Syntax and Type Analysis

Lecture Compilers Summer Term 2011

Prof. Dr. Arnd Poetzsch-Heffter

Software Technology Group
TU Kaiserslautern

Content of Lecture

2. Syntax and Type Analysis

Educational Objectives

- Tasks of different syntax analysis phases
- Interaction of syntax analysis phases
- Specification techniques for syntax analysis
- Generation techniques
- Usage of tools
- Lexical analysis
- Context-free analysis (parsing)
- Context-sensitive analysis

Syntax Analysis

Tasks of Syntax Analysis

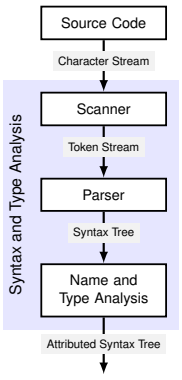
- Check if input is syntactically correct
- Dependent on result:
  - Error message
  - Generation of appropriate data structure for subsequent processing

Syntax and Type Analysis Phases

Lexical analysis:
Character stream → token stream (or symbol stream)

Context-free analysis:
Token stream → syntax tree

Context-sensitive analysis:
Syntax tree → syntax tree with cross references



Reasons for Separation of Phases

- Lexical and context-free analysis
  - Reduced load for context-free analysis, e.g., whitespaces are not required for context-free analysis
- Context-free and context-sensitive analysis
  - Context-sensitive analysis uses tree structure instead of token stream
  - Advantages for construction of target data structure
- For both cases
  - Increased efficiency
  - Natural process (cmp. natural language)
  - More appropriate tool support

2.1. Lexical Analysis

Lexical Analysis

Tasks

- Break input character stream into a token stream wrt. language definition
- Classify tokens into token classes
- Representation of tokens
  - Hashing of identifiers
  - Conversion of constants
- Elimination of
  - whitespaces (spaces, comments...)
  - external constructs (compiler directives...)

Lexical Analysis (2)

Terminology

- **Token/symbol**: a word over an alphabet of characters (often with additional information, e.g. token class, encoding, position..)
- **Token class**: a set of tokens (identifier, constants, ...); correspond to terminal symbols of a context-free grammar

Remark: the terms *token* and *symbol* refer to the same concept. The term token is in general used when talking about parsing technology, whereas the term symbol is used when talking about formal languages.

Lexical Analysis: Example

Input Line 23:

␣␣if␣(␣A␣<=␣3.14␣)␣␣␣B␣=␣B−−

Token Class	String	Token Information	Col:Row
IF	“if”		23:3
OPAR	“(”		23:5
ID	“A”	72 (Hash)	23:7
RELOP	“<=”	4 (Encoding)	23:9
FLOATCONST	“3.14”	3,14 (Constant Value)	23:12
CPAR	“)”		23:16
ID	“B”	84 (Hash)	23:20
...			

Specification

The specification of the lexical analysis is a part of the language specification.

The two parts of lexical analysis specification:

- Scanning algorithm (often only implicit)
- Specification of tokens and token classes

Examples: Scanning

1. Statement in C

   `B␣=␣B␣---␣A;`

   Problem: Separation (- - and - are tokens)
   Solution: Longest token is chosen, i.e,

   `B␣=␣B␣--␣-␣A;`

2. Java Fragment

   `class␣public␣{␣public␣m()␣{...}␣}`

   Problem: Ambiguity (keyword, identifier)
   Solution: Precedence rules

Standard Scan Algorithm (Concept)

Scanning is often implemented as a procedure:

- Procedure returns next token
- State is remainder of input
- In error cases, returns the `UNDEF` token and updates the input

Standard Scan Algorithm (Pseudo Code)

```
CharStream inputRest := input;

Token nextToken() {
  Token curToken := longestTokenPrefix(inputRest);
  inputRest:= cut(curToken, inputRest);
  return curToken;
}
```

where `cut` is defined as

- if `curToken` $\neq$ `UNDEF`, `curToken` is removed from `inputRest`
- else `inputRest` remains unchanged.

Standard Scan Algorithm (2)

```
Token longestTokenPrefix(CharStream ir) {
  require availableChar(ir) > 0
  int curLength = 1;
  String curPrefix := prefix(curLength,ir);
  Token longestToken := UNDEF;

  while( curLength <= availableChar(ir)
      && isTokenPrefix(curPrefix) ) {
    if (isToken(curPrefix) {
      longestToken := curPrefix;
    }
    curLength++;
    curPrefix := prefix(curLength,ir);
  }
  return longestToken;
}
```

Standard Scan Algorithm (3)

Predicates to be defined:

- `isTokenPrefix`: String → boolean
- `isToken`: String → boolean

Remarks:

- Standard scan algorithm is used in many modern languages, but not, e.g., in FORTRAN because blanks are not special, except in literal tokens, e.g.
  - `DO 7 I = 1.25` ⇒ "`DO 7 I`" is an identifier.
  - `DO 7 I = 1,25` ⇒ "`DO`" is a keyword.
- Error cases are not handled
- Complete realization of `longestTokenPrefix` is discussed later.

Specification of Token Classes

- Token classes are defined by regular expressions (REs).
- REs specify the set of strings, which belong to a certain token class.

Regular Expressions

Let Σ be an *alphabet*, i.e. an non-empty set of characters. Σ^* is the set of all words over Σ, ϵ is the empty word.

Definition (Regular expressions, regular languages)

- ε is a RE and specifies the language $L = \{\epsilon\}$.
- Each $a \in \Sigma$ is a RE and specifies the language $L = \{a\}$.
- Let r and s be two RE specifying the languages R and S, resp. Then the following are RE and specify the language L:
  - $(r|s)$ with $L = R \cup S$ (union)
  - rs with $L = \{vw \mid v \in R, w \in S\}$ (concatenation)
  - r^* with $\{v_1 \ldots v_n \mid v_i \in R, 0 \leq i \leq n\}$ (Kleene star)

The language $L \subseteq \Sigma^*$ is called regular if there exists RE r defining L.

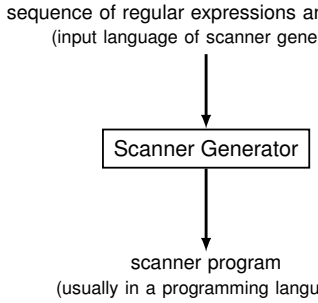
Regular Expressions (2)

Remarks:

- $L = \emptyset$ is not regular according to the definition, but is often considered regular.
- Other Operators, e.g. +, ?, ., [] can be defined using the basic operators, e.g.
  - $r^+ \equiv (r\ r^*) \equiv r^* \setminus \{\epsilon\}$
  - $[aBd] \equiv a \mid B \mid d$
  - $[a - g] \equiv a \mid b \mid c \mid d \mid e \mid f \mid g$

Caution: Regular expressions only define valid tokens and do not specify the program or translation units of a programming language.

Implementation of Scanners



Scanner Generator: JFlex

- Typical use of JFlex:

```
java -jar JFlex.jar Example.jflex
javac Yylex.java
```

  Actions are written in Java
- Examples :
  1. Regular expression in JFlex

```
[a-zA-Z_0-9] [a-zA-Z_0-9] *
```

  2. JFlex input with abbreviations

```
ZI = [0-9]
BU = [a-zA-Z_]
BUZI = [a-zA-Z_0-9]
%%
{BU}{BUZI}* { anAction(); }
```


A Complete JFlex Example

```
enum Token { DO, DOUBLE, IDENT, FLOATCONST, STRING;}
%%

%line
%column
%debug
%type Token // declare token type

ZI = [0-9]
BU = [a-zA-Z_]
BUZI = [a-zA-Z_0-9]
ZE = [a-zA-Z_0-9!?\]\[\. \t...]
WhiteSpace = [ \t\n]

%%
{WhiteSpace} { }
"double" { return Token.DOUBLE; }
"do" { return Token.DO; }
{BU}{BUZI}* { return Token.IDENT; }
{ZI}+\.{ZI}+ { return Token.FLOATCONST; }
\"({ZE}|\\\")*\" { return Token.STRING; }
<<EOF>> { System.out.println("FINISHED"); return null; }
```


Scanner Generators

- Scanner generation uses the equivalence between
  - Regular expressions
  - Non-deterministic finite automata (NFA)
  - Deterministic finite automata (DFA)
- Construction methods is based in two steps:
  - Regular expressions → NFA
  - NFA → DFA

Definition of NFA

Definition (Non-deterministic Finite Automaton)
A *non-deterministic finite automaton* is defined as a 5-tuple

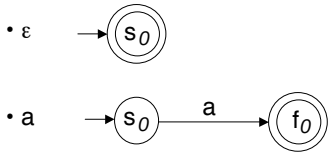
$$M = (\Sigma, Q, \Delta, q_0, F)$$

where

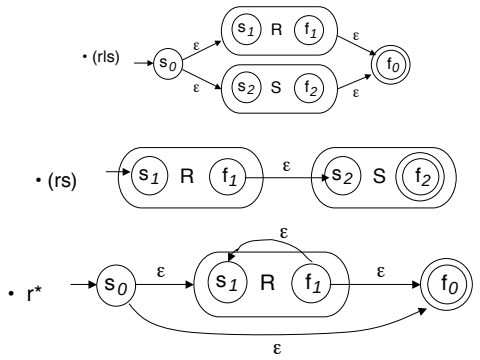
- Σ is the input alphabet
- Q is the set of states
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states
- $\Delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$ is the transition relation.

Regular Expressions → NFA

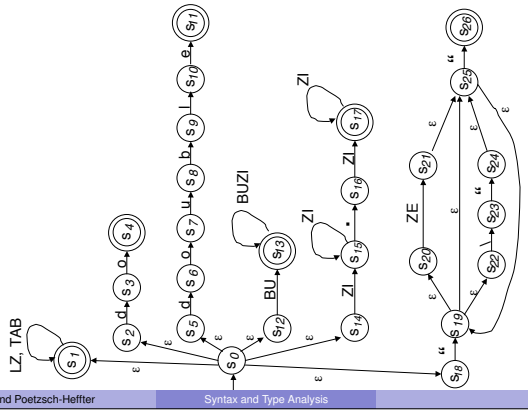
Principle: For each regular sub-expression, construct NFA with one start and end state that accepts the same language.



Regular Expressions → NFA (2)



Example: Construction of NFA



ϵ-closure

Function `closure` computes the ϵ-closure of a set of states $s_1, \ldots, s_n$.

Definition (ϵ-closure)
For an NFA $M = (\Sigma, Q, \Delta, q_0, F)$ and a state $q \in Q$, the ϵ-closure of q is defined by

$$\epsilon\text{-closure}(q) = \{p \in Q \mid \text{p reachable from q via } \epsilon\text{-transitions}\}$$

For $S \subseteq Q$, the ϵ-closure of S is defined by

$$\epsilon\text{-closure}(S) = \bigcup_{s \in S} \epsilon\text{-closure}(s)$$

Longest Token Prefix with NFA

```
Token longestTokenPrefix(char[] ir) {
  // length(ir) > 0
  StateSet curState := closure( {s0} );
  int curLength := 0;
  int tokenLength := undef;

  while (curLength <= length(ir) && !isEmptySet(curState) ) {
    if (contains(curState,FinalState)) {
      tokenLength := curLength;
    }
    curLength++;
    curState := closure(successor(curState,ir[curLength]));
  }
  return token(prefix(ir,tokenLength));
}
```


Longest Token Prefix with NFA (2)

Remark:

Problem of ambiguity:

If there are more than one token matching the longest input prefix, procedure `token` nondeterministically returns one of them.

NFA → DFA

Principle:

For each NFA, a DFA can be constructed that accepts the same language. (In general, this does not hold for NFA with output.)

Properties of DFA:

- No ϵ-transitions
- Transitions are deterministic given the input char

NFA → DFA (2)

Definition (Deterministic Finite State Automaton)

A *deterministic finite automaton* is defined as a 5-tuple

$$M = (\Sigma, Q, \Delta, q_0, F)$$

where

- Σ is the input alphabet
- Q is the set of states
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states
- $\Delta : Q \times \Sigma \to Q$ is the transition function.

NFA → DFA (3)

Construction: (according to John Myhill)

- The States of the DFA are subsets of NFA states (powerset construction). Subsets of finite sets are also finite.
- The start state of the DFA is the ϵ-closure of the *NFA* start state
- The final states of the DFA are the sets of states that contain an NFA final state.
- The successor state of a state S in the DFA under input a is obtained by
  - computing all successors p of $q \in S$ under a in the NFA
  - and adding the ϵ-closure of p

NFA → DFA (4)

- If working with character classes (e.g. [a-f]), characters and character classes at outgoing transitions must be disjoint.
- Completion of automaton for error handling:
  - Insert additional (final) state (nT)
  - For each state, add a transition for each character for which no outgoing transition exists to the nonToken state.

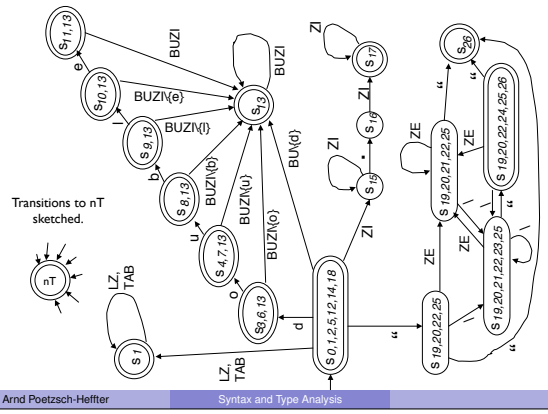
NFA → DFA (5)

Definition (DFA for NFA)

Let $M = (\Sigma, Q, \Delta, q_0, F)$ be a NFA. Then, the DFA M' corresponding to the NFA M is defined as $M' = (\Sigma, Q', \Delta', q_0', F')$ where

- the set of states is $Q' \subseteq \mathcal{P}(Q)$, power set of Q
- the initial state q_0' is the ϵ-closure of q_0
- the final states are $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$
- $\Delta'(S, a) = \epsilon\text{-closure}(\{p \mid (q, a, p) \in \Delta, q \in S\})$ for all $a \in \Sigma$.

Example: DFA



Longest Token Prefix with DFA

```
Token longestTokenPrefix(char[] ir) {
  // length(ir) > 0
  State curState : = StartState;
  int curLength := 0;
  int tokenLength := undef;

  while (curLength <= length(ir) && curState != nT)
    if (curState is FinalState) {
      tokenLength := curLength;
    }
    curLength++;
    curState := successor(curState,ir[curLength]));
  }
  return token(prefix(ir,tokenLength));
}
```


Longest Token Prefix with DFA (2)

Remarks:

- Computation of closure at construction time, not at runtime. (Principle: Do as much statically as you can!)
- Problem of ambiguity still not solved. However, many scanner generators allows the user to control which token is returned. For example, JFlex returns the token of the first rule in the JFlex file that matches the longest input prefix.

Longest Token Prefix with DFA (3)

Implementation Aspects:

- Constructed DFA can be minimized.
- Input buffering is important: often use of cyclic arrays (caution with maximal token length, e.g. in case of comments)
- Encode DFA in table
- Choose suitable partitioning of alphabet in order to reduce number of transitions (i.e. size of table)
- Interface with parser: usually parser asks proactively for next token

Recommended Reading

- Wilhelm, Maurer: Chap. 7, pp. 239-269 (More theoretical)
- Appel: Chap 2, pp. 16 - 37 (More practical)

Additional Reading:

- Aho, Sethi, Ullman: Chap. 3 (very detailed)