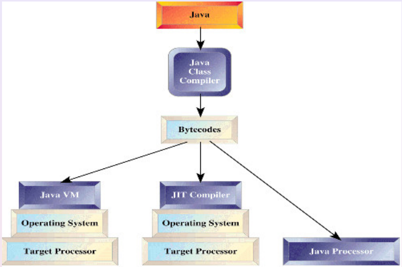
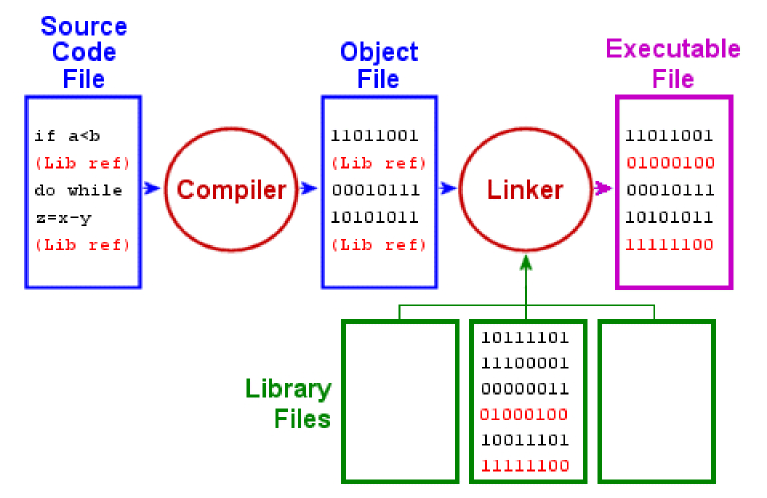
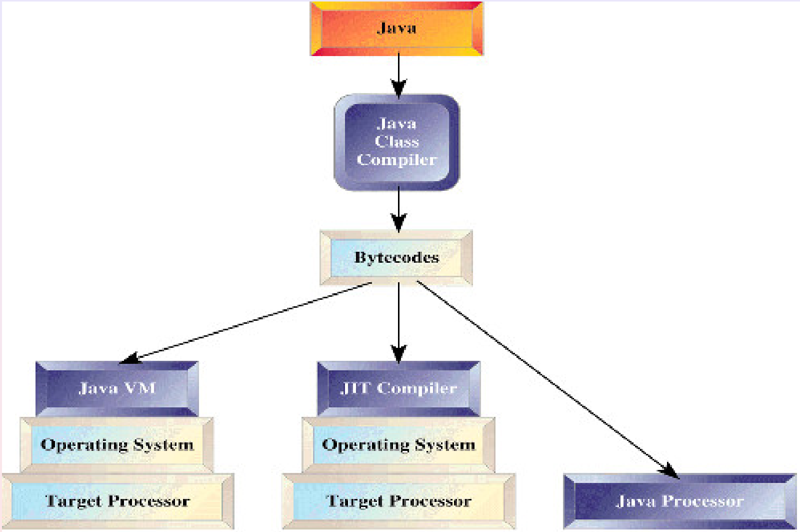


Short Notes on Compiler Construction

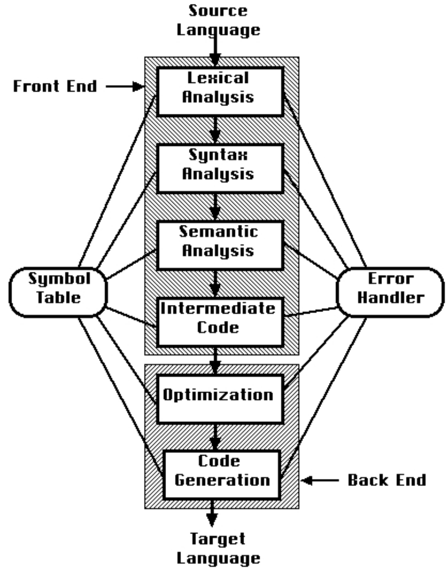
Compilation into native code



Compilation through an intermediate bytecode



Compilation phases



Lexical analysis

My open source Kitten compiler:
`https://github.com/HotMoka/Kitten`

Lexical analysis

- **input:** textual source code
- **output:** sequence of tokens
- **error:** unknown token

Input: Kitten source code

```
class Led {
  field boolean state

  constructor() {}

  method void on()
    this.state := true

  method void off()
    this.state := false

  method boolean isOn()
    return this.state

  method boolean isOff()
    return !this.state
}
```

Output: sequence of tokens

```
CLASS from 0 to 4
ID(Led) from 6 to 8
LBRACE from 10 to 10
FIELD from 14 to 18
BOOLEAN from 20 to 26
ID(state) from 28 to 32
CONSTRUCTOR from 37 to 47
LPAREN from 48 to 48
RPAREN from 49 to 49
LBRACE from 51 to 51
RBRACE from 52 to 52
METHOD from 57 to 62
VOID from 64 to 67
ID(on) from 69 to 70
...
```

Old friends: Regular expressions

An *alphabet* Λ is a finite collection of symbols (*characters*)

Regular Expressions $\mathcal{R}$ over Λ

- $\emptyset \in \mathcal{R}$ (empty set)
- $\varepsilon \in \mathcal{R}$ (empty string)
- $\Lambda \subseteq \mathcal{R}$ (single characters)
- if $r_1, r_2 \in \mathcal{R}$ then $r_1 r_2 \in \mathcal{R}$ (sequence)
- if $r_1, r_2 \in \mathcal{R}$ then $r_1 | r_2 \in \mathcal{R}$ (union)
- if $r \in \mathcal{R}$ then $r^* \in \mathcal{R}$ (iteration)

A regular expression denotes a language

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\mathcal{L}(a) = \{a\}$ for every $a \in \Lambda$
- $\mathcal{L}(r_1 r_2) = \{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1)$ and $s_2 \in \mathcal{L}(r_2)\}$
- $\mathcal{L}(r_1|r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
- $\mathcal{L}(r^*) = \{s_1 \cdots s_n \mid n \geq 0$ and $s_i \in \mathcal{L}(r)$ for all $0 \leq i \leq n\}$.

For instance

- `then` denotes the keyword $\{$`then`$\}$
- `[a-zA-Z][a-zA-Z0-9_]` denotes the identifiers

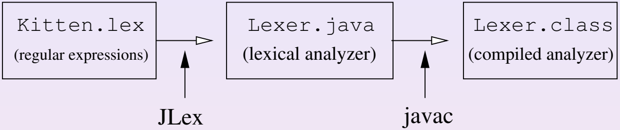
What you can <u>not</u> do with regular expressions

- you cannot define recursive languages
- you cannot count (match parentheses)

C, Java, Kitten source code is defined in a recursive way and is correct only if parentheses match

Building a lexical analyzer with JLex

https : //www.cs.princeton.edu/~appel/modern/java/JLex/




```
then                  {return tok(THEN, null);}
+                     {return tok(PLUS, null);}
:=                    {return tok(ASSIGN, null);}
[a-zA-Z][a-zA-Z0-9_]* {return tok(ID, yytext());}
[0-9]+                {return tok(INTEGER,
                                 new Integer(yytext()));}
[0-9]*.[0-9]+         {return tok(FLOATING,
                                 new Float(yytext()));}
```

Inside JLex

1. regular expressions $\Rightarrow$ NFA
2. NFA $\Rightarrow$ DFA
3. DFA $\Rightarrow$ `Lexer.java`

Disambiguation rules

- longest match
- rule priority

Syntactical analysis

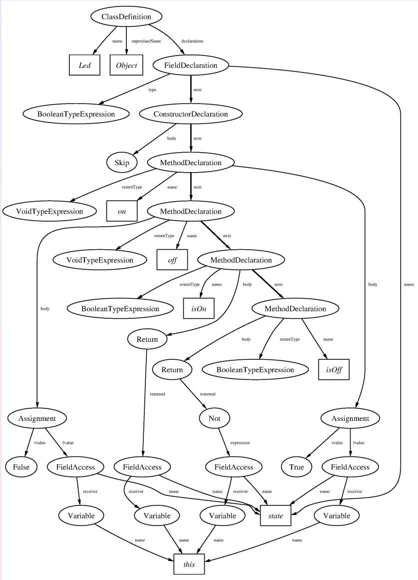
Syntactical analysis

- **input:** sequence of tokens
- **output:** abstract syntax tree
- **error:** syntax error

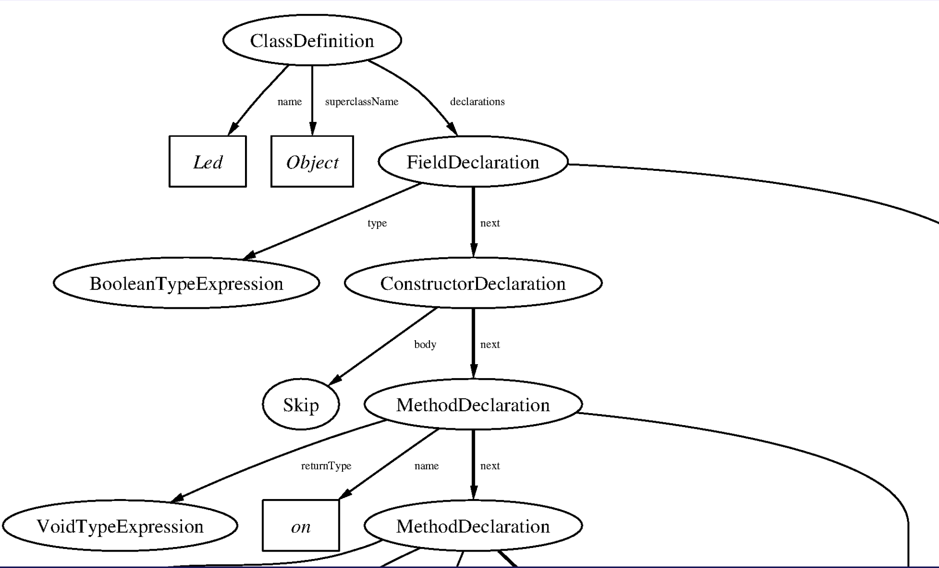
Input: sequence of tokens (yes, you have seen this already)

```
CLASS from 0 to 4
ID(Led) from 6 to 8
LBRACE from 10 to 10
FIELD from 14 to 18
BOOLEAN from 20 to 26
ID(state) from 28 to 32
CONSTRUCTOR from 37 to 47
LPAREN from 48 to 48
RPAREN from 49 to 49
LBRACE from 51 to 51
RBRACE from 52 to 52
METHOD from 57 to 62
VOID from 64 to 67
ID(on) from 69 to 70
...
```

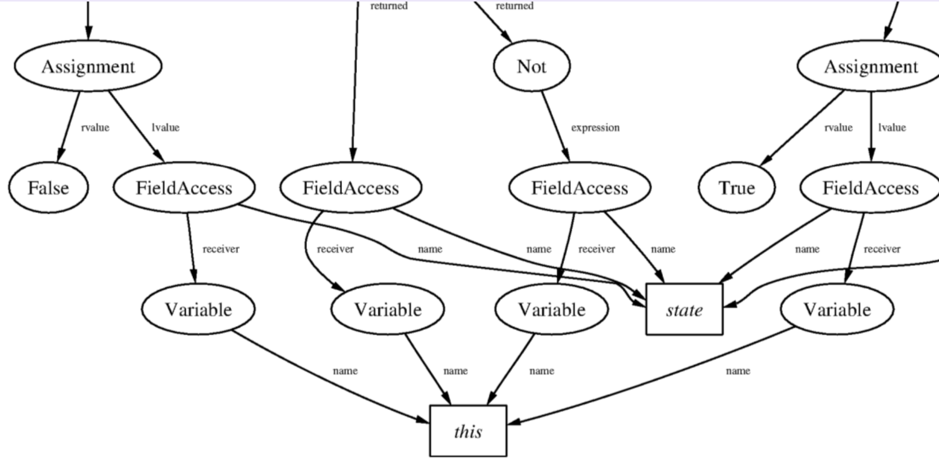
Abstract syntax *tree*



A closer look 1/2



A closer look 2/2



Context-free grammars

A *context-free grammar* over an alphabet Λ is a quadruple $\langle T, N, I, P \rangle$ where

- $T \subseteq \Lambda$ is a set of *terminals*
- N is a set of *non-terminals*
- $I \in N$ is the *starting non-terminal*
- P is a set of *productions*, that is, arrows such as $L \to \gamma$ where $L \in N$ and $\gamma \in (T \cup N)^*$.

For instance:

$$T = \{\mathtt{a}, \mathtt{b}\}$$
$$N = \{I\}$$
$$P = \{I \to \varepsilon,\ I \to \mathtt{a}I\mathtt{b}\}$$

Derivations

Given $G = \langle T, N, I, P \rangle$ we say that β is *derived in G in a step* from α iff there is $L \to \gamma \in P$ such that $\alpha = \eta L \delta$ and $\beta = \eta \gamma \delta$. We write it as $\alpha \Rightarrow \beta$. A *derivation* for G is a sequence of steps $\alpha \Rightarrow \beta_1 \Rightarrow \beta_2 \ldots$.

For instance:

$$
\begin{aligned}
\mathtt{ab}I\mathtt{b} &\Rightarrow \mathtt{aba}I\mathtt{bb} \\
\mathtt{aba}I\mathtt{bb} &\Rightarrow \mathtt{ababb} \\
I &\Rightarrow \mathtt{a}I\mathtt{b} \\
I &\Rightarrow^* I \\
\mathtt{ab}I\mathtt{b} &\Rightarrow^* \mathtt{ababb}.
\end{aligned}
$$

Language of a grammar

Given a grammar G on an alphabet Λ, its language $L(G)$ is

$$L(G) = \{\alpha \text{ ground} \mid I \Rightarrow^* \alpha\}.$$

For instance, for the previous example of grammar:

$$L(G) = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}.$$

Parse trees

A *parse tree* for $G = \langle T, N, I, P \rangle$ is a tree such that

1. its nodes are labeled with an element from N or from T or with ε;
2. its root is labeled with I
3. its leaves are labeled with elements from T or with ε
4. for every node labeled with L and its children labeled with $e_1, \ldots, e_n$ (left-to-right) we have $L \to e_1 \cdots e_n \in P$.

The frontier of the tree is the string derived by the tree from its root.

A parse tree stands for more derivations, by abstracting away the order or replacement of the non-terminals

Ambiguity 1/2

Consider the grammar

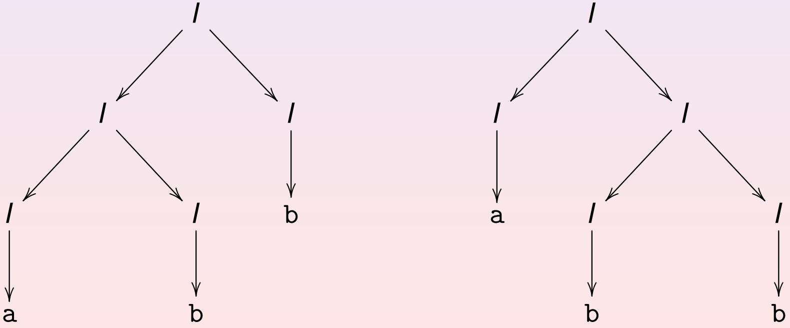
$$I \to \varepsilon$$

$$I \to \mathtt{a}$$

$$I \to \mathtt{b}$$

$$I \to II$$

It admits two parse trees for the same word abb:



Ambiguity 2/2

Why ambiguity is bad

- Ambiguity means that the same word can be given two structures, that is, potentially, two distinct interpretations
- With an ambiguous grammar, a compiler does not know which is the right interpretation of a program
- In a perfect world, grammars should be non-ambiguous but this often entails that they become complex and innatural
- We will give a concrete example later

Recursive descent parsing

$$I \rightarrow com\ \$$$
$$com \rightarrow exp\ \texttt{ASSIGN}\ \texttt{INTEGER}$$
$$exp \rightarrow \texttt{INTEGER}$$
$$exp \rightarrow \texttt{ID}$$
$$exp \rightarrow \texttt{MINUS}\ exp$$

We can implement it with a recursive descent parser:

```
public void parse() { parseI(); }
private void parseI() { parseCom(); eat(EOF); }
private void parseCom() { parseExp(); eat(ASSIGN); eat(INTEGER
private void parseExp() {
  switch (lookahead) {
    case ID: eat(ID); break;
    case INTEGER: eat(INTEGER); break;
    case MINUS: eat(MINUS); parseExp();
    default: syntax_error(lookahead);
  }
}
```

When does recursive descent parsing fail?

Recursive descent parsing fails

- when the first token is not enough to distinguish two productions for the same non-terminal
- when the grammar is left-recursive

For instance it fails for:

$$I \rightarrow A\$$$
$$A \rightarrow \mathtt{a}$$
$$A \rightarrow A\mathtt{a}$$

In practice, it fails for the grammars of all sensible programming languages

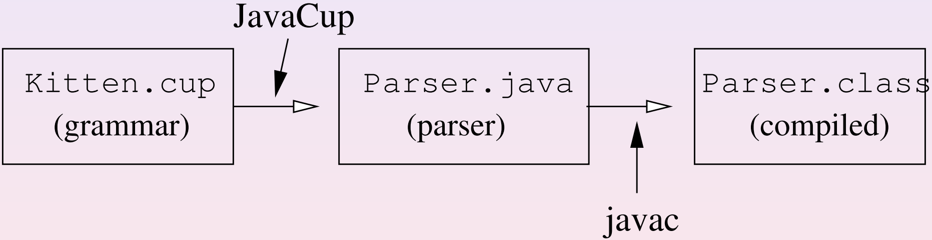
LR parsing

LR parsing uses a stack automaton to remember what has already been seen during parsing and which are the possible productions that we can use in the future, depending on the input token that will get processed by the compiler:

- details are long and boring: see any compilers book
- the construction of the parser is automatic from the grammar
- a special case of *LR* parsing is the standard approach for compiler construction
- it will anyway fail for ambiguous grammars

Automatic parser generation with JavaCup

http : //www2.cs.tum.edu/projects/cup/



The grammar of Kitten for JavaCup: types

```
type ::=
   ID
 | BOOLEAN
 | INT
 | FLOAT
 | ARRAY OF type ;

typeplus ::=
   type
 | VOID ;
```

The grammar of Kitten for JavaCup: leftvalues and expressions

```
lvalue ::=
    ID                              // variable
  | exp DOT ID                      // a field of an object
  | exp LBRACK exp RBRACK ;         // an element of an array

exp ::=
    lvalue                          // a leftvalue
  | TRUE                            // literals...
  | FALSE
  | INTEGER
  | FLOATING
  | STRING
  | NIL
  | NEW ID LPAREN expseq RPAREN     // object creation
  | NEW type LBRACK exp RBRACK      // array creation
```

The grammar of Kitten for JavaCup: more expressions

```
| exp AS type                       // cast into type
| exp PLUS exp                      // arithmetic....
| exp MINUS exp                     // AMBIGUITY
| exp TIMES exp
| exp DIVIDE exp
| MINUS exp                         // unary minus
| exp LT exp                        // comparisons...
| exp LE exp                        // AMBIGUITY
| exp GT exp
| exp GE exp
| exp EQ exp
| exp NEQ exp
| exp AND exp                       // logical operations...
| exp OR exp                        // AMBIGUITY
| NOT exp
| exp DOT ID LPAREN expseq RPAREN // method call
| LPAREN exp RPAREN ;               // parentheses
```

The grammar of Kitten for JavaCup: resolving the ambiguity

Ambiguity can be resolved with a non-ambiguous grammar (complex, innatural) or with *precedence and associativity* rules:

```
precedence left AND, OR;
precedence left NOT;
precedence nonassoc EQ, NEQ, LT, LE, GT, GE;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left UMINUS;
```

The grammar of Kitten for JavaCup: commands

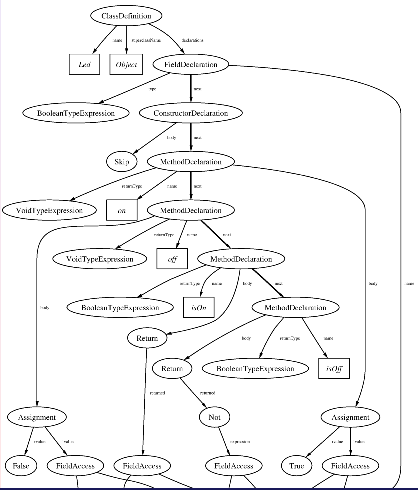
```
command ::=
    lvalue ASSIGN exp  // assignment
  | type ID ASSIGN exp // variable declaration
  | RETURN             // return from void method
  | RETURN exp         // return from non-void method
  | IF LPAREN exp RPAREN THEN command  // conditionals...
  | IF LPAREN exp RPAREN THEN command ELSE command
  | WHILE LPAREN exp RPAREN command    // while loop
  | FOR LPAREN command SEMICOLON exp
      SEMICOLON command RPAREN command // for loop
  | LBRACE statements RBRACE           // local scope
  | exp DOT ID LPAREN expseq RPAREN;  // method call, again!
```

The grammar of Kitten for JavaCup: classes

```
class ::=
   CLASS ID LBRACE class_members RBRACE
 | CLASS ID EXTENDS ID LBRACE class_members RBRACE ;

class_members ::=
 | FIELD type ID class_members
 | CONSTRUCTOR LPAREN formals RPAREN command
     class_members
 | METHOD typeplus ID LPAREN formals RPAREN command
     class_members ;
```

Building the abstract syntax

Given a grammar and a program, JavaCup checks that the program agrees with the grammar, otherwise it issues a syntax error. However, we want more: we want to build the *abstract syntax tree* of the program:



Semantical actions

JavaCup allows one to specify *semantical actions* that are fired when a derivation is used in a parse tree. We will use this possibility to build the syntax, recursively

Semantical actions for types

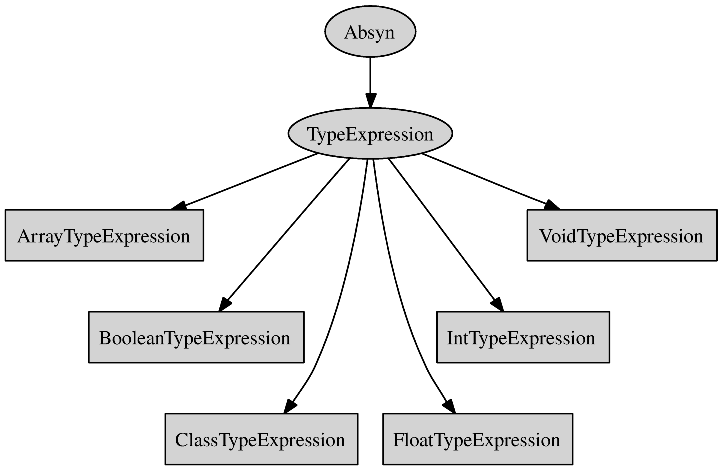
```
type ::=
   ID
 | BOOLEAN
 | INT
 | FLOAT
 | ARRAY OF type ;
```

become

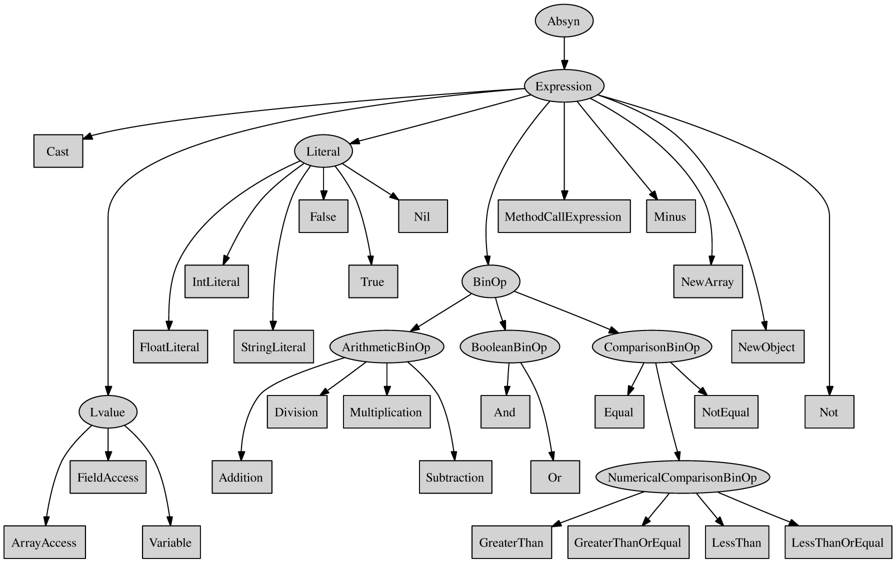
```
type ::=
   ID:id     {: RESULT = new ClassTypeExpression(idleft, id);
 | BOOLEAN:b {: RESULT = new BooleanTypeExpression(bleft); :}
 | INT:i     {: RESULT = new IntTypeExpression(ileft); :}
 | FLOAT:f   {: RESULT = new FloatTypeExpression(fleft); :}
 | ARRAY:a OF type:t
   {: RESULT = new ArrayTypeExpression(aleft, t); :} ;
```

Classes for the abstract syntax: types

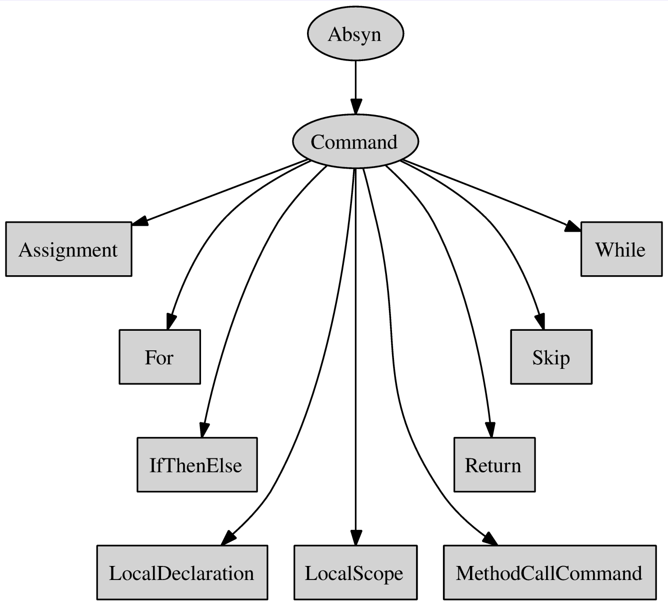
The classes for the abstract syntax are organized into a hierarchy, so that it will be simpler to share code among them in the future



Classes for the abstract syntax: expressions



Classes for the abstract syntax: commands



Semantical analysis

Semantical analysis

- **input:** abstract syntax tree
- **output:** annotated abstract syntax tree
- **error:** type error, unreachable code...

Why do we need a semantical analysis phase?

Grammars do not check for

- type consistency
- undeclared variables
- deadcode
- using the return value of a void method
- ...

The simplest implementation of those checks/inferences is through inductive definitions, that automatically translate into Java code that descends on the abstract syntax tree, recursively

Deadcode identification

```
while (x > 0) {
  x := x - 1;
  return;
  y := y + 1  <- a warning should be issued here
}
```

$$\boxed{\mathit{command} \vdash^{\mathrm{cdc}} \mathit{Boolean}}$$

command $\vdash^{\mathrm{cdc}}$ *true* means that the execution of *command* definitely ends with the last command(s) of *command*.

Deadcode rules

$$\frac{}{\texttt{Return}(\texttt{x}) \vdash^{\mathsf{cdc}} \mathit{true}}$$

$$\frac{}{\texttt{Assignment}(\texttt{lvalue}, \texttt{rvalue}) \vdash^{\mathsf{cdc}} \mathit{false}}$$

$$\frac{\texttt{then} \vdash^{\mathsf{cdc}} b_1 \quad \texttt{else} \vdash^{\mathsf{cdc}} b_2}{\texttt{IfThenElse}(\texttt{condition}, \texttt{then}, \texttt{else}) \vdash^{\mathsf{cdc}} b_1 \wedge b_2}$$

$$\frac{\text{if } com_1 \vdash^{\mathsf{cdc}} \mathit{true} \text{ issue a warning} \qquad com_2 \vdash^{\mathsf{cdc}} b}{com_1; com_2 \vdash^{\mathsf{cdc}} b}$$

$$\frac{\texttt{body} \vdash^{\mathsf{cdc}} b}{\texttt{While}(\texttt{condition}, \texttt{body}) \vdash^{\mathsf{cdc}} \mathit{false}}$$

Type inference and checking 1/4

$$\boxed{\rho \vdash \textit{expression} : \textit{type}}$$

$$\frac{\rho(\textit{name}) \text{ is defined}}{\rho \vdash \texttt{Variable}(\textit{name}) : \rho(\textit{name})}$$

$$\frac{\begin{array}{c}\rho \vdash \textit{receiver} : \kappa \quad \kappa \in \texttt{ClassType} \\ \textit{field} = \kappa.\texttt{fieldLookup}(\textit{name}) \quad \textit{field} \neq \texttt{null}\end{array}}{\rho \vdash \texttt{FieldAccess}(\textit{receiver}, \textit{name}) : \textit{field}.\text{getType}()}$$

$$\frac{\rho \vdash \textit{array} : t \quad t \in \texttt{ArrayType} \quad \rho \vdash \textit{index} : \texttt{INT}}{\rho \vdash \texttt{ArrayAccess}(\textit{array}, \textit{index}) : t.\text{get}\texttt{ElementsType}()}$$

Type inference and checking 2/4

$$\frac{}{\rho \vdash \texttt{False}() : \texttt{BOOLEAN}} \qquad \frac{}{\rho \vdash \texttt{True}() : \texttt{BOOLEAN}} \qquad \frac{}{\rho \vdash \texttt{Nil}() : \texttt{NIL}}$$

$$\frac{}{\rho \vdash \texttt{IntLiteral}() : \texttt{INT}} \qquad \frac{}{\rho \vdash \texttt{FloatLiteral}() : \texttt{FLOAT}}$$

$$\frac{}{\rho \vdash \texttt{StringLiteral}(\mathit{value}) : \texttt{ClassType.mk}(\texttt{STRING})}$$

$$\frac{\rho \vdash \mathit{expression} : \texttt{BOOLEAN}}{\rho \vdash \texttt{Not}(\mathit{expression}) : \texttt{BOOLEAN}} \qquad \frac{\rho \vdash \mathit{expression} : t \quad t \leq \texttt{FLOAT}}{\rho \vdash \texttt{Minus}(\mathit{expression}) : t}$$

$$\frac{\mathit{intoType} = \tau[\![\mathit{type}]\!] \quad \rho \vdash \mathit{expression} : \mathit{fromType} \quad \mathit{intoType} < \mathit{fromType}}{\rho \vdash \texttt{Cast}(\mathit{type}, \mathit{expression}) : \mathit{intoType}}$$

$$\frac{\texttt{ClassType.mk}(\mathit{className}) = \kappa \quad \rho \vdash \mathit{actuals} : \vec{\tau} \qquad \kappa.\texttt{constructorsLookup}(\vec{\tau}) = \{\mathit{constructor}\}}{\rho \vdash \texttt{NewObject}(\mathit{className}, \mathit{actuals}) : \kappa}$$

$$\frac{\rho \vdash \mathit{elementsType} : t \quad \rho \vdash \mathit{size} : \texttt{INT}}{\rho \vdash \texttt{NewArray}(\mathit{elementsType}, \mathit{size}) : \texttt{ArrayType.mk}(t)}$$

$$\frac{\rho \vdash \mathit{receiver} : \kappa \quad \kappa \in \texttt{ClassType} \quad \rho \vdash \mathit{actuals} : \vec{\tau} \qquad \kappa.\texttt{methodsLookup}(\mathit{name},\vec{\tau}) = \{\mathit{method}\} \quad r = \mathit{method}.\texttt{getReturnType}()}{\rho \vdash \texttt{MethodCallExpression}(\mathit{receiver}, \mathit{name}, \mathit{actuals}) : r}$$

$$\frac{\rho \vdash \mathit{left} : t_l \quad \rho \vdash \mathit{right} : t_r \quad t_l \leq \texttt{FLOAT} \quad t_r \leq \texttt{FLOAT}}{\rho \vdash \texttt{ArithmeticBinOp}(\mathit{left}, \mathit{right}) : t_l.\texttt{leastCommonSupertype}(t_r)}$$

$$\frac{\rho \vdash \mathit{left} : t_l \quad \rho \vdash \mathit{right} : t_r \quad (\text{either } t_l \leq t_r \text{ or } t_r \leq t_l)}{\rho \vdash \texttt{Equal}(\mathit{left}, \mathit{right}) : \texttt{BOOLEAN}}$$

Type inference and checking 4/4

$$\frac{t = \tau[\![type]\!] \quad \rho \vdash initialiser : i \quad i \leq t}{\rho \vdash \texttt{LocalDeclaration}(type, name, initialiser) : \rho[name \mapsto t]}$$

$$\frac{\rho \vdash lvalue : t_l \quad \rho \vdash rvalue : t_r \quad t_r \leq t_l}{\rho \vdash \texttt{Assignment}(lvalue, rvalue) : \rho}$$

$$\frac{\rho \vdash condition : \texttt{Type.BOOLEAN} \quad \rho \vdash then : \rho' \quad \rho \vdash else : \rho''}{\rho \vdash \texttt{IfThenElse}(condition, then, else) : \rho}$$

$$\frac{expression \neq \texttt{null} \quad \rho \vdash expression : t \quad \text{it is in a method that returns } r \geq t}{\rho \vdash \texttt{Return}(expression) : \rho}$$

$$\frac{\rho \vdash condition : \texttt{Type.BOOLEAN} \quad \rho \vdash body : \rho'}{\rho \vdash \texttt{While}(condition, body) : \rho}$$

Intermediate code generation

Generation of the intermediate Kitten bytecode

- **input:** annotated abstract syntax tree
- **output:** Kitten bytecode

Why not Java bytecode instead?

- JB is too low level
- JB is only implicitly typed
- JB uses integers for Booleans
- JB has many optimized variants of the same instruction
- ...

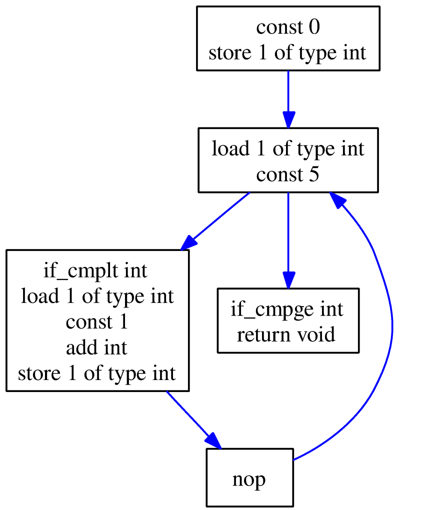
Intermediate code generation

Intermediate Kitten bytecode

```
Led():
  return void

on():
  load 0 of type Led
  const true
  putfield Led.state
  return void

off():
  load 0 of type Led
  const false
  putfield Led.state
  return void

isOn():
  load 0 of type Led
  getfield Led.state
  return boolean

isOff():
  load 0 of type Led
  getfield Led.state
  neg boolean
  return boolean
```

Intermediate code generation

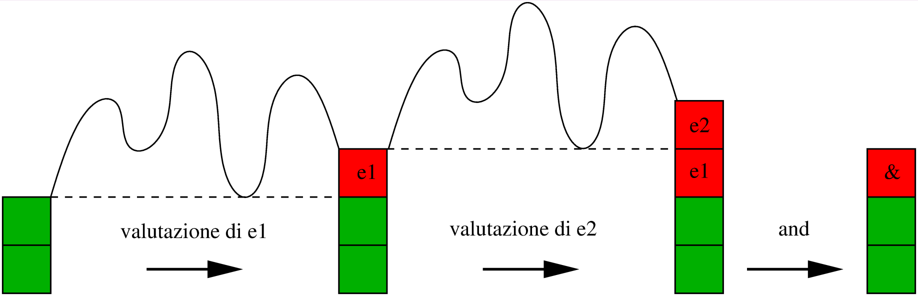
Intermediate Kitten bytecode: loops



Intermediate code generation for expressions 1/3

1. leave the original stack untouched
2. push on top the value of the expression

For instance, the evaluation of (e_1 `and` e_2) looks like:



Intermediate code generation for expressions 2/3

$$\boxed{\gamma[\![\_]\!] : \texttt{expression} \mapsto \texttt{block} \mapsto \texttt{block}}$$

$$\gamma[\![\texttt{True()}]\!](\beta) = \boxed{\texttt{const true}} \to \beta$$

$$\gamma[\![\texttt{Variable}(\mathit{name})]\!](\beta) = \boxed{\texttt{load}\ \ \mathit{num}\ \ \texttt{of}\ \ \texttt{type}\ \ \tau} \to \beta$$

$$\gamma[\![\texttt{And}(\mathit{left}, \mathit{right})]\!](\beta) = \gamma[\![\mathit{left}]\!]\left(\gamma[\![\mathit{right}]\!]\left(\boxed{\texttt{and}} \to \beta\right)\right)$$

$$\gamma[\![\texttt{FieldAccess}(\mathit{receiver}, \mathit{name})]\!](\beta) = \gamma[\![\mathit{receiver}]\!]\left(\boxed{\texttt{getfield}\ \ \mathit{field}} \to \beta\right)$$

$$\begin{aligned}\gamma[\![\texttt{ArrayAccess}(\mathit{array}, \mathit{index})]\!](\beta) \\ = \gamma[\![\mathit{array}]\!]\left(\gamma[\![\mathit{index}]\!]\left(\boxed{\texttt{arrayload from array of}\ \tau} \to \beta\right)\right)\end{aligned}$$

Intermediate code generation for expressions 3/3

$$\gamma[\![\texttt{Cast}(\mathit{type}, \mathit{expression})]\!](\beta) = \gamma[\![\mathit{expression}]\!] \left(\boxed{\texttt{cast from } \tau' \texttt{ into } \tau} \rightarrow \beta \right)$$

$$\gamma[\![\texttt{Addition}(\mathit{left}, \mathit{right})]\!](\beta) = \gamma^{\tau}[\![\mathit{left}]\!] \left(\gamma^{\tau}[\![\mathit{right}]\!] \left(\boxed{\texttt{add } \tau} \rightarrow \beta \right) \right)$$

$$\begin{aligned} &\gamma[\![\texttt{MethodCallExpression}(\mathit{receiver}, \mathit{name}, \mathit{actuals})]\!] \\ &\qquad = \gamma[\![\mathit{receiver}]\!] \left(\gamma^{\vec{t}}[\![\mathit{actuals}]\!] \left(\boxed{\texttt{virtualcall } \mathit{method}} \rightarrow \beta \right) \right) \end{aligned}$$

$$\begin{aligned} &\gamma[\![\texttt{NewObject}(\mathit{className}, \mathit{actuals})]\!](\beta) \\ &\qquad = \boxed{\begin{matrix}\texttt{new } \kappa \\ \texttt{dup } \kappa\end{matrix}} \rightarrow \gamma^{\vec{t}}[\![\mathit{actuals}]\!] \left(\boxed{\texttt{constructorcall } \mathit{con}} \rightarrow \beta \right) \end{aligned}$$

Intermediate code generation for Boolean expressions

$$\gamma^{test}[\![exp]\!](\beta_{true})(\beta_{false}) = \gamma[\![exp]\!]\left(\boxed{\texttt{nop}} \left\langle \begin{array}{l} \boxed{\texttt{if\_true}} \rightarrow \beta_{true} \\ \boxed{\texttt{if\_false}} \rightarrow \beta_{false} \end{array} \right.\right)$$

Passive compilation of leftvalues

$$\boxed{\gamma^{passive}[\![lvalue, rvalue]\!](\beta)}$$

$\texttt{Variable}(name)$

$$\gamma^{\tau}[\![rvalue]\!]\left(\boxed{\texttt{store } num \texttt{ of type } \tau} \to \beta\right)$$

$\texttt{FieldAccess}(receiver, name)$

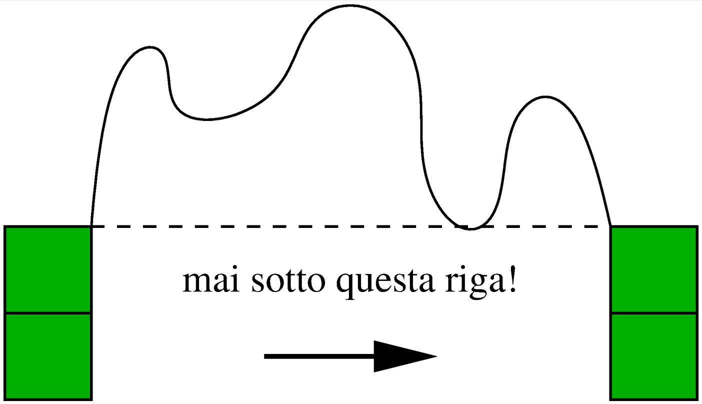
$$\gamma[\![receiver]\!]\left(\gamma^{\tau}[\![rvalue]\!]\left(\boxed{\texttt{putfield } field} \to \beta\right)\right)$$

$\texttt{ArrayAccess}(array, index)$

$$\gamma[\![array]\!]\left(\gamma[\![index]\!]\left(\gamma^{\tau}[\![rvalue]\!]\left(\boxed{\begin{array}{c}\texttt{arraystore into}\\ \texttt{array of } \tau\end{array}} \to \beta\right)\right)\right)$$

Intermediate code generation for the commands 1/2

1. leave the original stack untouched



Intermediate code generation for the commands 2/2

$$\gamma[\![\texttt{Skip()}]\!](\beta) = \beta \qquad \gamma[\![\texttt{LocalScope}(\textit{body})]\!](\beta) = \gamma[\![\textit{body}]\!](\beta)$$

$$\gamma[\![\texttt{IfThenElse}(\textit{cond}, \textit{then}, \textit{else})]\!](\beta) = \gamma^{\textit{test}}[\![\textit{cond}]\!](\gamma[\![\textit{then}]\!](\beta))(\gamma[\![\textit{else}]\!](\beta))$$

$$\gamma[\![\texttt{Assignment}(\textit{lvalue}, \textit{rvalue})]\!](\beta) = \gamma^{\textit{passive}}[\![\textit{lvalue}, \textit{rvalue}]\!](\beta)$$

$$\gamma[\![\texttt{While}(\textit{cond}, \textit{body})]\!](\beta) = \underbrace{\boxed{\texttt{nop}}}_{\textit{pivot}} \rightarrow \gamma^{\textit{test}}[\![\textit{cond}]\!](\gamma[\![\textit{body}]\!](\textit{pivot}))(\beta)$$

Java bytecode generation

Generation of the object Java bytecode

- **input:** Kitten bytecode
- **output:** Java bytecode

1. each Kitten bytecode is translated into one or more Java bytecodes, exploiting potential optimized variants
2. each block of Kitten bytecodes becomes a sequential snippet of Java bytecodes
3. such snippets are flattened into a linear sequence of Java bytecodes, by adding `goto`'s
4. everything is finally packaged into class files
5. a Java bytecode manipulation library such as BCEL does most of the work: `http://commons.apache.org/proper/commons-bcel`

Security guarantees of the Java bytecode

A controlled low-level language

- code cannot be modified at runtime ⇒ no metamorphic code
- at each given program point, number and static types of stack elements and local variables are fixed and statically known, for all possible execution paths ⇒ strong types, you cannot blow up the stack
- casts are checked ⇒ you cannot pretend that a frog is a prince
- unitialized local variables cannot be used ⇒ you cannot exploit stale values
- new objects cannot be used before calling one of their constructors ⇒ you cannot forge raw data
- all jumps go to static targets, inside the same method ⇒ you cannot dynamically build your jump targets

Identical constraints hold for the Dalvik Android bytecode