given grammar an decide if there are any conflicts.
- Attribute grammars. You should be able to compute the values for synthesized and inherited attributes for a given attribute grammar.

Review

Lecture numbering may have changed.

F14: Machine code generation

Overall knowledge about

- Machine architecture with CPU, registers, and memory

F12: Memory Management

Overall knowledge

- The difference between manual and automatic memory management
- Terminology: fragmentation, memory leak, dangling pointer, compaction, root pointer, ...
- Main ideas in the main algorithms: reference counting, mark-sweep, copying, generation-based, conservative, incremental, ...
- Main benefits and drawbacks of the different algorithms

You don't have to

- memorize the details of the algorithms

F12-B: LR parsing

You should understand ...

- The principles for how an LR parser works, LR items.
- Why LR is more powerful than LL
- Typical kinds of unambiguous grammars that can be handled by an LR parser but not by an LL parser
- shift and reduce actions
- What is meant by a Shift/Reduce or Reduce/Reduce conflict?

You should be able to construct

- a NFA and DFA labeled by LR(0) items for a given grammar.
- decide if there are any conflicts.

F12-A: Attribute grammars

You should understand ...

- General idea.
- What is the difference between inherited and synthesized attributes?

You should be able to

- compute values for synthesized and inherited attribute for a given attribute grammar.
- make name analysis using synthesized and inherited attributes.

F11: Optimization

SSA form (Static Single Assignment)

- a powerful representation for optimization

Typical optimizations at the intermediate code level

- dominance
- copy propagation
- constant propagation
- ...

Typical optimizations at the machine code level

- register allocation
- instruction scheduling (to take advantage of pipelining)

F11: Intermediate Code

You should know ...

- What different kinds of intermediate code are there?
- Why temporary variables are needed and how they are handled
- Advantages of using intermediate code
- Difference between intermediate code and machine code
- Difference between a virtual machine and a real machine
- Translate a program to ICode
- How to implement code generation based on the AST

You don't have to ...

- memorize the details of ICode — you may use the ICode sheet on the exam

F10: Run-time systems

You should know ...

- Terminology: activation record, stack, stack pointer, frame pointer, static link, dynamic link, return address, object, heap, heap pointer, ...
- How procedure calls work, with parameter and return value transmission
- How object creation works
- How local and non-local variables in procedures are accessed
- How different kinds of variables are accessed in an OO language
- What v-tables are and how they are used in OO languages for method calls
- Draw the execution state at a given point in a given program

F9: Semantic analysis

You should know

- Terminology: name analysis, type analysis, scope, block, homogeneous blocks, declaration-before-use, bindings, symbol table, ...
- Different kinds of scope rules
- The difference between IdDecls and IdUses
- How to implement name analysis based on the AST
- Typical kinds of errors that can occur during compilation, and what different compiler phases they are identified in

F8: Modularized computations on the AST

You should know ...

- The Visitor pattern and how to use it
- Intertype declarations (static Aspect-oriented programming) and how to use them
- The benefits and drawbacks of these techniques, compared to each other and compared to writing tangled code
- Implement various computations using Visitors and Intertype declarations, e.g., unparsing, metrics, interpretation, name analysis, type checking, computation of information needed for code generation, ...

F7: Abstract syntax trees

You should know ...

- the difference between a parse tree and an abstract syntax tree
- the difference between a CFG and an abstract grammar
- how to design an object-oriented abstract grammar with good names
- write down an abstract grammar using the JastAdd notation
- how to build ASTs using semantic actions
- how to build the AST when an LL parser is used
- ...

You don't have to ...

- memorize the API for generated JastAdd classes — you may use the JastAdd manual page on Abstract Syntax on the exam.
- memorize the JJTree way for building ASTs

F6: LL parsing

- The principles for how an LL parser works
- Intuitive definitions: nullable, FIRST, FOLLOW
- Construct the nullable, FIRST, and FOLLOW tables for any CFG
- Construct the LL(1) table for a CFG.
- decide if a grammar is LL(1) or not

F5: LL parsing

You should know ...

- the different names for LL parsing
- how to implement an LL parser by hand using recursive procedures
- typical kinds of grammars that an LL(1) parser cannot accept
- given a CFG with some of these typical problems, construct an equivalent CFG that is LL(1)
- what is the difference between local lookahead and global lookahead?
- what the "dangling else" problem is and how to handle it in an LL parser generator
- why it is sometimes useful to extend a CFG by an EOF-rule, and how to do it
- ...

F4: CFGs for LL(1) parsing

You should know ...

- What is meant by ambiguous and unambiguous grammars
- Given an ambiguous grammar for expressions, construct an equivalent unambiguous grammar (given associativity and precedence rules)
- Typical kinds of unambiguous grammars that cannot be handled by an LL(1) parser
- When could such grammars be LL(k)?
- Construct equivalent grammars that are LL(1)

F3: Scanning

You should know ...

- typical kinds of tokens and non-tokens
- how to define typical tokens and non-tokens using regular expressions
- what typical ambiguities may occur for a set of token definitions?
- how can such ambiguities be resolved?
- what a finite automaton (FA) is
- the difference between a deterministic and nondeterministic FA
- how to translate an NFA to a DFA
- how to implement a scanner based on FAs, including handling ambiguities between regular expressions

F2: CFGs and REs

CFGs

- how to design a clear and simple CFG for a language (disregarding ambiguities, non-LL-ness, etc.)
- Terminology: terminals, nonterminals, productions, start symbol
- The formal definition of a CFG, $G = (N, T, P, S)$, and what it means
- the different notation forms for CFGs
- given a grammar on EBNF form, how to construct an equivalent grammar on canonical form, and vice versa
- what is meant by (leftmost/rightmost) derivation
- show that a string belongs to a given language REs
- typical notation for regular expressions
- the difference between REs and CFGs

F1: Introduction

You should know ...

- the typical phases in a compiler
- the typical representations of a program inside a compiler
- the separation into analysis and synthesis
- the separation into front end and back end
- typical applications of compiler construction techniques (in addition to the typical source-to-machine code compiler)

Compiler research at LTH (LUCAS)

www.lucas.lth.se

Examples of projects:

- GC for real-time systems
- The JastAdd system: AOP techniques, Reference attributed grammars, Eclipse integration, ...
- JastAdd applications: Java grammar, WCET computations, Design-pattern support, robot languages, integration with visual languages, ...
- Java implementations for embedded systems: Java-to-C compiler, Java-to-PRE compiler (very small VM)
- Optimizing Compilers for Multiprocessors
- ...

Interesting Master's thesis and PhD thesis projects!