Introduction to Compilers

Lecture 1

Objectives

By the end of this lecture you should be able to:

1. Identify the functions of different kinds of language processors.
2. Describe the structure of a typical compiler.
3. Identify the function of each component of a compiler.

Outline

1. Language Processors

2. The Structure of a Compiler

Outline

1. Language Processors
2. The Structure of a Compiler

Languages

Definition

An alphabet is a non-empty, finite set of symbols.

Definition

A string is a finite sequence of symbols over some alphabet.

Definition

A language over some alphabet Σ is a set of strings over Σ.

This is fine as far as it goes, but it does not go far enough.

Languages

Definition

An alphabet is a non-empty, finite set of symbols.

Definition

A string is a finite sequence of symbols over some alphabet.

Definition

A language over some alphabet Σ is a set of strings over Σ.

This is fine as far as it goes, but it does not go far enough.

Languages

Definition

An alphabet is a non-empty, finite set of symbols.

Definition

A string is a finite sequence of symbols over some alphabet.

Definition

A language over some alphabet Σ is a set of strings over Σ.

This is fine as far as it goes, but it does not go far enough.

Languages

Definition

An alphabet is a non-empty, finite set of symbols.

Definition

A string is a finite sequence of symbols over some alphabet.

Definition

A language over some alphabet Σ is a set of strings over Σ.

This is fine as far as it goes, but it does not go far enough.

What are Languages Good for?

- Languages are used for
  1. representation
     - The language may be a private language of thought, for example.
  2. communication
     - The language must be a public communication language.
- In either case, strings in the language must be *meaningful*.
  - Whatever that means?

Programming Languages

- Programming languages are languages (sets of strings).
- Each string is a program.
- Programs are meaningful in that they describe computations which can be carried out by people and machines.
- The meaning of programs (semantics) lies in this transformation from programs (syntax) to structural configurations that cause actions.

Programming Languages

- Programming languages are languages (sets of strings).
- Each string is a program.
- Programs are meaningful in that they describe computations which can be carried out by people and machines.
- The meaning of programs (semantics) lies in this transformation from programs (syntax) to structural configurations that cause actions.

Programming Languages

- Programming languages are languages (sets of strings).
- Each string is a program.
- Programs are meaningful in that they describe computations which can be carried out by people and machines.
- The meaning of programs (semantics) lies in this transformation from programs (syntax) to structural configurations that cause actions.

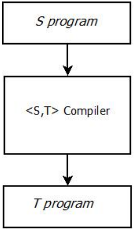
Programming Languages

- Programming languages are languages (sets of strings).
- Each string is a program.
- Programs are meaningful in that they describe computations which can be carried out by people and machines.
- The meaning of programs (semantics) lies in this transformation from programs (syntax) to structural configurations that cause actions.

Machine Language

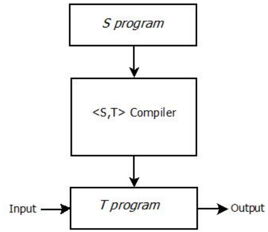
- A machine language is a language over the alphabet $\{0, 1\}$.
- A machine language program is a sequence of instructions.
- When loaded into the machine's instruction register, an instruction causes a unique behavior of the hardware.

Compilers



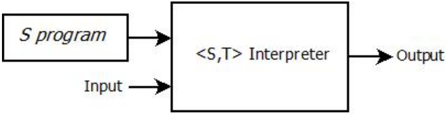
- A compiler is a software system which translates programs in a source language S into equivalent programs in a target language T.
-

Compilers



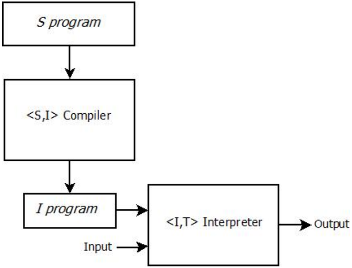
- A compiler is a software system which translates programs in a source language S into equivalent programs in a target language T.
- If T is the machine language of some machine M, then an $\langle S, T \rangle$ compiler makes S-programs meaningful for M.

Interpreters



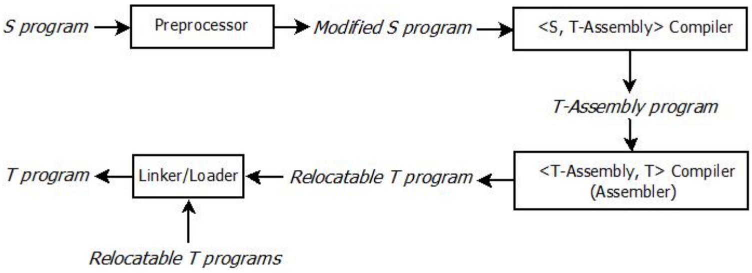
- Unlike a compiler, an interpreter does not produce a target program.
- It appears to be directly executing the source program on the input.
- Actually, it translates one statement of the S program into an equivalent piece of T program which is executed on the input.

Hybrid Compilers



- Sometimes, the S program is compiled into an intermediate I program, which is later interpreted by an $\langle I, T \rangle$ interpreter.
- For example, Java programs are compiled into bytecode, which is later interpreted by a virtual machine.

Structure of a General Language Processor

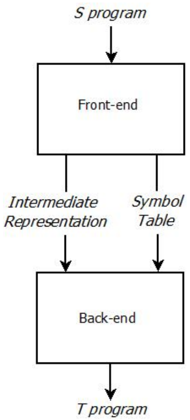


Outline

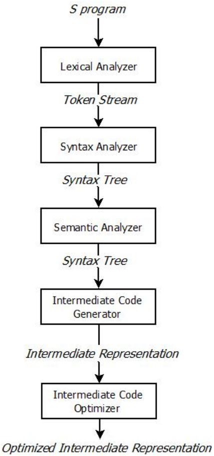
1. Language Processors
2. The Structure of a Compiler

Overall Structure

- The front-end of the compiler analyzes the S program into an intermediate representation.
- The back-end synthesizes the T program.
- The symbol table is a data structure containing a record for each identifier occurring in the S program.
  - What do you think is stored in these records?
- An $\langle S, T_1 \rangle$-compiler and an $\langle S, T_2 \rangle$-compiler may share the front-end. Similarly, An $\langle S_1, T \rangle$-compiler and an $\langle S_2, T \rangle$-compiler may share the back-end.



The Front-End



The Lexical Analyzer (I)

- Lexical analysis (or scanning)
  1. segments the input symbol stream into units called lexemes,
  2. identifies a certain class of symbols (or lexical category) of which the lexeme is a token, and
  3. produces a sequence of tokens of the form

$$\langle L, p \rangle$$

  where L is the name of a lexical category and p is a (possible) pointer to an entry for the token in the symbol table.

The Lexical Analyzer (II)

Example

- Segmentation: *How to recognize speech* vs. *How to wreck a nice peach*.
- Categorization: *How to recognize speech* $\Rrightarrow$ $[Adv, \{Prep, Part, Adv\}, V, N]$

Note that natural languages are lexically ambiguous.

The Lexical Analyzer (III)

Example

- Segmentation: "`position = initial + rate*60`" $\Rightarrow$ [`position`, `=`, `initial`, `+`, `rate`, `*`, `60`]
- Tokenization: $[\langle \textbf{id}, 1\rangle, \langle = \rangle, \langle \textbf{id}, 2\rangle, \langle + \rangle, \langle \textbf{id}, 3\rangle, \langle * \rangle, \langle \texttt{60} \rangle]$

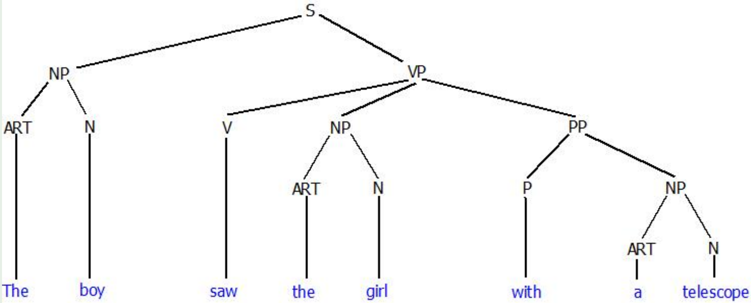
position	...
initial	...
rate	...

The Syntax Analyzer (I)

- Syntax analysis (or parsing) uncovers the recursive structure of a token stream by identifying meaningful sub-streams thereof.
- Typically, such structure is represented by a syntax tree.
- The syntax tree is often necessary for semantic interpretation.
- As an important side-effect, syntax analysis discovers grammatical errors.

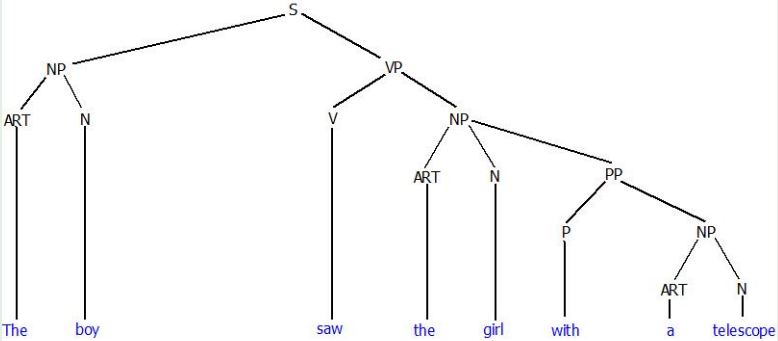
The Syntax Analyzer (II)

Example



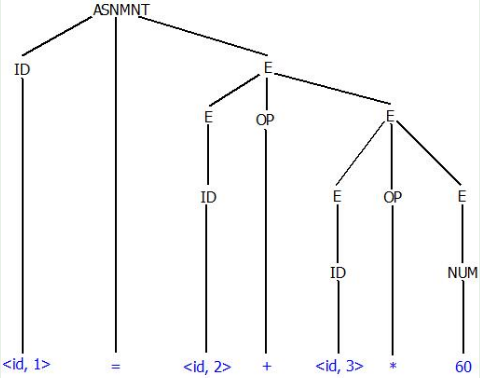
The Syntax Analyzer (III)

Example



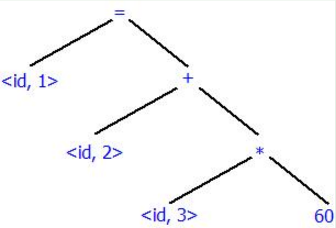
The Syntax Analyzer (IV)

Example



The Syntax Analyzer (V)

Example



Note that the tree indicates the order of evaluating expressions.

The Semantic Analyzer (I)

- The semantic analyzer makes sure that the input (program/sentence) is meaningful.
- English: *The boy saw the girl with a flower.*
- Programming languages: Type checking.

The Semantic Analyzer (I)

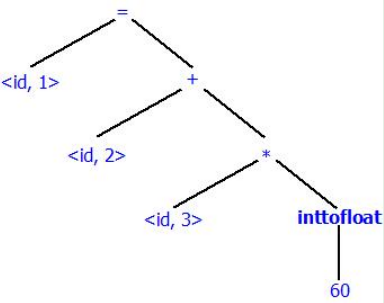
- The semantic analyzer makes sure that the input (program/sentence) is meaningful.
- English: *The boy saw the girl with a flower.*
- Programming languages: Type checking.

The Semantic Analyzer (I)

- The semantic analyzer makes sure that the input (program/sentence) is meaningful.
- English: *The boy saw the girl with a flower.*
- Programming languages: Type checking.

The Semantic Analyzer (II)

Example



The Intermediate Code Generator (I)

- The intermediate representation should be
  1. easy to generate and
  2. easy to translate into the target language.

Example

Three-address code is a common intermediate representation:

- Assembly-like instructions.
- Each with at most three operands.
- Each with at most one operator and a single assignment.

The Intermediate Code Generator (I)

- The intermediate representation should be
  1. easy to generate and
  2. easy to translate into the target language.

Example

Three-address code is a common intermediate representation:

- Assembly-like instructions.
- Each with at most three operands.
- Each with at most one operator and a single assignment.

The Intermediate Code Generator (II)

Example

```
position = initial + rate * 60
```

$$\Rrightarrow$$

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization (I)

- Code optimization attempts to transform the intermediate representation into a “better” target intermediate representation.
- “Better” may mean
  - faster,
  - shorter, or
  - less power-consuming.

Code Optimization (II)

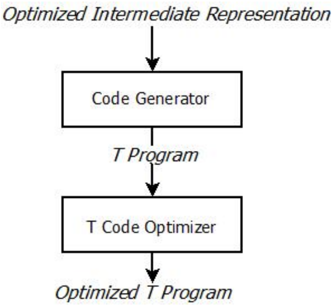
Example

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

$$\Rrightarrow$$

```
t1 = id3 * 60.0
id1 = id2 + t1
```

The Back-End (I)



The Back-End (II)

- The code generator translates the optimized intermediate representation into equivalent T code.
- If T is a machine language, registers or memory locations are allocated to the variables used.
- A carefully designed code generator needs to consider the crucial aspect of choosing variables that will be assigned to registers.
- The resulting T code may be further specifically optimized for the target machine.

The Back-End (III)

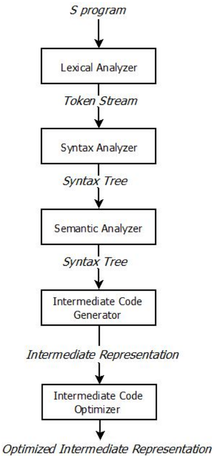
Example

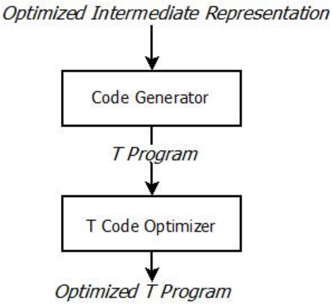
```
t1 = id3 * 60.0
id1 = id2 + t1
```

$\Rrightarrow$

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

Summary


S program
Lexical Analyzer
Token Stream
Syntax Analyzer
Syntax Tree
Semantic Analyzer
Syntax Tree
Intermediate Code Generator
Intermediate Representation
Intermediate Code Optimizer
Optimized Intermediate Representation


Optimized Intermediate Representation
Code Generator
T Program
T Code Optimizer
Optimized T Program