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

Elements of language

• What is a language?

• A programming language comprises of

  • syntax: the collection of things by which we represent meaningful things and refer to meaningful things, i.e., the form or structure of the expressions, statements and program units

  • semantics: the collection of things that express the meanings of the language, i.e., the meaning of expressions, statements, program units
Semantics $\longleftrightarrow$ 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 (learning)
  - one which ensures that each syntactic representation has a unique meaning (evaluation)

Syntax

- The semantic representation takes 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
Example of grammars and tree structures
(not strict BNF form)

Consider the expression

\[
\begin{align*}
\text{exp} & \rightarrow \text{exp } + \text{ term } | \\
& \rightarrow \text{exp } - \text{ term } | \\
& \rightarrow \text{term} \\
\text{term} & \rightarrow \text{term } * \text{ factor } | \\
& \rightarrow \text{term } / \text{ factor } | \\
& \rightarrow \text{factor} \\
\text{factor} & \rightarrow \text{identifier } | \text{constant } | \text{('exp ')}
\end{align*}
\]

Example: \(b*b-4*a*c\)
Abstract syntax tree

Annotated abstract syntax tree
Syntax

- The annotated AST shows type and location information has been added to the AST.
- The storage structures used are (CPU) registers and a stack data structure in main memory. For the stack, locations are indicated as offsets from the stack pointer (sp).
- 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.

BACKUS-NAUR FORM

Here is an example of a grammar:

\[
\begin{align*}
\text{<identifier>} &::= \text{<letter>} \\
&\quad | \text{<identifier>} \text{<digit>} \\
&\quad | \text{<identifier>} \text{<letter>} \\
\text{<letter>} &::= \text{a}|\text{b}|\text{c}|\text{d}| \ldots \text{x}|\text{y}|\text{z} \\
\text{<digit>} &::= 0|1|2|3|4|5|6|7|8|9
\end{align*}
\]

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.
What are the legal expressions in this language?

- Usually, but not necessarily, there will be a 'top-level' (or 'start') non-terminal symbol which stands for the 'legal expressions' in the language. In this case `<identifier>` is the start symbol.

- An identifier is a letter, or an identifier followed by a digit, or an identifier followed by a letter.

The formal grammar gives a basis for deriving legal expressions:

The derivation of `ch1` is:

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

So `ch1` is a legal expression (here an `<identifier>`).
BNF

- **Sentential forms**: Each stage in a derivation is known as a sentential form.

- **Sentence**: The final sentential form which does not contain any non-terminals

---

Tree Structures

- Such derivations can also be represented by tree structures:

```
<identifier>
  <identifier> <digit>
    <identifier> <letter> 1
      <letter> h
```

- At each stage, the leftmost nonterminal has been replaced by the right part of one of its defining production rules. Each line in a derivation is a sentential form, and the final line (which contains no nonterminals) is a sentence.
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.
- BNF rules may be recursive. They may be left_recursive (eg <identifier>), or right_recursive or self_embedding (see later).
- It is inherent in BNF that the left side of each rule consists of a single nonterminal symbol, eg context-free grammars. There are kinds of formal grammar which do not have this constraint, eg context-sensitive grammars.

Arithmetic Expressions

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

- There are four operators (+, -, *, and /), with two levels of precedence.
- In the expression a * (b + c) the parentheses indicate a <factor>.
- In the expression a * b + c there are no parentheses, so its derivation tree would be different.
- 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 precedences of the operators.
Two Derivations

\[
\begin{align*}
&\text{<exp>} &\text{<exp>} \\
&\text{<term>} &\text{<exp>} + \text{<term>} \\
&\text{<term>} * \text{<factor>} &\text{<term>} + \text{<term>} \\
&\text{<factor>} * \text{<factor>} &\text{<term>} * \text{<factor>} + \text{<term>} \\
&\text{<identifier>} * \text{<factor>} &\text{<factor>} * \text{<factor>} + \text{<term>} \\
&\text{a} * \text{<factor>} &\text{<identifier>} * \text{<factor>} + \text{<term>} \\
&\text{a} * (\text{<exp>}) &\text{a} * \text{<factor>} + \text{<term>} \\
&\text{a} * (\text{<exp>} + \text{<term>}) &\text{a} * \text{<identifier>} + \text{<term>} \\
&\text{a} * (\text{<term>} + \text{<term>}) &\text{a} * \text{b} + \text{<term>} \\
&\text{a} * (\text{<factor>} + \text{<term>}) &\text{a} * \text{b} + \text{<factor>} \\
&\text{a} * (\text{<identifier>} + \text{<term>}) &\text{a} * \text{b} + \text{<identifier>} \\
&\text{a} * (\text{b} + \text{<term>}) &\text{a} * \text{b} + \text{c} \\
&\text{a} * (\text{b} + \text{<factor>}) & \\
&\text{a} * (\text{b} + \text{<identifier>}) & \\
&\text{a} * (\text{b} + \text{c}) & \\
\end{align*}
\]

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:

```plaintext
<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.

• How is the sentential form
  
  ```plaintext
  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)

- Here are two derivation trees for the above:

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

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

31Y4 Spring 2004
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>
Another ambiguous grammar:

\[
\text{<exp>} ::= \text{<identifier>} \\
| \text{<exp>} + \text{<exp>} \\
| \text{<exp>} * \text{<exp>}
\]

E.g. \(x \times y + z\) has two derivations (and is both left and right recursive).

The extra complication of our original rules for arithmetic expressions is necessary to avoid ambiguity.

This involves the introduction of the two other nonterminals term and factor and allowing the use of parentheses.

Syntax Diagrams and Extended BNF (EBNF)

Extended BNF allows iteration instead of recursion:

\[
\text{term} \rightarrow \text{factor} \{ (\times | \div) \text{factor} \}
\]
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.

\[
\text{exp} \rightarrow \text{term} \{ (\text{"+"} | \text{"-"}) \text{term} \}
\]

\begin{center}
\includegraphics[width=0.5\textwidth]{syntax_diagram_exp.png}
\end{center}

Notes:
1. Rectangular boxes are used for nonterminals.
2. Circles are used for terminals.
3. Arrows indicate the direction of flow.
4. Alternatives are shown in parallel.
5. Iteration is shown by arrows which loop back.
6. All possible routes through the diagram yield legal expressions.

\[
\text{factor} \rightarrow \text{\textbackslash ( \ exp \ \textbackslash )} | \text{identifier}
\]

\begin{center}
\includegraphics[width=0.5\textwidth]{syntax_diagram_factor.png}
\end{center}
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.

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 function 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 like the following:

  - Assignment Statements
    \[
    E |- \langle \text{exp} \rangle \Rightarrow v \\
    E |- \langle \text{identifier} \rangle = \langle \text{exp} \rangle \Rightarrow E[\langle \text{identifier} \rangle \rightarrow v]
    \]

  - Here the environment \( E \) is updated to reflect the new binding

- Sequence of Statements
  \[
  E |- \langle \text{statement} \rangle \Rightarrow E' \\
  E' |- \langle \text{prog} \rangle \Rightarrow E'' \\
  E |- \langle \text{statement} \rangle ; \langle \text{prog} \rangle \Rightarrow E''
  \]

- 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
    \( (R(e)) \times := e(R(x)) \)
  • while sequencing program statements have a rule of inference:
    \[
    \begin{align*}
    \{P\} S_1 (R) (R) S_2 (Q) \\
    \{P\} S_1;S_2 (Q)
    \end{align*}
    \]
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 \} Sqrt \{ 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.

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

Compiling vs. Interpretation

• Front-end: analysis of the source language text
• Back-end: synthesis of the target language text

• In principle, the front-end need know nothing of the target language; the back-end need know nothing of the source language; the only common aspect is the semantic representation

• If all required input data are available, the compiler could perform the actions specified by the semantic representation rather than re-express these in a different form:
  - code-generating back-end is then replaced by an interpreting back-end
  - the whole program is then called an interpreter
Compiling vs. Interpretation (cont'd)

- Diagrammatically, we have

```
Source code -> Executable code -> Machine
  preprocessing  processing
      \                          /
        \                       /
          Compilation

Source code -> Intermediate code -> Interpreter
  preprocessing  processing
```

Advantages of this:
- 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
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)
Translation of a statement

- Example of translating into assembly code
  \[
  \text{position} := \text{initial} + \text{rate} \times 60
  \]

- 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
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