Compiler Construction

Lecture 1: Introduction & Lexing

Sjaak Smetsers Mart Lubbers Jordy Aaldering

January 30th, 2025

Radboud University

Mart Lubbers
M1 01.07
mart@cs.ru.nl

Sjaak Smetsers
M1 01.10
s.smetsers@cs.ru.nl

Jordy Aaldering
M1 01.14
jordy.aaldering@ru.nl

Part I

Introduction

What is a compiler?

Compiler

What is a compiler?

Compiler

What is a compiler?

- A program that translates written text into text written in another language.
- Why translate source code?
  - Higher level.
  - Lower level.

Compiler

What is a compiler?

- A program that translates written text into text written in another language.
- Why translate source code?
  - Higher level.
  - Lower level.

What is an interpreter

- A program that translates written text into an intermediate representation and immediately executes this.

Intrinsic merit

Compiler construction is challenging and fun

Intrinsic merit

Compiler construction is challenging and fun

- Many complex compilation steps
- Interesting conversion and analysis problems
- New architectures, new languages, new challenges

Intrinsic merit

Compiler construction is challenging and fun

- Many complex compilation steps
- Interesting conversion and analysis problems
- New architectures, new languages, new challenges

- Knowledge not only useful for making compilers:

Intrinsic merit

Compiler construction is challenging and fun

- Many complex compilation steps
- Interesting conversion and analysis problems
- New architectures, new languages, new challenges
- Knowledge not only useful for making compilers:

Compiler construction poses some of the most interesting problems in computing.

Qualities of a compiler

Qualities of a compiler

1. Correct code
2. Fast code
3. Fast compiler

Qualities of a compiler

1. Correct code
2. Fast code
3. Fast compiler
4. Proportional compile time

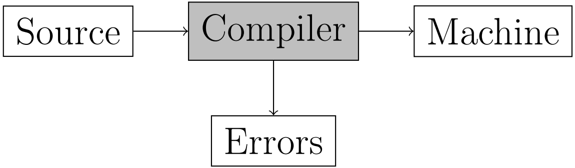
Qualities of a compiler

1. Correct code
2. Fast code
3. Fast compiler
4. Proportional compile time
5. Good diagnostics
6. Debugging support
7. Precise but flexible type system
8. Foreign function interface
9. Consistent and predictable optimisations
10. Energy saving executables
11. ...

Abstract view of a compiler

Compiler

Abstract view of a compiler

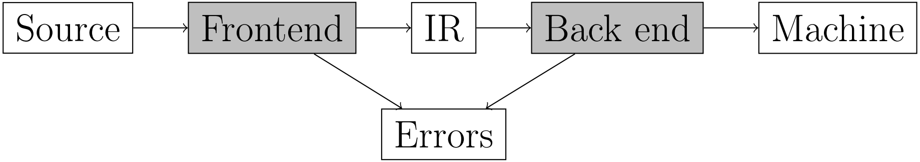


Abstract view of a compiler

Traditional two pass compiler

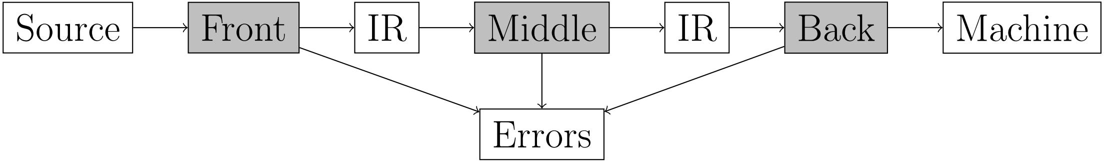
Abstract view of a compiler

Traditional two pass compiler



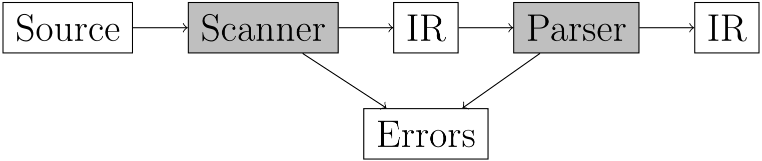
Abstract view of a compiler

Traditional ~~two~~three pass compiler

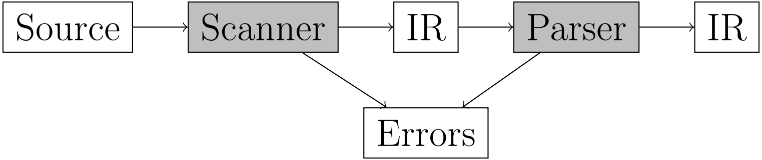


Frontend

Frontend



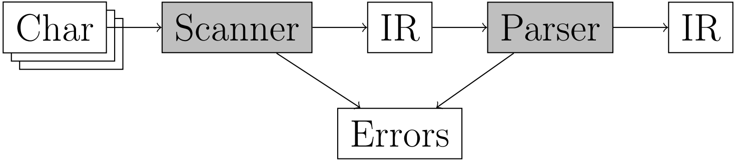
Frontend



Duty of the frontend

- Recognise the (context-free) syntax
- Produce IR
- Report errors

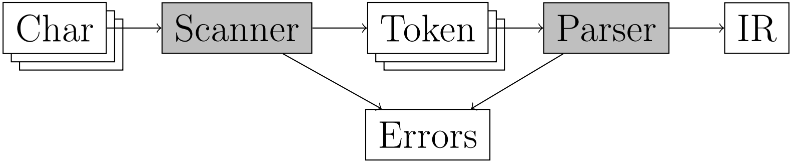
Frontend



Duty of the scanner

- Translate source code:
  `x = val + 42;`

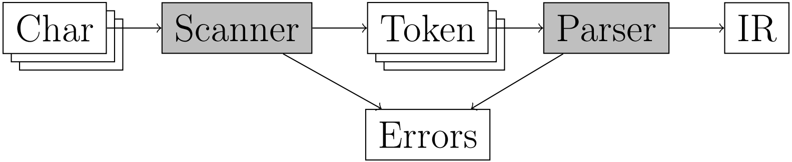
Frontend



Duty of the scanner

- Translate source code:
  `x = val + 42;`
- to regular tokens
  `<id,x> <sym,=> <id,val> <sym,+> <int,42> <sym,;>`
- and remove unneeded info
- I.e. preprocess for the parser

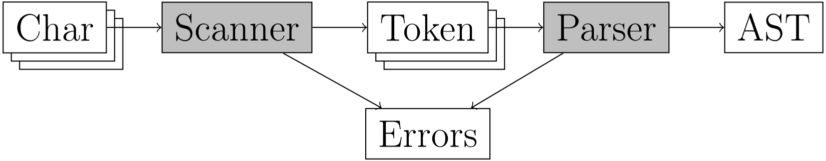
Frontend



Duty of the parser

- Translate tokens:
  <id,x> <sym,=> <id,val> <sym,+> <int,42> <sym,;>

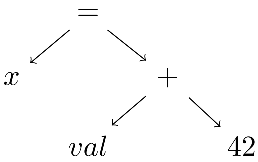
Frontend



Duty of the parser

- Translate tokens:

  `<id,x> <sym,=> <id,val> <sym,+> <int,42> <sym,;>`
- To an abstract syntax tree:



Parser

Context-free syntax

Grammar

$$
\begin{array}{rcl}
\mathit{list} & ::= & \mathit{nil} \\
 & | & \mathit{cons}\ \ \mathit{elem}\ \ \mathit{list}
\end{array}
$$

Parser

Context-free syntax

Grammar

$$\begin{array}{lcl} \mathit{list} & ::= & \mathit{nil} \\ & | & \mathit{cons}\ \ \mathit{elem}\ \ \mathit{list} \end{array}$$

Backus-Naur form (BNF)

A grammar is:

$$G = (S, N, T, P)$$

Parser

Context-free syntax

Grammar

$$
\begin{array}{lcl}
\mathit{list} & ::= & \mathit{nil} \\
 & | & \mathit{cons}\ \ \mathit{elem}\ \ \mathit{list}
\end{array}
$$

Backus-Naur form (BNF)

A grammar is:

$$G = (S, N, T, P)$$

where

S	is the start symbol
N	is the set of non-terminal symbols
T	is the set of terminal symbols
P	is the set of productions: $P : N \to (N \cup T)^+$

Expression grammar

$$
\begin{array}{rrcl}
1 & goal & ::= & expr \\
2 & expr & ::= & expr\ op\ term \\
3 & & | & term \\
4 & term & ::= & number \\
5 & & | & id \\
6 & op & ::= & + \\
7 & & | & -
\end{array}
$$

Parser

Example grammar

Expression grammar

$$
\begin{array}{llcl}
1 & goal & ::= & expr \\
2 & expr & ::= & expr\ op\ term \\
3 & & | & term \\
4 & term & ::= & number \\
5 & & | & id \\
6 & op & ::= & + \\
7 & & | & -
\end{array}
$$

BNF

$$
\begin{array}{ll}
S & = goal \\
T & = number\ , id\ , +\ , - \\
N & = goal\ , expr\ , term\ , op \\
P & = 1, 2, 3, 4, 5, 6, 7
\end{array}
$$

Given a grammar, valid sentences can be derived by repeated substitution.

Substitution

$$x + 2 - y$$

Expression grammar

$$
\begin{array}{llcl}
1 & goal & ::= & expr \\
2 & expr & ::= & expr\ op\ term \\
3 & & | & term \\
4 & term & ::= & number \\
5 & & | & id \\
6 & op & ::= & + \\
7 & & | & -
\end{array}
$$

Parser

Substitution

Given a grammar, valid sentences can be derived by repeated substitution.

Substitution

$$\begin{array}{ll} 5 & term \; + 2 - y \\ & x + 2 - y \end{array}$$

Expression grammar

$$\begin{array}{lrcl} 1 & goal & ::= & expr \\ 2 & expr & ::= & expr\ op\ term \\ 3 & & \mid & term \\ 4 & term & ::= & number \\ 5 & & \mid & id \\ 6 & op & ::= & + \\ 7 & & \mid & - \end{array}$$

Parser

Substitution

Given a grammar, valid sentences can be derived by repeated substitution.

Substitution

6	
3	$expr \ + 2 - y$
5	$term \ + 2 - y$
	$x + 2 - y$

Expression grammar

$$
\begin{array}{llcl}
1 & goal & ::= & expr \\
2 & expr & ::= & expr\ op\ term \\
3 & & \mid & term \\
4 & term & ::= & number \\
5 & & \mid & id \\
6 & op & ::= & + \\
7 & & \mid & -
\end{array}
$$

Parser

Substitution

Given a grammar, valid sentences can be derived by repeated substitution.

Substitution

1	$goal$
2	$expr$
5	$expr\ op\ term$
7	$expr\ op\ y$
2	$expr\ - y$
4	$expr\ op\ term\ - y$
6	$expr\ op\ 2 - y$
3	$expr\ + 2 - y$
5	$term\ + 2 - y$
	$x + 2 - y$

Expression grammar

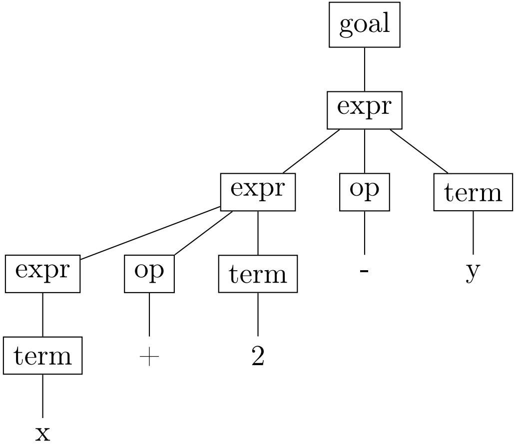
1	$goal$	$::=$	$expr$
2	$expr$	$::=$	$expr\ op\ term$
3		$\mid$	$term$
4	$term$	$::=$	$number$
5		$\mid$	id
6	op	$::=$	$+$
7		$\mid$	$-$

Parser

Parse tree

The result of a parse can be represented by a parse tree or syntax tree.

Parse tree

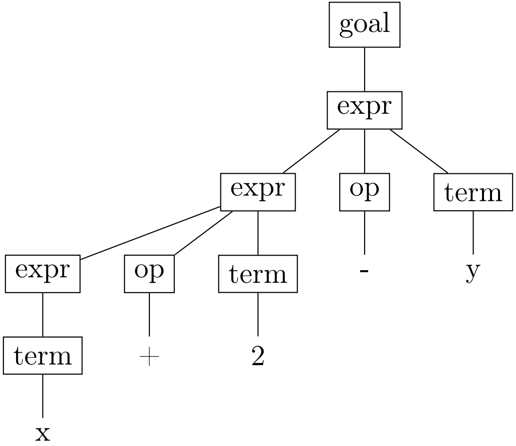


Parser

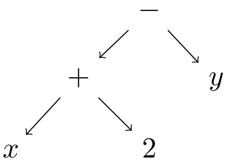
Parse tree

The result of a parse can be represented by a parse tree or syntax tree.

Parse tree

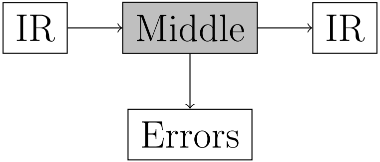


Abstract syntax tree



Middle end

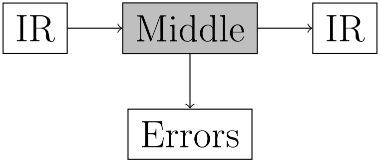
Middle end



Duty of the middle end

- Semantic analyses
- Static analyses
- Typing
- Constant propagation, folding
- Common sub-expression elimination

Middle end



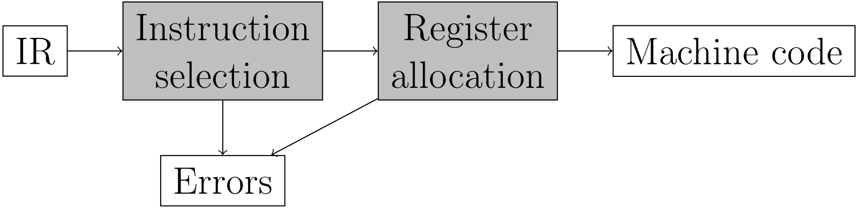
Duty of the middle end

- Semantic analyses
- Static analyses
- Typing
- Constant propagation, folding
- Common sub-expression elimination
- Redundant code elimination
- Dead code elimination
- Return path analysis
- ...

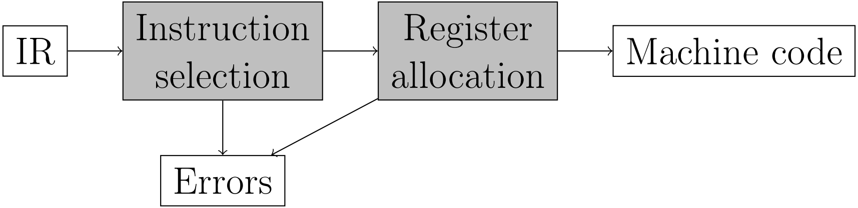
Backend

Backend

Backend


IR
Instruction selection
Register allocation
Machine code
Errors

Backend



Duty of the backend

- Translate the IR to machine code
- Decide what registers to use
- Ensure conformance with system interfaces

SPL

Simple Programming Language (SPL)

Properties

- Designed at UU

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments
- Booleans, Integers, Characters, and Lists

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments
- Booleans, Integers, Characters, and Lists
- Strong type system

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments
- Booleans, Integers, Characters, and Lists
- Strong type system
- First-order functions

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments
- Booleans, Integers, Characters, and Lists
- Strong type system
- First-order functions
- Some built-in overloaded operators

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments
- Booleans, Integers, Characters, and Lists
- Strong type system
- First-order functions
- Some built-in overloaded operators
- Automatic memory allocation but no garbage collection

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments
- Booleans, Integers, Characters, and Lists
- Strong type system
- First-order functions
- Some built-in overloaded operators
- Automatic memory allocation but no garbage collection
- Incomplete specification

Simple Programming Language (SPL)

Properties

- Designed at UU
- Pascal/C-like language
- Imperative
- Assignments
- Booleans, Integers, Characters, and Lists
- Strong type system
- First-order functions
- Some built-in overloaded operators
- Automatic memory allocation but no garbage collection
- Incomplete specification
- Compilers will differ, examples may not always run

SPL Examples

Hello world!

```
main ( ) : Void {
  print ('H':'e':'l':'l':'o':' ':'w':'o':'r':'l':'d':'!':[]);
}
```

SPL Examples

Hello world!

```
main ( ) : Void {
  print ('H':'e':'l':'l':'o':' ':'w':'o':'r':'l':'d':'!':[]);
}
```

With a mini extension:

```
main ( ) : Void {
  print ("Hello world!");
  print (42);
  print (True);
}
```

SPL Examples

Factorial

```
facR (n : Int) : Int {
  if (n < 0) {
    return 1;
  } else {
    return n * facR (n - 1);
  }
}
```

SPL Examples

Factorial

```
facR (n : Int) : Int {
  if (n < 0) {
    return 1;
  } else {
    return n * facR (n - 1);
  }
}
```

```
facI (n) {
  var r = 1;
  while (n > 1) {
    r = r * n;
    n = n - 1;
  }
  return r;
}
```

SPL Examples

List examples

```
product ( list : [Int] ) : Int {
  if (isEmpty(list)) {
    return 1;
  } else {
    return list.hd
      * product (list.tl);
  }
}
```

SPL Examples

List examples

```
product ( list : [Int] ) : Int {
  if (isEmpty(list)) {
    return 1;
  } else {
    return list.hd
      * product (list.tl);
  }
}
```

```
fromTo (from, to) {
  if (from <= to) {
    return from : fromTo (from+1,
   to);
  } else {
    return [];
  }
}
```

SPL Examples

Polymorphism

```
reverse ( list : [t] ) : [t] {
  var accu = [];
  while (isEmpty(list)) {
    accu = list.hd : accu;
    list = list.tl;
  }
  return accu;
}
```

Project

Compiler construction

6ECTS

Compiler construction

6ECTS

Compiler construction

6ECTS

In this course

Project information

- Build a compiler for SPL (Simple Programming Language)
  - Grammar and semantics provided
  - Pick your favourite language
  - Example programs provided

In this course

Project information

- Build a compiler for SPL (Simple Programming Language)
  - Grammar and semantics provided
  - Pick your favourite language
  - Example programs provided
- Target machine: SSM (Simple Stack Machine)
  - Simulator is provided
  - Example programs are available

In this course

Project information

- Build a compiler for SPL (Simple Programming Language)
  - Grammar and semantics provided
  - Pick your favourite language
  - Example programs provided
- Target machine: SSM (Simple Stack Machine)
  - Simulator is provided
  - Example programs are available
- Four phases:
  1. Lexical analyses
  2. Semantic Analyses
  3. Code generation
  4. Extension

In this course

Project information

- Build a compiler for SPL (Simple Programming Language)
  - Grammar and semantics provided
  - Pick your favourite language
  - Example programs provided
- Target machine: SSM (Simple Stack Machine)
  - Simulator is provided
  - Example programs are available
- Four phases:
  1. Lexical analyses
  2. Semantic Analyses
  3. Code generation
  4. Extension
- Write up a report in every phase
- Deadlines are on brightspace
- Tested in an oral exam

In this course

Grading & Progress

- Mandatory to work in git (`https://gitlab.science.ru.nl/compilerconstruction`).

In this course

Grading & Progress

- Mandatory to work in git (`https://gitlab.science.ru.nl/compilerconstruction`).
- Tip: use gitlab features such as CI, Milestones, issues, etc.
- The gitlab group contains other public repos:
  - `ssm`: Simple Stack Machine simulator.
  - `material`: Report skeleton, example programs.
  - `2425/`: Group containing your private repos Mart creates when the groups are known.

In this course

Grading & Progress

- Mandatory to work in git (`https://gitlab.science.ru.nl/compilerconstruction`).
- Tip: use gitlab features such as CI, Milestones, issues, etc.
- The gitlab group contains other public repos:
  - `ssm`: Simple Stack Machine simulator.
  - `material`: Report skeleton, example programs.
  - `2425/`: Group containing your private repos Mart creates when the groups are known.
- Done in groups of two.
- Every phase ends with a presentation and a report section.
- Every group presents one phase.

In this course

Grading & Progress

- Mandatory to work in git (`https://gitlab.science.ru.nl/compilerconstruction`).
- Tip: use gitlab features such as CI, Milestones, issues, etc.
- The gitlab group contains other public repos:
  - `ssm`: Simple Stack Machine simulator.
  - `material`: Report skeleton, example programs.
  - `2425/`: Group containing your private repos Mart creates when the groups are known.
- Done in groups of two.
- Every phase ends with a presentation and a report section.
- Every group presents one phase.
- Compiler extension is a case study.
- As soon as possible: Choose a partner and a language (brightspace group enroll).
- During the course: Think about a nice extension and discuss this with us.

Version control with git

- We will create a repo for you and grant you access.

[1] jordy.aaldering@ru.nl

- We will create a repo for you and grant you access.

[1] jordy.aaldering@ru.nl

Version control with git

- We will create a repo for you and grant you access.
- Setup your `.gitignore` carefully.

[1] `jordy.aaldering@ru.nl`

Version control with git

- We will create a repo for you and grant you access.
- Setup your `.gitignore` carefully.
- Never use force flags (`-f`) flags.

[1] `jordy.aaldering@ru.nl`

Version control with git

- We will create a repo for you and grant you access.
- Setup your `.gitignore` carefully.
- Never use force flags (`-f`) flags.
- Ask Jordy[1] for guidance with complex matters.

[1] `jordy.aaldering@ru.nl`

Version control with git

- We will create a repo for you and grant you access.
- Setup your `.gitignore` carefully.
- Never use force flags (`-f`) flags.
- Ask Jordy[1] for guidance with complex matters.
- If you want to understand it better:
  Git from the Bottom Up — John Wiegley:
  `https://jwiegley.github.io/git-from-the-bottom-up/`

[1] `jordy.aaldering@ru.nl`

Take home

- Log in on gitlab.

Take home

- Log in on gitlab.
- Enroll for a group.

Take home

- Log in on gitlab.
- Enroll for a group.
- Check the schedule *Contents*→*Overview*.

Take home

- Log in on gitlab.
- Enroll for a group.
- Check the schedule *Contents→Overview*.
- In case of questions/ideas/wishes
  don't hesitate to contact us.

Take home

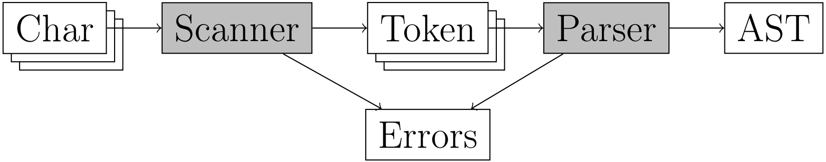
- Log in on gitlab.
- Enroll for a group.
- Check the schedule *Contents*→*Overview*.
- In case of questions/ideas/wishes don't hesitate to contact us.
- N.B. The tutorial session is a Q&A session, send an email to register (`mart@cs.ru.nl`).

Part II

Scanners

Recap

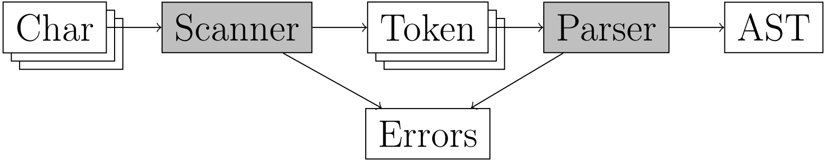
Remember the frontend



Duty of the frontend

- Recognise the (context-free) syntax
- Produce IR
- Report errors

Remember the frontend



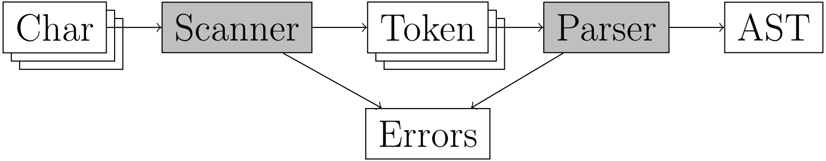
Duty of the scanner

- Translate source code:

  `x = val + 42;`
- to regular tokens

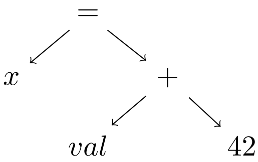
  `<id,x> <sym,=> <id,val> <sym,+> <int,42> <sym,;>`
- and remove unneeded info
- I.e. preprocess for the parser

Remember the frontend



Duty of the parser

- Translate tokens:
  <id,x> <sym,=> <id,val> <sym,+> <int,42> <sym,;>
- To an abstract syntax tree:



Scanner

Specifying allowed input

Allowed input

- Parser: CFG (sentences)
- Scanner: RE (words)

Specifying allowed input

Allowed input

- Parser: CFG (sentences)
- Scanner: RE (words)

Easy tokens

- Keywords: `case`, `if`, `module`, `while`
- Comments: anything after `//` or between `/*` and `*/`
- Whitespace

Specifying allowed input

Allowed input

- Parser: CFG (sentences)
- Scanner: RE (words)

Easy tokens

- Keywords: `case`, `if`, `module`, `while`
- Comments: anything after `//` or between `/*` and `*/`
- Whitespace

Tricky tokens

- Integers: perhaps a sign followed by digits
- Decimal: integer followed by a `'.'`, scientific notation?
- Identifier: `'_'` or letter followed by letters, digits or `'_'`.
- Characters: Single character between single quotes, or an escape sequence.

Regular expressions

$$letter \quad ::= \quad a \mid b \mid \ \ldots \ \mid z \mid A \mid B \mid \ \ldots \ \mid Z$$

Regular expressions

$$
\begin{aligned}
\mathit{letter} \quad &::= \quad a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\
\mathit{digit} \quad &::= \quad 0 \mid 1 \mid \ldots \mid 9
\end{aligned}
$$

Regular expressions

$$\begin{array}{lcl} \mathit{letter} & ::= & a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\ \mathit{digit} & ::= & 0 \mid 1 \mid \ldots \mid 9 \\ \mathit{ident} & ::= & \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^* \end{array}$$

Regular expressions

$$
\begin{array}{lcl}
\mathit{letter} & ::= & a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\
\mathit{digit} & ::= & 0 \mid 1 \mid \ldots \mid 9 \\
\mathit{ident} & ::= & \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^* \\
\\
\mathit{integer} & ::= & [+ \mid -]\ (0 \mid (1 \mid \ldots \mid 9)\ \mathit{digit}^*)
\end{array}
$$

Regular expressions

$$
\begin{array}{lcl}
\mathit{letter} & ::= & a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\
\mathit{digit} & ::= & 0 \mid 1 \mid \ldots \mid 9 \\
\mathit{ident} & ::= & \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^* \\
\\
\mathit{integer} & ::= & [+ \mid -]\ (0 \mid (1 \mid \ldots \mid 9)\ \mathit{digit}^*) \\
\mathit{decimal} & ::= & \mathit{integer}\ .\ \mathit{digit}^*
\end{array}
$$

Regular expressions

$$
\begin{array}{lcl}
\mathit{letter} & ::= & a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\
\mathit{digit} & ::= & 0 \mid 1 \mid \ldots \mid 9 \\
\mathit{ident} & ::= & \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^* \\
 & & \\
\mathit{integer} & ::= & [+ \mid -]\ (0 \mid (1 \mid \ldots \mid 9)\ \mathit{digit}^*) \\
\mathit{decimal} & ::= & \mathit{integer}\ .\ \mathit{digit}^* \\
\mathit{real} & ::= & (\mathit{integer} \mid \mathit{decimal})\ (E \mid e)\ \mathit{integer}
\end{array}
$$

Regular expressions

$$
\begin{array}{lcl}
\mathit{letter} & ::= & a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\
\mathit{digit} & ::= & 0 \mid 1 \mid \ldots \mid 9 \\
\mathit{ident} & ::= & \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^* \\
\\
\mathit{integer} & ::= & [+ \mid -]\ (0 \mid (1 \mid \ldots \mid 9)\ \mathit{digit}^*) \\
\mathit{decimal} & ::= & \mathit{integer}\ .\ \mathit{digit}^* \\
\mathit{real} & ::= & (\mathit{integer} \mid \mathit{decimal})\ (E \mid e)\ \mathit{integer}
\end{array}
$$

Recognising regular expressions

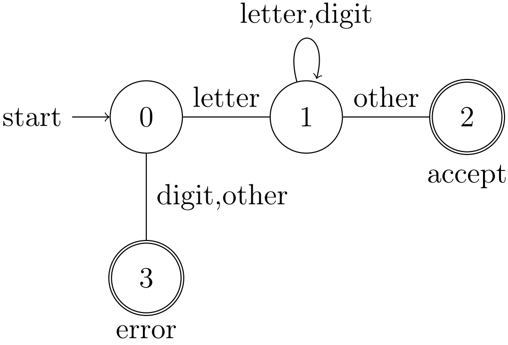
Construct a DFA

$$\begin{array}{lcl} \mathit{letter} & ::= & a \mid b \mid \ \ldots \ \mid z \mid A \mid B \mid \ \ldots \ \mid Z \\ \mathit{digit} & ::= & 0 \mid 1 \mid \ \ldots \ \mid 9 \\ \mathit{ident} & ::= & \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^* \end{array}$$

Recognising regular expressions

Construct a DFA

$$
\begin{array}{lcl}
\mathit{letter} & ::= & a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \\
\mathit{digit} & ::= & 0 \mid 1 \mid \ldots \mid 9 \\
\mathit{ident} & ::= & \mathit{letter}\ (\mathit{letter} \mid \mathit{digit})^*
\end{array}
$$



Haskell Example

Creating a lexer by hand

In Haskell: `scanner.hs`

```
data Token
    = MinusToken | PlusToken | TimesToken | DivideToken
    | BraceOpen | BraceClose
    | NumToken Integer | IdentToken String
    | ErrToken String
  deriving (Show, Eq)
```

Creating a lexer by hand

In Haskell: `scanner.hs`

```
data Token
    = MinusToken | PlusToken | TimesToken | DivideToken
    | BraceOpen | BraceClose
    | NumToken Integer | IdentToken String
    | ErrToken String
  deriving (Show, Eq)

tokenise :: [Char] → [Token]
tokenise ('/':'*':xs) = gulp xs
```

Creating a lexer by hand

In Haskell: `scanner.hs`

```
data Token
    = MinusToken | PlusToken | TimesToken | DivideToken
    | BraceOpen | BraceClose
    | NumToken Integer | IdentToken String
    | ErrToken String
  deriving (Show, Eq)

tokenise :: [Char] → [Token]
tokenise ('/':'*':xs) = gulp xs
    where
        gulp ('*':'/':rest) = tokenise rest
        gulp (c:rest) = gulp rest
        gulp [] = []
```

Creating a lexer by hand

In Haskell: `scanner.hs`

```
data Token
    = MinusToken | PlusToken | TimesToken | DivideToken
    | BraceOpen | BraceClose
    | NumToken Integer | IdentToken String
    | ErrToken String
  deriving (Show, Eq)

tokenise :: [Char] → [Token]
tokenise ('/':'*':xs) = gulp xs
    where
        gulp ('*':'/':rest) = tokenise rest
        gulp (c:rest) = gulp rest
        gulp [] = []
tokenise ('/':'/':xs) = tokenise $ dropWhile (/='\n') xs
```

Creating a lexer by hand

In Haskell: `scanner.hs`

```
data Token
    = MinusToken | PlusToken | TimesToken | DivideToken
    | BraceOpen | BraceClose
    | NumToken Integer | IdentToken String
    | ErrToken String
  deriving (Show, Eq)

tokenise :: [Char] → [Token]
tokenise ('/':'*':xs) = gulp xs
    where
        gulp ('*':'/':rest) = tokenise rest
        gulp (c:rest) = gulp rest
        gulp [] = []
tokenise ('/':'/':xs) = tokenise $ dropWhile (/='\n') xs
tokenise ('(':xs) = BraceOpen:tokenise xs
```

Creating a lexer by hand

In Haskell: `scanner.hs`

```haskell
data Token
    = MinusToken | PlusToken | TimesToken | DivideToken
    | BraceOpen | BraceClose
    | NumToken Integer | IdentToken String
    | ErrToken String
  deriving (Show, Eq)

tokenise :: [Char] → [Token]
tokenise ('/':'*':xs) = gulp xs
    where
        gulp ('*':'/':rest) = tokenise rest
        gulp (c:rest) = gulp rest
        gulp [] = []
tokenise ('/':'/':xs) = tokenise $ dropWhile (/='\n') xs
tokenise ('(':xs) = BraceOpen:tokenise xs
tokenise (')':xs) = BraceClose:tokenise xs
```

Creating a lexer by hand

In Haskell: `scanner.hs`

```
data Token
    = MinusToken | PlusToken | TimesToken | DivideToken
    | BraceOpen | BraceClose
    | NumToken Integer | IdentToken String
    | ErrToken String
  deriving (Show, Eq)

tokenise :: [Char] → [Token]
tokenise ('/':'*':xs) = gulp xs
    where
        gulp ('*':'/':rest) = tokenise rest
        gulp (c:rest) = gulp rest
        gulp [] = []
tokenise ('/':'/':xs) = tokenise $ dropWhile (/='\n') xs
tokenise ('(':xs) = BraceOpen:tokenise xs
tokenise (')':xs) = BraceClose:tokenise xs
...
```

Recognising regular expressions

Haskell code: `scanner.hs` (2)

```
...

tokenise (c:xs)
    | isSpace c = tokenise xs
```

Recognising regular expressions

Haskell code: `scanner.hs` (2)

```
...

tokenise (c:xs)
    | isSpace c = tokenise xs
    | isDigit c = spanToken isDigit (NumToken . read) (c:xs)
```

Recognising regular expressions

Haskell code: `scanner.hs` (2)

```
...

tokenise (c:xs)
    | isSpace c = tokenise xs
    | isDigit c = spanToken isDigit (NumToken . read) (c:xs)
    | isAlpha c = spanToken isAlphaNum IdentToken (c:xs)
```

Recognising regular expressions

Haskell code: `scanner.hs` (2)

```
...

tokenise (c:xs)
    | isSpace c = tokenise xs
    | isDigit c = spanToken isDigit (NumToken . read) (c:xs)
    | isAlpha c = spanToken isAlphaNum IdentToken (c:xs)
    | otherwise = ErrToken ("Unrecognised character: " ++ show c) :
   tokenise xs
```

Recognising regular expressions

Haskell code: `scanner.hs` (2)

```
...

tokenise (c:xs)
    | isSpace c = tokenise xs
    | isDigit c = spanToken isDigit (NumToken . read) (c:xs)
    | isAlpha c = spanToken isAlphaNum IdentToken (c:xs)
    | otherwise = ErrToken ("Unrecognised character: " ++ show c) :
   tokenise xs
tokenise [] = []
```

Recognising regular expressions

Haskell code: `scanner.hs` (2)

```
...

tokenise (c:xs)
    | isSpace c = tokenise xs
    | isDigit c = spanToken isDigit (NumToken . read) (c:xs)
    | isAlpha c = spanToken isAlphaNum IdentToken (c:xs)
    | otherwise = ErrToken ("Unrecognised character: " ++ show c) :
   tokenise xs
tokenise [] = []

spanToken :: (Char → Bool) → ([Char] → Token) → [Char] → [Token]
spanToken pred tfun cs = tfun tchars:tokenise rest
  where (tchars, rest) = span pred cs
```

C/C++ Example

Lexer generators

using (f)lex in C/C++

- Extra compilation step
  - `lex -t scanner.l > scanner.c`
  - `cc scanner.c -o scanner`
  - or use make.
- Generate efficient scanner code
- Generate an automaton as a jump table
- lex for C/C++, alex for Haskell, see: `https://en.wikipedia.org/wiki/Comparison_of_parser_generators#Regular_languages`

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
```

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
"//".*\n        printf("line comment\n");
```

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
"//".*\n        printf("line comment\n");
[ \n\t]         ; // eat whitespace
```

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
"//".*\n        printf("line comment\n");
[ \n\t]         ; // eat whitespace
\-              printf("minus\n");
```

Lexer generators using (f)lex in C/C++

`scanner.l`

```
%{
#include <stdio.h>
%}
%%
"//".*\n        printf("line comment\n");
[ \n\t]         ; // eat whitespace
\-              printf("minus\n");
"+"             printf("plus\n");
"*"             printf("times\n");
"/"             printf("divide\n");
"("             printf("brace open\n");
")"             printf("brace close\n");
```

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
"//".*\n       printf("line comment\n");
[ \n\t]        ; // eat whitespace
\-             printf("minus\n");
...
[+-]?[0-9]+    printf("number: %d\n", atoi(yytext));
[_a-zA-Z][_a-zA-Z0-9-]* printf("ident: %s\n", yytext);
```

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
"//".*\n         printf("line comment\n");
[ \n\t]          ; // eat whitespace
\-               printf("minus\n");
...
[+-]?[0-9]+      printf("number: %d\n", atoi(yytext));
[_a-zA-Z][_a-zA-Z0-9-]* printf("ident: %s\n", yytext);
.                printf("illegal character: '%c' (%02x)\n", yytext[0],
    yytext[0]);
```

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
"//".*\n       printf("line comment\n");
[ \n\t]        ; // eat whitespace
\-             printf("minus\n");
...
[+-]?[0-9]+    printf("number: %d\n", atoi(yytext));
[_a-zA-Z][_a-zA-Z0-9-]* printf("ident: %s\n", yytext);
.              printf("illegal character: '%c' (%02x)\n", yytext[0],
   yytext[0]);
%%
```

Lexer generators using (f)lex in C/C++

scanner.l

```
%{
#include <stdio.h>
%}
%%
"//".*\n      printf("line comment\n");
[ \n\t]       ; // eat whitespace
\-            printf("minus\n");
...
[+-]?[0-9]+   printf("number: %d\n", atoi(yytext));
[_a-zA-Z][_a-zA-Z0-9-]* printf("ident: %s\n", yytext);
.             printf("illegal character: '%c' (%02x)\n", yytext[0],
   yytext[0]);
%%
int main (void)
{
  return yylex();
}
```

Lexer generators using (f)lex in C/C++

`scanner.l` (2) Multiline comments

- Complex regex

Lexer generators using (f)lex in C/C++

`scanner.l` (2) Multiline comments

- Complex regex
- `"/*"([^*]|(\*+[^*/]))*\*+\/`

Lexer generators using (f)lex in C/C++

`scanner.l` (2) Multiline comments

- Complex regex
- `"/*"([^*]|(\*+[^*/]))*\*+\/`
- Start conditions (context)

Lexer generators using (f)lex in C/C++

`scanner.l` (2) Multiline comments

- Complex regex
- `"/*"([^*]|(\*+[^*/]))*\*+\/`
- Start conditions (context)
- Conditionals in the automaton

Lexer generators using (f)lex in C/C++

`scanner.l` (2) Multiline comments

- Complex regex
- `"/*"([^*]|(\*+[^*/]))*\*+\/`
- Start conditions (context)
- Conditionals in the automaton

Lexer generators using (f)lex in C/C++

`scanner.l` (2) Multiline comments

- Complex regex
- `"/*"([^*]|(\*+[^*/]))*\*+\/`
- Start conditions (context)
- Conditionals in the automaton

```
%x IN_COMMENT
...
%%
<INITIAL>{
"/*"            BEGIN(IN_COMMENT);
"//".*\n        printf("line comment\n");
...
}
```

Lexer generators using (f)lex in C/C++

`scanner.l` (2) Multiline comments

- Complex regex
- `"/*"([^*]|(\*+[^*/]))*\*+\/`
- Start conditions (context)
- Conditionals in the automaton

```
%x IN_COMMENT
...
%%
<INITIAL>{
"/*"            BEGIN(IN_COMMENT);
"//".*\n        printf("line comment\n");
...
}
<IN_COMMENT>{
"*/"            BEGIN(INITIAL);
[^*\n]+         ; // eat comment in chunks
"*"             ; // eat the lone star
\n              yylineno++;
}
```

Conclusion

Scanners in practise

- Most languages have regular tokens.

Scanners in practise

- Most languages ~~have~~pretend to have regular tokens.

Scanners in practise

- Most languages ~~have~~pretend to have regular tokens.
- e.g. Nested comments.

Scanners in practise

- Most languages ~~have~~pretend to have regular tokens.
- e.g. Nested comments.
- Some deem lexers obsolete (lecture 3).

Scanners in practise

- Most languages ~~have~~pretend to have regular tokens.
- e.g. Nested comments.
- Some deem lexers obsolete (lecture 3).
- Many languages have lexers written by hand

Scanners in practise

- Most languages ~~have~~pretend to have regular tokens.
- e.g. Nested comments.
- Some deem lexers obsolete (lecture 3).
- Many languages have lexers written by hand
- Many languages use lexer generators

Scanners in practise

- Most languages ~~have~~pretend to have regular tokens.
- e.g. Nested comments.
- Some deem lexers obsolete (lecture 3).
- Many languages have lexers written by hand
- Many languages use lexer generators
- There is such a thing as the Lexer hack (lecture 3)

Abstracter view of compilers (extra)

Abstracter view of compilers (extra)

- Notation: the outcome of a program P, written in language A, on input x:

$$\llbracket P \rrbracket^{A}(x)$$

Abstracter view of compilers (extra)

- Notation: the outcome of a program P, written in language A, on input x:

  $$[\![P]\!]^A(x)$$

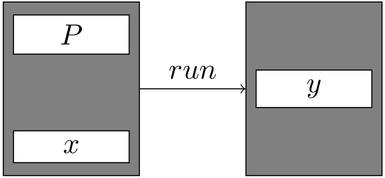
- How can we compute $[\![P]\!]^A(x)$ on a computer?

Abstracter view of compilers (extra)

- Notation: the outcome of a program P, written in language A, on input x:

  $$[\![P]\!]^A(x)$$

- How can we compute $[\![P]\!]^A(x)$ on a computer?
- Special case: P in machine language M



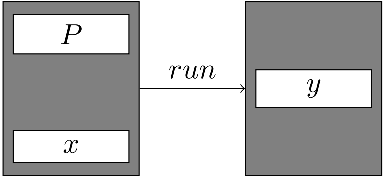
$$y = [\![P]\!]^M(x)$$

Abstracter view of compilers (extra)

- Notation: the outcome of a program P, written in language A, on input x:

$$[\![P]\!]^{A}(x)$$

- How can we compute $[\![P]\!]^{A}(x)$ on a computer?
- Special case: P in machine language M

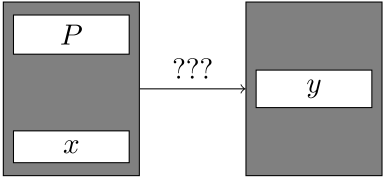


$$y = [\![P]\!]^{M}(x)$$

- General case?

Implementation (extra)

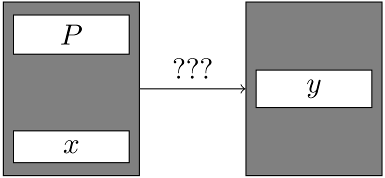
- An implementation of a language A is a method with which we can compute $[\![P]\!]^A(x)$ (for arbitrary P and x):



$$y = [\![P]\!]^A(x)$$

Implementation (extra)

- An implementation of a language A is a method with which we can compute $[\![P]\!]^A(x)$ (for arbitrary P and x):



$$y = [\![P]\!]^A(x)$$

- We already have an implementation for a machine language. What to do with other languages?

Compilers (extra)

- A compiler from A to B (written in C) is a program F such that for any P and x

$$\left[\!\left[[\![F]\!]^C (P) \right]\!\right]^B (x)$$

Compilers (extra)

- A compiler from A to B (written in C) is a program F such that for any P and x

$$\left[\!\left[[\![F]\!]^C (P) \right]\!\right]^B (x)$$

Compilers (extra)

- A compiler from A to B (written in C) is a program F such that for any P and x

$$\left[\!\left[[\![F]\!]^{C}(P) \right]\!\right]^{B}(x) = [\![P]\!]^{A}(x)$$

Compilers (extra)

- A compiler from A to B (written in C) is a program F such that for any P and x

$$\left[\!\left[[\![F]\!]^C (P) \right]\!\right]^B (x) = [\![P]\!]^A (x)$$

- In other words: $[\![F]\!]^C (P)$ does the same as P (but in a different language)

Compilers (extra)

- A compiler from A to B (written in C) is a program F such that for any P and x

$$\left[\!\left[[\![F]\!]^{C}(P) \right]\!\right]^{B}(x) = [\![P]\!]^{A}(x)$$

- In other words: $[\![F]\!]^{C}(P)$ does the same as P (but in a different language)
- Notation: $F : A \xrightarrow{C} B$

Applicative style (extra)

- We introduce a more concise notation

$$P \underset{A}{\cdot} x = [\![P]\!]^{A}(x)$$

Applicative style (extra)

- We introduce a more concise notation

$$P \underset{A}{\cdot} x = [\![P]\!]^A(x)$$

- Special case: machine language, $A = M$

$$P \underset{M}{\cdot} x = P \,!\, x$$

Applicative style (extra)

- We introduce a more concise notation

$$P \underset{A}{\cdot} x = [\![P]\!]^{A}(x)$$

- Special case: machine language, $A = M$

$$P \underset{M}{\cdot} x = P \, ! \, x$$

- Example: if $F : A \xrightarrow{C} B$ then

$$F \underset{C}{\cdot} P \underset{B}{\cdot} x = P \underset{A}{\cdot} x$$

Applicative style (extra)

- We introduce a more concise notation

$$P \underset{A}{\cdot} x = [\![P]\!]^A(x)$$

- Special case: machine language, $A = M$

$$P \underset{M}{\cdot} x = P \,!\, x$$

- Example: if $F : A \xrightarrow{C} B$ then

$$F \underset{C}{\cdot} P \underset{B}{\cdot} x = P \underset{A}{\cdot} x$$

- Applications associate to the left

$$F \underset{C}{\cdot} P \underset{B}{\cdot} x = (F \underset{C}{\cdot} P) \underset{B}{\cdot} x$$

Example 1 (extra)

- We have a compiler F (in machine language) from Java to machine language

$$F : J \xrightarrow{M} M$$

Example 1 (extra)

- We have a compiler F (in machine language) from Java to machine language

$$F : J \overset{M}{\to} M$$

- Wanted: an implementation in Java

Example 1 (extra)

- We have a compiler F (in machine language) from Java to machine language

$$F : J \xrightarrow{M} M$$

- Wanted: an implementation in Java
- We express $P \underset{J}{\cdot} x$ in terms of !

$$P \underset{J}{\cdot} x \quad = (F \underset{M}{\cdot} P) \underset{M}{\cdot} x$$

Example 1 (extra)

- We have a compiler F (in machine language) from Java to machine language

$$F : J \overset{M}{\to} M$$

- Wanted: an implementation in Java
- We express $P \underset{J}{\cdot} x$ in terms of !

$$\begin{aligned} P \underset{J}{\cdot} x &= (F \underset{M}{\cdot} P) \underset{M}{\cdot} x \\ &= (F \; ! \; P) \; ! \; x \end{aligned}$$

Example 1 (extra)

- We have a compiler F (in machine language) from Java to machine language

$$F : J \xrightarrow{M} M$$

- Wanted: an implementation in Java
- We express $P \underset{J}{\cdot} x$ in terms of !

$$\begin{aligned} P \underset{J}{\cdot} x &= (F \underset{M}{\cdot} P) \underset{M}{\cdot} x \\ &= (F \,!\, P) \,!\, x \end{aligned}$$

- From this we know what we need to do:

Example 1 (extra)

- We have a compiler F (in machine language) from Java to machine language

$$F : J \xrightarrow{M} M$$

- Wanted: an implementation in Java
- We express $P \underset{J}{\cdot} x$ in terms of !

$$\begin{aligned} P \underset{J}{\cdot} x &= (F \underset{M}{\cdot} P) \underset{M}{\cdot} x \\ &= (F \mathbin{!} P) \mathbin{!} x \end{aligned}$$

- From this we know what we need to do:
  1. Load F as program, P as input
  2. Run
  3. Load the output from 2 as program, x as input
  4. Run

Example 2 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \xrightarrow{M} M$

Example 2 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \stackrel{M}{\to} M$
  - a compiler G from B to A in machine language: $G : B \stackrel{M}{\to} A$

Example 2 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \xrightarrow{M} M$
  - a compiler G from B to A in machine language: $G : B \xrightarrow{M} A$
- Wanted: an implementation B

$$P \underset{B}{\cdot} x \quad = G \underset{M}{\cdot} P \underset{A}{\cdot} x$$

Example 2 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \overset{M}{\to} M$
  - a compiler G from B to A in machine language: $G : B \overset{M}{\to} A$
- Wanted: an implementation B

$$
\begin{aligned}
P \underset{B}{\cdot} x &= G \underset{M}{\cdot} P \underset{A}{\cdot} x \\
&= F \underset{M}{\cdot} (G \underset{M}{\cdot} P) \underset{M}{\cdot} x
\end{aligned}
$$

Example 2 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \overset{M}{\to} M$
  - a compiler G from B to A in machine language: $G : B \overset{M}{\to} A$
- Wanted: an implementation B

$$\begin{aligned} P \underset{B}{\cdot} x &= G \underset{M}{\cdot} P \underset{A}{\cdot} x \\ &= F \underset{M}{\cdot} (G \underset{M}{\cdot} P) \underset{M}{\cdot} x \\ &= F \ ! \ (G \ ! \ P) \ ! \ x \end{aligned}$$

Example 3 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \xrightarrow{M} M$

Example 3 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \xrightarrow{M} M$
  - a compiler G from B to A in A: $G : B \xrightarrow{A} A$

Example 3 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \overset{M}{\to} M$
  - a compiler G from B to A in A: $G : B \overset{A}{\to} A$
- Wanted: an implementation B

$$P \underset{B}{\cdot} x \quad = G \underset{A}{\cdot} P \underset{A}{\cdot} x$$

Example 3 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \xrightarrow{M} M$
  - a compiler G from B to A in A: $G : B \xrightarrow{A} A$
- Wanted: an implementation B

$$\begin{aligned} P \underset{B}{\cdot} x &= G \underset{A}{\cdot} P \underset{A}{\cdot} x \\ &= F \,!\, G \,!\, P \underset{A}{\cdot} x \end{aligned}$$

Example 3 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \xrightarrow{M} M$
  - a compiler G from B to A in A: $G : B \xrightarrow{A} A$
- Wanted: an implementation B

$$
\begin{aligned}
P \underset{B}{\cdot} x &= G \underset{A}{\cdot} P \underset{A}{\cdot} x \\
&= F \,!\, G \,!\, P \underset{A}{\cdot} x \\
&= F \,!\, (F \,!\, G \,!\, P) \,!\, x
\end{aligned}
$$

Example 3 (extra)

- We have
  - a compiler F from A to machine language in machine language: $F : A \xrightarrow{M} M$
  - a compiler G from B to A in A: $G : B \xrightarrow{A} A$
- Wanted: an implementation B

$$\begin{aligned} P \underset{B}{\cdot} x &= G \underset{A}{\cdot} P \underset{A}{\cdot} x \\ &= F \,!\, G \,!\, P \underset{A}{\cdot} x \\ &= F \,!\, (F \,!\, G \,!\, P) \,!\, x \end{aligned}$$

- Take a good look again: which run occurs when?

Efficiency (extra)

- We have

Efficiency (extra)

- We have
  - a compiler F from J to M in M ($F : J \stackrel{M}{\to} M$)

Efficiency (extra)

- We have
  - a compiler F from J to M in M ($F : J \xrightarrow{M} M$)
  - a compiler G from J to M in J ($G : J \xrightarrow{J} M$)

Efficiency (extra)

- We have
  - a compiler F from J to M in M ($F : J \xrightarrow{M} M$)
  - a compiler G from J to M in J ($G : J \xrightarrow{J} M$)
- Wanted: an implementation of J

Efficiency (extra)

- We have
  - a compiler F from J to M in M ($F : J \overset{M}{\to} M$)
  - a compiler G from J to M in J ($G : J \overset{J}{\to} M$)
- Wanted: an implementation of J
- Include efficiency information: M^+, M^-

Efficiency (extra)

- We have
  - a compiler F from J to M in M ($F : J \xrightarrow{M} M$)
  - a compiler G from J to M in J ($G : J \xrightarrow{J} M$)
- Wanted: an implementation of J
- Include efficiency information: M^+, M^-
  - a compiler F from J to M^- in M ($F : J \xrightarrow{M^-} M^-$)

Efficiency (extra)

- We have
  - a compiler F from J to M in M ($F : J \xrightarrow{M} M$)
  - a compiler G from J to M in J ($G : J \xrightarrow{J} M$)
- Wanted: an implementation of J
- Include efficiency information: M^+, M^-
  - a compiler F from J to M^- in M ($F : J \xrightarrow{M^-} M^-$)
  - a compiler G from J to M^+ in J ($G : J \xrightarrow{J} M^+$)

Solutions (extra)

Remember our compilers

$$F : J \xrightarrow{M^-} M^- \quad G : J \xrightarrow{J} M^+$$

Solutions (extra)

Remember our compilers

$$F : J \overset{M^-}{\rightarrow} M^- \quad G : J \overset{J}{\rightarrow} M^+$$

Solution 1

- Use only F

$$P \underset{J}{\cdot} x = (F \; !^- \; P) \; !^- \; x$$

Solutions (extra)

Remember our compilers

$$F : J \overset{M^-}{\rightarrow} M^- \quad G : J \overset{J}{\rightarrow} M^+$$

Solution 1

- Use only F

$$P \underset{J}{\cdot} x = (F \,!^-\, P) \,!^-\, x$$

- Two inefficient runs for each P and x

Solutions (extra)

Remember our compilers

$$F : J \overset{M^-}{\to} M^- \quad G : J \overset{J}{\to} M^+$$

Solution 1

- Use only F

$$P \underset{J}{\cdot} x = (F \; !^- \; P) \; !^- \; x$$

- Two inefficient runs for each P and x

Solutions (extra)

Remember our compilers

$$F : J \xrightarrow{M^-} M^- \quad G : J \xrightarrow{J} M^+$$

Solution 1

- Use only F

$$P \underset{J}{\cdot} x = (F \;!^-\; P) \;!^-\; x$$

- Two inefficient runs for each P and x

Solution 2

- Note that

$$F \;!^-\; G : J \xrightarrow{M^-} M^+$$

Solutions (extra)

Remember our compilers

$$F : J \overset{M^-}{\to} M^- \quad G : J \overset{J}{\to} M^+$$

Solution 1

- Use only F

$$P \underset{J}{\cdot} x = (F \,!^-\, P) \,!^-\, x$$

- Two inefficient runs for each P and x

Solution 2

- Note that

$$F \,!^-\, G : J \overset{M^-}{\to} M^+$$

- Thus

$$P \underset{J}{\cdot} x = (F \,!^-\, G) \,!^-\, P \,!^+\, x$$

Solutions (extra)

Remember our compilers

$$F : J \xrightarrow{M^-} M^- \quad G : J \xrightarrow{J} M^+$$

Solution 1

- Use only F

$$P \underset{J}{\cdot} x = (F \,!^-\, P) \,!^-\, x$$

- Two inefficient runs for each P and x

Solution 2

- Note that

$$F \,!^-\, G : J \xrightarrow{M^-} M^+$$

- Thus

$$P \underset{J}{\cdot} x = (F \,!^-\, G) \,!^-\, P \,!^+\, x$$

- One inefficient run for each P and x

Solutions (extra)

Remember our compilers

$$F : J \xrightarrow{M^-} M^- \quad G : J \xrightarrow{J} M^+$$

Solutions (extra)

Remember our compilers

$$F : J \xrightarrow{M^-} M^- \qquad G : J \xrightarrow{J} M^+$$

- Idea

$$G \underset{J}{\cdot} G : J \xrightarrow{M^+} M^+$$

Solutions (extra)

Remember our compilers

$$F : J \overset{M^-}{\rightarrow} M^- \quad G : J \overset{J}{\rightarrow} M^+$$

- Idea

$$G \underset{J}{\cdot} G : J \overset{M^+}{\rightarrow} M^+$$

- Thus

$$P \underset{J}{\cdot} x \quad = (G \underset{J}{\cdot} G) \; !^+ \; P \; !^+ \; x$$

Solutions (extra)

Remember our compilers

$$F : J \overset{M^-}{\to} M^- \quad G : J \overset{J}{\to} M^+$$

- Idea

$$G \underset{J}{\cdot} G : J \overset{M^+}{\to} M^+$$

- Thus

$$\begin{aligned} P \underset{J}{\cdot} x &= (G \underset{J}{\cdot} G) \,!^+\, P \,!^+\, x \\ &= ((F \,!^-\, G) \,!^-\, G) \,!^+\, P \,!^+\, x \end{aligned}$$

Solutions (extra)

Remember our compilers

$$F : J \xrightarrow{M^-} M^- \quad G : J \xrightarrow{J} M^+$$

- Idea

$$G \underset{J}{\cdot} G : J \xrightarrow{M^+} M^+$$

- Thus

$$\begin{aligned} P \underset{J}{\cdot} x &= (G \underset{J}{\cdot} G) \,!^+\, P \,!^+\, x \\ &= ((F \,!^-\, G) \,!^-\, G) \,!^+\, P \,!^+\, x \end{aligned}$$

Recipe

1. run G with F: inefficient G
2. run G with inefficient G: efficient G
3. run P with G
4. run P with x

Solutions (extra)

Remember our compilers

$$F : J \overset{M^-}{\rightarrow} M^- \quad G : J \overset{J}{\rightarrow} M^+$$

- Idea

$$G \underset{J}{\cdot} G : J \overset{M^+}{\rightarrow} M^+$$

- Thus

$$\begin{aligned} P \underset{J}{\cdot} x &= (G \underset{J}{\cdot} G) \,!^+\, P \,!^+\, x \\ &= ((F \,!^-\, G) \,!^-\, G) \,!^+\, P \,!^+\, x \end{aligned}$$

Recipe

1. run G with F: inefficient G
2. run G with inefficient G: efficient G
3. run P with G
4. run P with x

- Inefficient just runs once
- F is needed once:

Solutions (extra)

Remember our compilers

$$F : J \stackrel{M^-}{\to} M^- \quad G : J \stackrel{J}{\to} M^+$$

- Idea

$$G \underset{J}{\cdot} G : J \stackrel{M^+}{\to} M^+$$

- Thus

$$\begin{aligned} P \underset{J}{\cdot} x &= (G \underset{J}{\cdot} G)\ !^+\ P\ !^+\ x \\ &= ((F\ !^-\ G)\ !^-\ G)\ !^+\ P\ !^+\ x \end{aligned}$$

Recipe

1. run G with F: inefficient G
2. run G with inefficient G: efficient G
3. run P with G
4. run P with x

- Inefficient just runs once
- F is needed once: bootstrapping

Bootstrapping in real life

GCC

- hex0
- hex1
- hex2
- cc_x86
- M2-planet
- mes (mescc)
- tinycc
- ...
- tinycc
- gcc (musl)
- gcc
- gcc
- ...
- gcc

`https://bootstrappable.org/`

Take home

- Log in on gitlab.
- Enroll for a group.
- Check the schedule *Contents*→*Overview*.
- In case of questions/ideas/wishes
  don't hesitate to contact us.
- N.B. The tutorial session is a Q&A session, send an email to register (`mart@cs.ru.nl`).
- What to do with the practical session?