Hochschule für Technik Stuttgart

Concepts of Programming Languages

8^{th} Week

Parsing Strategies

Prof. Dr. Peter Heusch

Recap

- Any compilation process begins with the same two tasks:
  - Lexical analysis constructs lexemes (token) from single characters
  - Syntax analysis constructs a syntax tree from the tokens
- Historically the problems were addressed by different tools (lex & yacc, flex & bison)
- Modern compiler compilers combine the descriptions for both steps into a single file

Goals for Today

- Goals of parsing in general: Parse Trees
- Parsing strategies: Top-down and bottom-up

Parsing

- In some sense parsing must revert the work of the programmer:
  - Derive a parse tree for syntactically correct programs
  - Produce error messages for syntactically incorrect programs while still building up as much of the parse tree as possible
- The parse tree is the central data structure for all subsequent compilation processes
- Parsing may remove syntactic sugar and useless symbols

Syntactic Sugar

- Syntactic Sugar are constructs that are only needed for parsing, not for the execution:
    - Parentheses
    - Colons, Semicolons
- Perl is especially known for syntactic diabetes, all of the following statements produce the same code:

```
if ($a<10) { print "Hello"; }
unless (!($a<10)) { print "Hello"; }
print "Hello" if ($a<10);
print "Hello" unless (!$a<10);
!($a<10) || print „Hello“;
```

Goals for Today

- Goals of parsing in general
- Parsing strategies: Top-down and bottom-up

Parsing Techniques

- Top-Down: Start at the root
  - Select applicable rule based on lookup
  - Expand
  - Creates left-most derivation
- Bottom-Up: Start at the bottom
  - Consume token until the right side of a rule is found
  - Replace set of tokens by left side of the rule
  - Creates reverse of right-most derivation

- Input is always consumed left-to-right
- Algorithms are classed as LL or LR

Top-Down Parsers

- Given a sentential form xAα choose the correct rule for the expansion of A based on the first token produced by A
- Two approaches are possible:
  - Recursive descent (handwritten parsers)
  - Parsing tables (machine generated parsers)
- Not every context free grammar is parseable by a top-down parser, but every deterministic context free language has a grammar that is parsable by a top-down parser

Bottom-Up Parsers

- Given a sentential form α determine the rule to reduce some prefix of α such that the previous sentential form (previous in the generation tree) is produced, until only the start symbol is left
- Cover a larger class of grammars than Top Down parsers
- Bottom-Up parsers are generally too complex to produce manually
- Normally tools are used e.g:
    - Eli
    - Bison, Yacc

Goals for Today

- Goals of parsing in general
- Parsing strategies: Top-down and bottom-up

Top-Down: Recursive Descent Parsing

- Recursive Descent parsing is especially suited for EBNF grammars, because they need few nonterminals
- Every nonterminal corresponds to a method in the parser, the method must be able to parse text that is produced from this nonterminal
- Options and repetitions can be converted to if- and while-statements

Recursive Descent Sample

<expr> --> <term> [(+ | -) <term>]

is parsed by method

```
void expr() {
  term();
  Token t = peekToken();
  if (t.equals("+") || t.equals("-")) {
    consumeToken();
    term();
  }
}
```

General Rules

What if there are several?

- For every nonterminal, call the corresponding method
- If a terminal is needed, read the next token and match it, signal mismatch as error
- If an expansion is optional (or repeated) peek the next token and check whether this token can occur as first token of the expansion:
  - If yes, take the expansion
  - If no, leave it

This is cheating… but we call it Lookahead.

Multiple Right Hand Sides

- If a nonterminal has multiple right hand sides, select the one that can produce the first token of the input
- If the first token can be produced by multiple right hand sides, this is an error in the parser
- If the first token can be produced by no right hand side, this is an error in the input

Making Grammars LL-Parseable

- If grammars contain left recursive productions, they are not LL-parseable
  - Left recursion can be direct (in the same rule) or spread across several rules
  - Eliminate by reformulating as tail recursion (this may introduce ε productions)
- If two right hand sides can both produce the same first token, the grammar is not LL-parseable
  - Ambiguity can be removed by factoring out the identical parts

Determine Parseability: FIRST Sets

For each grammar symbol, use the grammar rules to determine which initial terminals it can produce (after one or several derivation steps)

- Terminals produce themselves: First(t) = {t}
- Non-Terminals: Add First(X) to First(N)
  for rule N -> X Y Z (expansion of X will eventually produce the first terminal)
- If the grammar allows ε productions (N -> ε):
  - Add ε to First(N) if N-> ε exists
  - First (N) u First(X) u First(Y) u First (Z) u (ε)
    if X, Y and Z all contain an ε production (else, stop at the first non-terminal that doesn't have an ε production and don't include (ε))

Example: FIRST Sets

Grammar:
```
Expr -> Term + Term
Term -> ( Expr ) | id
```

- FIRST sets for the LHS symbols
  - FIRST(`Term`) = {`(`, `id`}
  - FIRST(`Expr`) = FIRST(`Term`) (b/c `Term` is the left-most non-terminal and will produce the first terminal)

- FIRST sets for the rules
  - FIRST(`Term -> ( Expr )`) = {`(`}
  - FIRST(`Term -> id`) = {`id`}
  - FIRST(`Expr -> Term + Term`)
    = FIRST(`Term->(Expr)`) ∪ FIRST(`Term->id`)
    = FIRST(`Term`)

Lookahead

- Given the FIRST sets, we find the applicable rule by looking at the next unprocessed token in the input: Lookahead
- If the first token can be produced by multiple right hand sides, this is an error in the parser
  - **FIRST/FIRST conflict** (First sets for several RHS produce the same terminal: Which rule to use?)
  - FIRST/FIRST conflicts are also caused by left recursion (given `E -> E + T | T` : FIRST(E+T) == FIRST(T))
- Sometimes it might be necessary to check more than one token lookahead (e.g. Java‘s labeled loops)
- This changes the grammar type from LL(1) to LL(n) – but: more lookahead means greater parsing effort

Goals for Today

- Goals of parsing in general
- Parsing strategies: Top-down and bottom-up

Bottom-Up Parsing

- Reduce the current data on the stack and replace it by a right hand side, based on the first symbol of the remaining input:

```
<E> → <T> {(+ | -) <T>}
<T> → <F> {(* | /) <F>}
<F> → id | ( <E> )
```

```
| id * id + id
id | * id + id
F | * id + id
F * | id + id
F * id | + id
F * F | + id
T | + id
T + | id
T + id |
T + F
T + T
E
```

Advantages of Bottom-Up Parsers

- They will work for nearly all grammars that describe programming languages.
- They work on a larger class of grammars than top-down algorithms, but are as efficient as top-down parsers.
    - The LR class of grammars is a superset of the LL class of grammars.
- They can detect syntax errors as soon as it is possible.

Bottom-Up Parser Construction

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
- There are only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Structure of Bottom-Up Parsers

- Bottom-Up parsers use two tables to work:
  - The action table defines what to do in a certain situation based on the current stack state and the next token of the input
  - The goto table defines which state is to be entered in case a reduction is performed
- Details: Next week!

Parsing Results

- Usually the result of parsing is a parse tree
  - Compilers translate this parse tree to (byte)code
  - Interpreters immediately execute the commands given by the parse tree
- The parse tree is constructed during the rule expansion process
- In most cases another side effect of the parsing process is the construction of a symbol table
- Symbol tables are necessary to check the context dependent syntax

Parse Trees

```
               E
     T        +      T
  F  *  F            F
  id   id            id
```

Top Down: By rule expansion

Bottom Up: By reduction

```
| id * id + id
id | * id + id
F | * id + id
F * | id + id
F * id | + id
F * F | + id
T | + id
T + | id
T + id |
T + F
T + T
E
```

Goals for Today

- Goals of parsing in general
- Parsing strategies: Top-down and bottom-up