Hochschule für Technik Stuttgart

Concepts of Programming Languages

7th Week

Syntax and Semantics

Prof. Dr. Peter Heusch

Introduction

- The syntax of a programming language describes its form and structure:
    - From the perspective of a programmer the syntax describes how to write formally correct code
    - From the perspective of the compiler the syntax describes how to check code for formal correctness
- The semantics of a programming language describes the meaning of a program
- Together, syntax and semantics define the programming language

Topics

- How to describe syntax and semantics
  - Formal description methods: languages and grammars
  - Attribute grammars: consistency beyond sequence
- (Dynamic semantics: the Meaning of a Program)

Some Formal Language Theory

- An alphabet ∑ is a finite set of symbols
- A sentence or word $w = w_1 \ldots w_n$ is a finite sequence of symbols from ∑, its length is n
- A language is defined as a set of words
- For programming languages we usually have to different „layers“ of formal languages:
  - The lower layer transforms characters to tokens
  - The upper layer transforms tokens to programs

Grammars define Languages

- Natural languages are described by grammars:
  Sentence ==> Subject Predicate Object
- A grammar is a symbol replacement process:

```
Expr ==> Term | Term '+' Expr
Term ==> Factor | Factor '*' Term
Factor ==> '0' | ... | '9' | '(' Expr ')'
```

- Grammars are useful for both directions of programming language processing:
  - Generating a program
  - Recognizing a program

Generators and Recognizers

- A Generator is an (often human) entity that generates programs according to the syntax of a programming language
- A Recognizer reads characters from an input and decides whether the given character sequence is a sentence of the language
  - Syntax analysis (context free and context sensitive)
  - Output generation (usually machine code or byte code or direct execution)

The lower level: Token Recognition

- Transform the given character input into a sequence of lexemes:

  ```
  public class MyClass { public static main(
  String args[]){System.out.println("Hello
  World");}}
  ```

  becomes

  ```
  'public' 'class' 'MyClass' '{' 'public'
  'static' 'main' '(' 'String' 'args' '[' ']'
  ')' '{' 'System' '.' 'out' '.' 'println' '('
  '"Hello World"' ')' ';' '}' '}‘
  ```

- Assign types to the lexemes:
  - '`public`', '`class`', '`static`' are keywords
  - '`String`', '`out`', '`main`' are identifiers

Regular Expressions

- A typical grammar on the lower level consists of regular expressions
  - Every character is a regular expression matching itself (some characters like . \ etc. are special)
  - A set of characters is matched by [abc], [a-z], [^0-9]
  - A quantor designates optional (?), arbitrary (*), at least one (+) or exactly defined ({2,5}) repetition of the preceeding r.e.
  - The | denotes alternatives
- In practice, regular expressions can be much more complex and syntax varies (somewhat)

Regular Expressions in JavaCC

```
TOKEN :
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f",
     "A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
}
```

RegExps and Finite Automata

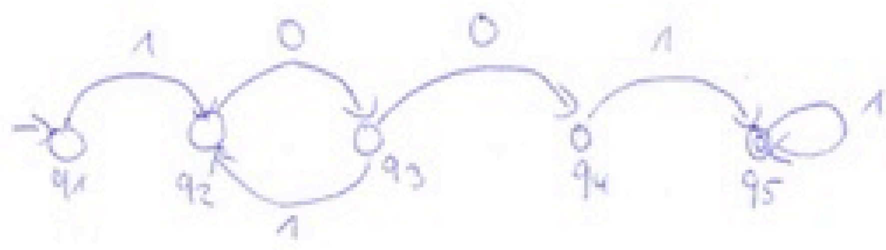
- Internally, regular expressions are represented by finite automata consisting of:
  - An alphabet ∑
  - A set of states Q
  - A state transition function δ : Q x ∑ -> Q
  -
- Processing the RegEx means stepping through the automaton
  - Most efficient type of known grammar classes: Worst-case performance is O(n)

Sample Automaton

Task: Recognize words from the language $(10)^n01^m$, m and n > 0
(and only those)

Ex: 10101001 is in the language
10001 is not in the language

The automaton:



The higher level: Context Free Grammars

- Context Free Grammars were developed in the 50s by Noam Chomsky to describe the recursive properties of natural languages:
  Diejenige Person, welche denjenigen Übeltäter, der dasjenige Schild, welches an der Kreuzung, wo Bleicherstraße und Schellingstraße sich treffen, stand, umgefahren hat, anzeigt, erhält eine Belohnung
- Programming languages during the 50s had mostly regular grammars
- This is no longer true: Much more complexity

Context Free Grammars

- A CFG consists of
  - A vocabulary **V** containing
    - A set of terminal symbols **T**
    - A set of non-terminal symbols N := V \ T, one of which is the start symbol **S**
  - A set of Productions (rules) **P** from N x T*
- CFGs are processed by stack automata: Stack allows „counting“ (e.g., making sure that brackets are balanced)
- Expressivity comes at the price of efficiency: Worst-case performance is $O(n^3)$

(Extended) Backus-Naur-Forms

- BNF and EBNF were developed to describe context free grammars
- Like Regular Expressions (E)BNF is a meta-language that describes another language
- First used in the description of Algol 58
- BNF-rules group elements of the description:
  - Variables or Nonterminals are used inside the BNF but do not appear in the final program
  - Terminals appear within the final program, denoted by surrounding '-' signs

BNF Fundamentals

- Every BNF Grammar contains of a set of rules
  - The left hand side (LHS) is a single nonterminal
  - The right hand side (RHS) is a sequence of terminals and nonterminals
  - A rule is allowed to have multiple right hand sides
  -
- Example:

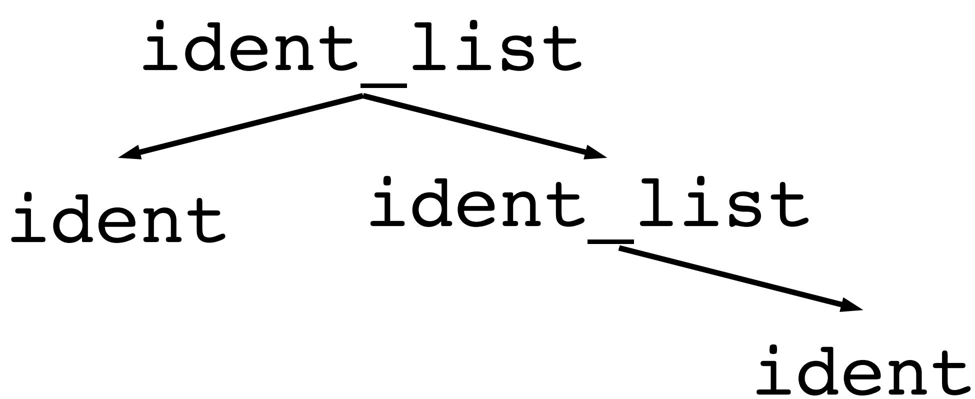
```
<tri_digit> ->  1
             |  2
             |  3 ;
```

(Recursive) Derivation in BNF

- Lists are described by using recursion:

```
<ident_list> ->  ident
               | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
- Derivations are usually written as trees



A sample grammar

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

EBNF

- BNF is not very expressive, hence some typical tasks are clumsy:
    - Optional element (if/else)
    - Repetition (statement list, arithmetic expression)
    - Alternative parts (arithmetic expressions)
- In 1972 Niklaus Wirth extended BNF to EBNF for the description of Pascal
- EBNFs expressive power is equal to the one of BNF

Innovations of EBNF

- Optional parts are enclosed in brackets ([]):

```
if_stmt ->   if ( <cond> )
<stmt_or_block>
[ else <stmt_or_block> ]
```

- Repetitions are denoted by braces:

```
<program> -> <stmt> { ; <stmt> }
```

- Alternatives can be placed in parentheses, separated by |:

```
expr -> <term> (+|-) <term>
```

- The resulting notation is much more concise

Semantics: Beyond CFGs

- It is not possible to describe the whole ruleset of a programming language in BNF/EBNF:
  - Declarations and types of variables
  - Application of operators (Multiplication of Strings?)
- The following is „perfectly legal“ Java:

```
public class MyClass {
  String foo;
  public static void bar(int x) {
    return x / foo;
  }
}
```

- Need to cover **static semantics** (describe legal form of programs, can be checked at compile time)
-

Attribute Grammars

- Attribute Grammars extend EBNF by adding attributes to terminals and nonterminals
- Attributes can contain things like:
  - Definedness
  - Type
  - Reachability
- Moreover attributed grammes define rules („predicate functions“) that govern whether derivations are allowed:

```
<assign> -> <variable> = <expr>
   iff <expr>.type == <variable>.type
```

Defining Attribute Grammars

- To define an attribute grammar, a context free grammar G = (V, T, P, S) is taken and extended by attributes:
  - For every grammar element x from V, an attribute set A(x) is defined
  - Every rule gets a set of functions that compute the attributes of the nonterminals in the rules
- Predicate functions on rules check attribute consistency

Inferring Attributes

- Intrinsic attributes are defined by the scanner initially
- When X_0 -> X_1 ... X_n is a rule, then:
  - Synthesized attributes compute attributes of X_0 from attributes of X_1 ... X_n
  - Inherited attributes compute attributes of X_i from attributes of X_0 ... X_n except X_i

Attribute Grammar Example

```
<assign>  -> <var> = <expr>
<expr>    -> <var> + <var> | <var>
<var>   -> A | B | C
```

Attributes:

- `actual_type`: synthesized for <assign>, `<var>` and `<expr>` (as left-hand sides)
- `expected_type`: inherited for `<expr>` on the right-hand side

Type Inference by Attributes

- Syntax rule: `<expr> -> <var>[1] + <var>[2]`

  Semantic rule:
  `<expr>.act_type <- <var>[1].act_type`

  Predicate:
  `<var>[1].act_type==<var>[2].act_type`
  `<expr>.exp_type == <expr>.act_type`

- Syntax rule: `<var> -> id`

  Semantic rule:
  `<var>.act_type ← lookup(<var>.string)`

Type Evaluation Steps

```
<expr>.expected_type <- inherited from parent
<var>[1].actual_type <- lookup (A)
<var>[2].actual_type <- lookup (B)
<var>[1].actual_type =? <var>[2].actual_type
<expr>.actual_type <- <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type
```

Dynamic Semantics

- The dynamic semantics of a program describes the intended meaning of a program:
  - The operational semantics specifies what happens when the program is executed
  - The axiomatic semantics specifies how the input is transformed to the output
- Unfortunately there is no standardized way of specifying either:
  - Operational semantics is specified plain text
  - Axiomatic semantics gets specified by code