Programming Languages - ST0244

Introduction

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2019-2

Administrative Information

Course Coordinator

Juan Francisco Cardona Mc'Cormick

Course web page

`http://www1.eafit.edu.co/asr/courses/programming-languages-st0244`

Exams, programming labs, course's repository, etc.

See course web page.

Administrative Information

Conventions

- The numbers assigned to examples, exercises, figures, pages and theorems correspond to the numbers in the textbook [Lee 2017].
- The source code examples are in course's repository.

Textbook's First Paragraph

'A career in computer science is a commitment to a lifetime of learning. You will not be taught every detail you will need in your career while you are a student. The goal of a computer science education is to give you the tools you need so you can teach yourself new languages, frameworks, and architectures as they come along.' (p. v)

Course Outline

Course contents

- Programming Languages Introduction
- Syntax
- Assembly Language
- Object-Oriented Programming
- Functional Programming
- Compiling Haskell
- Logic Programming

Course Outline

From textbook's Preface

'This text covers these three paradigms [object-oriented/imperative programming, functional programming, and logic programming] while using each of them in the implementation of a non-trivial programming language.' (p. v)

Programming Paradigms

Definition

A **programming language** is a formal language for writing computer programs.

Programming Paradigms

Definition

A **programming language** is a formal language for writing computer programs.

Question

What means the 'formal' adjective in the above definition?

Programming Paradigms

Definition

Paradigm: 'A model of something, or a very clear and typical example of something.' (Cambridge Dictionary)

Definition

Programming paradigms are:

'Ways of thinking about programming.' (p. v)

'High-level approaches for viewing computation.' [Turbark and Gifford 2008, p. 16]

'A way to classify programming languages based on their features.' (Wikipedia, 2019-07-13)

Programming Paradigms

Motivation

A cognitive bias: ‘if all you have is a hammer, everything looks like a nail’

Programming Paradigms

Motivation

A cognitive bias: ‘if all you have is a hammer, everything looks like a nail’

Three programming paradigms

- Imperative/object-oriented programming

  E.g. C, C++, COBOL, Fortran, Java, Pascal and Python.

- Functional programming

  E.g. Haskell, Scheme and Standard ML.

- Logic programming

  E.g. CLP(R) and Prolog.

Historical Perspective

Remark

The development of programming languages is based in theoretical and engineering and developments.

Historical Perspective

Time line*

- c. 1675 Gottfried Wilhelm Leibniz. *Characteristica universalis* (a universal symbolic language). Mechanical calculators.
- 1822 Charles Babbage. Difference engine (mechanical machine for tabulating polynomial functions).
- 1928 David Hilbert and Wilhelm Ackermann. The *Entscheidungsproblem* (decision problem) [Hilbert and Ackermann (1928) 1950].
- 1935-6 Alonzo Church. Lambda-calculus (computability model) and negative solution to the *Entscheidungsproblem* [Church 1935; Church 1936].
- 1936 Alan Turing. Turing machine (computability model) and negative solution to the *Entscheidungsproblem* [Turing 1936].

Continued on next slide

*A time line must start in some point and it is necessarily incomplete.

Historical Perspective

Time line (continuation)

1939 John Atanasoff and Clifford Berry. The ABC or Atanasoff-Berry Computer. United States.

c. 1940 Alonzo Church, Alan Turing and Stephen Kleene. The Church-Turing thesis.

1943 Tommy Flowers. The Colossus computer. England.

1945 John von Neumann. Storing the computer programs (there is controversy about the author(s) of this idea).

1946 John Mauchly and J. Presper Eckert. The ENIAC (Electronic Numerical Integrator and Computer). United States.

1949 Alan Turing. Design for stored programs and verification of programs [Turing 1949].

Continued on next slide

Historical Perspective

Time line (continuation)

1957 John Backus and others. FORTRAN [Backus, Beeber, Best, Goldberg, Haibt, Herrick, Nelson, Sayre, Sheridan, Stern, Ziller, Hughes and Nutt 1957].

1958 John McCarthy. Lisp [McCarthy 1960].

1960 John Backus and others. ALGOL 60 [Backus, Bauer, Green, Katz, McCarthy, Perlis, Rutishauser, Samelson, Vauquois, Wegstein, Wijngaarden and Woodger 1960].

c. 1960 John Backus and Peter Naur. BNF (Backus-Naur Format)

1965 J. A. Robinson. The resolution principle [Robinson 1965].

1972 Alain Colmerauer and Philippe Roussel. Prolog.

Models of Computation

Components of von Neumann's architecture

- A processing unit and processor registers.
- A control unit that contains an instruction register and a program counter (PC) which kept track of the next instruction to execute.
- Memory that stores both data and instructions.
- External mass storage and input and output mechanisms.

Models of Computation

Components of von Neumann's architecture

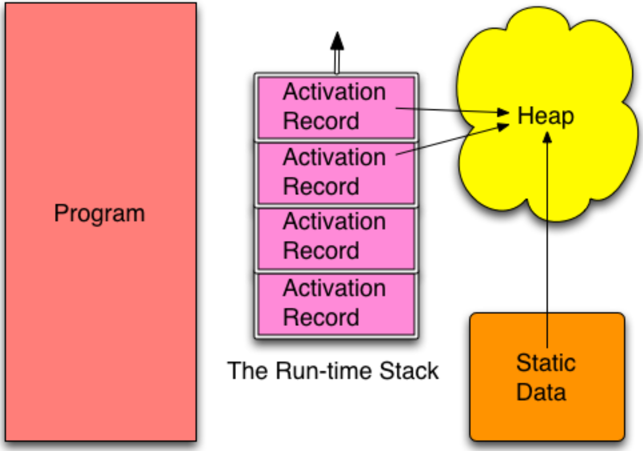
- A processing unit and processor registers.
- A control unit that contains an instruction register and a program counter (PC) which kept track of the next instruction to execute.
- Memory that stores both data and instructions.
- External mass storage and input and output mechanisms.

Remark

von Neumann's architecture was not enough for handling more structured and complex programs.

Models of Computation: The Imperative Model

Description (Fig. 1.4)



Models of Computation: The Imperative Model

Features

- Decomposition of a program in sub-programs (functions, procedures, sub-routines).
- Structural programming (top-down or bottom-up design).
- Activation records for functions/procedures.
- Division of the data area.

Models of Computation: The Imperative Model

Activation records for each function/procedure invocation

- Local variables.
- The return address (program counter's value before the function/procedure was called).
- Value of parameters.

Models of Computation: The Imperative Model

Division of the data area

- Static or global area

  Area for storing data and functions that are accessible globally in the program (e.g. constants, global variables, and built-in functions)

- The run-time stack

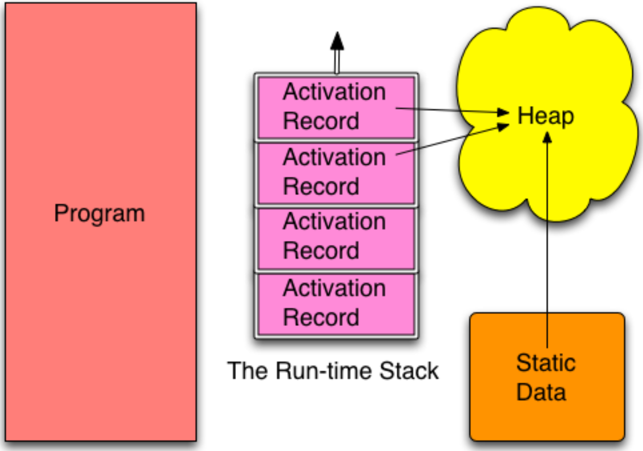
  Area for storing activation records using a LIFO order.

- The heap

  Area for dynamic memory allocation (data created at run-time) via references and pointers without pattern to the allocation and deallocation.

Models of Computation: The Functional Model

Description (Fig. 1.4)



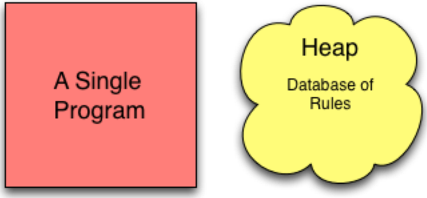
Models of Computation: The Functional Model

Features

- Persistent data (data is not updated)
- Functions are first-class citizens.
- No difference between program and data.
- Since all the work is made via calling functions the run-time is more important than in the imperative model.
- The programmer does not interact with the heap.
- The functional programming is more abstract (good) but the programmer has minor control (bad).

Models of Computation: The Logic Model

Description (Fig. 1.5)



Features

- The programmer does not write a program but a database with facts and rules (both are axioms from the logical point of view).
- It is debatable whether we should talk of a division of the data are in the logical model of computation.

Brief History of Some Programming Languages

Reading

To read the brief history of C, C++, Java, Prolog, Python and Standard ML in the textbook.

Language Implementation

Definition

Machine language is the (binary) language that is read, interpreted and executed by the CPU.

Language Implementation

Definition

Machine language is the (binary) language that is read, interpreted and executed by the CPU.

Definition

A **platform** is a specific combination of hardware and operating system.

Remark

Machine languages are platform-dependent.

Language Implementation

Definition

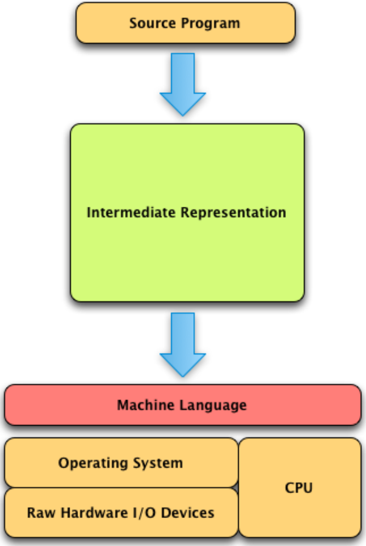
An **assembly language** is a symbolic representation (human readable) of the machine language.

Remark

Assembly languages are hardware-dependent.

Language Implementation

From source program to machine language (Fig. 1.11)



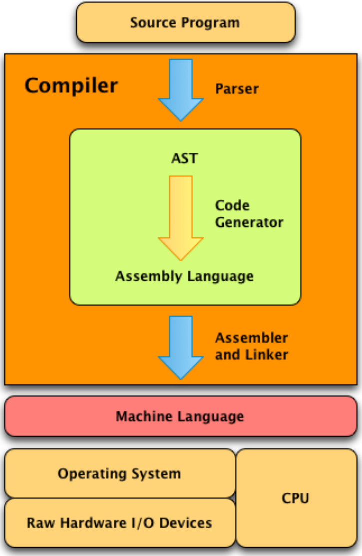
Language Implementation

Languages can be implemented in different ways

- A language can be compiled to a machine language.
- A language can be interpreted.
- A language can be implemented by combining compilation and interpretation.

Language Implementation: Compilation

Implementation via compilers (Fig. 1.12)



Language Implementation: Compilation

Definition

A **compiler** is a program that converts a source program to machine language.

Features

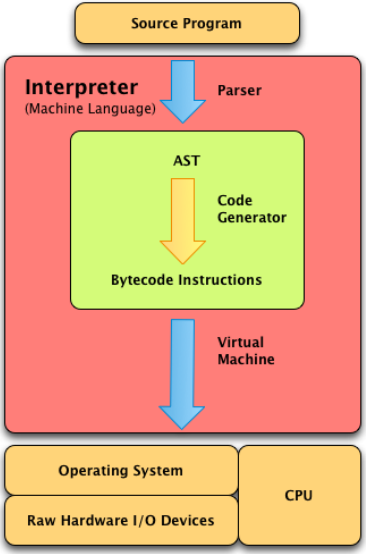
- Abstract syntax tree (AST): Internal representation of the source program.
- If you change your source code you need to recompile.

Example

C, C++, COBOL, Fortran, Haskell and Pascal are compiled languages.

Language Implementation: Interpretation

Implementation via interpreters (Fig. 1.13)



Language Implementation: Interpretation

Definition

An **interpreter** is a program that executes other programs.

Remark

Interpreters are platform-dependent.

Language Implementation: Interpretation

Definition

'A **virtual machine** is a program that provides insulation from the actual hardware and operating system of a machine while supplying a consistent implementation of a set of low-level instructions, often called **bytecode**.' (p. 23)

Remark

Virtual machines are platform-dependent.

Remark

Bytecode instructions are platform-independent.

Language Implementation: Interpretation

Features

- You execute your source programming by running the interpreter.
- Research problem: Heap memory management.
- Advantage: Portability (the interpreter insulates your program from CPU architecture and operating system dependencies).
- Disadvantage: Speed of execution.

Language Implementation: Interpretation

Features

- You execute your source programming by running the interpreter.
- Research problem: Heap memory management.
- Advantage: Portability (the interpreter insulates your program from CPU architecture and operating system dependencies).
- Disadvantage: Speed of execution.

Example

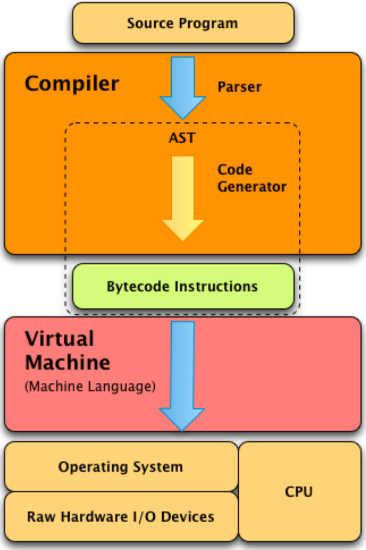
Bash, Haskell, Lisp, Prolog, Python, Ruby and Standard ML are interpreted languages.

Remark

Haskell programs can be both compiled or interpreted.

Language Implementation: Virtual Machines

Implementation via virtual machines (Fig. 1.14)



Language Implementation: Virtual Machines

Features

- Separation of the virtual machine from the compiler.
- The programs are compiled to bytecode.
- The bytecode programs are interpreted.
- The interpretation of bytecode programs is faster than the interpretation of source code.
- The programs implemented via virtual machines are more portable than programs implemented via compilers.
- Programs can be distributed in binary (bytecode) form.

Language Implementation: Virtual Machines

Example

C# Java, Python, Standard ML and Visual Basic.Net are implemented via virtual machines.

Remark

Python and Standard ML programs can be both implemented via interpreters or virtual machines.

Types and Type Checking

Types in logic and mathematics

Types as ranges of significance of propositional functions. Let $\varphi(x)$ be a (unary) propositional function. The type of $\varphi(x)$ is the range within which x must lie if $\varphi(x)$ is to be a proposition [Russell (1903) 1938, Appendix B: The Doctrine of Types].

In modern terminology, Rusell's types are domains of propositional functions.

Example

Let $\varphi(x)$ be the propositional function 'x is a prime number'. Then $\varphi(x)$ is a proposition only when its argument is a natural number.

$$\varphi : \mathbb{N} \to \{\text{False}, \text{True}\}$$
$$\varphi(x) = x \text{ is a prime number.}$$

Types and Type Checking

Types in programming languages

- 'They [programming languages] define types to specify which operations make sense on which types of data' (p. 26).
- 'A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.' [Kiselyov and Shan 2008, p. 8]

Example

Examples of types include integers, booleans, floating point numbers, characters, strings, lists, Cartesian products (tuples), discriminated unions, sets, functions, recursive/inductive types and user-defined types.

Types and Type Checking

Type checking

- Dynamically typed programming languages

  Type checking occurs in run-time. E.g. Python.

- Statically typed programming languages

  Type checking occurs in compile-time. E.g. C, C++, Haskell, Java and Standard ML.

Types and Type Checking

Type checking

- Dynamically typed programming languages

  Type checking occurs in run-time. E.g. Python.

- Statically typed programming languages

  Type checking occurs in compile-time. E.g. C, C++, Haskell, Java and Standard ML.

Discussion

What do you prefer, dynamically or statically typed languages? Why?

Types and Type Checking

Definition

Let P be a program.

A type system is **sound** iff

$$P \text{ passed the type checker} \Rightarrow P \text{ is a correctly typed program.}$$

A type system is **complete** iff

$$P \text{ is a correctly typed program} \Rightarrow P \text{ will pass the type checker.}$$

Example

The Standard ML type system is sound and complete.

References

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., Wijngaarden, A. van and Woodger, M. (1960). Report on the Algorithmic Language ALGOL 60. Communications of the ACM 3.5. Ed. by Naur, Peter, pp. 299–314. DOI: 10.1145/367236.367262 (cit. on p. 15).

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A. and Nutt, R. (1957). The FORTRAN Automatic Coding System. In: Proceedings Western Joint Computer Conference, pp. 188–198 (cit. on p. 15).

Church, Alonzo (1935). An Unsolvable Problem of Elementary Number Theory. Preliminar Report (Abstract). Bulletin of the American Mathematical Society 41.5, pp. 332–333. DOI: 10.1090/S0002-9904-1935-06102-6 (cit. on p. 13).

— (1936). An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics 58.2, pp. 345–363. DOI: 10.2307/2371045 (cit. on p. 13).

References

Hilbert, D. and Ackermann, W. [1928] (1950). Principles of Mathemalical Logic. Translation of *Grundzüge der Theoretischen Logik*, Springer, 1928. Translated by Lewis M. Hammond, George G. Leckie and F. Steinhardt. Edited and with notes by Robert E. Luce. Chelsea Publising Company (cit. on p. 13).

Kiselyov, Oleg and Shan, Chung-chieh (2008). Interpreting Types as Abstract Values. Formosan Summer School on Logic, Language and Computacion (FLOLAC 2008) (cit. on p. 42).

Lee, Kent D. (2017). Foundations of Programming Languages. 2nd ed. Springer (cit. on p. 3).

McCarthy, John (1960). Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. Communications of the ACM 3.4, pp. 184–195. DOI: 10.1145/367177.367199 (cit. on p. 15).

Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12.1, pp. 23–41. DOI: 10.1145/321250.321253 (cit. on p. 15).

References

Russell, Bertrand [1903] (1938). The Principles of Mathematics. 2nd ed. W. W. Norton & Company, Inc (cit. on p. 41).

Turbark, Franklyn and Gifford, David (2008). Design Concepts in Programming Languages. MIT Press (cit. on p. 9).

Turing, Alan M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. Proc. London Math. Soc. 42, pp. 230–265. DOI: 10.1112/plms/s2-42.1.230 (cit. on p. 13).

— (1949). Checking a Large Routine. In: Report of a Conference on High Speed Automatic Calculating (cit. on p. 14).

Programming Languages - ST0244

Syntax

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2019-2

Introduction

Syntax and Semantics

- Syntax is how programs look (well-formed programs)
- Semantics is how programs work (meaning of programs)

Question

When you are learning/using a programming language are its syntax and its semantics equally of important?

Terminology

Syntax and semantics issues

- Syntax issues (static, compile-time)
- Static semantics issues (compile-time)
- Dynamic semantics issues (run-time)

Terminology

Example (p. 32)

Is the code

```
a = b + c;
```

a correct C++ statement?

Some questions about the code:

1. Do `b` and `c` have values? (answered in run-time, dynamic semantic issue or answered in compile-time, static semantic issue)
2. Have `b` and `c` been declared as a type that allows the `+` operation? (answered in compile-time, static semantic issue)
3. Is `a` assignment compatible with the result of the expression `b + c`? (answered in compile-time, static semantic issue)
4. Does the assignment statement have the proper form? (answered in compile-time, syntactic issue)

Terminology

Definition

A **terminal** symbol (or **token**) is an elementary symbol of the language.

Example

Keywords, types, operators, numbers, identifiers, among others are terminal symbols in a programming language.

Terminology

Definition

A **non-terminal** symbol (or **syntactic category** or **syntactic variable**) represents a sequence of terminal symbols.

Example

- C++, Java, Python and other

  ⟨statement⟩, ⟨expression⟩, ⟨if-statement⟩, among others.

- Haskell, Standard ML and other

  ⟨type⟩, ⟨expression⟩, ⟨function application⟩, ⟨function abstraction⟩, among others.

Backus-Naur Form (BNF)

Definition

Backus Naur-Form (BNF) is a formal (i.e. non-ambiguous) meta-language (i.e. a language for describing or analysing other language) for describing language syntax.

Backus-Naur Form (BNF)

Example

A BNF describing the λ-calculus.

$$\langle \text{variable} \rangle ::= \text{x} \mid \text{x'} \mid \text{x''} \mid \dots$$

$$\begin{aligned} \langle \lambda\text{-term} \rangle ::= {} & \langle \text{variable} \rangle \\ & \mid \lambda \ \langle \text{variable} \rangle \ . \ \langle \lambda\text{-term} \rangle \\ & \mid (\ \langle \lambda\text{-term} \rangle \ \langle \lambda\text{-term} \rangle \ ) \end{aligned}$$

Remark

Note the recursive definition of λ-terms.

Backus-Naur Form (BNF)

Example

A BNF describing a part of Java (pp. 33–34).

$$\langle\text{primitive-type}\rangle ::= \text{boolean} \mid \text{char} \mid \text{byte} \mid \text{short} \mid \text{int} \mid \text{long} \mid \text{float} \mid \ldots$$

$$\begin{aligned}\langle\text{argument-list}\rangle ::= {} & \langle\text{expression}\rangle \\ \mid {} & \langle\text{argument-list}\rangle \text{ , } \langle\text{expression}\rangle\end{aligned}$$

$$\begin{aligned}\langle\text{selection-statement}\rangle ::= {} & \text{if (} \langle\text{expression}\rangle \text{) } \langle\text{statement}\rangle \\ \mid {} & \text{if (} \langle\text{expression}\rangle \text{) } \langle\text{statement}\rangle \text{ else } \langle\text{statement}\rangle \\ \mid {} & \text{switch (} \langle\text{expression}\rangle \text{) } \langle\text{block}\rangle\end{aligned}$$

Continued on next slide

Backus-Naur Form (BNF)

Example (continuation)

$\langle$m[ethod]-declaration$\rangle$ ::=
$\langle$modifiers$\rangle$ $\langle$type-specifier$\rangle$ $\langle$m-declarator$\rangle$ $\langle$throws-clause$\rangle$ $\langle$m-body$\rangle$
| $\langle$modifiers$\rangle$ $\langle$type-specifier$\rangle$ $\langle$m-declarator$\rangle$ $\langle$m-body$\rangle$
| $\langle$type-specifier$\rangle$ $\langle$m-declarator$\rangle$ $\langle$throws-clause$\rangle$ $\langle$m-body$\rangle$
| $\langle$type-specifier$\rangle$ $\langle$m-declarator$\rangle$ $\langle$m-body$\rangle$ $\langle$m-body$\rangle$

Backus-Naur Form (BNF)

Extended BNF (EBNF)

We shall extended BNF with the following definitions:

(a) ‘item?’ or ‘[item]’ means the item is optional.

(b) ‘item*’ or ‘{item}’ means zero or more occurrences of the item are allowable.

(c) ‘item+’ means one or more occurrences of the item are allowable.

(d) Parentheses may be used for grouping.

Context-Free Grammars

Definition

A **context-free grammar** is a 4-tuple

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S}),$$

where

$$\begin{aligned}
\mathcal{N}&: \text{ A finite set of non-terminal symbols} \\
\mathcal{T}&: \text{ A finite set of terminal symbols} \\
\mathcal{P}&: \text{ A finite set of productions of the form } A \to \alpha, \\
&\quad \text{ where } A \in \mathcal{N} \text{ and } \alpha \in \{\mathcal{N} \cup \mathcal{T}\}^* \\
\mathcal{S} \in \mathcal{N}&: \text{ The start symbol}
\end{aligned}$$

The grammar G is context-free because every non-terminal symbol in a sentence is independent of the context in which it is used.

Context-Free Grammars

Example (Infix expressions grammar (§.2.3.1))

We can define a context-free grammar for infix expressions by

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E),$$

where

$$\begin{aligned}
\mathcal{N} &= \{E, T, F\}, \\
\mathcal{T} &= \{\text{identifier}, \text{number}, +, -, *, /, (,)\},
\end{aligned}$$

and the productions $\mathcal{P}$ are

$$\begin{aligned}
E &\to E + T \mid E - T \mid T \\
T &\to T * F \mid T/F \mid F \\
F &\to (\ E\ ) \mid \text{identifier} \mid \text{number}
\end{aligned}$$

Derivations

Definition

A **sentence** of a grammar G is a string of tokens (terminal symbols) from G.

Derivations

Definition

A **sentence** of a grammar G is a string of tokens (terminal symbols) from G.

Example

Sentences for the infix expressions grammar (previous example).

$$(5 * x) + y \quad \text{and} \quad)4 + +(.$$

Derivations

Definition

A **sentential form** of a grammar G is a string of terminals and non-terminals symbols from G.

Definition

A **derivation** of a sentence S in a grammar G is a sequence of sentential forms of G that starts with the start symbol of G and ends with S.

Every sentential form in the derivation is obtained from the previous one by replacing $A \in \mathcal{N}$ (non-terminal symbol) by $\alpha \in \{\mathcal{N} \cup \mathcal{T}\}^*$ (string of terminals and non-terminals symbols), if $A \to \alpha$ is a production of G.

Definition

A sentence S of a grammar G is **valid** iff there exists at least one derivation for S in G.

Derivations

Example

A derivation of the sentence $(5 * x) + y$ in the infix expressions grammar.

$$
\begin{aligned}
\text{(start symbol) } \underline{E} &\Rightarrow \underline{E} + T && \text{(sentential form)} \\
&\Rightarrow \underline{T} + T && \text{(sentential form)} \\
&\Rightarrow \underline{F} + T && \text{(sentential form)} \\
&\Rightarrow (\underline{E}) + T && \text{(sentential form)} \\
&\Rightarrow (\underline{T}) + T && \text{(sentential form)} \\
&\Rightarrow (\underline{T} * F) + T && \text{(sentential form)} \\
&\Rightarrow (\underline{F} * F) + T && \text{(sentential form)} \\
&\Rightarrow (5 * \underline{F}) + T && \text{(sentential form)} \\
&\Rightarrow (5 * x) + \underline{T} && \text{(sentential form)} \\
&\Rightarrow (5 * x) + \underline{F} && \text{(sentential form)} \\
&\Rightarrow (5 * x) + y && \text{(valid sentence)}
\end{aligned}
$$

Derivations

Definition

Let G be a grammar. The **language** of G, denoted $L(G)$, is the set of valid sentences of G.

Derivations

Types of derivations

- Left-most derivation (always replace the left-most non-terminal symbol).
- Right-most derivation (always replace the right-most non-terminal symbol).

Derivations

Example

Left-most and right-most derivations of $(5 * x) + y$ in the infix expressions grammar.

(left-most)
$$
\begin{aligned}
\underline{E} &\Rightarrow \underline{E} + T \\
&\Rightarrow \underline{T} + T \\
&\Rightarrow \underline{F} + T \\
&\Rightarrow (\underline{E}) + T \\
&\Rightarrow (\underline{T}) + T \\
&\Rightarrow (\underline{T} * F) + T \\
&\Rightarrow (\underline{F} * F) + T \\
&\Rightarrow (5 * \underline{F}) + T \\
&\Rightarrow (5 * x) + \underline{T} \\
&\Rightarrow (5 * x) + \underline{F} \\
&\Rightarrow (5 * x) + y
\end{aligned}
$$

(right-most)
$$
\begin{aligned}
\underline{E} &\Rightarrow E + \underline{T} \\
&\Rightarrow E + \underline{F} \\
&\Rightarrow \underline{E} + y \\
&\Rightarrow \underline{T} + y \\
&\Rightarrow \underline{F} + y \\
&\Rightarrow (\underline{E}) + y \\
&\Rightarrow (\underline{T}) + y \\
&\Rightarrow (T * \underline{F}) + y \\
&\Rightarrow (\underline{T} * x) + y \\
&\Rightarrow (\underline{F} * x) + y \\
&\Rightarrow (5 * x) + y
\end{aligned}
$$

Derivations

Prefix expressions

In prefix expressions the operator appears before the operands.

Example

$$
\begin{aligned}
4 + (a - b) * x \quad & \text{(infix expression)} \\
+4 * -abx \quad & \text{(prefix expression)}
\end{aligned}
$$

Derivations

Example (Prefix expressions grammar (§ 2.4.3))

We can define a context-free grammar for prefix expressions by

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E),$$

where

$$\begin{aligned} \mathcal{N} &= \{E\}, \\ \mathcal{T} &= \{\text{identifier}, \text{number}, +, -, *, /\}, \end{aligned}$$

and the productions $\mathcal{P}$ are

$$\begin{aligned} E \to\ & +EE \mid -EE \mid *EE \mid /EE \\ & \mid \text{identifier} \mid \text{number} \end{aligned}$$

Parser Trees

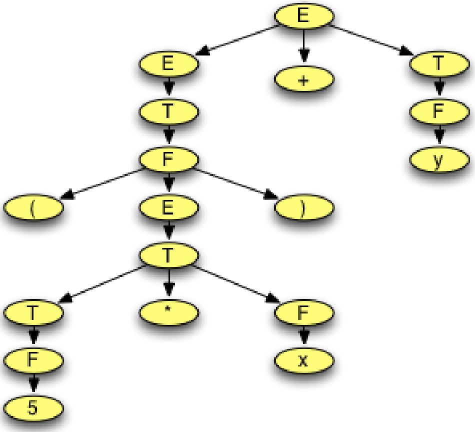
Definition

Let G be a grammar. A **parser tree** is a tree representing of a sentence of $L(G)$ (i.e. a valid sentence of G).

Parser Trees

Example

Parser tree for the sentence $(5 * x) + y$ in the infix expressions grammar (Fig. 2.1).



Parser Trees

Remark

Recall that a sentence of a grammar can have various derivations.

Definition

A grammar G is **ambiguous** iff there is (at least) a sentence in $L(G)$ that has more than one parse tree.

Abstract Syntax Trees (AST)

Definition

An **abstract syntax tree** is a parser tree without non-essential information required for evaluating (generate code in compilation or execute in interpretation) the sentence (p. 38):

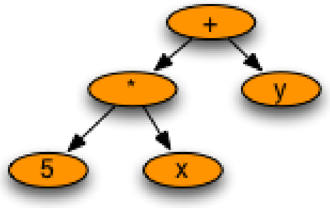
(a) 'Non-terminal nodes in the tree are replaced by nodes that reflect the part of the sentence they represent.'

(b) 'Unit productions in the tree are collapsed.'

Abstract Syntax Trees (AST)

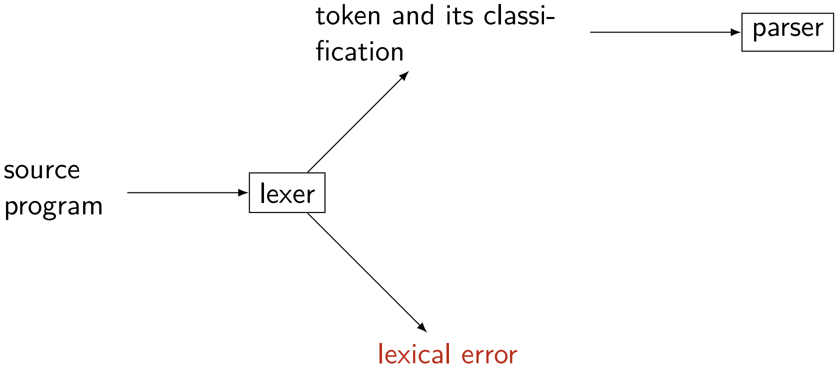
Example

AST for the sentence $(5 * x) + y$ in the infix expressions grammar (Fig. 2.2).



Lexical Analysis

Introduction



Lexical Analysis

Remark

Tokens (terminal symbols) of a language can be described by regular expressions (other language).

Lexical Analysis

The language of regular expressions

The context-free grammar for the language of regular expressions is defined by

$$\mathrm{RE} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E),$$

$$\begin{aligned} \mathcal{N} &= \{E, T, K, F\}, \\ \mathcal{T} &= \{\text{character}, *, +, ., (,)\}, \end{aligned}$$

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T.K \mid K \\ K &\to F^* \mid F \\ F &\to (\ E\ ) \mid \text{character} \end{aligned}$$

Continued on next slide

Lexical Analysis

The language of regular expressions (continuation)

Operators and their precedence (from highest to lowest):

$()$	
$*$	(Kleene star or Kleene closure (zero or more occurrences))
$\cdot$	(concatenation operator)
$+$	(choice operator)

The operators . and $+$ are left-associative.

Lexical Analysis

Example

Recall that the terminal symbols (tokens) of the infix expressions grammar are

$$\mathcal{T} = \{\text{identifier}, \text{number}, +, -, *, /, (,)\}.$$

These terminal symbols are defined by the regular expression

$$letter.letter^* \ + \ digit.digit^* \ + \ `+\text{'} \ + \ `-\text{'} \ + \ `*\text{'} \ + \ `/\text{'} \ + \ `(\text{'} \ + \ `)\text{'}$$

where

$$\begin{array}{rll} letter & \text{abbreviates} & \mathrm{A} + \mathrm{B} + \cdots + \mathrm{Z} + \mathrm{a} + \mathrm{b} + \cdots + \mathrm{z} \quad \text{and} \\ digit & \text{abbreviates} & 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 \end{array}$$

Finite State Machines

Introduction

Relation between regular expressions and finite state machines (mathematical model).

Finite State Machine

Definition

A **finite state machine** is a 5-tuple

$$M = (\Sigma, S, F, s_0, \delta),$$

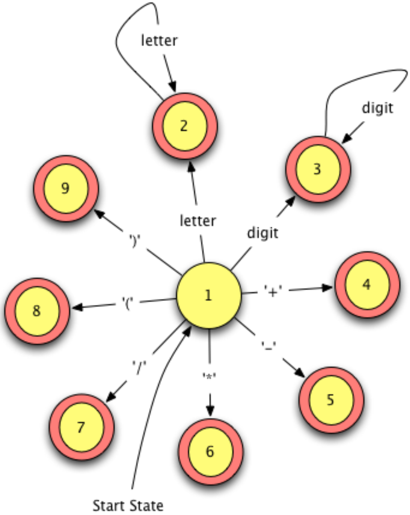
where

$$\begin{aligned}
\Sigma&\text{: Input alphabet}\\
S&\text{: Set of states}\\
F \subseteq S&\text{: A set of final (or accepting) states}\\
q_0 \in S&\text{: The start state}\\
\delta : \Sigma \times S \to S&\text{: A transition function}
\end{aligned}$$

Finite State Machine

Example

Finite state machine for the language of infix expression tokens (Fig. 2.3).



Lexer Generators

Definition

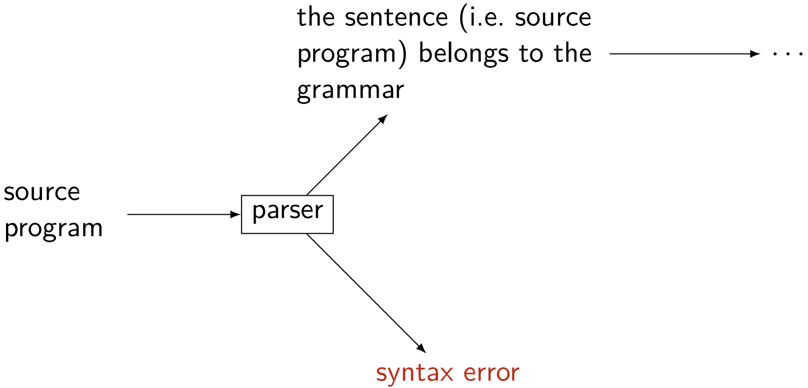
A **lexer generator** is a tool for building a lexer.

lexer source $\longrightarrow$ $\boxed{\text{lexer generator}}$ $\longrightarrow$ lexer program

Example

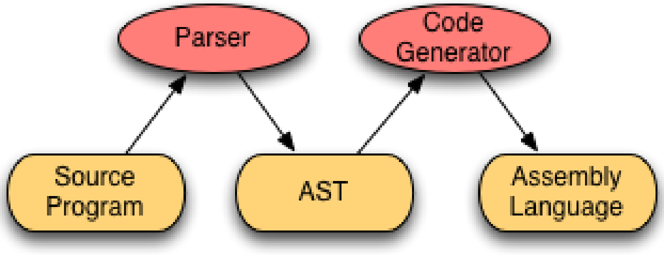
Lex (for C), Flex (in Linux) and Alex (for Haskell).

Introduction



Parsing

Introduction (Fig. 2.4)



Parsing

Types of parsing

- Top-down parser (starts with the root)
- Bottom-up parser (starts with the leaves)

Top-Down Parsers

Features

- A top-down parser performs a left-most derivation of the sentence (the source program).
- Top-downs parsers are also called recursive descent parsers because they can be implemented by a set of mutually recursive functions.

Top-Down Parsers

Definition

'An LL(1) grammar is simply a grammar where the next choice in a left-most derivation can be deterministically chosen based on the current sentential form and the next token in the input.' (p. 43)

Top-Down Parsers

Definition

'An LL(1) grammar is simply a grammar where the next choice in a left-most derivation can be deterministically chosen based on the current sentential form and the next token in the input.' (p. 43)

Remark

LL(1) parser (L: left to right, L: left-most derivation, 1: only one symbol of look-ahead)

Top-Down Parsers

Example (§ 2.9.1)

The prefix expressions grammar is an LL(1) grammar.

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E),$$

$$\mathcal{N} = \{E\},$$
$$\mathcal{T} = \{\text{identifier}, \text{number}, +, -, *, /\},$$

$$E \to +EE \mid -EE \mid *EE \mid /EE \mid \text{identifier} \mid \text{number}$$

Top-Down Parsers

Example (§ 2.9.2)

The infix expressions grammar is not an LL(1) grammar.

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E),$$

$$\begin{aligned}
\mathcal{N} &= \{E, T, F\}, \\
\mathcal{T} &= \{\text{identifier}, \text{number}, +, -, *, /, (,)\},
\end{aligned}$$

$$\begin{aligned}
E &\to E + T \mid E - T \mid T \\
T &\to T * F \mid T/F \mid F \\
F &\to (\ E\ ) \mid \text{identifier} \mid \text{number}
\end{aligned}$$

Continued on next slide

Top-Down Parsers

Example (continuation)

A left-most derivation for $5 * y$.

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow 5 * F \Rightarrow 5 * 4.$$

Can we choose a production looking at 5? No.

Top-Down Parsers

Example (§ 2.9.3)

An LL(1) grammar for infix expressions where ϵ denotes the empty production.

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E),$$

$$\begin{aligned}
\mathcal{N} &= \{E, RestE, T, RestT, F\}, \\
\mathcal{T} &= \{\text{identifier}, \text{number}, +, -, *, /, (,)\},
\end{aligned}$$

$$\begin{aligned}
E &\to T\ RestE \\
RestE &\to +T\ RestE \mid -T\ RestE \mid \epsilon \\
T &\to F\ RestT \\
RestT &\to *F\ RestT \mid /F\ RestT \mid \epsilon \\
F &\to (\ E\ ) \mid \text{identifier} \mid \text{number}
\end{aligned}$$

Bottom-Up Parsers

Features

- 'A bottom-up parser constructs a right-most derivation of a source program in reverse (p. 45)'.
- LALR(1) parser (LA: look ahead, L: left to right R: right-most derivation, 1: only one symbol of look-ahead)
- Pushdown automaton: Finite state machine + stack

Bottom-Up Parsers

Example

Let $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ be the following grammar for infix expressions:

$$\mathcal{N} = \{E, T, F\},$$
$$\mathcal{T} = \{\text{identifier}, \text{number}, +, *, (,)\},$$

$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow \text{identifier}$$
$$F \rightarrow \text{number}$$
$$F \rightarrow (\ E\ )$$

Continued on next slide

Bottom-Up Parsers

Example (continuation)

Right-most derivation for the expression $5 * 4 + 3$.

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow E + F \\
&\Rightarrow E + 3 \\
&\Rightarrow T + 3 \\
&\Rightarrow T * F + 3 \\
&\Rightarrow T * 4 + 3 \\
&\Rightarrow F * 4 + 3 \\
&\Rightarrow 5 * 4 + 3
\end{aligned}
$$

Used productions

$$E \to E + T \qquad (1)$$

$$E \to T \qquad (2)$$

$$T \to T * F \qquad (3)$$

$$T \to F \qquad (4)$$

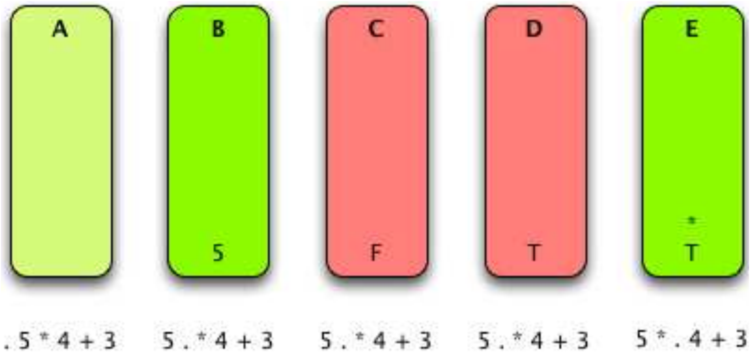
$$F \to \text{number} \qquad (5)$$

$$F \to (\ E\ ) \qquad (6)$$

Continued on next slide

Bottom-Up Parsers

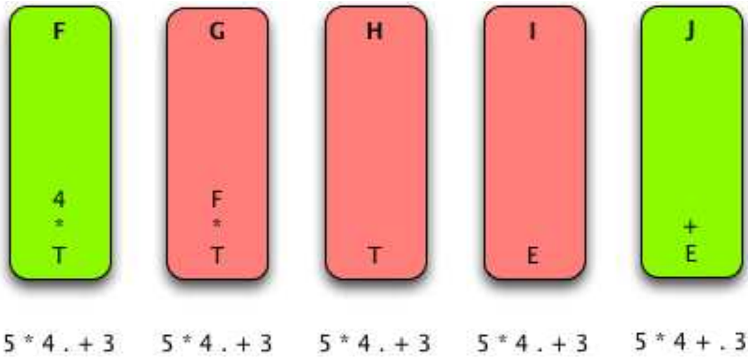
Example (continuation (Fig. 2.6))



Continued on next slide

Bottom-Up Parsers

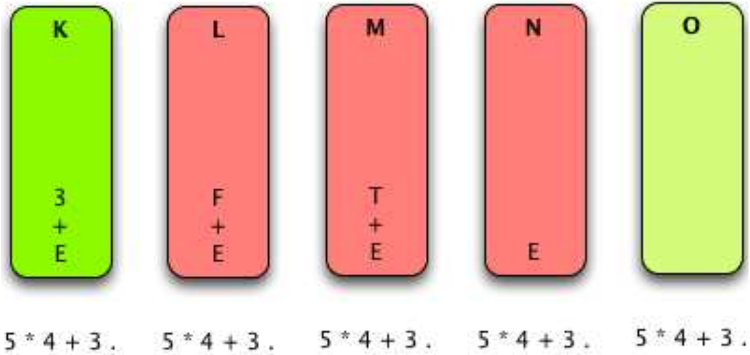
Example (continuation (Fig. 2.6))



Continued on next slide

Bottom-Up Parsers

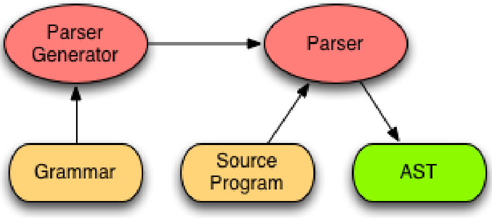
Example (continuation (Fig. 2.6))



Bottom-Up Parsers

Definition

A **parser generator** is a tool for building a parser (Fig. 2.5).



Example

Yacc (for Unix), Bison (in Linux for C, C++ and Java) and Happy (for Haskell).

Limitations of Syntactic Definitions

Some limitations

- The syntax of a programming language is an incomplete description of it (e.g. $5 + 4/0$).
- 'The set of programs in any interesting language is not context-free.' (p. 50) (e.g. $a + b$)
- A (context-free) grammar does not specify the semantics of a (programming) language.

Limitations of Syntactic Definitions

Example (Context-sensitive issues (p. 50–51))

- In an array declaration in C++, the array size must be a non-negative value.
- Operands for the && operation must be boolean in Java.
- In a method definition, the return value must be compatible with the return type in the method declaration.
- When a method is called, the actual parameters must match the formal parameter types.

Programming Languages - ST0244

Assembly Language

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2019-2

Introduction

- Python is an object-oriented language and it is an interpreted language via a virtual machine (see Fig. 1.14).
- We shall study an assembly language via an Python virtual machine.
- Python virtual machine is internal. We shall use a virtual machine for a subset of Python called JCoCo.

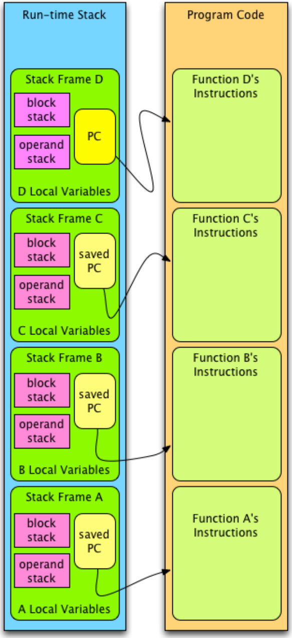
The JCoCo Virtual Machine

Some features

- JCoCo is implemented in Java.
- JCoCo reads an assembly language and internally build a sequence of bytecode instructions for each function.
- JCoCo includes a disassembler (from machine language/bytecode to assembly language) for Python.
- JCoCo is a stack based architecture (see Fig 3.1 in next slide).

The JCoCo Virtual Machine

Fig. 3.1



The JCoCo Virtual Machine

Getting Started

See whiteboard.

Example

See `asm/addtwo.py` and `asm/addtwo.casm` files.

Input/Output

Example

See `asm/io.[py,casm]` files.

Remarks

- Input/output is based on the built-in functions `input` and `print`.
- On calling functions:
  1. The function must be pushed on the operand stack.
  2. The actual arguments also must be pushed on the operand stack.
  3. The instruction `CALL_FUNCTION n` calls the function with its `n` arguments.
  4. The function leaves its return value on the operand stack.

If-Then-Else Statements

Example (if-then-else statement)

See `asm/ite.[py,casm,casm.txt]` files.

Remarks

- A **label** provides a symbolic target to jump to in the code.
- The labels disappear when JCoCo assembles the code.
- The instructions of each function are at zero-based offsets from the beginning of the function.

If-Then-Else Statements

Example (if-then statement)

See `asm/iif.[py,casm]` files.

Remarks

- Assembly languages can have instructions like `POP_JUMP_IF_FALSE` or `POP_JUMP_IF_TRUE`.
- The instruction `JUMP_FORWARD label00` is not necessary.

While Loops

Example

Recall the sequence of Fibonacci numbers

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

defined by

$$\begin{aligned} F_1 &= 1, \\ F_2 &= 1, \\ F_n &= F_{n-1} + F_{n-2}, \quad \text{for } n > 2. \end{aligned}$$

See `asm/while.py`, `asm/while.casm` and `asm/while.casm.txt` files.

While Loops

Remarks

- The instruction `SETUP_LOOP` is required for handling Python `break` instruction. This virtual machine instruction uses the block stack.
- The while loops and the if-then-else statements are implemented using the same instructions.

Exception Handling

Example

See `asm/exception.[py,casm,casm.txt]` files.

Remarks

- The instruction `SETUP_EXCEPT` handles the exceptions. This virtual machine instruction uses the block stack.
- JCoCo only has one type of exception, called `Exception`.

List Constants

Example

See `asm/list-constants.[py,casm]` files.

Remark

The instruction `BUILD_LIST` is used for building lists.

Methods Call

Example

See `asm/method-call.[py,casm,casm.txt]` files.

Remark

The instruction `LOAD_ATTR` gets the methods from the objects.

Iteration over Lists

Iteration

Iterating through a sequence (e.g. lists, tuples or strings) requires that JCoCo supports iterators.

Example

See `asm/list-iteration.[py,casm,casm.txt]` files.

Remark

The instructions for iteration over a list are `GET_ITER` and `FOR_ITER`.

Range Objects and Lazy Evaluation

Indexing

We can also iterating through a sequence (e.g. lists, tuples or strings) by indexing it. Usually indexes are zero based.

Evaluation strategies

An **evaluation strategy** defines when evaluate the arguments to a function, method or operation.

- In **eager evaluation** the arguments are evaluated at the time of call.
- In **lazy evaluation** the arguments are evaluated at the time they are actually used.

Range Objects and Lazy Evaluation

Remark

Eager evaluation is also called strict evaluation or call-by-valued, and lazy evaluation is also called non-strict evaluation or call-by-need. Unfortunately, this terminology varies between authors.

Range Objects and Lazy Evaluation

Example

In Python 2 the `range` function is *eagerly evaluated*. In Python 3 this function is *lazily evaluated*.

```
$ python --version
Python 2.7.16

>>> range(pow(10,11))
...
MemoryError

$ python --version
Python 3.7.4

>>> range(pow(10,11))
range(0, 100000000000)
```

Range Objects and Lazy Evaluation

Example

See `asm/lazy-evaluation.[py,casm,casm.txt]` files.

Remark

The instructions for ranging objects are `GET_ITER`, `FOR_ITER` and `BINARY_SUBSCR`.

Functions and Closures

Functions

Python allows the definition of functions inside a function (nested functions).

Example

See `asm/closure.py` file.

Functions and Closures

Functions

Python allows the definition of functions inside a function (nested functions).

Example

See `asm/closure.py` file.

Question

Why is the output of the program `asm/closure.py`? Why?

Functions and Closures

Static and Dynamic Scoping

The **scope** of a variable is the part of the program where the variable exists.

- Static (or lexical) scoping
  The variables refer to the environment (part of the source code) where were defined.
- Dynamic scoping
  The variables refer to the environment (execution context) where were called.

Functions and Closures

Static and Dynamic Scoping

The **scope** of a variable is the part of the program where the variable exists.

- Static (or lexical) scoping
  The variables refer to the environment (part of the source code) where were defined.

- Dynamic scoping
  The variables refer to the environment (execution context) where were called.

Example

Most of programming languages including C, C++, Haskell, Java, Prolog, Python and Standard ML are statically scoped.

Functions and Closures

Definition

'A **closure** is the environment in which a function is defined and the code for the function itself.' (p. 85).

Closures are used for implementing statically scoped languages.

Programming Languages - ST0244

Object-Oriented Programming

Andrés Sicard-Ramírez

Universidad EAFIT

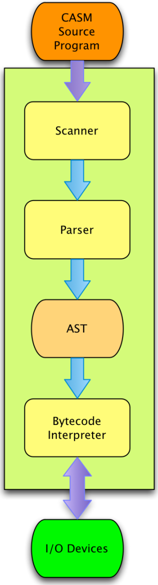
Semester 2019-2

Introduction

- Features object-oriented programming (Java and C++) via the implementation of the JCoCo virtual machine which was written in Java.
- Since Java and C++ are statically typed programming languages, type errors are caught in compile-time.
- 'Run-time errors are still possible, but those run-time errors are due to logic problems and not due to type errors.' (p. 111).
- JCoCo was written in Java. JCoCo consists of 56 source files and ~ 8.900 LOC (lines of code). Structuring large programs is of the higher importance.

The JCoCo Virtual Machine

- The scanner (lexer) is implemented via a finite state machine.
- The grammar is LL(1). The parser is implemented as a top-down (recursive descent) parser.
- 'Each non-terminal of the grammar is a function in the parser. The right hand sides of rules for each non-terminal defines the body of each function in the parser.' (p. 114).
- The AST consists of *function* and *class* definitions returned by the parser.
- Recall that the run-time stack consists of activation records (or frames) (Fig. 1.4). The bytecode interpreter evaluates the AST using frames and it interacts with the I/O devices. The figure to the right is Fig. 4.1.



The JCoCo Virtual Machine

Example

Two first lines of the `asm/addtwo.casm` file.

```
Function: main/0
Constants: None, 5, 6
```

Tokens from this file include a `Function` keyword, a colon, a `main` identifier, a slash, an integer `0`, a `Constants` keyword, another colon, a `None` keyword, a comma, an integer `5`, another comma, an integer `6`, and so on.

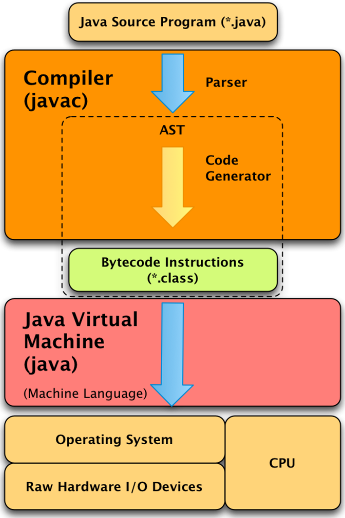
The Java Environment

Tools

- Java compiler (command `javac` on Linux): Source code to bytecode.
- Java Virtual Machine (JVM) (command `java` on Linux): Executes the bytecode.

The Java Environment

Java compiler and virtual machine (Fig. 4.4)



The Java Environment

Example

Compile and run the `oop/HelloWorld.java` program by running the followings commands:

```
$ javac HelloWorld.java
$ java HelloWorld
```

The C++ Environment

The C++ environment

See Fig. 4.5.

The C++ Environment

The C++ environment

See Fig. 4.5.

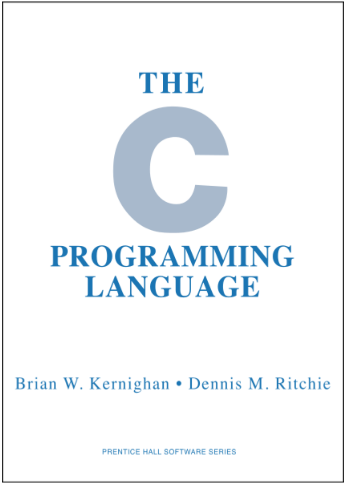
Example

Compile and run the `oop/hello-world.cpp` program by running the followings commands:

```
$ g++ hello-world.cpp
$ ./a.out
```

The C++ Environment

About the ‘hello world’ example



‘The first program to write is the same for all languages: Print the words `hello, world`.’ [1978, §1.1]

The C++ Environment

Definition

The C++ **macro processor** is a program that processes **directives**, which give instructions (e.g. for including files, for conditional compilation, for macro definition and expansion, among other) to the compiler to preprocess the source code before the compilation starts.

The C++ Environment

Definition

The C++ **macro processor** is a program that processes **directives**, which give instructions (e.g. for including files, for conditional compilation, for macro definition and expansion, among other) to the compiler to preprocess the source code before the compilation starts.

Example

From the line

```
#include <iostream>
```

in `oop/hello-world.cpp`, the macro processor includes the `iostream` library.

The C++ Environment

Example

The C++ macro processor is called cpp. Using cpp and gcc -E.

The C++ Environment

The make tool

'The make tool is a program that can be used to compile programs that are composed of modules and utilise separate compilation.' (p. 120)

The make tool

'The make tool is a program that can be used to compile programs that are composed of modules and utilise separate compilation.' (p. 120)

Example (make rule)

```
PyObject.o : PyObject.cpp PyObect.h
           g++ -g -c -std=c++0x PyObject.cpp
```

The C++ Environment

The make tool

'The make tool is a program that can be used to compile programs that are composed of modules and utilise separate compilation.' (p. 120)

Example (make rule)

```
PyObject.o : PyObject.cpp PyObect.h
           g++ -g -c -std=c++0x PyObject.cpp
```

Example (Making the CoCo executable)

```
coco : main.o PyObject.o PyInt.o PyType.o ...
     g++ -o coco -std=c++0x main.o \
     PyObject.o PyInt.o PyType.o
```

Namespaces

Description

In programming languages **namespaces** are context for identifiers. They help to uniquely identify the names of variables, functions, classes, etc.

Namespaces

Example (C++)

The line

```
using namespace std;
```

in `oop/hello-world.cpp` opens the `std` (standard) namespace. If we remove this line, we should replace the line

```
cout << "hello, world" << endl;
```

by the line

```
std::cout << "hello, world" << std::endl;
```

where `::` is a scope qualifier.

Namespaces

Example (Python)

- The line

  ```
  from disassembler import *
  ```

  merges the the namespace of the `disassembler` module with the current module.

  E.g., `disassemble(main)`.

- The line

  ```
  import disassembler
  ```

  preserves the namespace of the current module.

  E.g., `disassembler.disassemble(main)`.

Namespaces

Example (Java)

Namespaces in Java are handle by packages (named collection of classes).

- From the line

  ```
  import java.io.File;
  ```

  we can write `File` instead of `java.io.File`.

- From line

  ```
  import java.io.*;
  ```

  we don't need qualified names when using the classes in `java.io`.

Namespaces

Remark

'The safest way to program is to not open up namespaces or merge them together. But, that is also inconvenient since the whole name must be written each time. What is correct for your program depends on the program being written.' (p. 122)

Dynamic Linking

Static linking

Some issues when using static linking are (p. 120):

- 'The size of the linked executable program would be huge taking up a lot of space in memory as it was executing.'
- 'Any change in any library would require each program that uses it to be re-linked to get the new version of the library.'
- 'There is no reason to have multiple copies of libraries, one for each program that uses it. This wastes space in addition to the overhead of having to manage multiple copies of libraries.'

The Main Function

Reading

To read Section 4.5 Defining the Main Function.

I/O Streams

Reading

To read Section 4.6 I/O Streams.

Garbage Collection

Features

- It removes automatically objects (variables, data structures, functions, or methods) from the heap (i.e., dynamically created) when are no longer needed.
- Trade-off between programmer control and automatically memory management.
- It avoids memory leaks.
- It impacts the run-time performance of a system.
- Languages with garbage collection require a run-time system (i.e., virtual machine) for executing the programs.
- Java, Haskell and Python have garbage collection. C and C++ haven't.
- It runs in a thread.

Threading

TODO

The Java PyToken Class

Object-Oriented programming

From the textbook (p. 127):

> *Object-Oriented programming is all about creating objects. Objects have **state information**, sometimes just called **state**, and **methods** that operate on that state, sometimes altering the state. If we alter the state of an object we call it a **mutable** object. If we cannot alter the object's state once it is created, the object is called **immutable**. A **class** defines the state information maintained by an object and the methods that operate on that state.*

The Java PyToken Class

Using the JCoCo (student version) repository

Cloning and generating the JAR file:

```
$ git clone https://github.com/kentdlee/JCoCo.git \
  JCoCo-student
$ cd JCoCo-student
$ ant compile
$ ant jar
```

Running the JCoCo virtual machine:

```
$ cd dist
$ java -jar JCoCoStudent.jar file-name.casm
```

The Java PyToken Class

The PyToken class*

- The class defines the tokens of the JCoCo virtual machine.

*File `src/jcoco/PyToken.java` in the JCoCo repository.

The Java PyToken Class

The PyToken class*

- The class defines the tokens of the JCoCo virtual machine.
- The JCoCo code is packaged (`jcoco` package).

*File `src/jcoco/PyToken.java` in the JCoCo repository.

The Java PyToken Class

The PyToken class*

- The class defines the tokens of the JCoCo virtual machine.
- The JCoCo code is packaged (`jcoco` package).
- The `TokenType` enumeration defines the types of tokens.

*File `src/jcoco/PyToken.java` in the JCoCo repository.

The Java PyToken Class

The PyToken class*

- The class defines the tokens of the JCoCo virtual machine.
- The JCoCo code is packaged (`jcoco` package).
- The `TokenType` enumeration defines the types of tokens.
- The `TokenType` enumeration is formed by constant names (e.g. `PYEOFTOKEN`) which can be used in the source code.

Continued on next slide

*File `src/jcoco/PyToken.java` in the JCoCo repository.

The Java PyToken Class

The PyToken class (continuation)

- The object state is defined by the variables

```
private String lexeme;
private TokenType type;
private int line;
private int col;
```

  Only class' methods may access these variables directly because they were declared private.

Continued on next slide

The Java PyToken Class

The PyToken class (continuation)

- A `PyToken` object can be created using the `PyToken` constructor.

```
PyToken t;
t = new PyToken(type, lex, line, column);
```

The C++ PyToken Class

Using the CoCo repository

Cloning and generating the executable:

```
$ git clone https://github.com/asr/CoCo.git CoCo-asr
$ cd CoCo-asr
$ git checkout pl-st0244
$ ./rebuild  # If necessary
$ ./configure
$ make
```

Running the CoCo virtual machine:

```
$ ./coco file-name.casm
```

The C++ PyToken Class

The PyToken class

- Header file (`PyToken.h`) and methods implementation file (`PyToken.cpp`) from the CoCo repository.

The C++ PyToken Class

The PyToken class

- Header file (`PyToken.h`) and methods implementation file (`PyToken.cpp`) from the CoCo repository.
- In `PyToken.h` the destructor is declared in the line

  ```
  virtual ~PyToken();
  ```

  Since Java has garbage collector, a destructor is not required.

Continued on next slide

The C++ PyToken Class

Pointers and references

From the textbook (p. 130):

> ***Pointers*** *are the address of data in the memory of the computer. Pointers can be used in expressions to create new pointers using pointer arithmetic. In a programming language a pointer can point anywhere. A* ***reference*** *is much more controlled. References are somewhat like pointers except that they cannot be used in arithmetic expressions. They also don't directly point to locations in memory. When a reference is dereferenced using a dot, the run-time system does the lookup in a reference table.*
>
> *This difference between references and pointers means that we can safely rely on every reference pointing to a real object where we don't necessarily know if a pointer is pointing to space that might be safely freed or not since the pointer might be the result of some pointer arithmetic. References are safe for garbage collection. Pointers are not.*

The C++ PyToken Class

The PyToken class (continuation)

- In C++ `this` is a pointer (we use the arrow operator). In Java `this` is a reference (we use the dot notation).

Inheritance and Polymorphism

Description

From the textbook (p. 131):

> ***Inheritance** is the mechanism we employ to re-use code in software we are currently writing. **Polymorphism** is the mechanism we employ to customize the behavior of code we have already written.*

Inheritance and Polymorphism

Example (C++)

- All methods and data values in Python are objects.
- The common behaviour is implemented in the `PyObject` class (files `PyObject.h` and `PyObject.cpp` in the CoCo repository).
- C++ implements polymorphism via a virtual function table.
- The classes `PyInt` and `PyList` inherit from the `PyObject` class.
- The `toString()` method is implemented in the classes `PyObject`, `PyInt` and `PyList`.
- The `toString()` method is virtual in the above classes because only in run-time we know which `toString()` to call.