CSE302: Compiler Design

Instructor: Dr. Liang Cheng
Department of Computer Science and Engineering
P.C. Rossin College of Engineering & Applied Science
Lehigh University

January 18, 2007
Today’s Outline

- Recap
- A simple syntax-directed translator (Chapter 2)
  - Introduction (Section 2.1)
  - Syntax definition (Section 2.2)
  - Parsing (Section 2.4)
- Summary and homework
Six Compilation Phases

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Code optimization
- Code generation
Outline

- Recap
- A simple syntax-directed translator (Chapter 2)
  - Introduction (Section 2.1)
  - Syntax definition (Section 2.2)
  - Parsing (Section 2.4)
- Summary and homework
Contents in Chapter 2

- Illustrate some compiler techniques via developing a simple language translator coded in Java (Appendix A)
  - Source language
  - Target language: three-address code
- The front-end of a compiler
Six Compilation Phases

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Code optimization
- Code generation
A Simplified Model of A Compiler

Front End

- do i=i+1; while (a[i]>v);

1: i = i + 1

  t1 = a[i]

  if t1>v goto 1
Outline

- Recap
- A simple syntax-directed translator (Chapter 2)
  - Introduction (Section 2.1)
  - Syntax definition (Section 2.2)
  - Parsing (Section 2.4)
- Summary and homework
A Formal Method of Describing Syntax

- Backus-Naur Form (1959)
  - For Algol 58 (John Backus, the Peter Naur)
  - BNF is equivalent to context-free grammars
    - Context-free grammars were developed by Noam Chomsky in mid-1950s to define a class of languages called context-free languages
      - A BNF grammar defines a language
        - Recognizer vs. generators
      - A BNF grammar or description is a production-rule collection
BNF Rules

- A BNF rule defines an abstraction for syntactic structure
  \[
  \texttt{<while_stmt>} \rightarrow \texttt{while( <logic_expr> ) <stmt>}
  \]

- Abstractions are used to represent classes of syntactic structures: they act like syntactic variables (also called nonterminal symbols)

- Terminal symbols: lexemes and tokens
A rule has a **left-hand side** (LHS) and a **right-hand side** (RHS), and consists of terminal and/or nonterminal symbols.

An abstraction (or nonterminal symbol) can have more than one RHS:

\[
<\text{stmt}> \rightarrow <\text{single\_stmt}>
\quad | \quad \text{begin } <\text{stmt\_list}> \text{ end}
\]
BNF Functionality

- Describe grammars and derivations
  - Describe lists of similar constructs
  - Parse trees
- Powerful enough to avoid grammar ambiguity
  - Operator precedence and associativity
Describing Lists

- Syntactic lists are described using recursion
  \[ \text{<ident\_list> } \rightarrow \text{ident} \]
  \[ \quad \text{| ident, <ident\_list>} \]
- A rule is \text{recursive} if its LHS appears in its RHS

BNF Functionality
- Describing Lists
- Grammar & Derivation
- Parse Trees
- Avoiding Ambiguity
A BNF Grammar

- A BNF grammar defines a language
- Sentences of a language are generated through applications of rules, beginning with a start symbol
- A grammar for a small language

\[
\begin{align*}
\text{<program>} & \rightarrow \text{<stmts>} \\
\text{<stmts>} & \rightarrow \text{<stmt>} \mid \text{<stmt>} \, ; \, \text{<stmts>} \\
\text{<stmt>} & \rightarrow \text{<var>} \, = \, \text{<expr>} \\
\text{<var>} & \rightarrow \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \\
\text{<expr>} & \rightarrow \text{<term>} \, + \, \text{<term>} \mid \text{<term>} \, - \, \text{<term>} \\
\text{<term>} & \rightarrow \text{<var>} \mid \text{const}
\end{align*}
\]
A sentence generation is called a derivation.

An example derivation:

\[
\text{<program> } \Rightarrow \text{ <stmts>}
\]
\[
\Rightarrow \text{ <stmt>}
\]
\[
\Rightarrow \text{ <var> = <expr>}
\]
\[
\Rightarrow a = \text{ <expr>}
\]
\[
\Rightarrow a = \text{ <term> + <term>}
\]
\[
\Rightarrow a = \text{ <var> + <term>}
\]
\[
\Rightarrow a = b + \text{ <term>}
\]
\[
\Rightarrow a = b + \text{ const}
\]
Terms in Derivation

- Every string of symbols in the derivation is a **sentential form**
- A **sentence** is a sentential form that has only terminal symbols
- A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

<program> => <stmts>
  => <stmt> => <var> = <expr>
  => a = <expr>
  => a = <term> + <term>
  => a = <var> + <term>
  => a = b + <term>
  => a = b + const
A hierarchical representation of a derivation

\[
\begin{align*}
\text{<program>} & \Rightarrow \text{<stmts>} \\
& \Rightarrow \text{<stmt>} \\
& \Rightarrow \text{<var>} = \text{<expr>} \\
& \Rightarrow a = \text{<expr>} \\
& \Rightarrow a = \text{<term>} + \text{<term>} \\
& \Rightarrow a = \text{<var>} + \text{<term>} \\
& \Rightarrow a = b + \text{<term>} \\
& \Rightarrow a = b + \text{const}
\end{align*}
\]
Lists, Grammar, and Parse Trees

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

BNF Functionality
- Describing Lists
- Grammar & Derivation
- Parse Trees
- Avoiding Ambiguity
Avoiding Ambiguity

- A grammar is **ambiguous** iff it generates a sentential form that has two or more distinct parse trees
  - Operator precedence ambiguity
  - Operator associativity ambiguity
  - ...

BNF Functionality
- Describing Lists
- Grammar & Derivation
- Parse Trees
- Avoiding Ambiguity
An Ambiguous Expression Grammar

\[
<\text{expr}> \rightarrow <\text{expr}> \ <\text{op}> \ <\text{expr}> \ | \ \text{const} \\
<\text{op}> \rightarrow / \ | \ -
\]

BNF Functionality
- Describing Lists
- Grammar & Derivation
- Parse Trees
- Avoiding Ambiguity
An Unambiguous Expression Grammar

If we use an additional nonterminal to indicate precedence levels of the operators, we will not have such ambiguity.

\[
<\text{expr}> \rightarrow <\text{expr}> - <\text{term}> \mid <\text{term}>
\]

\[
<\text{term}> \rightarrow <\text{term}> / \text{const} \mid \text{const}
\]
Another Ambiguity

- Operator associativity ambiguity

\[ <\text{expr}> \rightarrow <\text{expr}> - <\text{expr}> \mid \text{const} \] (ambiguous)
Avoid Ambiguity

- Operator associativity can also be indicated by a BNF description

<expr> -> <expr> - <expr> | const (ambiguous)
<expr> -> <expr> - const | const (unambiguous)
Outline

- Recap
- A simple syntax-directed translator (Chapter 2)
  - Introduction (Section 2.1)
  - Syntax definition (Section 2.2)
  - Parsing (Section 2.4.1-2.4.4)
- Summary and homework
Parsing is the process of determining how a string of terminals can be generated by a grammar.

- Parse tree generation
- Parsers make a single left-to-right scan over the input tokens, look ahead of one terminal at a time, and construct the parse tree.

- Top-down parsing vs. bottom-up parsing
Predictive Parsing (1)

- A Recursive-Descent Parser
  - Directly following the BNF grammar
  - \(<expr> \rightarrow <term> + <term> \mid <term> - <term> \mid <term>\)
  - \(<term> \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9\)

- Pseudo code
  ```
  void expr() {
    term();
    if( token==plus_op or token==minus_op) {
      match(token);
      term();
    } else error();
  }

  void term() {
    match(int_literal);
  }

  void match(expectedToken) {
    if(token==expectedToken) {
      getNextToken();
    } else error();
  }
  ```

- Demo
  - A homework question
Predictive Parsing (2)

- A BNF grammar
  - stmt → expr; | for(optexpr; optexpr; optexpr) stmt | other
  - optexpr → ε | expr
- Pseudo code
  void stmt() {
    switch ( lookahead ) {
    case expr:
      match(expr); match(';'); break;
    case for:
      match(for); match('('); optexpr(); match('); optexpr(); match('); optexpr(); match('); optexpr(); match('); stmt(); break;
    case other:
      match(other); break;
    default: report("Syntax error");
    }
  }
  void optexpr() { if ( lookahead == expr ) match(expr); }
  void match(terminal t) {
    if ( lookahead == t ) lookahead = nextToken; else report("Syntax error"); }
Predictive Parsing (3)

- A BNF grammar
  - stmt → expr; | if(expr) stmt | for(optexpr; optexpr; optexpr) stmt | other
  - optexpr → ε | expr

- A statement
  - for ( ; expr ; expr ) other
Outline

■ Recap
■ A simple syntax-directed translator (Chapter 2)
■ Summary and homework
You should now be able to …

- Evaluate whether a grammar is ambiguous;
- Use BNF to specify operator precedence;
- Write a simple recursive-descent parser.
Grammar Ambiguity

- A grammar is ambiguous iff it generates a sentential form that has two or more distinct parse trees.
- Every derivation with an unambiguous grammar has a unique parse tree, although that tree can be represented by different derivations.
You should now be able to …

- Evaluate whether a grammar is ambiguous;
- Use BNF to specify operator precedence and associativity;
- Write a simple recursive-descent parser.
Operator Precedence by BNF

- Guidelines
  - Use terminal operator lexemes
  - Use additional nonterminals as operands

```
<expr>  →  <expr> <op> <expr>  |  const
<expr>  →  <expr> - <term>  |  <term>
<op>    →  /  |  -
```

An ambiguous grammar

```
<term>  →  <term> / const  |  const
```

An unambiguous grammar
Operator Associativity by BNF

- Operator associativity can also be indicated by a BNF description
  \[
  \text{<expr> -> <expr> + <expr> | const (ambiguous)}
  \]
  \[
  \text{<expr> -> <expr> + const | const (unambiguous)}
  \]

- Guidelines
  - A rule's LHS appears at the beginning of its RHS, which is a left-recursive rule. Then it indicates left associativity.
  - Right associativity (LHS is the operand at the right-hand side, not at the left)
    - Exponentiation operator
      \[
      \text{<expr> -> <base> ** <expr> | <base>}
      \]
      \[
      \text{<base> -> id | const}
      \]
You should now be able to …

- Evaluate whether a grammar is ambiguous;
- Use BNF to specify operator precedence and associativity;
- Write a simple recursive-descent parser.
1.1. (10 points) Just as shown in Figure 1.7 in the textbook (page 7), detail the six steps of compilation for the statement: A=B*5+C/3 where A, B, C are variables of floating-point type.

1.2. (10 points) Consider the context-free grammar:

   \[ S \rightarrow + S S \mid - S S \mid a \]

   a) Show how the string +a-aa can be generated by this grammar.

   b) Construct a parse tree for this same string.

1.3. (10 points) Implement an executable and correct recursive-descent parser based on the pseudo code for the BNF grammar illustrated in this lecture:

   \[ <expr> \rightarrow <term> + <term> \mid <term> - <term> \mid <term> \]

   \[ <term> \rightarrow 0 \mid 1 \mid 2 \mid ... \mid 9 \]
Reading Assignment

  - The Grammar of the Java Programming Language
  - http://java.sun.com/docs/books/jls/