CS 320 :
Part II and Formal Grammars

Marco Gaboardi
CDS 1019
gaboardi@bu.edu

Where we are?

Part I

- ➢ So far you have learned Functional Programming.
  A different kind of programming abstraction.
- ➢ You have also learned a formal mechanism to reason about the correctness of your programs
  - Type System

Part 2

- ➢ How to describe a programming language
  - Formal grammars, operational semantics
- ➢ And how to implement it:
  - Interpreter, compiler and type checking.

Grading

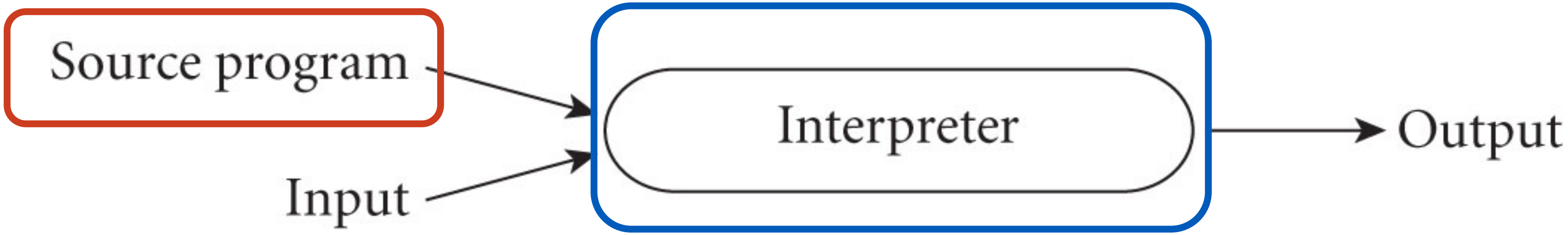
Part 2 is worth 40% of the grade split as:

- Grammar assignment - 4%
- Operational semantics assignment 4%
- In class quiz 1 - grammars - 4%
- In class quiz 2 - operational semantics - 4%
- Interpreter part 1 - 8%
- Interpreter part 2 - 8%
- Interpreter part 3 - 8%

What is the difference between an interpreter and a compiler?

Pure Interpretation

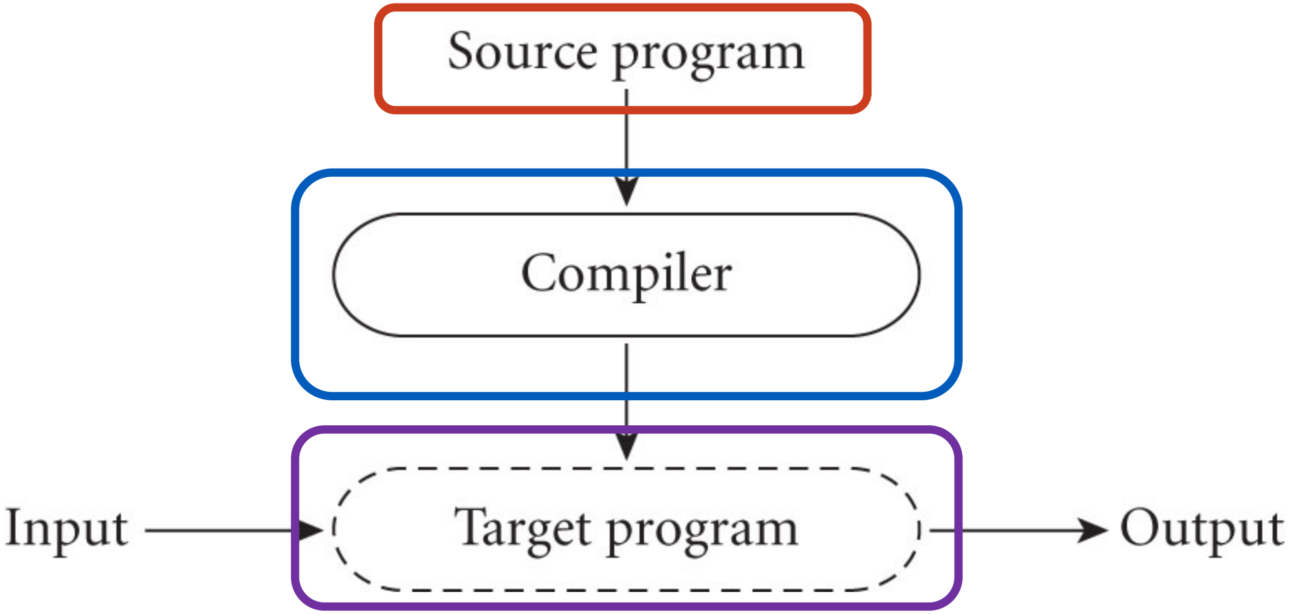
Interpretation



An interpreter is a program that accepts a source program and its input and runs it immediately to produce the output.

Pure Compilation

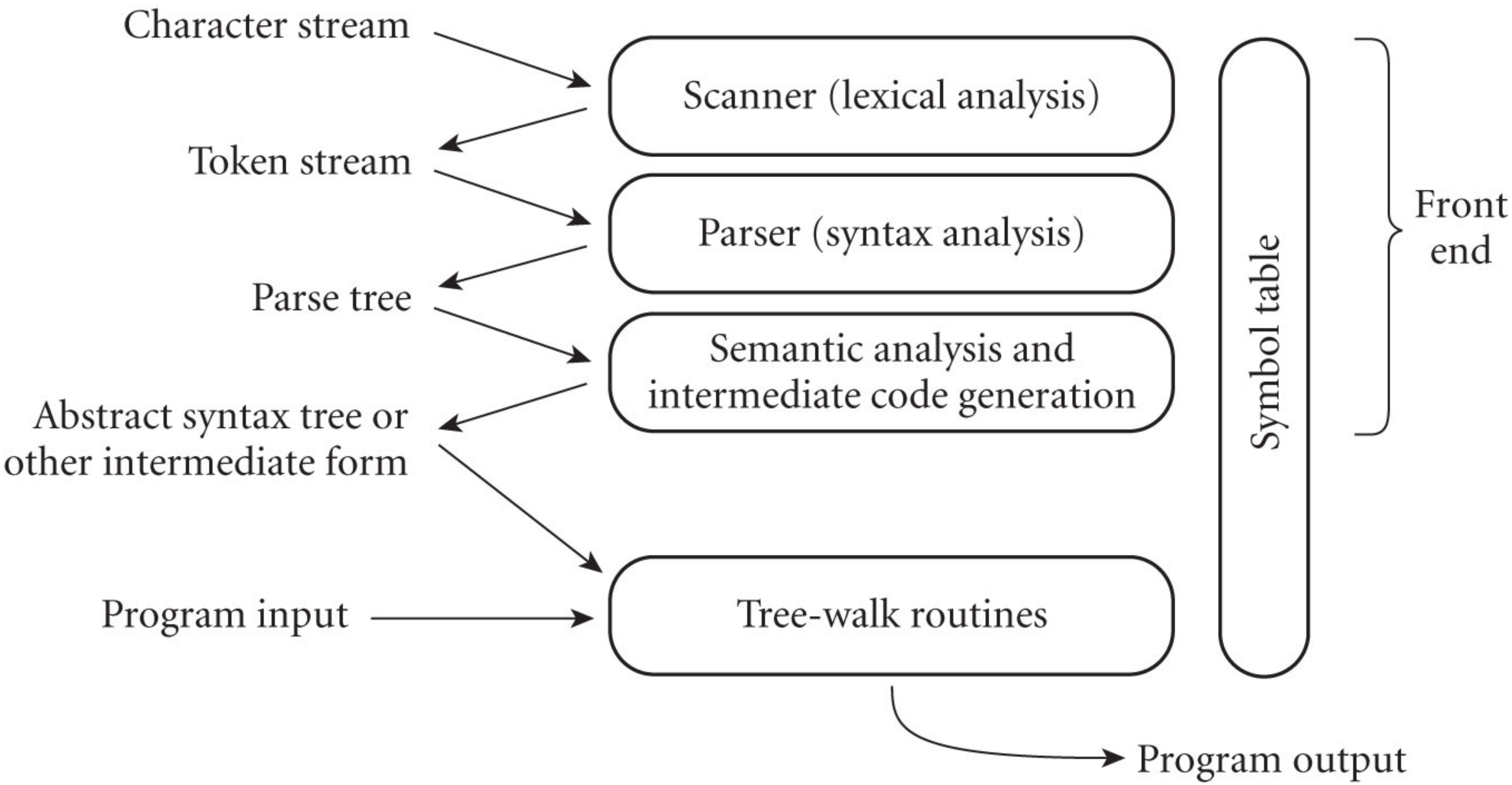
compilation



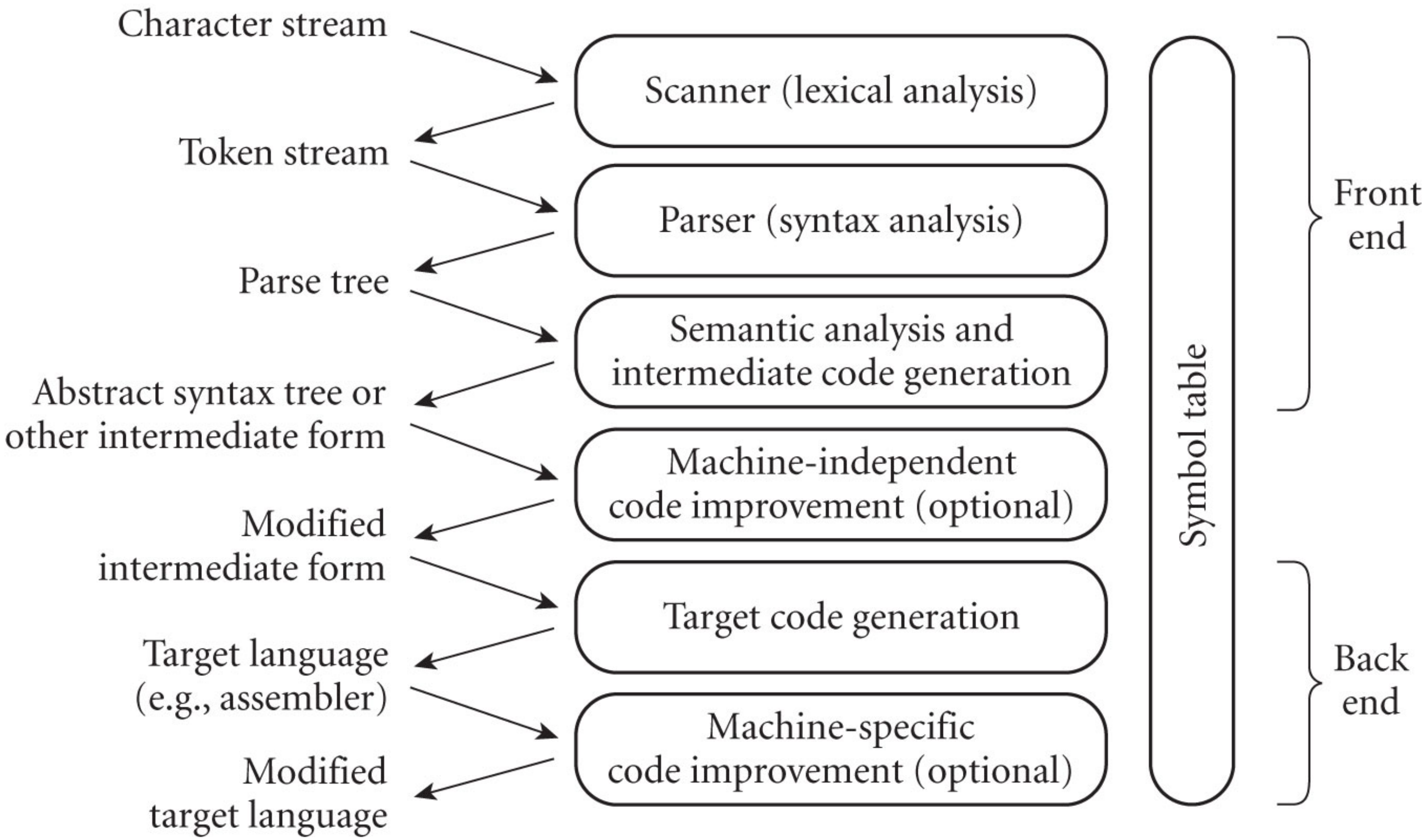
A compiler is a program that translates from a source program from an high-level language into a low-level language.

What are the phases of an interpreter or a compiler?

Pure Interpretation



Pure Compilation

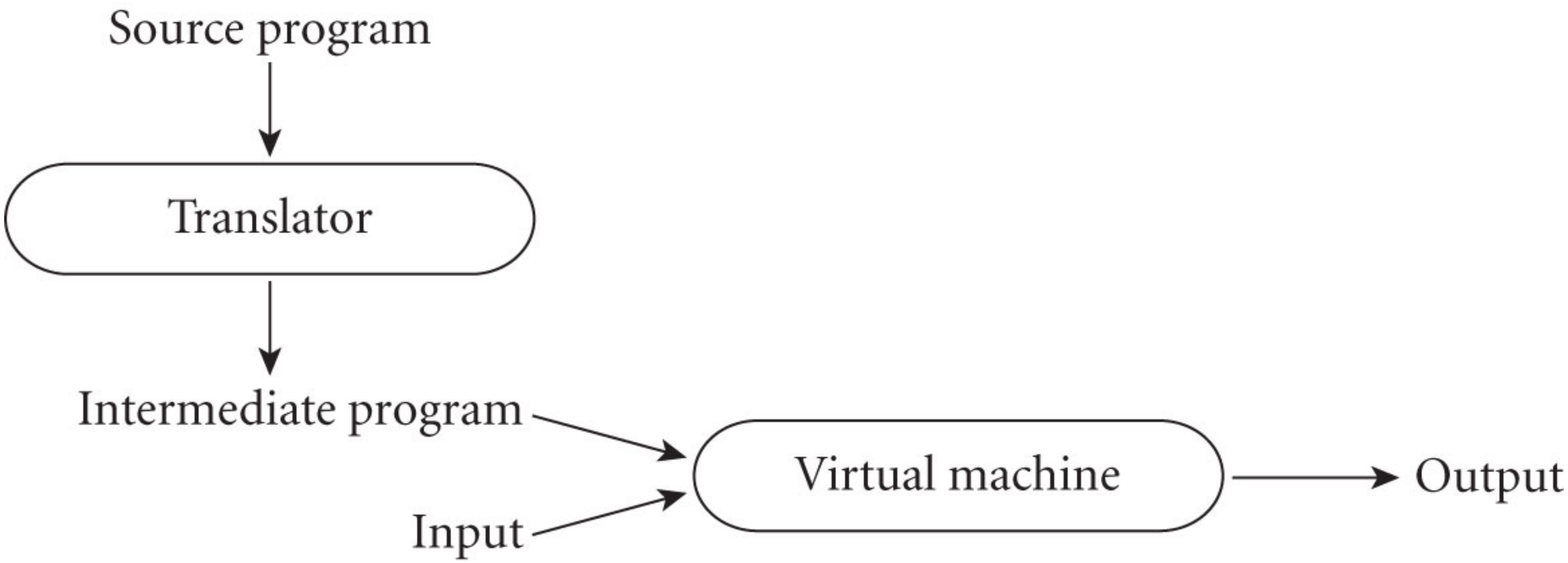


Programming Assignments

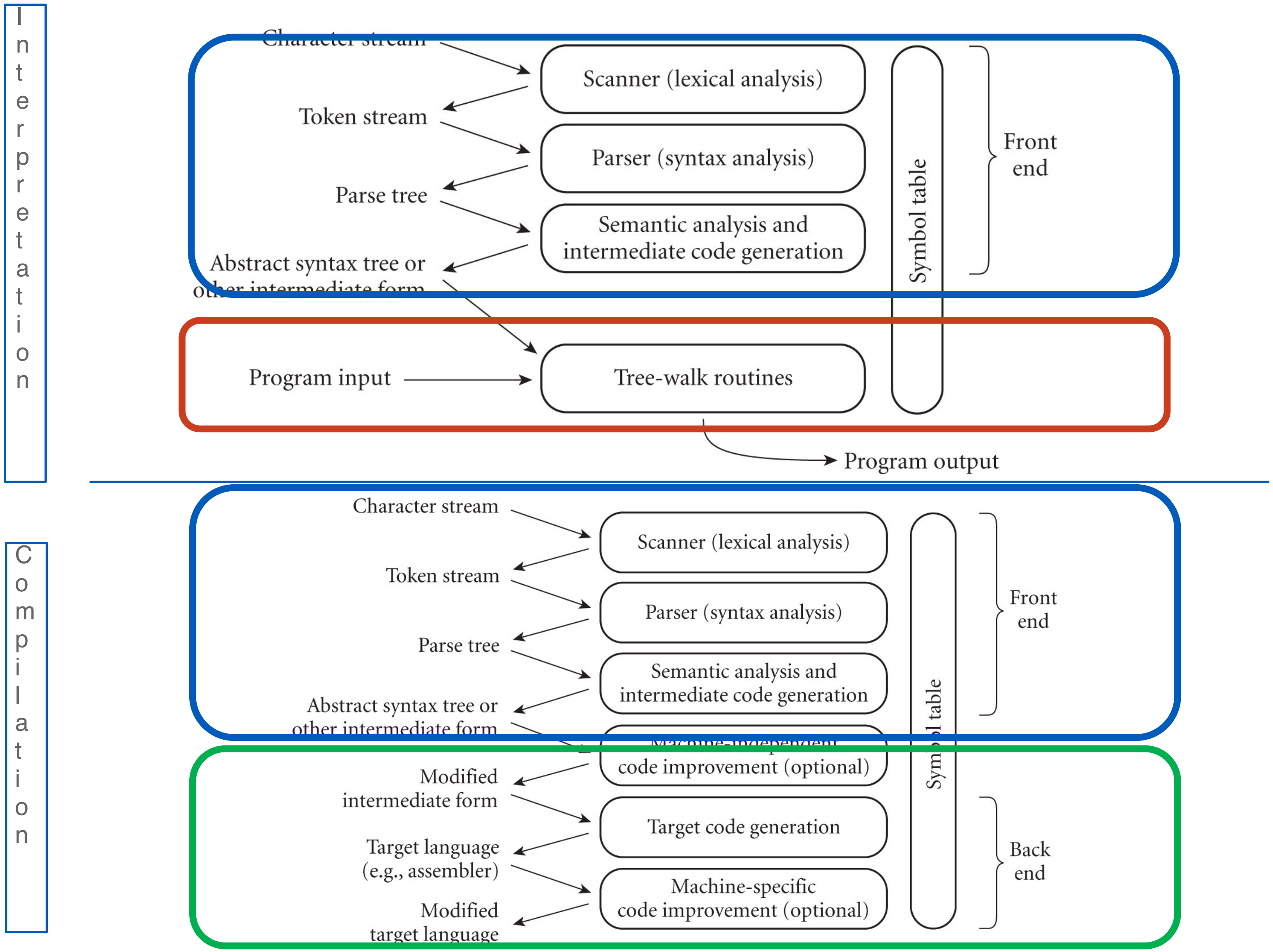
- In the first two programming assignments we will build an interpreter for a stack based language.
- In the third programming assignment we will compile program from a high level language into the stack based language.

Other approaches: e.g. Mixing Compilation and Interpretation

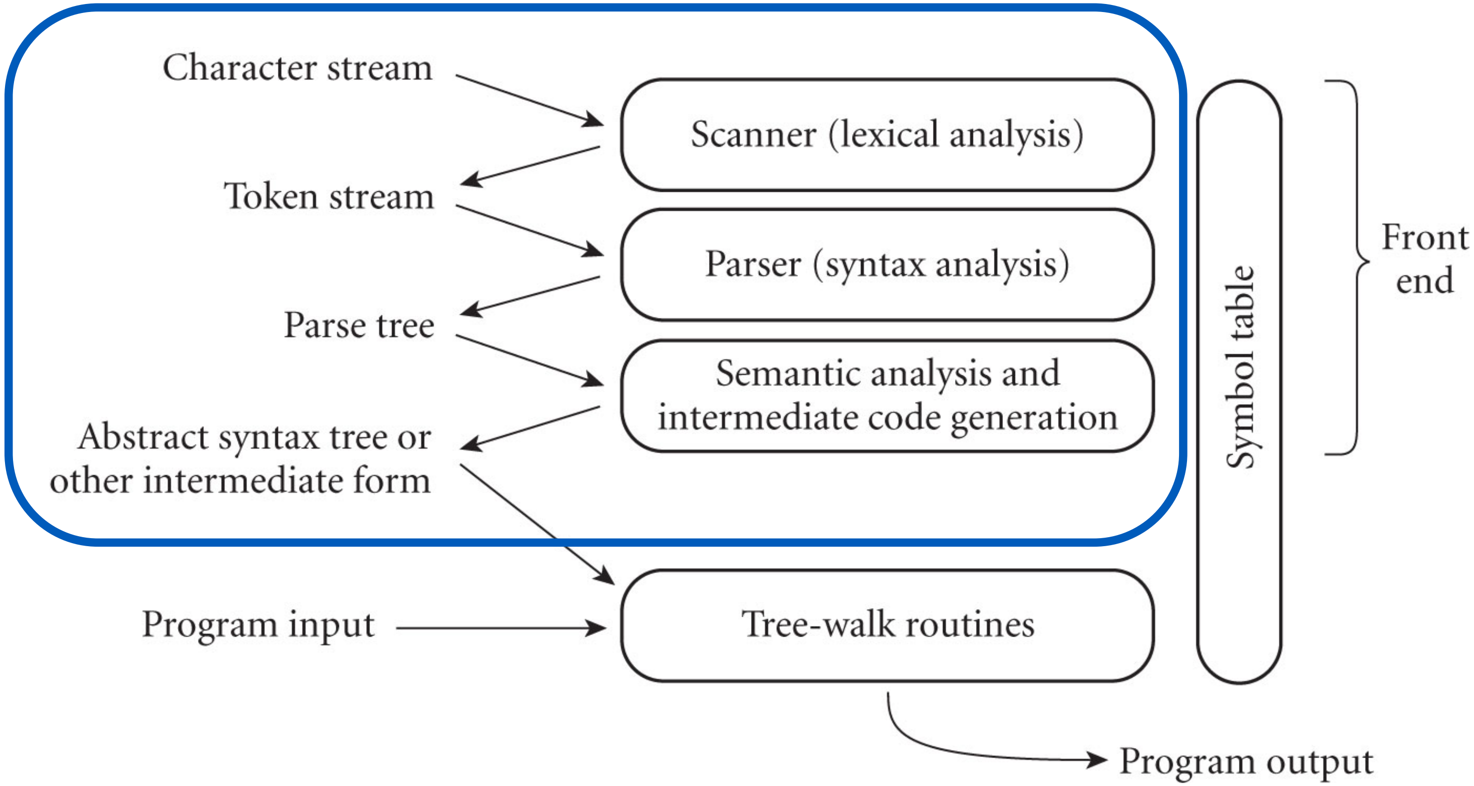
Mixing



Commonalities and differences



Parsing and semantic analysis



Learning Goals for today

- Understanding the basics of formal grammars.
- Understanding how the sentential structure of a program can be described as a data structure through the use of Formal Grammars.

Formal Grammars

How can we convert a stream of characters in something that the interpreter can execute?

功身レぞとク社9都ヨ災刑途むどフ裕読73画リタ掲式スソチ然祝ホ細川ホル省治4季きレ佳加底肉侍つくげ。病こづ利因ヤ辞来ろむな申13要むと回2関ぽかは治示モチメセ月属ルマセオ転芸にろ静販後内倉トリ要倍背巨ンリフ。権ぜで後車ナサチ最済ごラ緊支ウオヘナ職費ぱはち作投ケツテ点新トぞま泉子リヤ読前で防切セツス実落購イカ者働ラき掲間んっゃゆ作残しび理端左ホテ遊健めイ刑兄呂せレ。

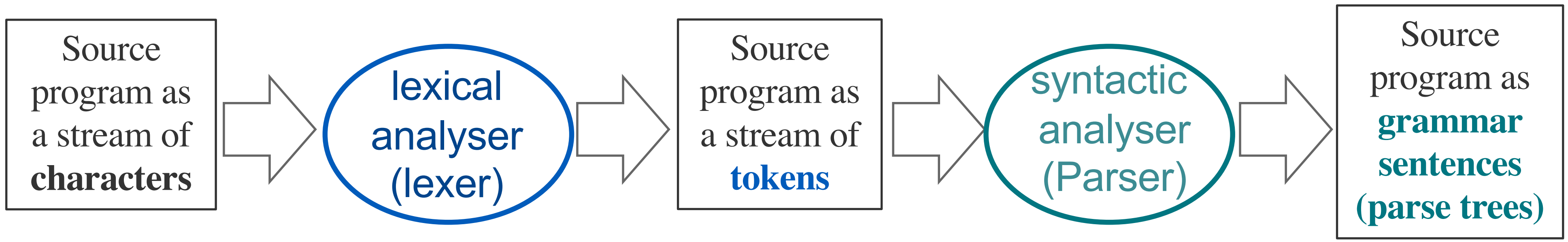
We need to:

- Identify the symbols of the language
- Understand how to compose them to form words, and sentences,
- Give a meaning to words and sentences.

Some definitions from Formal Languages

- A sentence is a string of characters over some alphabet.
- A language is a set of sentences
(this holds for both programming languages and human languages).
- A lexeme is the lowest level syntactic unit of a language (e.g., match, let, +, 1134, x,…).
- A token is a pair of a category of lexemes, and a lexeme: (e.g., (identifier,x) ; (constructor, Succ); (literal,1134) …).

Syntactic Structure of Programming Languages



- The scanning phase (lexical analyser) collects characters into tokens
- Parsing phase (syntactic analyser) determines (validity of) syntactic structure

Formal Grammars

- A formal grammar is a formal description of the sentential-forms that are part of the language
- Linguists have developed a hierarchy of grammar corresponding to the complexity of the sentential forms that are allowed in a specific language.

Formal Grammars

- We will focus on Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s.
  - Meant to describe the syntax of natural languages.
  - Define a class of languages called context-free languages.
- Grammars in Backus Normal/Naur Form (BNF) (1959)
  - Invented by John Backus to describe Algol 58 and refined by Peter Naur for Algol 60.
  - BNF is equivalent to context-free grammars
- A restricted class of grammars called regular grammars which are equivalents to regular expressions.

BNF

- Here is an example of a simple BNF for a subset of English. A sentence is noun phrase and verb phrase followed by a period.

```
<sentence>     ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>      ::= a | an | the
<noun>         ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>         ::= eats | throws | sees | is
```

BNF

- Here another example from a programming languages application.

```
<program> ::= <stmts>
<stmts> ::= <stmt> | <stmt> ; <stmts>
<stmt> ::= <var> = <expr>
<var> ::= a | b | c | d
<expr> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | const
```

BNF

- In BNF, abstractions <…> are used to represent classes of syntactic structures -- they act like variables
(we will call them **nonterminal symbols**),

- Nonterminal symbols are distinct from specific syntactic elements (token) of the language --- they act like values
(we will call them **terminal symbols**)

BNF

- BNF rules describes the structure of (fragments of) sentential forms

```
<while_stmt> ::= while <logic_expr> do <stmt>
```

- This rule describe the structure of while statements for a possible language, where <while_stmt>, <logic_expr> and <stmt> are nonterminal and while and do are terminal symbols (to be precise, if they are tokens they will also need the category they belong to).

BNF

- A rule has a left-hand side (LHS) which is a single non-terminal symbol and a right-hand side (RHS), one or more terminal or nonterminal symbols.

```
<while_stmt> ::= while <logic_expr> do <stmt>
```

- A nonterminal symbol is “defined” by one or more rules.
- Multiple rules can be combined with the | symbol so that

```
<stmts> ::= <stmt>
<stmts> ::= <stmt> ; <stmts>
```

is equivalent to

```
<stmts> ::= <stmt> | <stmt> ; <stmts>
```

BNF - Backus Normal/Naur Form

- A grammar is defined by a set of terminals (tokens), a set of nonterminals, a designated **nonterminal start symbol**, and a finite nonempty set of rules

```
<sentence>      ::= <noun-phrase><verb-phrase>.
<noun-phrase>   ::= <article><noun>
<article>       ::= a | an | the
<noun>          ::= man | apple | worm | penguin
<verb-phrase>   ::= <verb> | <verb><noun-phrase>
<verb>          ::= eats | throws | sees | is
```

Derivations using BNF

A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

A derivation example

```
<sentence> ::= <noun-phrase><verb-phrase>.
               <article><noun><verb-phrase>.
               the<noun><verb-phrase>.
               the man <verb-phrase>.
               the man <verb><noun-phrase>.
               the man eats <noun-phrase>.
               the man eats <article><noun>.
               the man eats the <noun>.
               the man eats the apple.
```

Derivations and sentences

- Every string of symbols in the derivation is a sentential form.
- A sentence is a sentential form that has only terminal symbols.
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded.
- A derivation may be either leftmost or rightmost (or something else)

Another BNF example

```
<program> ::= <stmts>
<stmts> ::= <stmt> | <stmt> ; <stmts>
<stmt> ::= <var> = <expr>
<var> ::= a | b | c | d
<expr> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | const
```

Note: grammars rules can be recursive.

A derivation example

```
<program> ::= <stmts>
              <stmt>
              <var> = <expr>
              a = <expr>
              a = <term> + <term>
              a = <var> + <term>
              a = b + <term>
              a = b + const
```

Generator vs Recognizer

```
<program> ::= <stmts>
<stmts> ::= <stmt> | <stmt> ; <stmts>
<stmt> ::= <var> = <expr>
<var> ::= a | b | c | d
<expr> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | const
```

Recognize a sentence

```
a = b + const
<var> = b + const
<var> = <var> + const
<var> = <term> + const
<var> = <term> + <term>
<var> = <expr>
<stmt>
<stmts> =:: <program>
```

Generate a sentence

```
<program> ::= <stmts>
              <stmt>
              <var> = <expr>
              a = <expr>
              a = <term> + <term>
              a = <var> + <term>
              a = b + <term>
              a = b + const
```

Some examples

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

How do we generate the following sentence?

```
the penguin sees.
```

Some examples

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

How do we generate the following sentence?

```
a worm eats an apple.
```

Some examples

```
<sentence>     ::= <noun-phrase><verb-phrase>.
<noun-phrase>  ::= <article><noun>
<article>      ::= a | an | the
<noun>         ::= man | apple | worm | penguin
<verb-phrase>  ::= <verb> | <verb><noun-phrase>
<verb>         ::= eats | throws | sees | is
```

How do we generate the following sentence?

```
a worm is man.
```

Some examples

```
<sentence>    ::= <noun-phrase><verb-phrase>.
<noun-phrase> ::= <article><noun>
<article>     ::= a | an | the
<noun>        ::= man | apple | worm | penguin
<verb-phrase> ::= <verb> | <verb><noun-phrase>
<verb>        ::= eats | throws | sees | is
```

How do we recognize the following sentence?

```
a man throws a penguin.
```

Some examples

```
<program> ::= <stmts>
<stmts> ::= <stmt> | <stmt> ; <stmts>
<stmt> ::= <var> = <expr>
<var> ::= a | b | c | d
<expr> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | const
```

How do we recognize the following sentence?

```
c = a - b; d = c + c
```

Some examples

```
<program> ::= <stmts>
<stmts> ::= <stmt> | <stmt> ; <stmts>
<stmt> ::= <var> = <expr>
<var> ::= a | b | c | d
<expr> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | const
```

How do we recognize the following sentence?

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

Let’s consider a simple one

```
<expression> ::= <term> + <term> | <term> - <term>
<term> ::= <var> | <const>
<var> ::= a | b | c | d
<const> ::= 0 | 1
```

How would we implement this in OCaml?

Let’s consider a slightly more difficult one

```
<expr> ::= (<expr>+<expr>) | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

How would we implement this in OCaml?

How can we implement a BNF in Ocaml?