SWEN430 - Compiler Engineering

Lecture 3 - Compiler Architecture and Parsing

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Compiler Architecture

- The usual structure of a compiler is:
  - Front end: Read program, check for errors, create intermediate form (e.g. AST)
  - Back end: Translate AST to machine/vm code and optimise;
    or just execute (interpreter);
    or compile while executing (JIT)
- This gives a “2^+ pass” compiler — front and back ends may each make several passes over the program (e.g. see JKit slide lecture 1).
- Intermediate form(s) may be written to a file or passed as a data structure, or front and back ends may run as coroutines.
- Compiled form usually written to a file which is then loaded, along with run-time system. May just write to memory and execute.
- A “one pass compiler” does everything in one pass!
  (Ex: What problems does that pose?)

Front end – more closely

The front end has several steps, mirroring the language definition:

- Scanner (Lexer, Tokeniser):
  - Read input as a sequence characters/lines
  - Output a sequence of tokens/lexemes/symbols
  - Report lexical errors (illegal symbols)
  - Strip out comments and white space
- Parser:
  - Read input as sequence of tokens
  - Build a parse tree (perhaps implicitly) with the input as its fringe
  - Report syntax errors (illegal combinations of symbols)
  - Build symbol table containing identifiers declared in the program
- Context checking, type checking and static analysis:
  - Check for context conditions such as variables used and methods called must be declared
  - Check for type correctness: methods and operators must be applied to arguments of the correct type
  - Check for properties such as definite assignment and dead code

Scanner design

- Set of tokens designed to form a (deterministic) *regular language*.
- The scanner is based on a *finite state acceptor*.
- Read one character at a time, and decide what to do next based on the next character, i.e. one character look ahead (or maybe 2 or 3) — localises reading/counting lines and checking for end of file.
- May be table driven or generated from a regular grammar, or hand coded — in which case the FSA is implicit in the program code.
- Often the most time consuming part of a compiler, so must be fast.
- May run scanner over the entire input and create a token string which is passed to the parser;
  or call scanner as a `nextToken` method from the parser;
  or run scanner and parser in parallel, e.g. as coroutines.
- Ex: Look at the code for the While lexer.

Scanner design

- Basic outline:

```
while there is more input do
  ch <- getNextChar
  case what kind of token can ch start of
    number: scanNumericCont;
    string: scanStringConst;
    ident: scanIdentifier;  // includes keywords
    operator: scanOperator;
    whiteSpace: scanWhiteSpace;
    otherwise: error
```

- Each method scans one kind of token and advances over all of the characters in that token.
- Sometimes look for white space before each token.
- Adding an *eof* char avoids checking everywhere for end of input:

```
ch <- getNextChat
while ch neq eof do
  case what kind of token can ch start of
```

Parser design

- The set of syntactically valid programs is designed to be a (deterministically parsable) *context-free language*, defined by a form of *context-free grammar*.
- The parser is based on a *push-down automaton*.
- Read one token at a time and decide what to do next based on that token, i.e. one symbol look ahead (or sometimes ...).
- May build a parse tree from root down to leaves (top-down, LL(1)); or from leaves up to root (bottom-up, LR(1)).
- May be table driven or generated from a context-free grammar, or hand coded — in which case the PDA is implicit in the program code.
- Lots of tools for generating scanners and parsers (yacc and lex, bison, antlr, ...).

Recursive Descent Parsing

- We'll use a form of top-down hand coded parser called *recursive descent* or *predictive parsing*.
- Simple but powerful deterministic parsing method — uses one (or more) lookahead symbols to determine what to look for next.
- Grammars and languages for which recursive descent works are called LL(k), where k is the number of lookahead symbols needed.
- Most programming languages structures turn out to be LL(1).
- For each nonterminal *N* in the grammar, define a method *parseN* to recognise an instance of *N* as a prefix of the input, and build an AST for it.
- Logic of the parse methods reflects the structure of the grammar. Can be coded directly from the grammar once it is in LL(1) form.
- Easy to extend to do error analysis/reporting/recovery, semantic checking and building AST.

Recursive Descent Parsing

- A *Context-Free Grammar* (CFG) is a set of rules of the form

$$N \longrightarrow A_1 \mid \ldots \mid A_n$$

where, N is a *non-terminal*, and $A_1, \ldots, A_n$ are strings of *terminals* and/or non-terminals.

- Terminals are symbols that actually appear in a program
Non-terminals are names of structural components of a program

- The parser method for such a rule (ignoring tree building) is:

```
parseN()
  if nextSym can start A1 then recognise A1
  ...
  else if nextSym can start An then recognise An
  else error(nextSym can't start N)
```

Recursive Descent Parsing

- If A_i is $X_1 \ldots X_m$
- Then `recognise Ai` is:

  ```
  recognise X1;
  ...
  recognise Xm;
  ```

- Where `recognise Xj` calls `parseXj` if X_j is a nonterminal, and looks for terminal X_j otherwise.
- This needs some more plumbing to handle errors and build AST.
- In practice, we can simplify the parser a bit.

Recursive Descent Parsing

- Example: $$\begin{array}{lcl} E & \longrightarrow & N \mid V \mid \texttt{(}E\texttt{+}E\texttt{)} \\ N & \longrightarrow & [\texttt{0-9}]^{+} \\ V & \longrightarrow & [\texttt{a-zA-Z\_}]^{+}[\texttt{a-zA-Z0-9\_}]^{*} \end{array}$$

  Note that this uses Unix regular expression notation to define sets of terminals.

- Parser is:

```
parse E
  if nextSym is N then advance
  else if nextSym is V then advance
  else if nextSym is '(' then
         parse('(');
         parseE;
         parse('+');
         parseE;
         parse(')');
  else error(nextSym can't start E)
```

When does it work? LL(1) conditions

- Must be able to decide what to do on basis of next input symbol.
- So, given $N \longrightarrow A|B$, no symbol that can start an A can also start a B.
- Define *first* sets:

  Let γ be a sequence of terminal and non-terminal symbols. Then *first*(γ) is the set of all terminal symbols which begin a string derived from γ.
- Code `nextSym can start N` as `nextSym in first(N)`.
- We can now state the **Choice Condition**:

  For any rule $N \longrightarrow \alpha|\beta$, it must hold that *first*$(\alpha) \cap$ *first*$(\beta) = \emptyset$.

LL(1) — first() sets

- Example: $$\begin{array}{llr} S \longrightarrow T\ \mathtt{a} \mid U\ \mathtt{b} & & (1,2) \\ T \longrightarrow \mathtt{d}\ T \mid \mathtt{e} & & (3,4) \\ U \longrightarrow \mathtt{c}\ U \mid \mathtt{f} & & (5,6) \end{array}$$

- $\mathit{first}(T) = \{\mathtt{d}, \mathtt{e}\}$
  $\mathit{first}(U) = \{\mathtt{c}, \mathtt{f}\}$

- Does this satisfy the choice condition?

LL(1) — first() sets

- What are the first() sets for these grammars?

1. $S \longrightarrow \texttt{a}\ S \mid \texttt{a}$

2. $T \longrightarrow \texttt{c}\ T \mid \texttt{d}$
   $U \longrightarrow \texttt{c}\ U \mid \texttt{e}$

3. $S \longrightarrow T\ \texttt{a} \mid U\ \texttt{b}$
   $S \longrightarrow T\ \texttt{a} \mid U\ \texttt{c}$
   $T \longrightarrow \texttt{d}\ T \mid \texttt{e}$
   $U \longrightarrow \texttt{c}\ U \mid \epsilon$

- Do they satisfy the choice condition?

Questions to ponder

- How do we extend the Choice Condition to $N \longrightarrow \alpha_1|...|\alpha_n$?
- Are there any cases the Choice Condition doesn't handle?

  i.e. anywhere else the parser has to make a choice?
- How can we extend this to handle extended BNF grammars, where we can write:
  - $[\alpha]$ to mean that α is optional.
  - α^* to mean that α is repeated 0 or more times.
  - α^+ to mean that α is repeated 1 or more times.