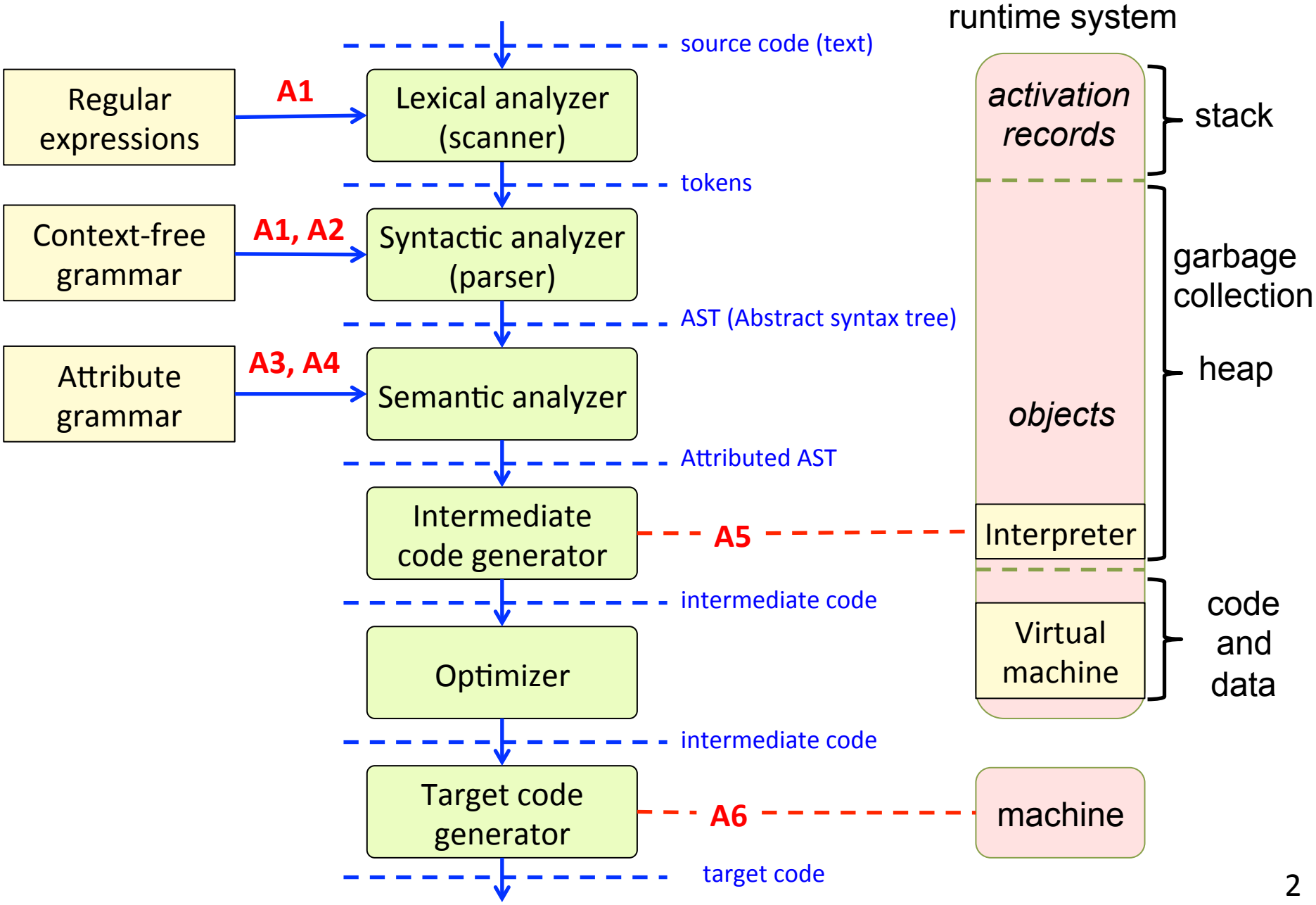
EDAN65: Compilers, Lecture 14

Review of important concepts

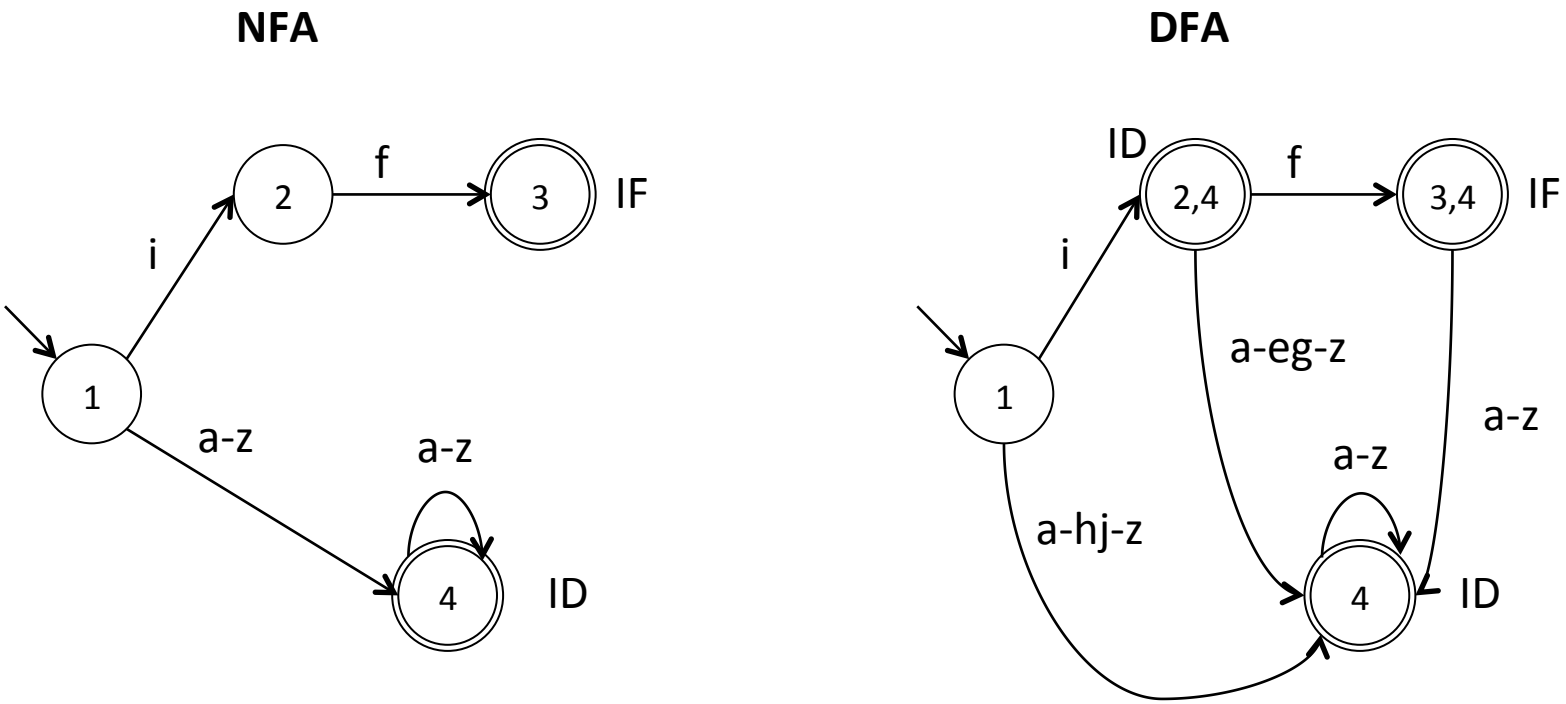
Görel Hedin

Revised: 2016-10-11

Course overview



Regular expressions and scanning



Given some informal description, formulate a regular expression or automaton.
Translate between regular expressions, NFAs, DFAs.
Know how to combine automata representing tokens.
Know how to handle rule priority and longest match.

Context-free grammars

```
Exp -> Exp "+" Exp
Exp -> Exp "*" Exp
Exp -> INT
```

Given some informal description, formulate a language as a context-free grammar.

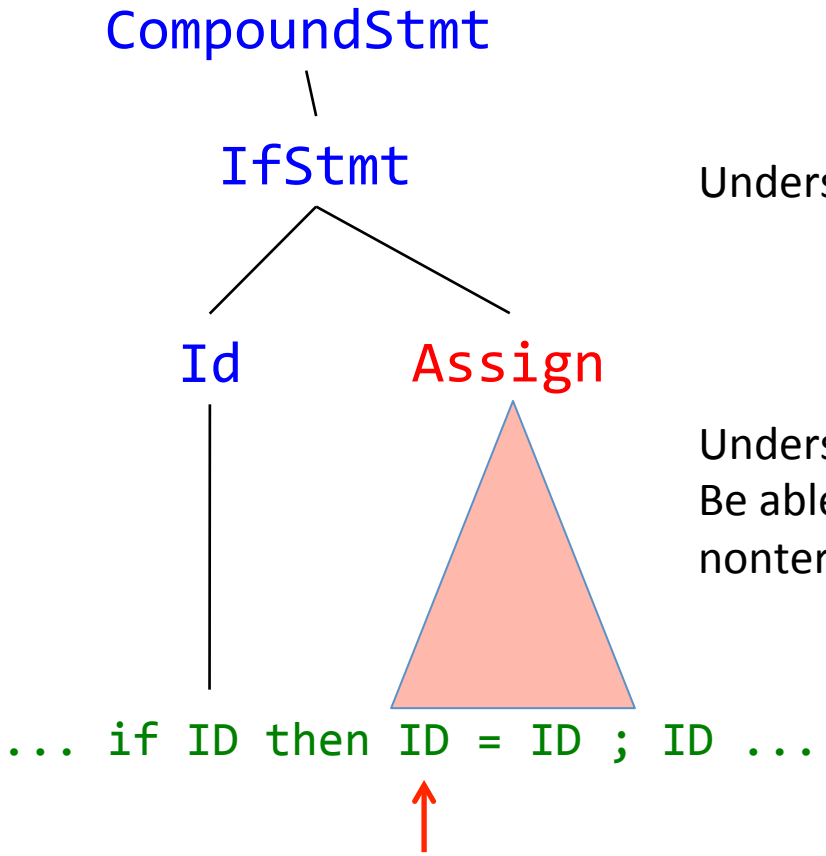
The elements of a context-free grammar G=(N, T, P, S):
nonterminal symbols, terminal symbols, productions, start symbol

Understand what the language defined by a grammar is.
Understand the difference between regular expressions and context-free grammars.

```
Exp
=> Exp "+" Exp
=> INT "+" Exp
...
```

Derivations. How to prove that a sentence belongs to a language.
Parse trees. How a parse tree corresponds to a derivation.

LL Parsing



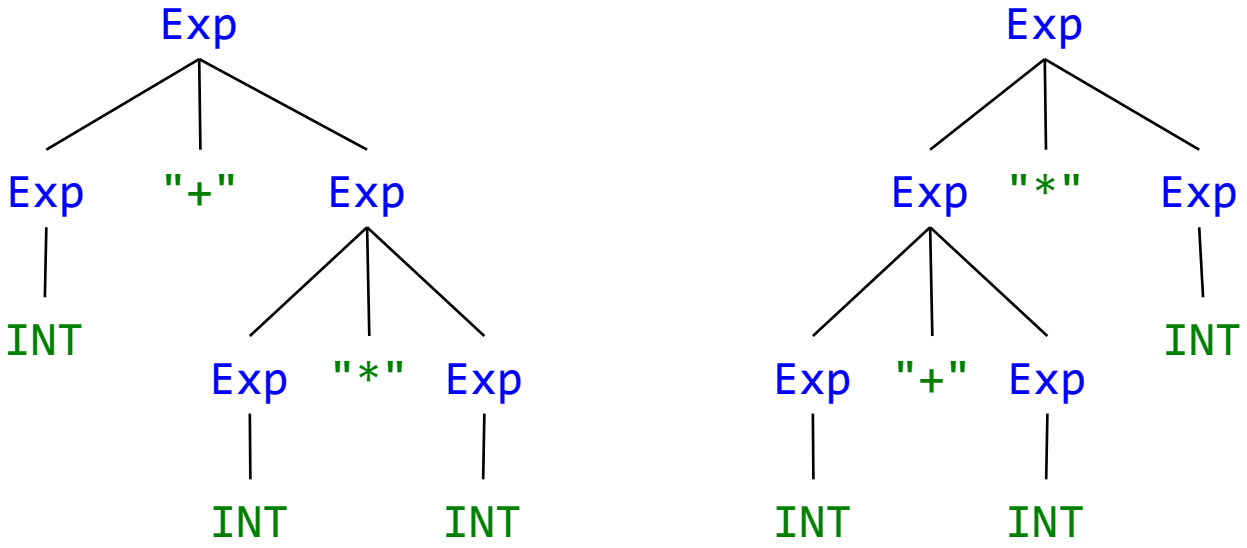
Understand the main idea of how LL parsing works

Understand the concepts of FIRST, Nullable, and FOLLOW.
Be able to compute them for a given
nonterminal in a grammar.

Be able to construct a recursive-descent parser for a given LL(1) grammar.

Understand what left recursion and common prefix mean.
Understand why grammars with these properties are not LL(1).
Be able to transform such a grammar to an equivalent LL(1) grammar.

Ambiguities in context-free grammars



Understand what it means for a grammar to be ambiguous.
Understand what it means for two grammars to be equivalent.

Be able to prove that a grammar is ambiguous for common ambiguities (expressions and dangling else).

Be able to transform an ambiguous grammar to an equivalent unambiguous grammar, for common ambiguities.

Notations for context-free grammars

```
A -> B d e C f
A -> g A
```

Canonical form

```
C -> D a b | b E F | a C
```

BNF

```
G -> H* i | (d E)+ F | [d C]
```

EBNF

Be able to formulate grammars on canonical form, as BNF, and EBNF
Be able to translate between these forms.

Abstract grammars

Abstract grammar

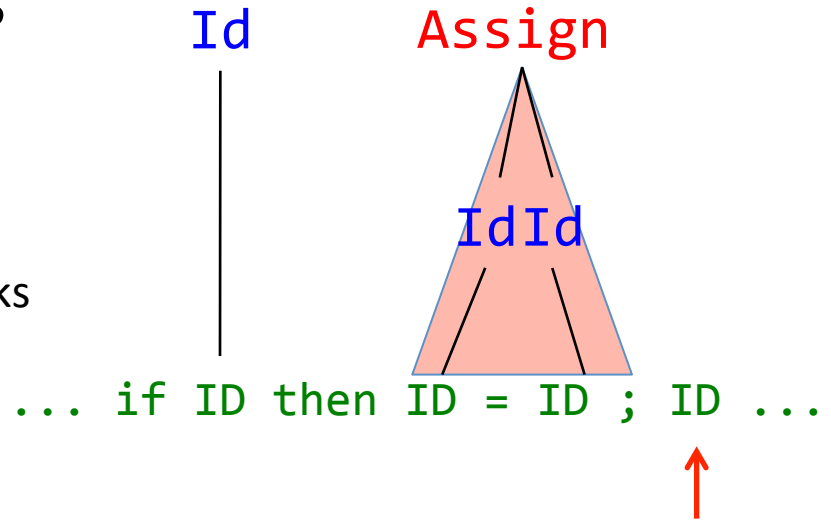
```
abstract Stmt;
IfStmt : Stmt ::= Expr Stmt;
Assignment : Stmt ::= IdUse Expr;
IdUse : Expr ::= <ID:String>;
```

Understand the difference between an abstract grammar and a context-free grammar.

Understand how an abstract grammar corresponds to an object-oriented model.
Be able to design an abstract grammar for a simple language, corresponding to a good object-oriented model.

Be able to design a high-level (possibly ambiguous) concrete grammar that is very close to an abstract grammar.

LR parsing



Understand the main idea of how LR parsing works with shift, reduce, and accept actions.

Understand why LR is more powerful than LL.

Understand what an LR item is.
Understand what an LR state is.

```
E -> E • "+" E     ?
E -> E "*" E •     "+"
```

Understand what an LR conflict is.
For a given LR conflict, be able to construct a program that would expose the conflict (for a simple case).

I will **not** ask you to construct an LR DFA or an LR parsing table on the exam.

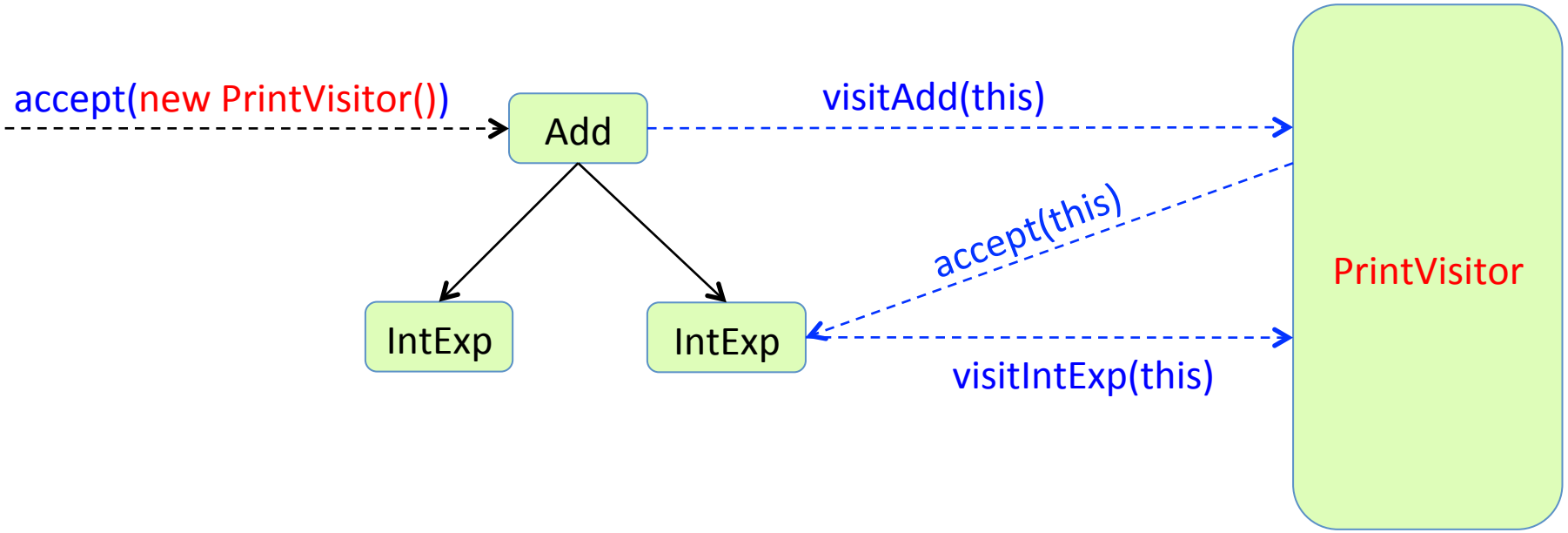
Static aspects with intertype declarations

```
aspect Evaluator {
  abstract int Expr.value();
  int Add.value() { return getLeft().value() + getRight().value(); }
  int Sub.value() { return getLeft().value() – getRight().value(); }
  int IntExpr.value() { return String.parseInt(getINT()); }
}
```

Understand how static aspect-oriented programming with inter-type declarations works.

Be able to program problems using methods declared in aspects.

Visitors

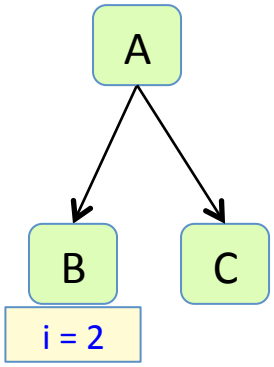


Understand how the visitor pattern works.

Be able to program problems using visitors.

Reference attribute grammars

```
inh int B.i();
eq  A.getB().i() = 2;
```

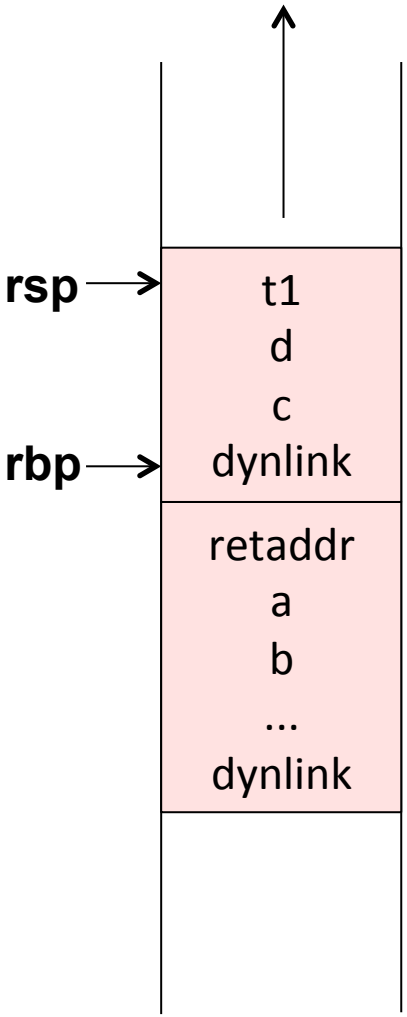


Understand how reference attribute grammars work, with different kinds of attributes: synthesized, inherited, parameterized, NTAs, circular, and collection attributes.

Be able to program problems using reference attribute grammars.

You **don't** have to memorize details in the JastAdd syntax –
copies of the JastAdd reference manual will be available at the exam.

Runtime systems and code generation



Understand how an activation stack works, with frame pointer, stack pointer, and dynamic link.

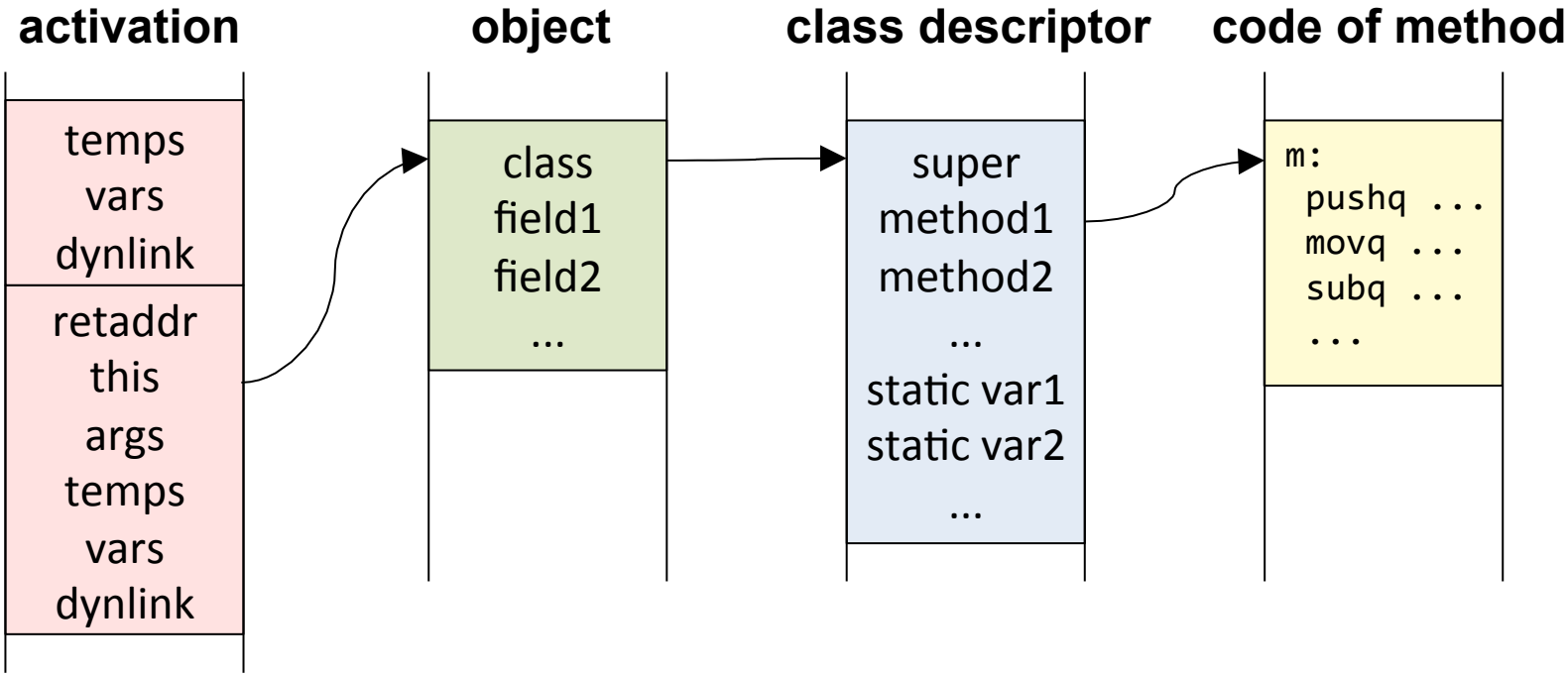
Understand simple calling conventions, like passing arguments on the stack and the return value in a register.

Be able to write down pseudo assembly code for procedure calls, procedure activation and returns, and simple computations like loops, assignments, expression evaluation, etc.

Be able to draw the contents of the stack for a given execution point in a program.

You **don't** have to memorize x86 instructions.

Runtime systems for object-oriented languages



Understand the role of the static link ("this" pointer).
Understand how prefixing works for single inheritance of fields.
Understand how vtables work for single inheritance dynamic dispatch.
Have an overall idea about how dynamic adaptive compilation works.

You **don't** have to memorize how to handle multiple inheritance.

Preparing for the exam

Read slides, book, papers, lab descriptions.
Do the quizzes and the exercises.
Study old exams.

See http://cs.lth.se/edan65

Ask at the **forum** if you have questions!

Don't forget to **sign up** for the exam by Monday 2016-Oct-17.

You need to **finish the assignments/labs** before the exam.

Catch-up lab session: Tuesday Oct 18, 13:00-15:00, E:Alfa

Extra guest lecture

Monday Oct 17, 10:15-11:00, MA:3

Deconstructing Dotty, a next generation Scala Compiler

Felix Mulder, M.Sc., Programming Methods Laboratory, EPFL, Lausanne

Abstract: Dotty is a next generation compiler aimed at trying new language concepts and compiler technologies for Scala. In this talk I aim to explore the future of Scala from a compiler engineer's perspective. I will explore the compiler pipeline, type inferencing, library rewrite rules and compiler optimizations as well as giving an overall view of the differences between Scala and the Dotty version of Scala.