Syntax and Semantics

In this section of the course we will address:

- Relationship between syntax and semantics
- Syntax analysis
    - Grammars
        - BNF (Derivations,Tree Structures, Ambiguous Grammars)
        - Syntax Diagrams (EBNF)
- Semantics
    - General principles
    - Operational, Axiomatic (briefly)
- Role of syntax and semantics in compilers & interpreters

Chapter 12

Elements of language

- What is a language?
- A programming language comprises of
    - syntax: the allowed phrases of the language
    - semantics: what those phrases mean

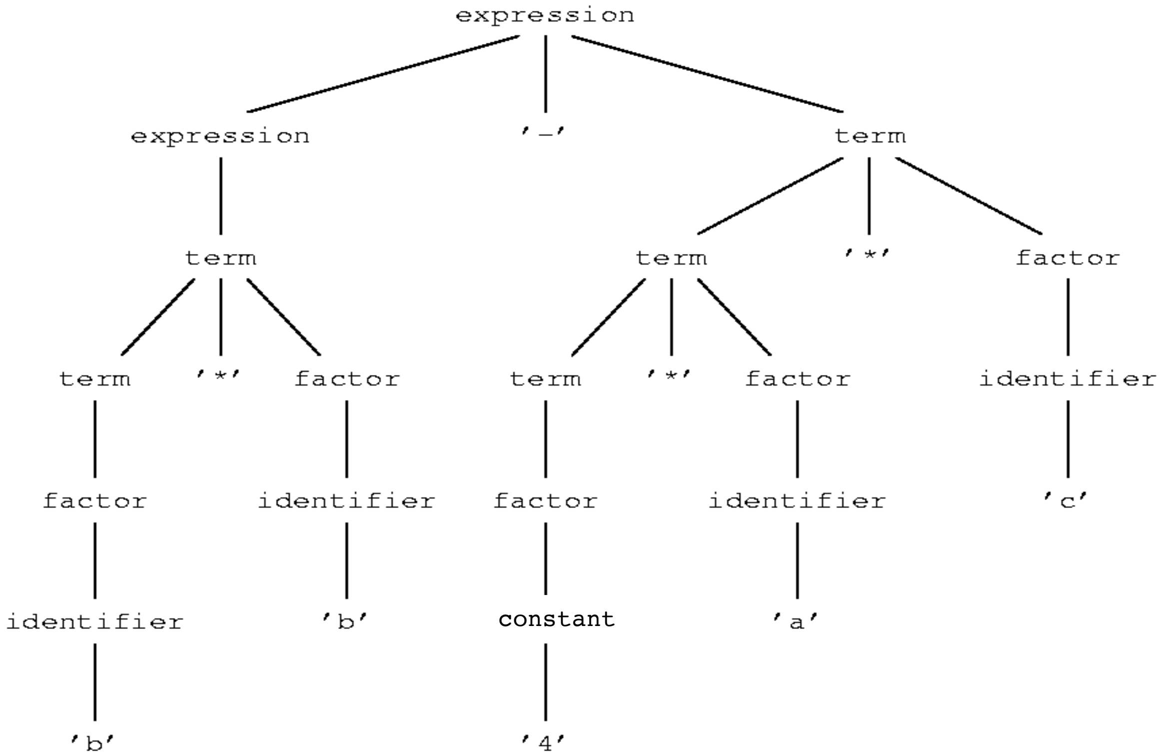
Relating Semantics to Syntax

- There are two relationships involving the semantics and the syntax:
  - one which ensures that each semantic element (meaningful thing) has at least one syntactic representation
  - one which ensures that each syntactic representation has a unique meaning

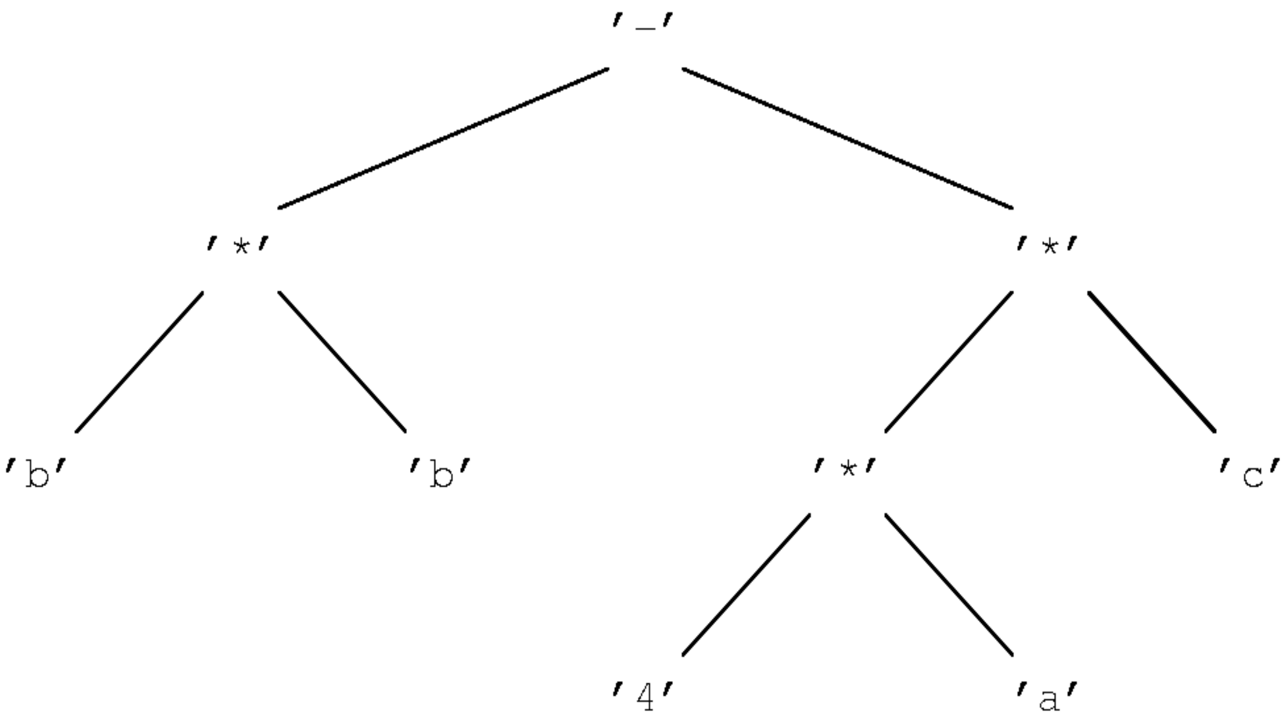
Syntax

- The semantic representation can take the form of a data structure often called the “intermediate” code of the compiler:
  - form is usually an **annotated abstract syntax tree**
- A **syntax tree** of a program text is a data structure showing precisely how segments of the program text are viewed in terms of the **grammar**:
  - obtaining the syntax tree is called **parsing**; sometimes we use the term **parse tree** instead of syntax tree
  - parsing is often called **syntax analysis**
- The parse tree is not always best for further work:
  - modified form is called an **abstract syntax tree** (AST)
  - detailed semantic information can be attached to the nodes of this tree using **annotations**; hence, **annotated abstract syntax tree**

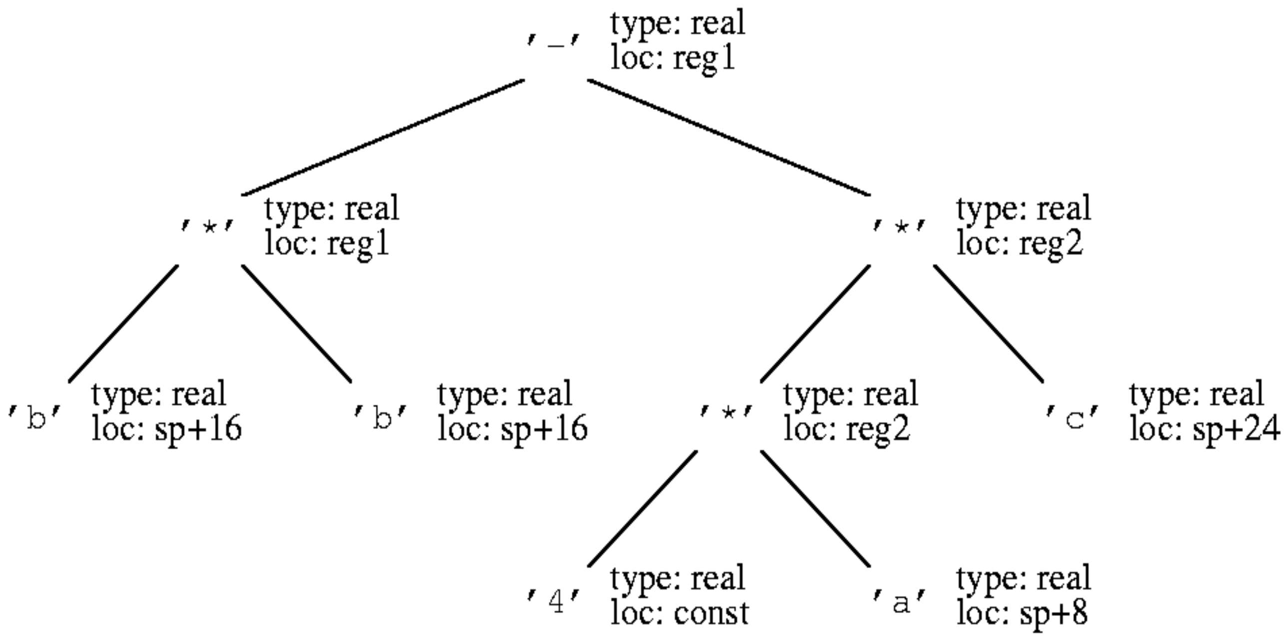
Parse tree for b*b-4*a*c



Abstract syntax tree for b*b-4*a*c



Annotated abstract syntax tree for b*b-4*a*c



Syntax

- Normally, the **grammar** of a programming language is not specified in terms of input characters but by **tokens**:
  - examples of tokens are identifiers (length or a5), strings ("Hello!", "!@#"), numbers (0, 123e-5), keywords (begin, end), compound operators (++, :=), separators (:, [), etc.
- Producing tokens is the task of **lexical analysis** (see later)

Backus-Naur Form & Grammar

- Backus-Naur / Backus-Normal Form (BNF) is a metalanguage
- By metalanguage, we mean a language used to define another language.
- Using BNF to express a language, we can clearly identify which constructs are legal in a language and which are not.

Key features of Backus-Naur Form

- Non-Terminals: defined by a production rule
- Terminals: These are the basic components of the language being defined, e.g. symbols, keywords, variable identifiers, etc in the language being defined
- Production Rule: Each production rule has a non-terminal symbol on the left-hand side, and the right-hand side may contain nonterminals or terminal symbols, possibly in specified sequences.
- Start Symbol: A `top-level' non- terminal symbol which stands for the ‘legal expressions' in the language.

Backus-Naur Form

Here is an example of a grammar:

```
<identifier> ::=  <letter>
                 | <identifier> <digit>
                 | <identifier> <letter>
<letter>     ::= a|b|c|d| ... x|y|z
<digit>      ::= 0|1|2|3|4|5|6|7|8|9
```

The essential features of the BNF formalism are:

1. Angle brackets. These signify non-terminal symbols.
2. The symbol ::= which is read `is defined as'.
3. The symbol | which means ‘or'.
4. The idea of a production rule.
5. A terminal symbol : anything not enclosed in angle brackets.

BNF

```
<identifier> ::=  <letter>
                | <identifier> <digit>
                | <identifier> <letter>
<letter>      ::=  a|b|c|d| ... x|y|z
<digit> ::=   0|1|2|3|4|5|6|7|8|9
```

- What are the legal expressions in this language?
- How would you express in English what an identifier is?

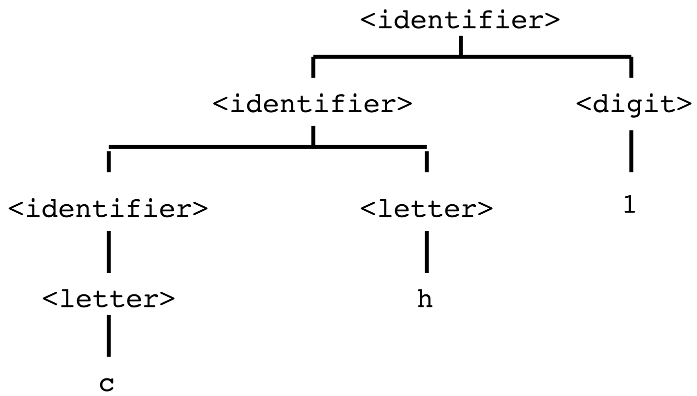
The formal grammar gives a basis for deriving legal expressions. E.g. Is `ch1` is a legal expression?

The **derivation** of ch1 is:

```
<identifier>
<identifier><digit>
<identifier><letter><digit>
<letter><letter><digit>
c<letter><digit>
ch<digit>
ch1
```


Tree Structures

- Such derivations can also be represented by tree structures:



Syntax Analysis

- One of the tasks of a compiler is syntax analysis. This consists precisely of checking that the program as a whole has a corresponding derivation tree, starting from a suitable start symbol, eg <program>.
- A compiler may take a top-down approach or a bottom-up approach in building such a tree.

Phrase Structure and Arithmetic Expressions

```
<exp>       ::= <exp> + <term>
              | <exp> - <term>
              | <term>
<term>      ::= <term> * <factor>
              | <term> / <factor>
              | <factor>
<factor>    ::= ( <exp> )
              | <identifer>
```

- There are four operators (+, -, * and /), with two levels of precedence.
- The grammar imposes a ***phrase structure*** on expressions. In `a * b + c` the subexpression `a * b` is a phrase because it corresponds to a subtree of the derivation tree. This phrase structure gives effect to the precedence of the operators.
- The derivation of `a * (b + c)` the parentheses indicate a <factor>, so its derivation tree would be different.

Two Derivations

```
<exp>
<term>
<term> * <factor>
<factor> * <factor>
<identifier> * <factor>
a * <factor>
a * (<exp>)
a * (<exp> + <term>)
a * (<term> + <term>)
a * (<factor> + <term>)
a * (<identifier> + <term>)
a * (b + <term>)
a * (b + <factor>)
a * (b + <identifier>)
a * (b + c)
```

```
<exp>
<exp> + <term>
<term> + <term>
<term> * <factor> + <term>
<factor> * <factor> + <term>
<identifier> * <factor> + <term>
a * <factor> + <term>
a * <identifier> + <term>
a * b + <term>
a * b + <factor>
a * b + <identifier>
a * b + c
```


Ambiguity

A *derivation* or a *derivation tree* represents the structure of the expression.

Problem: given a legal expression, can we be sure that there is only one derivation?

Answer: No - A *grammar may be ambiguous*.

Ambiguous Grammars

- Another example of a BNF grammar:

```
<statement>        ::=   <conditional statement>
                   | . . .
                   | . . .

<conditional statement> ::=
    if <condition> then <statement>
  | if <condition> then <statement> else <statement>
```

- We presume that <statement> has appropriate other alternative forms, and that <condition> is defined elsewhere.

Ambiguous Grammars

- How is the sentential form

```
if <condition>
then if <condition>
then <statement>
else <statement>
```

to be interpreted?

- This is a well-known problem, the so-called `dangling else' problem.
- The problem is: to which if _ then _ does the else belong?

Ambiguity (continued)

- Demonstrate this grammar is ambiguous by showing there are two derivation trees for

```
if <condition>
then if <condition>
then <statement>
else <statement>
```


Ambiguity (continued)

- The grammars of Pascal and of C are ambiguous - but the compiler decides which interpretation to choose. In this case the first is chosen - an else is always paired with the most recent as yet unpaired then.
- In general it is not possible to decide whether grammars are ambiguous, but certain circumstances are known to lead to ambiguity.
- A grammar is bound to be ambiguous if it is any two of
    - left-recursive
    - self-embedding
    - right-recursive
  - with respect to any one nonterminal symbol.

Ambiguity (continued)

- Left- Recursion

  `<identifier> ::= <identifier> <letter>`

(as the nonterminal being defined is the leftmost symbol in the rhs.)

- Right- Recursion

  `<identifier> ::= <letter> <identifier>`

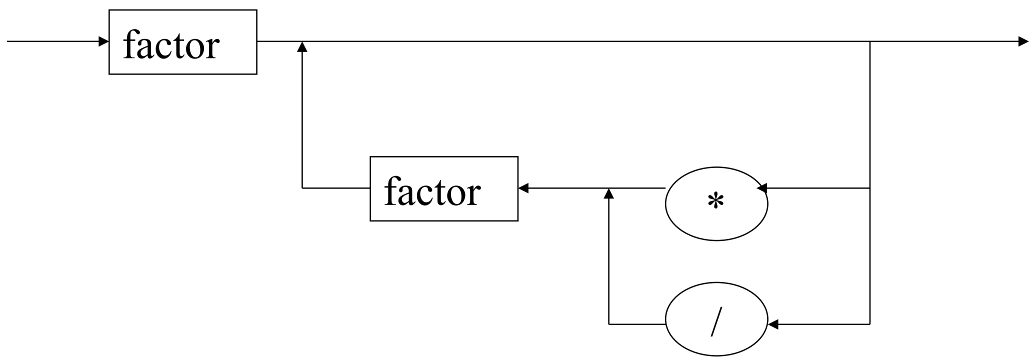
- Self-Embedding

  `<identifier> ::= <letter> <identifier> <letter>`

Syntax Diagrams and Extended BNF (EBNF)

Extended BNF allows iteration instead of recursion, but it describes the same set of languages.

```
term -> factor { ('*' | '/') factor }
```

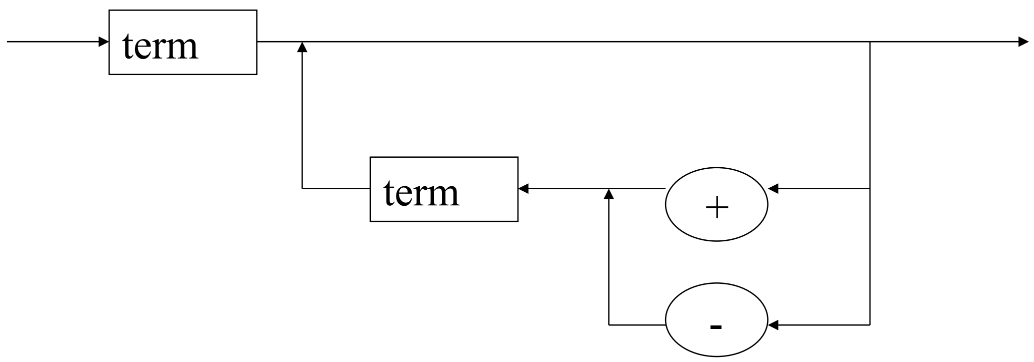


Syntax Diagrams

Syntax diagrams are a convenient way to represent EBNF rules.

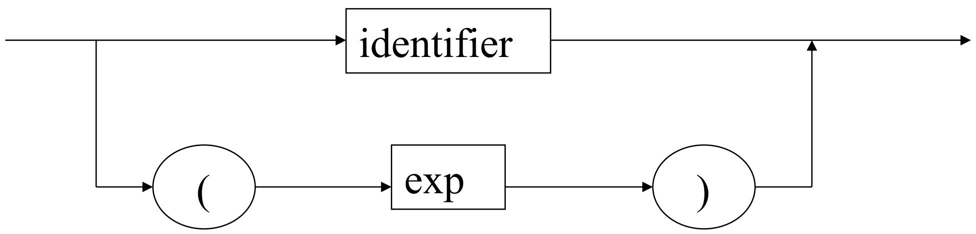
There is one diagram for each nonterminal. The nonterminal is defined by the possible paths through its defining diagram.

```
exp -> term { ('+' | '-') term }
```



Syntax Diagrams

```
factor -> '(' exp ')' | identifier
```

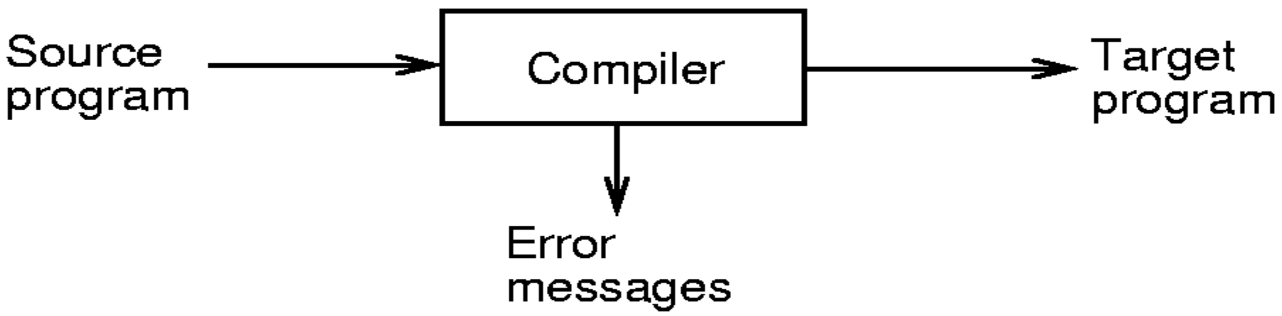


How might unary negation be represented?

Semantics

- **Syntax** is concerned with the *form* of programs.
- **Semantics** is concerned with the *meaning* of programs.
- In a programming language, the meaning of a program can be understood in several different ways:
  - in terms of the executable program produced
  - as a sequence of execution steps defined by certain rules. This is the basis of operational semantics.
  - as a mathematical function, mapping its inputs to its outputs. This is the basis of denotational semantics.
  - in terms of the logical conditions that are true before and after it is executed. This is the basis of axiomatic semantics.
- It is preferable to define the language semantics in terms of something that is itself precisely defined, e.g. mathematical notation.

Syntax, Semantics, Compilers and Interpreters



- A compiler is a program - a language translator.
- It accepts as input a program text written in one language - the source language - and translates it into an equivalent program in another language - the target language
- Part of the translation process is that the compiler reports to the user the presence of errors in the source program
- Normally, the source and target languages differ greatly

Language translation

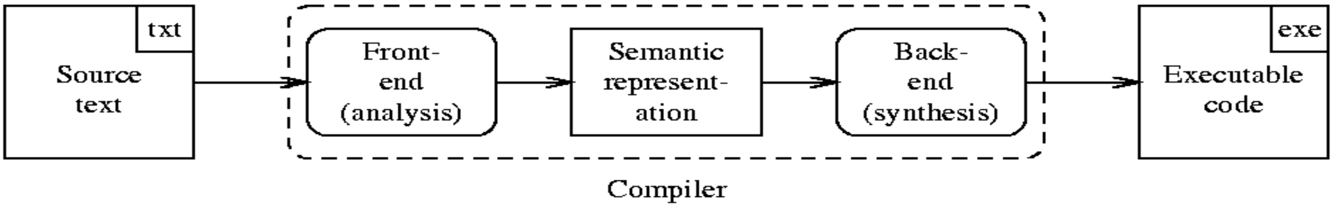
- The language in which the compiler program is written is called the implementation language
- The target program may now run on an actual computer hardware
- There are two questions:
  – what is the translation process?
  – How do we get a compiler in the first place?

Conceptual structure of a compiler

- A compiler is a program which performs a specific task:
  – the input is a language and hence has structure, which is described in the language reference manual
  – the input has meaning, i.e., semantics, which is described in terms of the structure and is attached to the structure in some way
- These properties enable the compiler to understand the input and collect the semantics in a semantic representation
- The target (output) has the same two properties
- The compiler re-forms the collected semantics in terms of the target language

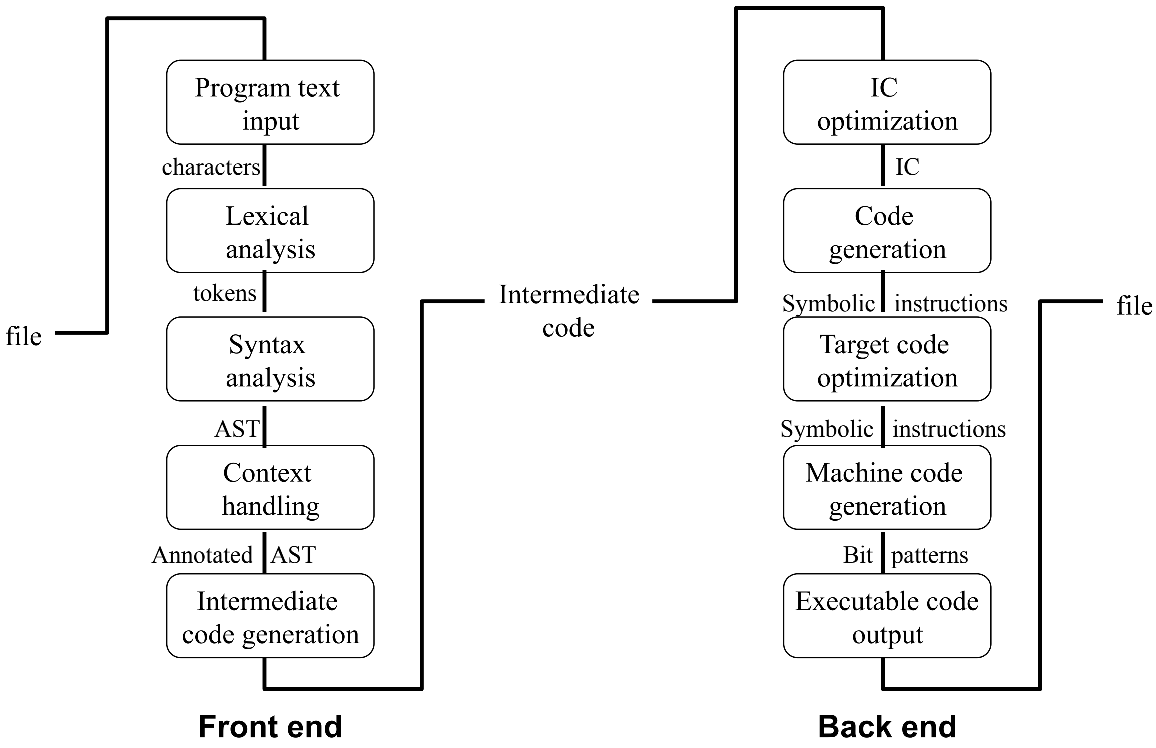
Conceptual structure (cont' d)

- The compiler, therefore, analyses the input, constructs the semantic representation, and synthesises the output from it



- The front-end/semantic representation/back-end structure simplifies the development of compilers for L languages for M machines:
  - no common semantic representation means that we require L*M compilers
  - with a common semantic representation we require L+M modules
- The analysis-synthesis paradigm is very powerful and widely applicable

Realistic compiler

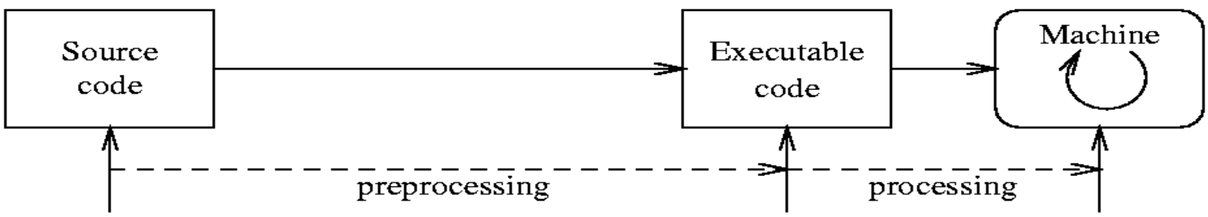


Notable features

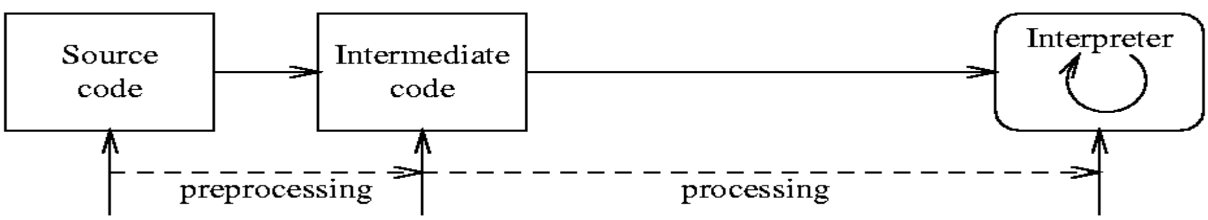
- Important features:
  - symbol-table management: a database of identifiers used in the source program and their corresponding attributes including type, scope, storage allocation information; for procedure/method names, such things as number and type of parameters, method of parameter passing, type of result (if any)
  - context-handler: collects information from various places in the program, and annotates nodes with results. Examples are: relating type information from declarations to expressions; connecting "goto" statements to their program labels, in imperative languages; deciding which routine calls are local and which are remote, in distributed languages
  - error handler: e.g., input characters which don't make up a token, tokens that fail to satisfy the grammar, wrong use of an operation with respect to types (adding an array identifier to a procedure identifier)

Compiling vs. Interpretation

- Diagrammatically, we have



Compilation



Interpretation

Compiling vs. Interpretation (cont' d)

- Advantages of interpretation:
  - interpreters normally written in high-level languages and will, therefore run on most machine types - i.e., better portability
  - writing an interpreter is much less work than writing a back-end (code generator, optimiser, …)
  - allows better error checking and reporting to be done
  - increased security possible by interpreters
  - added flexibility

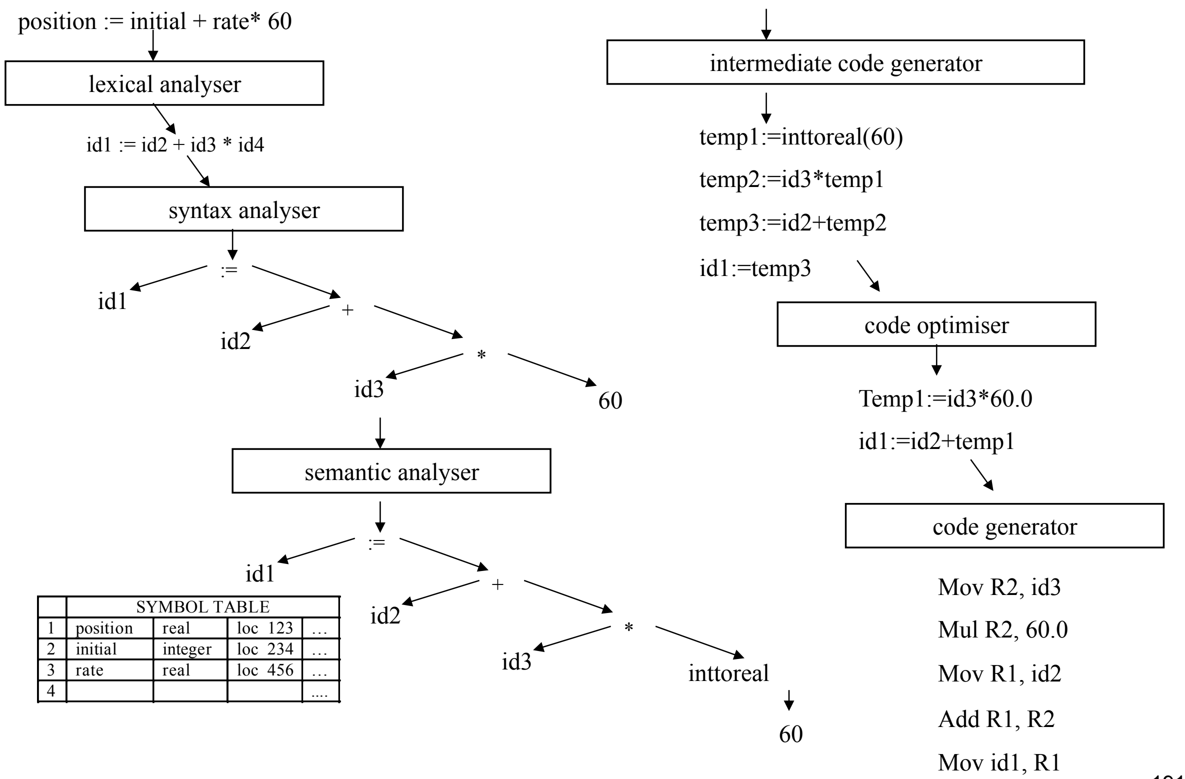
Translation of a statement

- Example of translating into assembly code

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

- Associated with this example we would expect to see a symbol table
- The assembly code is "assembled" by the assembler program into relocatable machine code or object code
- The object code produced via the compiler may require the services of a number of pre-compiled subprograms; the object code plus these subprograms are combined/linked by the linker into a load module (absolute machine code) which the loader places in memory starting at an approved location
- The final product is an executable program

position := initial + rate * 60



Operational Semantics

- Operational semantics is the most low-level of the methods we shall look at.
- It describes the behaviour of programs by giving rules showing how each language construct is to be evaluated.
- There are various approaches. We shall look at structured operational semantics which was used to define the functional language ML.
- We need some basic concepts, e.g.:
  - VAR : a set of variables
  - VAL : a set of values
- We think of a program state or environment E as a mapping from variables to values.

Operational Semantics

- A program is executed within an environment. Execution of the program results in a new environment (and possibly a value as well). We assume the syntax of the language is defined in BNF. The semantics is defined by rules such as the following:
- Assignment Statements

$$\frac{\text{E |- <exp> => v}}{\text{E |- <identifer> = <exp> => E[<identifer> |-> v]}}$$

- Here the environment E is updated to reflect the new binding

- Sequence of Statements

$$\frac{\text{E |- <statement> => E' \quad E' |- <prog> => E''}}{\text{E |- <statement> ; <prog> => E''}}$$

- Operational semantics gives a great deal of information about the details of the execution of a program. This is very useful if, for example, you wish to write a compiler. However, for some purposes, this amount of detail is too low-level.

Axiomatic Semantics

- The effect of a program can be expressed in terms of the conditions which are true before execution (the pre-condition) and the conditions which are true after execution (the post-condition). This is the basis of axiomatic semantics.

- The basic formalism is

  {P} S {Q}

- Here P denotes a pre-condition, S denotes a program segment, and Q denotes a post-condition, and the line above is read:

  `Given the truth of pre-condition P initially, execution of S results in the truth of Q.'

Axiomatic Semantics

- For example,
- assignment statements have the axiom

  {R(e)} x := e {R(x)}

- while sequencing program statements have a rule of inference:

$$\frac{\{P\}\ S1\ \{R\}\ \{R\}\ S2\ \{Q\}}{\{P\}\ S1;S2\ \{Q\}}$$

Axiomatic Semantics

- Axiomatic semantics can be used to develop proofs of correctness. The correctness property is expressed in terms of pre-and post-conditions attached to the program.
- For example, given a program Sqrt, the correct behaviour of the program might be specified as follows:

  { true } y := Sqrt(x) { y * y = x }
- To prove that the program is correct we use the rules of the axiomatic semantics to show that the post-condition above does indeed result from the execution of Sqrt with the pre-condition true.
- This approach is used in the language Eiffel. This language allows pre-conditions and post-conditions to be inserted by the programmer, to allow automatic checking for correctness as the program is being developed.

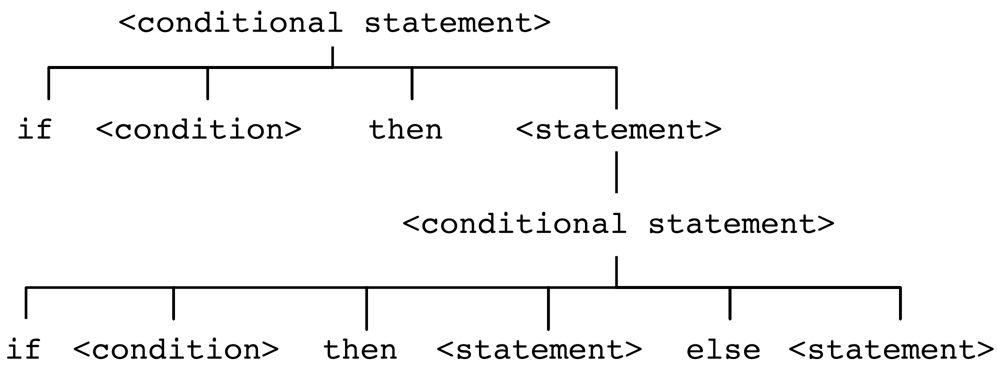
Summary

We have addressed:

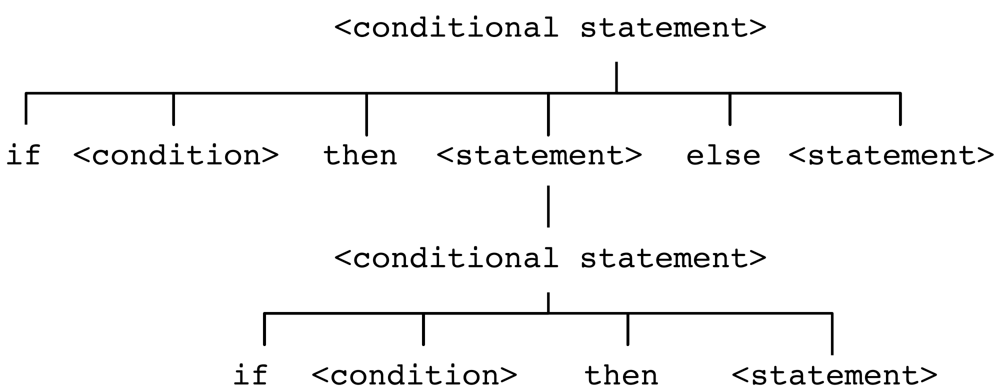
- Syntax:
    - Definition, Grammars (BNF : grammar, derivations, tree structures, ambiguous grammars; syntax diagrams; EBNF)
- Semantics:
    - Operational, Axiomatic (briefly)
- Relationship between syntax and semantics
- Role of syntax and semantics in compilers/interpreters

Ambiguity (continued)

- Here are two derivation trees for the above:



Ambiguity (continued)



Example of grammars and tree structures (not strict BNF form)

Consider the expression

exp → exp '+' term |
exp '-' term |
term

term → term '*' factor |
term '/' factor |
factor

factor → identifier | constant | '(' exp ')'

Example : b*b-4*a*c