EDAN65 Compilers - Study Guide

Jakub Gorski (dt07jg8)

October 26, 2014

Lund University

Contents

What are the major compiler phases?

1. **Lexical analysis** (scanning of textual source code).
   Returns tokens.
2. **Syntactic analysis** (Parsing of tokens).
   Returns AST (Abstract syntax tree).
3. **Semantic analysis** (Attributes AST).
   Returns attributed AST.
4. **Intermediate Code Generation**.
   Returns intermediate code.
5. **Optimization**.
   Returns optimized intermediate code.
6. **Target code generation**
   Returns architecture-specific assembly code.
   Assembler and Linker do the rest.

What is the difference between **analysis** and **synthesis** phases?

- Analysis belongs to the **front-end**, where Lexical- Syntactic- and Semantic analysis are performed **independent of targeted platform language**. Intermediate code is generated.
- Synthesis takes intermediate code, optionally **optimizes** it, and synthesizes an **executable** the target platform understands.

Why do we use **intermediate code**?

- Intermediate code opens possibility of having **several front-ends and back-ends**.
- Each back-end may optimize intermediate code for target platform differently.
- Debugging of front-end is easier using an **interpreter** than a target machine.

What is the advantage of **separating** the front and back ends?

- Because it is more rational to implement **m** front-ends and **n** back-ends, instead of programming **m*n** compilers.

What is a **lexeme**?

- A lexeme is a string corresponding to a **token**.
- Example: Token corresponding to **terminal** macro "`ID = [a-zA-Z]+`" will have whole words as **lexemes**.

What is a **token**?

- A token corresponds to a predefined **terminal**, which either has a **symbolic name**, or is defined by a **regular expression**.

What is a **parse tree**?

- Locally generated and used by **Syntactic analysis** (Parsing of tokens).
- A tree of **non-terminal** tokens which branches its way down to **all terminal** tokens.

What is an **abstract syntax tree**?

- Generated and returned by **Syntactic analysis**.
- A tree of **non-terminal** tokens which branches its way down to **only most essential terminal** tokens.

Lund University

What is **intermediate code**?

- Generated and returned by **Intermediate code generation**.
- Assembly-like code that is **independent** of target-platform, and source language.
- Employs a **virtually unlimited** stack for all **memory**/register operations.

What is the difference between **assembly code**, **object code**, and **executable code**?

- **Assembly code** is an unsafe, **architecture-specific language**.
- **Object code** is returned by the **Assembler**, which contains global symbols and **(still) relocatable addresses**.
- **Executable code** is returned by the **Linker**, where global symbols and relocatable addresses have been replaced by **absolute addresses**.

What is **bytecode**, an **interpreter**, a **virtual machine (VM)**?

- **Bytecode** is low-level **platform-independent code** conventionally executed on **VM**:s.
- **Interpreter** is a computer program that **directly executes**, i.e. performs, instructions written in a programming or scripting language, **without previously compiling** them into a machine language program.
- **A VM is an interpreter** that executes low-level, usually platform-independent code. In other contexts, a **VM** may refer to system virtualization.

What is a **JIT compiler**?

- A **JIT compiler** is an interpreter or VM that **compiles** a program **during** its **execution** (run-time) – rather than prior to execution.

Lund University

What kind of errors can be caught by a compiler? A runtime system?

1. **Lexical analysis**; Lexical errors.
   Text that cannot be interpreted as a token.
2. **Syntactic analysis**; Syntactic errors.
   Tokens in the wrong order.
3. **Semantic analysis**; Static-semantic errors.
   Illegal use of names, types, and other high-level language entities.
4. **Intermediate Code Generation**; No errors.
   Returns intermediate code.
5. **Optimization** (Optional); Possibly no errors.
   Returns optimized intermediate code.
6. **Target code generation**.
   Returns architecture-specific assembly code.
   Assembler and Linker do the rest. Runtime errors may occur.

What is a **formal language**?

- An **alphabet** Σ is a set of symbols (**nonempty**, **finite** length).
- A **String** is a sequence of symbols (**finite** length).
- A **formal language** Γ is a set of strings (may be **infinite**).

What is a **regular expression**?

- It is a sequence of characters that forms a **non-recursive search pattern**, mainly for use in pattern matching with strings, or string matching.

What is meant by an **ambiguous lexical definition**?

- If **two production rules** can be used to **match** the same sequence of characters **then the rule is ambiguous**.

Give some typical examples of ambiguities and how they may be resolved.

- Given the two productions rules "`if`" and "`ifff`". The ambiguity can be resolved by **longest match**, meaning if there is another rule that can match a longer token, the latter rule will be chosen. This way the scanner will match the longest token possible.
- Additionally **rule priority** is employed when two rules match the same sequence of characters, in which case the first one takes priority.

What is a lexical action?

- Each production rule has the form `regular-expression {lexical action}`, where each lexical action consists of arbitrary Java code. This **code is run when the token is matched**, consequently the method `yytext()` returns a lexeme (the token string value).

Show how to construct an NFA for a given lexical definition.

- Read **L02 p.25** `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2014/lectures/L02.pdf`

Show how to construct an NFA for a given lexical definition.

- Read **L02 p.29** `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2014/lectures/L02.pdf`

What is the difference between a DFA and and NFA?

NFA (Non-deterministinc Finite Automaton)

- Two outgoing edges may have **overlapping character sets**.
- ε edges are **permitted**.
- Every NFA can be translated into an equivalend DFA.
- Every DFA is also an NFA.

DFA (Deterministinc Finite Automaton)

- Outgoing edges must have **disjoint character sets**.
- ε edges are **prohibited**.

Give an example of how to implement a DFA in Java.

Table driven implementation

- Represent the automaton by a **table**.
- Additional table to keep track of **final states** and **token kinds**.
- A **global variable** keeps track of the **current state**.

Switch statement implentation

- Each state is implemented as a **switch statement**.
- Each case implements a state transition as a **jump** (to another switch statement).
- The current state is represented by the **program counter**.

How is rule priority handled in the implementation?

Longest match?

- If **two** rules can be used to **match the same** sequence of characters, the **first** one takes **priority**.
- The general idea:
  1. When a token is matched **don't stop scanning**.
  2. When the **error** state is reached, **return** the **last token** matched.
  3. **Push** read characters that are **unused** back into the file, so they can be **scanned again**.
  4. Use a `PushbackFile` to accomplish this.

EOF?

- Construct an **explicit EOF** token when the `EOF` character is read.

Lund University

How is rule priority handled in the implementation? Cont.

Whitespace?

- Whitespaces are viewed as tokens of a **special kind**.
- Scan the as **normal tokens**, but create **no objects** for them.
- **Loop** in `next()` **until** a **real token** has been found.

Errors?

- Construct an **explicit ERROR** token to be returned **when no valid token** can be found.

What are lexical states? When are they useful?

- Some tokens are **difficult** or **impossible** (recursive) to define **using regular expressions**.
- **Lexical states** (sets of token production rules) give the possibility to **switch token sets** (DFAs) during scanning.
- Useful for multi-line comments, HTML, scanning multi-language documents.

Construct a CFG for a simple part of a programming language.

```
Stmt -> WhileStmt | AssignStmt | ...
WhileStmt -> WHILE LPAR Exp RPAR Stmt
Exp -> ID | ...
...
```

- Where the following `terminals` = {`WHILE`, `LPAR`, `RPAR`, `ID`}, and the rest are non-terminal production rules.

What is a **nonterminal symbol**?

- A non-token symbol, which are represented by **inner nodes** in the pares tree.

What is a **terminal symbol**?

- Terminals are tokens which are **leafs** in the parse tree.

What is a **production**?

- A grammar rule defining a **non-terminal node**.

What is a **start symbol**?

- The start symbol is the **root** in the parse tree.
- **Starting point** for the grammar.

What is a **parse tree**?

- It is a **derivation tree**, which is the representation of a set of derivations based on a language grammar.
- A parse tree **must be unambiguous** for it to fulfill its task.

What is a **left-hand side** of a production?

- The left-hand side X of a production rule is a **non-terminal**.

What is a **right-hand side** of a production?

- The right-hand side $Y_1 Y_2 ... Y_{n-1} Y_n$ is a sequence of **nonterminals and terminals**.
- If the right-hand side for a production is **empty** we write ε.

Lund University

Given a grammar G, what is meant by the language $L(G)$?

- G defines a language $L(G)$ over the alphabet T.
- Where $G = (N, T, P, S)$.
  - N - set of **nonterminal** symbols.
  - T - set of **terminal** symbols.
  - P - set of **production rules**.
  - S - **start symbol**, which is a **nonterminal**.
- $T*$ is the set of **all possible terminals** symbol sequences.
- $L(G)$ is a **subset** of T^*, which can be derived by traversing the **parse tree**, beginning with the **root** start symbol S, then **following** the **production rules** P.

What is a **derivation step**?

- It is a process where we can **replace** a **nonterminal** i a given **sequence** of terminals and nonterminals by **applying production rules**.

What is a **derivation**?

- It is a sequence of **derivation steps**.

What is **leftmost derivation**?

- In a leftmost derivation, the **leftmost** nonterminal is **replaced** in each derivation step.

What is **rightmost derivation**?

- In a rightmost derivation, the **rightmost** nonterminal is **replaced** in each derivation step.

How does a derivation correspond to a parse tree?

- Nonterminals are **nodes** and terminals are **leafs** in the tree.

What does it mean for a grammar to be **ambiguous**?

- `CFG` (Context Free Grammar) is ambiguous if a sentence in the language can be derived by **two (or more) different parse trees**.

What does it mean for a grammar to be **unambiguous**?

- `CFG` is unambiguous if each sentence in the language can be derived by **only one syntax tree**.

LUND UNIVERSITY

Give an example an ambiguous **CFG**.

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

Ambiguity due to the **lack of operator precedence**, thus multiplication and addition has equal evaluation rights.

What is the difference between an **LL** and an **LR** parser?

- **LL**: **L**eft-to-right scan **L**eftmost derivation.
  - Builds tree **top-down**.
  - Simple to understand.
- **LR**: **L**eft-to-right scan **R**ightmost derivation.
  - Builds tree **bottom-up**.
  - Can scan more `CFG`:s than **LL**, therefore more powerful.

What is the difference between **LL(1)** and **LL(2)**? Or between **LR(1)** and **LR(2)**?

- LL(k) and LR(k), where k denotes the **number** of **lookahead tokens**.
- May **look** k tokens **ahead** into the scanner in order to **decide** which nonterminal **production rule** to apply.

Construct a **recursive descent** parser for a simple language.

```
A -> B | C | D
B -> e C f D
C -> ...
D -> ...
```

Where, `terminals = {e, f}` are predefined.

LUND UNIVERSITY

Give typical examples of grammars that cannot be handled by a recursive-descent parser.

- Grammars that have **circular dependencies**, which cause **left-recursion**, and those that are ambiguous.

Explain why context-free grammars are more powerful than regular expressions.

- Because a `CFG` can **handle recursion**.

In what sense are context-free grammars "context-free"?

- CFG is not constrained to one grammar (tree), thus may **change evaluation rules** depending on "**subtree**", which can be an entire language.

What does it mean for a grammar to be **ambiguous**?

- A CFG is ambiguous if there is a sentence in the language that can be derived by **two (or more) different parse trees**.

What does it mean for **two grammars** to be **equivalent**?

- Two grammars, $G1$ and $G2$, are equivalent if they **generate the same language**.

Exemplify some common kinds of **ambiguities**.

- Operators with different **priorities**.
- Associativity of operators with **same priority**.
- Dangling `else`.

Exemplify how **expression** grammars can be **disambiguated**.

- **Subtrees** for `INT "+" INT "*" INT`, with inherent **rule-priority**.

What is the "dangling else"-problem, and how can it be solved?

- Two possible parse trees pose a problem. Three possible solutions:
  1. **Rewrite** to equivalent **unambiguous grammar** possible. However this results in more **complex** grammar.
  2. Use the ambiguous grammar:
     - Use **rule priority** such that the parser can select the **correct rule**.
     - **Works** for the dangling else problem, but **not** for **ambiguous grammars** in general.
     - Not all parser generators support this well.
  3. Change the language:
     - E.g. **add a terminal** `fi` that closes the if-statement.
     - Restrict the `then` part to be a block: { ... }.
     - Not always an option.

When should we use **canonical form**, and when **BNF** or **EBNF**?

- Canonical sequence of terminals and nonterminals.
- **BNF** (Backus-Naur Form) **alternative** productions (... | ... | ...).
- **EBNF** (Extended Backus-Naur Form) **repetition** ($*$ and +), parentheses (...).

What is a **common prefix**?

- **Two** derivations **rules** starting with **same token**.

Exemplify how a **common prefix** can be eliminated.

- Rewrite to an **equivalent grammar** without the common prefix.

What is **left factoring**?

- **Refactoring** the **leftmost** common prefix.

What is **left recursion**?

- The **leftmost expression** could go to **endless recursion**.

Exemplify how **left recursion** can be eliminated in a grammar on **canonical form**.

Left recursive:

- ```
  E -> E "+" T
  E -> T
  T -> ID
  ```

1. Rewrite to right-recursion!
   - ```
     E -> T "+" E
     E -> T
     T -> ID
     ```
2. Eliminate the common prefix. The grammar is now $LL(1)$:
   - ```
     E -> T E'
     E' -> "+" E
     E' -> ε
     T -> ID
     ```

Lund University
     ```

Exemplify how left recursion can be eliminated using EBNF.

- Left recursive:
  - ```
    E -> E "+" T
    E -> T
    T -> ID
    ```

1. Rewrite to EBNF!
   - ```
     E -> E ( "+" T )*
     E -> ID
     ```

Which parsing algorithms handle common prefixes and left recursion?

- *LL*(*k*), **no**.
- *LR*(*k*), **yes**.

LUND UNIVERSITY

How does LR differ from LL parsers?

- $LL(k)$: The tree is built **top-down**.
- $LR(k)$: The tree is built **bottom-up**.

What does it mean to **shift**?

- **Shift**: move the input token to the **top of the stack**.

What does it mean to **reduce**?

- **Reduce**: modify the stack by **applying a production**.
  1. If γ is on the top of the stack, we can pop γ and push X. This is called reducing γ to X.
  2. Accept – when the parser is about to shift \$, the parse is complete, at the same time, build this part of the tree.

Lund University

Explain how LR parsing works on an example.

- See **Lecture 05 p.28**: `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2014/lectures/L05.pdf`

What is an LR item?

- An $LR(1)$ item is: `X ->` $\alpha \bullet \beta$
- A production extended with: a dot ($\bullet$), corresponding to the position in the input sentence.
- One or more possible **lookahead** terminal symbols `t,s` (`?` when the lookahead doesn't matter).
- α and β are sequences of terminal and nonterminal symbols.

Lund University

What does an LR state consist of?

- The states in the DFA are sets of LR items.

What does it mean to take the closure of a set of LR items?

- Adding new productions for nonterminals following the dot, until no more productions can be added, is called taking the closure of the LR item set.

What do the edges in an LR DFA represent?

- Actions.

How can an LR table be constructed from an LR DFA?

1. For each edge from state j to state `k` labelled by token `t`, add an action `s k` (shift `t` and goto state `k`) to $table[j, t]$.
2. For each edge from state j to state `k` labelled by nonterminal `X`, add an action `g k` (goto state `k`) to $table[j, X]$.
3. For a state j containing an **LR** item with the dot to the left of \$, add an action `a` (accept) to $table[j, \$]$.
4. For each state j that contains an **LR** item for a production `p`, and where the dot is at the end, and the lookahead is `t`, add an action `r p` (reduce `p`) to $table[j, t]$.

LUND UNIVERSITY

How is the LR table used for parsing?

1. Use a symbol stack and a state stack
2. The current state is the state stack top.
3. Push state 1 to the state stack
4. Perform an action for each token:
   - Case Shift *s*:
     1. Push the token to the symbol stack.
     2. Push *s* to the state stack.
     3. The current state is now *s*.
   - Case Reduce p:
     1. Pop symbols for the *rhs* of p.
     2. Push the lhs symbol *X* of p.
     3. Pop the same number of states.
     4. Let $s_1 =$ the top of the state stack.
     5. Let $s_2 = table[s1, X]$.
     6. Push s_2 to the state stack.
     7. The current state is now s_2.
   - Case Accept: Report successful parse.

What is meant by a shift-reduce conflict and a reduce-reduce conflict?

- A shift-reduce conflict:

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

- A reduce-reduce conflict:

  ```
  A -> B C • t
  D -> C • t
  ```

How can such a conflict be analyzed?

- Shift-reduce conflicts can sometimes be solved with precedence rules. In particular for binary expressions with priority and associativity.
- For other cases, you need to carefully analyze the shift-reduce conflicts to see if precedence rules are applicable, or if you need to change the grammar.
- For reduce-reduce conflicts, it is advisable to think through the problems, and change the grammar.

How can precedence rules be used in an LR parser?

```
%precedence nonassoc EQ
%precedence left PLUS, MINUS
%precedence left TIMES, DIV
%precedence right POWER
```

Where the above terminals are denoted by their respective operator strings inside the scanner.

What is LR(0) and SLR parsing?

- LR(0):
  - LR items without lookahead.
  - Not very useful in practice.
- SLR (Simple LR):
  - Look at the FOLLOW set to decide where to put reduce actions.
  - Can parse many useful grammars.

What is the difference between LALR(1) and LR(1)?

- LALR(1) (Lookahead LR(1)) Merges states that have the same LR items, but different look-aheads.
- Leads to much smaller tables than LR(1).
- LR(1) Slightly more powerful than LALR(1), not used in practice.
- The tables become very large.

Explain why the LALR(1) algorithm is most commonly used in parser generators.

- Reasonably powerful. Tables do not become too large.
- Used by most well known tools: Yacc, CUP, Beaver, SableCC, etc.

What is a GLR parser?

- GLR (Generalized LR) can parse any context free grammar. Including ambiguous grammars!
- Returns a parse forest (all possible parse trees).
- Can parse grammars with shift-reduce and reduce-reduce conflicts (spawns parallel parsers).
- Has cubic worst-case complexity (in the length of the input).
- Is often much better than that in practice. But still slower than LALR. Used in several research systems.

What is the Expression Problem?

- Being able to define language constructs in a modular way.
- Define computations in a modular way.
- Compose these modules as we like, preferrably with separate compilation of the modules.
- Attain full type safety (without need for casts).

Why is solving the Expression Problem desirable for implementing compilers?

- In order to make the compiler a more modular entity.
- Visitors: an OO (Object Oriented) design pattern.
  - Modularize through clever indirect calls.
  - Not full modularization, not composition.
  - Supported by many parser generators.
  - Reasonably useful, commonly used in industry.
- Static Aspect-Oriented Programming (AOP).
  - Also known as inter-type declarations (ITDs)
  - Use new language constructs (aspects) to factor out code.
  - Solves the expression problem in a nice simple way.
  - The drawback: requires a new language: AspectJ, JastAdd.
- Advanced language constructs.
  - Use more advanced language constructs: virtual classes in *gbeta*, traits in *Scala*, typeclasses in *Haskell* etc.
  - Drawbacks: More complex than static AOP.

What are different ways of solving the Expression Problem?

- Edit the AST classes (i.e., actually not solving the problem)
- Visitors: an OO design pattern.
- Static Aspect-Oriented Programming (AOP), Advanced language constructs.

Why is it a bad idea to edit generated code?

- Non-modular, non-compositional.
- It is always a *VERY BAD IDEA* to edit generated code!
- Sometimes used anyway in industry.

What is an inter-type declaration?

Static Aspect-Oriented Programming (AOP) also known as inter-type declarations (ITDs).

What is aspect-oriented programming?

Aspect-oriented programming is a wider concept that usually focuses on dynamic behavior.

How does static AOP differ from dynamic AOP?

Read **Lecture 06 p.28** `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2014/lectures/L06.pdf`

Explain how the Visitor pattern can be implemented.

- Add boilerplate code that allows delegation to a Visitor object.

Implement a computation over the AST using static aspects.

```
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()); }
}
```

The same computations can be implemented with Visitors using *accept* methods.

Why can traversing visitors be useful?

Counting identifiers becomes sequential, as opposed to aspect-oriented programming.

What are advantages and disadvantages of static AOP as compared to Visitors?

Read **Lecture 06 p.28** `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2014/lectures/L06.pdf`

LUND UNIVERSITY

How does the Lookup pattern work?

- `decl` – the name binding.
- `lookup(String)` – finds the declaration.
- `localLookup(String)` – looks locally.
- `eq child.lookup(String)` – delegates to localLookup and lookup attributes, according to scope rules.

What is demand evaluation?

Attributes are not evaluated until demanded.

Why are attributes cached?

Eliminate complexity.

What is the Null Object pattern?

Use a real object instead of null. Give the object suitable behavior. The code becomes simpler.

What is an NTA?

An NTA is both a node and an attribute.

How does the Root Attribute pattern work?

- Make an attribute in the root visible throughout the AST.
- Solution: Add an equation in the root, propagating the value to the children.
- Expose the attribute by declaring it as inherited where it is needed.
- Or declare it in ASTNode. Then it will be visible in all nodes.

Why is it useful to implement missing declarations and unknown types as AST nodes?

- Missing declaration errors will give type checking errors as well.

How can localLookup be implemented?

Use a local hash-map which is built on the first access. After that each access is done in constant time. Resulting complexity: $O(n)$.

What is type analysis and type checking?

Check if types are used correctly.

How can unnecessary error propagation be avoided?

Propagate a reference to the Program root (Root Attribute pattern).

What is a collection attribute?

- A collection attribute holds a **composite value**.
- **Contribution rules** can declare elements that should contribute to the composite value.
- The attribute evaluator will automatically traverse the AST starting from a given root and add the contributions, using a method `m` which must be commutative.

How can a collection of error message be implemented?

- Declare a collection attribute, and make AST node contribute error string to that collection.

Construct an LL(1) table for a grammar.

See **Lecture 07 p.6** `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2014/lectures/L08.pdf`

Why can it be useful to add an end-of-file rule to some grammars?

Usefulness not explained at all in lecture slide. Dealing with End of File problem is described however:

Lecture 07 p.8 `http://fileadmin.cs.lth.se/cs/Education/EDAN65/2014/lectures/L08.pdf`

LUND UNIVERSITY

How can we decide if a grammar is LL(1) or not?

- Collision in a table entry!
- If some entry has more than one element, then the grammar is not LL(1).

What is the definition of NULLABLE, FIRST, and FOLLOW?

- FIRST(γ): the tokens that can appear first in a γ derivation
- NULLABLE(γ): can the empty string be derived from γ?
- FOLLOW(X): the tokens that can follow an X derivation

What is a fixed-point problem?

- Computing NULLABLE(X) is an example of a fixed-point problem. These problems have the form:
    - $x == f(x)$
    - Can we find a value x for which the equation holds (i.e. a solution)?
    - x is then called a fixed point of the function f.

How can it be solved using iteration?

- Fixed-point problems can (sometimes) be solved using iteration:
    1. Guess an initial value x_0.
    2. Apply the function iteratively, until the fixed point is reached.

How can we know that the computation terminates?

- The computation will terminate because:
  - The variables are only changed monotonically (from false to true).
  - The number of possible changes is finite (from all false to all true).

What is an intrinsic attribute?

The terminal symbols (like ID) are intrinsic attributes – constructed when building the AST. They are **not** defined by equations.

What is an externally visible side-effect? Why are they not allowed in the equations?

Making changes outside the object itself.

What is a circular attribute?

- The attribute may depend on itself (solved using fixed-point iteration)
- A circular attribute may depend (transitively) on itself.

How is a circular attribute evaluated?

Circularity is checked at runtime (results in exception).

How can you know if the evaluation of a circular attribute will terminate?

- The attribute values (sets of states) can be arranged in a lattice.
- The lattice is of finite height (the number of states is finite).
- The equations are monotonic: they use set union.

Give examples of properties that can be computed using circular attributes.

Among others:

- Reachability.
- Enclosing function.

What is the difference between registers and memory?

- Execution is performed through the registers.
- Memory is only used to store data.

What typical segments of memory are used?

Typically divided into different segments:

- Global data.
- Code.
- Stack.
- Heap.

What is an activation?

- The data for each method call is stored in an activation.

Why are activations put on a stack?

- To persist the order of execution.

What are FP, SP, and PC?

- FP – Frame Pointer. The first word of the current activation
- SP – Stack Pointer. The first unused word of the stack
- PC – Program counter. The currently executing instruction.

What is the static link? Is it always needed?

- Static link: Frame of enclosing method/object.
- Required for return `ret` to function, otherwise `undefined behaviour` will be triggered.

What is the dynamic link?

- Points to the frame of the calling method.

What is meant by the return address?

- Saved PC (Program Counter) - where to jump at return.

How can local variables be accessed?

- By computing offset relative FP.

How can non-local variables be accessed?

- By computing offset relative SP.

How does the compiler compute offsets for variables?

- Compiler offsets are computed by numbering each function's variables.
- Each function should have an attribute denoting the amount of variables it its subtree.

What happens at a method call?

1. Transfer the arguments and the static link.
2. Store the return address in a register and jump to code of the called procedure.
3. Allocate the new activation and move FP.
4. Run the code for p2.
5. Store the return value in a register.
6. Deallocate the activation.
7. Move FP back.
8. Jump back to the return address.
9. Save the return value if needed.
10. Continue executing in p1.

What information does the compiler need to compute in order to generate code for accessing variables? For a method call?

- For variables and argument uses.
  - The offsets to use (relative to the Frame Pointer).
  - The number of static levels to use (0 for locals).
- For method calls.
  - The number of static levels to use (0 for local methods).
- For method declarations.
  - The space needed for local declarations and temporaries.

What is meant by **calling conventions**?

- **Calling conventions**: conventions for which activation has the responsibility of saving a particular register. Often platform-specific.

What is the difference between intermediate code and assembly code?

Expressions are broken down to one operation per instruction, introducing temporary variables for each non trivial expression.

- Variables have high level symbolic names.
- Control structures are implemented using branch instructions that jump to labels.

Machine code assembly code:

- Operations can only be done on registers.
- Values in memory need to be loaded to registers before performing the operation.
- Variable names are replaced by addresses, typically relative to the frame pointer.

Mention two kinds of typical intermediate code. When are they useful?

- Three address uses temp variables, code close to ordinary register based code, good for optimization.
- Stack code uses value stack, commonly used for interpreters and virtual machines.

Why is it not meaningful to minimize the number of temporaries in intermediate code?

- Typically, the intermediate code is optimized at a later stage.
- The optimizations transform the code and introduce new temporaries.
- Temporaries are optimized as a final step, as part of register allocation.
- Trying to minimize the number of temporaries at the code generation stage is therefore meaningless.

What is register allocation?

Goal of register allocation:

- Try to keep as many variables and temporaries as possible in registers, ”spilling” as few of them as possible into memory.

What information needs be computed before generating code?

- Expression evaluation, using temporaries, local variables, formal arguments.
- Method call, passing arguments and return values.
- Method activation and return, setting up a new frame, restoring it.
- Control structures, labels and branching.

How do explicit temporaries work? How do stacked temporaries work?

- Explicit temporaries: each operation puts its result in a new `temp`.
- Stacked temporaries: each expression puts its result in `rax`.

What are the advantages and disadvantages of these implementation techniques?

- Code generation is simpler for stacked temps–we don't need to compute addresses for temps.
- To generate code for method calls, we need to evaluate the arguments from right to left, to push them in the appropriate order on the stack.
- Not all languages allow this.

How can local variable numbers be computed using attributes?

Imperatively, it would be simple: traverse the tree and give each `VarDecl` an increasing `index`.

How can unique labels be computed?

Give each statement a "pathname" relative to the function.

- E.g. 3_2 means the 2nd statement in the 3rd statement in the function.
- Generate labels such as `m_3_2_whilestart` and `m_3_2_whileend`.

LUND UNIVERSITY

What is the difference between a text and a data segment in an assembly program?

- `.data` - data segment for global data
- `.text` - text segment for code, write protected

What needs to be done to run a program in assembly code?

- Link object code with library object code into executable code.

LUND UNIVERSITY

What is the difference between dynamic and static typing? Is Java statically typed?

- Dynamic typing:
  - At runtime, every object has a type.
- Static typing:
  - At runtime, every object has a type.
  - At compile time, every variable has a type.
  - At runtime, the variable points to an object of at least that type.

What is a heap pointer?

`HP` – Heap Pointer (where to allocate next object).

How are inherited fields represented in an object?

Subclass has all the superclass fields.

What is prefixing?

- Fields of the superclass are placed in front of local fields ("prefixing").
- Each field is thus located at an offset computed at compile time, regardless of the dynamic type of the object.

How can dynamic dispatch be implemented?

- Calling methods in presence of inheritance and overriding). Two common implementation methods:
  - Virtual tables (Uses static typing. Simula, C++).
  - Hash table (For dynamic typing. Smalltalk, ...).

LUND UNIVERSITY

What is a virtual table?

Virtual tables:

- Class descriptor contains virtual table (often called "vtable").
- Pointers to superclass methods are placed in front of locally declared methods ("prefixing").
- Each method pointer is located at an offset computed at compile time, using the static type.

Why is it not straightforward to optimize object-oriented languages?

Virtual tables:

- Many small methods – not much to optimize in each.
- Virtual methods slower to call.
- Virtual methods are difficult to inline.

LUND UNIVERSITY

What is an inline call cache?

Inline call caches a way to optimize method calls at runtime.

What is a polymorphic inline cache (PIC)?

- A generalization of inline call caches, handle several possible object types.
- Inline the prologues into the calling code. Check for several types.

How can code be further optimized when call caches are used?

- Inlining method bodies.
- Copy the called methods into the calling code.

What is meant by dynamic adaptive compilation?

- Keep track of execution profile, add PICs dynamically, order cases according to frequency.
- Inline the called methods if sufficiently frequent optimize the code if sufficiently frequent.