Compiler Design
Syntax Analysis
Introduction to Syntax Analysis

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

http://www.cs.upt.ro/~chirila

Outline

- Syntax Analysis
- Syntax Rules
- The Role of the Parser
- Types of Parsers
- Representative Grammars
- Ambiguous Grammars
- Syntax Error Handling

Syntax Analysis

- parsing methods typically used in compilers
- basic concepts
- techniques suitable for hand implementation
- algorithms used in automated tools
- recovery methods from common errors

Syntax Rules

- each programming language has precise rules that prescribe the syntactic structure of well formed programs
- e.g.: a C program
  - functions
    - declarations
    - statements
      - expressions
- can be expressed as
  - context-free grammars
  - BNF rules
- grammars offers benefits for both
  - language designers
  - compiler writers

Grammar Benefits

- precise syntactic specification of a programming language
- from certain classes of grammars efficient parsers can be automatically generated
- the structure disclosed by a grammar is useful for
  - translating source programs object code
  - detecting errors
- allows a language to
  - to evolve
  - to be developed iteratively and incrementally

Topics

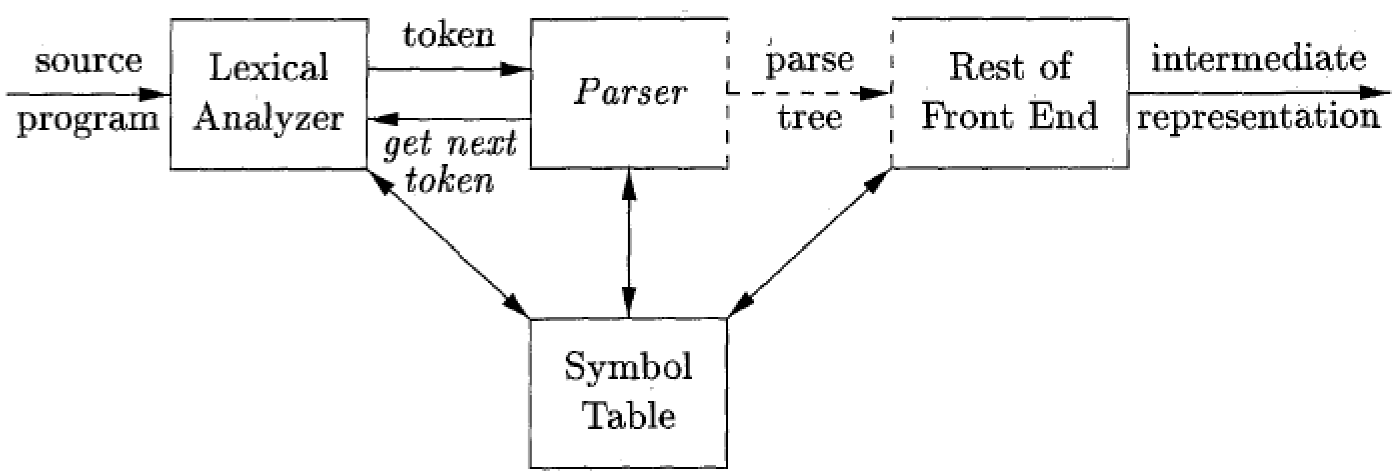
- how a parser fits into a typical compiler
- take a look at typical grammars for arithmetic expressions
  - carry over to most programming constructs
- error handling
  - finding that the input can not be generated by the grammar

The Role of the Parser

- to receive a string of tokens from the lexical analyzer
- to verify whether the string of token names can be generated by the grammar for a source language
- to report any syntax errors in an intelligible way
- to recover from commonly occurring errors
- to continue processing the remainder of the program

The Role of the Parser

- to construct a parse tree
- to pass it to the rest of the compiler for further processing
- to intersperse with checking and translations actions



Types of Parsers

- Universal
  - can parse any grammar
    - Cocke-Younger-Kasami parsing methods
    - Earley's algorithm
  - inefficient to be used in production of compilers
- Top-down - build parse trees
  - from top (root)
  - to the bottom (leaves)
- Bottom-up
  - start from leaves
  - work their way up to the root
- Both top-down and bottom-up
  - scan the input from left to right
  - one symbol at a time

Types of Parsers

- most efficient top-down and bottom-up work for subclasses of grammars
  - LL and LR are expressive enough to describe most of the syntactic constructs in modern programming languages
- by hand implemented parsers use LL grammars
- tool generated parsers use the larger class of LR grammars

Parser

- Input
  - Stream of tokens
- Output
  - Some representation of the parse tree
- Tasks
  - Collecting information about various tokens into the symbol table
  - Performing type, domain checking,…
  - Generating intermediate code

Representative Grammars

- constructs
  - starting with keywords
    - while, int
  - are easy to parse
  - keywords guide the choice of the grammar production that must be applied to match the input
- expressions
  - are more challenging
    - because of operators which have
      - association rules
      - precedence

Representative Grammars

- E – expressions of terms separated by + signs
- T – terms consisting of factors separated by * signs
- F – factors which can be parenthesized expressions or identifiers

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

- LR grammar
  - suitable for bottom-up parsing
  - can be adapted to handle additional
    - operators
    - levels of precedence
  - can not be used for top-down parsing because is left recursive !!!

Representative Grammars

- Non-left-recursive variant
- Suitable for top-down parsing

```
E  -> T E′
E′ -> + T E′ | ε
T  -> F T′
T′ -> * F T′ | ε
F  -> ( E ) | id
```

Ambiguous Grammar

```
E -> E + E | E * E | (E) | id
```

- operators + and * are treated alike
- the grammar permits more than one parse for the expression
  - a+b*c

Syntax Error Handling

- nature of syntactic errors
- general strategies of error recovery
- parsing only correct code
  - design and implementation – greatly simplified
- assisting the programmer to locate and track down errors
- few languages with error handling in design
- error induced behavior is not present in language specification
- error handling is left to compiler designer
- planning it from the beginning
  - simplifies the structure of the compiler
  - improves the handling of errors

Common Programming Error Levels

- lexical errors
  - misspelling of identifiers, keywords or operators
    - e.g. missing quotes around text intended as string
- syntactic errors
  - misplaced, extra, missing tokens:
    - semicolons, braces
    - e.g. case without enclosing switch (Java)
- semantic errors
  - type mismatches between operators and operands
  - e.g. return statement for a Java method with result type void
- logical errors
  - incorrect reasoning on the part of the programmer
  - e.g. using in C the assignment = operator instead of the comparison == operator

Viable Prefix Property

- precision of parsing methods allows efficient syntactic error detection
- LL and LR parsing methods detect an error as soon as possible
- Viable Prefix Property of parsing methods is to issue an error
  - when the token stream can not be parsed further according to the grammar for the language
  - when they see a prefix at the input that can not be completed to for a string in the language

Error Handler Goals

- report the presence of errors clearly and accurately
- recover from errors quickly in order to detect subsequent errors
- add minimal overhead to the processing of correct programs

However

- accurate detection of semantic and logical errors at compile time is in general a difficult task !!!
- common errors are simple ones
- straightforward error-handling mechanisms suffices

Error Reporting

- the place in the source program where the error is detected
- the actual error is probably around the 2-3 neighbor tokens
- common strategy
  - print the offending line
  - point to the position where the error was detected

Error Recovery Strategies

- when error detected -> the parser should recover
- no strategy universally acceptable
- few methods with broad applicability
  - to quit with an informative error message at first error
    - additional errors are uncovered if the parser restores itself to a state where processing of the input can continue with reasonable hopes that further processing is meaningful
    - if error number increases then the compiler
      - should stop after a given error number limit
      - will avoid issuing an avalanche of “spurious” messages

Panic Mode Recovery

- on discovering an error
- the parser discards input symbols
- one at a time
- until is found one of a designated set of synchronizing tokens
  - delimiters ; or }
  - have a clear and unambiguous role
  - must be selected by the compiler designer
- skips considerable amount of input
- no checking for additional errors
- simple
- guaranteed not to go on an infinite loop

Phrase-Level Recovery

- on discovering an error
- to perform local correction on the remaining input
- to replace the remaining input by some string that allows the parser to continue
- examples
  - to replace a comma by a semicolon
  - to delete an extraneous semicolon
  - to insert a missing semicolon
- the choice of local correction is left to the compiler designer
- to choose replacements that do not lead to infinite loops
- difficulty in coping with situations in which the actual error has occurred before the detection point

Error Productions

- to equip the grammar with productions which generate erroneous constructs
- such a parser detects the anticipated errors when an error production is used during parsing
- the parser can generate appropriate error diagnostics

Global Correction

- ideally a compiler would make as few changes as possible in processing an incorrect string
- algorithms for choosing the minimal sequence of changes to obtain globally a least-cost correction
  - given an incorrect input x
  - to find a parse tree for a related string y
  - such as the number of insertions, deletions and changes of tokens required to transform x into y is as small as possible
- too costly to implement in time and space
- only of theoretical interest
- a closest correct program may not have the same semantics as the intended erroneous one
- the least cost correction is used for
  - evaluating error recovery techniques
  - finding optimal replacement strings for phrase-level recovery

Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, 2007