CS132: Compiler Construction

Professor Palsberg

Thilan Tran

Fall 2020

Contents

CS132: Compiler Construction **2**

Introduction **2**
Compiler Overview . . . 2

Lexical Analysis **5**

Parsing **6**
Top-Down Parsing . . . 7
Grammar Hacking . . . 8
Achieving LL(1) Parsing . . . 10
Predictive Parsing . . . 10
Handling Epsilon . . . 11
Formal Definition . . . 12
JavaCC . . . 12

Handling Syntax Trees **14**
Visitor Pattern . . . 14
Java Tree Builder . . . 16

Appendix **19**
LL(1) Practice Examples . . . 19

CS132: Compiler Construction

Introduction

- a **compiler** is a program that *translates* an executable program in one language to an executable program in another language
- an **interpreter** is a program that *reads* an executable program and produces the results of running that program
  - usually involves executing the source program in some fashion, ie. *portions* at a time
- compiler construction is a *microcosm* of CS fields:
  - AI and algorithms
  - theory
  - systems
  - architecture
- in addition, the field is not a solved problem:
  - changes in architecture lead to changes in compilers
    - new concerns, re-engineering, etc.
  - compiler changes then prompt new architecture changes, eg. new languages and features
- some compiler motivations:
  1. correct output
  2. fast output
  3. fast translation (proportional to program size)
  4. support separate compilation
  5. diagnostics for errors
  6. works well with debugger
  7. cross language calls
  8. optimization
- for new languages, how are compilers written for them?
  - eg. early compilers for Java were written in C
  - eg. for early, low level languages like C, **bootstrapping** is done:
    - a little subset of C is written and compiled in machine code
    - then a larger subset of C is compiled using that smaller subset, etc.

Compiler Overview

- abstract compiler system overview:
  - *input*: source code
  - *output*: machine code or errors
  - recognizes illegal programs, and outputs associated errors
- *two-pass* compiler overview:
  - source code eg. Java compiles through a frontend to an **intermediate representation (IR)** like Sparrow
    - the **frontend** part of the compiler maps legal code into IR:
      - language *dependent*, but machine *independent*
      - allows for swappable front ends for different source languages
  - IR then compiles through a backend to machine code
    - the **backend** part maps IR onto target machine:
      - language *independent*, but machine / architecture *dependent*
- frontend overview:
  - *input*: source code
  - *output*: IR
  - *responsibilities*:
    - recognize legality syntactically
    - produce meaningful error messages
    - shape the code for the backend
  1. the scanner produces a stream of tokens from source code:
  - ie. *lexing* source file into tokens
  2. the parser produces the IR:
  - recognizes context free grammars, while guiding context sensitive analysis
  - both steps can be automated to some degree
- backend overview:
  - *input*: IR
  - *output*: target machine code
  - *responsibilities*:
    - translate to machine code
    - **instruction selection**:
      - choose specific instructions for each IR operation
      - produce compact, fast code
    - **register allocation**:
      - decide what to keep in registers at each points
      - can move loads and stores
      - optimal allocation is difficult
  - more difficult to automate
- specific frontends or backends can be swapped
  - eg. use special backend that targets ARM instead of RISC, etc.
- middleend overview:
  - *responsibilities*:

    * optimize code and perform code improvement by analyzing and changing IR
    * must preserve values while reducing runtime
  - optimizations are usually designed as a set of iterative passes through the compiler
  - eg. eliminating redundant stores or dead code, storing common subexpressions, etc.
  - eg. GCC has 150 optimizations built in

Lexical Analysis

- the role of the **scanner** is to map characters into **tokens**, the basic unit of syntax:
  - while eliminating whitespace, comments, etc.
  - the character string value for a token is a **lexeme**
  - eg. `x = x + y;` becomes `<id,x> = <id,x> + <id,y> ;`
- a scanner must recognize language syntax
  - how to define what the syntax for integers, decimals, etc.

1. use regular expressions to specify syntax patterns:
   - eg. the syntax pattern for an integer may be `<integer> ::= (+ | -) <digit>*`
2. regular expressions can then be constructed into a **deterministic finite automaton (DFA)**:
   - a series of states and transitions for accepting or rejecting characters
   - this step also handles state minimization
3. the DFA can be easily converted into code using a while loop and states:
   - by using a table that categorizes characters into their language specific identifier types or classes, this code can be language *independent*
     - as long as the underlying DFA is the same
   - a linear operation, considers each character once

- this process can be automated using **scanner generators**:
  - emit scanner code that may be direct code, or table driven
  - eg. `lex` is a UNIX scanner generator that emits C code

Parsing

- the role of the **parser** is to recognize whether a stream of tokens forms a program defined by some grammar:
  - performs context-free syntax analysis
  - usually constructs an IR
  - produces meaningful error messages
  - generally want to achieve *linear* time when parsing:
    - need to impose some restrictions to achieve this, eg. the **LL** restriction
- context-free syntax is defined by a **context-free grammar (CFG)**:
  - formally, a 4-tuple $G = (V_t, V_n, S, P)$ where:
    - V_t is the set of **terminal** symbols, ie. tokens returned by the scanner
    - V_n is the set of **nonterminal** symbols, ie. syntactic variables that denote substrings in the language
    - S is a distinguished nonterminal representing the **start symbol** or goal
    - P is a finite set of **productions** specifying how terminals and nonterminals can be combined
      - each production has a single nonterminal on the LHS
    - the **vocabulary** of a grammar is $V = V_t \cup V_n$
    - the motivation for using CFGs instead of simple REs for grammars is that REs are not powerful enough:
      - REs are used to classify tokens such as identifiers, numbers, keywords
      - while grammars are useful for counting brackets, or imparting structure eg. expressions
      - factoring out lexical analysis simplifies the CFG dramatically
  - general CFG notation:
    - $a, b, c, ... \in V_t$
    - $A, B, C, ... \in V_n$
    - $U, V, W, ... \in V$
    - $\alpha, \beta, \gamma, ... \in V^*$, where V^* is a sequence of symbols
    - $u, v, w, ... \in V_t^*$, where V_t^* is a sequence of terminals
    - $A \to \gamma$ is a production
    - $\Rightarrow, \Rightarrow^*, \Rightarrow^+$ represent derivations of 1, ≥ 0, and ≥ 1 steps
    - if $S \Rightarrow^* \beta$ then β is a **sentential form** of G
    - if $L(G) = \{\beta \in V^* | S \Rightarrow^* \beta\} \cap V_t^*$, then $L(G)$ is a **sentence** of G, ie. a derivation with all nonterminals
- grammars are often written in **Backus-Naur form (BNF)**:
  - non-terminals are represented with angle brackets

    - terminals are represented in monospace font or underlined
    - productions follow the form `<nont> ::= ...expr...`
- the productions of a CFG can be viewed as rewriting rules:
    - by repeatedly rewriting rules by replacing nonterminals (starting from goal symbol), we can **derive** a sentence of a programming language
        * **leftmost derivation** occurs when the *leftmost* nonterminal is replaced at each step
        * **rightmost derivation** occurs when the *rightmost* nonterminal is replaced at each step
    - this sequence of rewrites is a **derivation** or **parse**
    - *discovering* a derivation (ie. going backwards) is called **parsing**
- can also visualize the derivation process as construction a tree:
    - the goal symbol is the root of tree
    - the children of a node represents replacing a nonterminal with the RHS of its production
    - note that the ordering of the tree dictates how the program would be *executed*
        * can multiple syntaxes lead to different parse trees depending on the CFG used?
    - parsing can be done **top-down**, from the root of the deriviation tree:
        * picks a production to try and match input using backtracking
        * some grammars are backtrack-free, ie. *predictive*
    - parsing can also be done **bottom-up**:
        * start in a state valid for legal first tokens, ie. start at the leaves and fill in
        * as input is consumed, change state to encode popssibilities, ie. recognize valid prefixes
        * use a stack to store state and sentential forms

Top-Down Parsing

- try and find a linear parsing algorithm using top-down parsing
- general top-down parsing approach:
    1. select a production corresponding to the current node, and construct the appropriate children
        - want to select the right production, somehow guided by input string
    2. when a terminal is added to the *fringe* that doesn't match the input string, backtrack
    3. find the next nonterminal to expand
- problems that will make the algorithm run worse than linear:

    - too much backtracking
    - if the parser makes the wrong choices, expansion doesn't even terminate
        * ie. top-down parsers *cannot* handle left-recursion
- top-down parsers may backtrack when they select the wrong production:
    - do we need arbitrary **lookahead** to parse CFGs? Generally, yes.
    - however, large subclasses of CFGs *can* be parsed with *limited* lookahead:
        * **LL(1)**: left to right scan, left-most derivation, 1-token lookahead
        * **LR(1)**: left to right scan, right-most derivation, 1-token lookahead
- to achieve LL(1) we roughly want to have the following initial properties:
    - no left recursion
    - some sort of *predictive* parsing in order to minimize backtracking with a lookahead of only one symbol

Grammar Hacking

Consider the following simple grammar for mathematical operations:

```
<goal> ::= <expr>
<expr> ::= <expr> <op> <expr> | num | id
<op> ::= + | - | * | /
```

- there are multiple ways to rewrite the same grammar:
    - but each of these ways may build *different* trees, which lead to *different* executions
    - want to avoid possible grammar issues such as precendence, infinite recursion, etc. by rewriting the grammar
    - eg. classic precedence issue of parsing `x + y * z` as `(x+y) * z` vs. `x + (y*z)`
- to address **precedence**:
    - additional machinery is required in the grammar
    - introduce extra **levels**
    - eg. introduce new nonterminals that group higher precedence ops like multiplication, and ones that group lower precedence ops like addition
        * the higher precedence nonterminal cannot reduce down to the lower precedence nonterminal
        * forces the *correct* tree

Example of fixing precedence in our grammar:

```
<expr> ::= <expr> + <term> | <expr> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= num | id
```

- **ambiguity** occurs when a grammar has more than one derivation for a single sequential form:
  - eg. the classic **dangling-else** ambiguity `if A then if B then C else D`
  - to address ambiguity:
    - rearrange the grammar to select one of the derivations, eg. matching each `else` with the closest unmatched `then`
  - another possible ambiguity arises from the context-free specification:
    - eg. **overloading** such as `f(17)` , could be a function or a variable subscript
    - requires context to disambiguate, really an issue of type
    - rather than complicate parsing, this should be handled separately

Example of fixing the dangling-else ambiguity:

```
<stmt> ::= <matched> | <unmatched>
<matched> ::= if <expr> then <matched> else <matched> | ...
<unmatched> ::= if <expr> then <stmt> | if <expr> then <matched> else <unmatched>
```

- a grammar is **left-recursive** if $\exists A \in V_n s.t. A \Rightarrow^* A\alpha$ for some string α:
  - top-down parsers fail with left-recursive grammars
  - to address left-recursion:
    - transform the grammar to become right-recursive by introducing new nonterminals
  - eg. in grammar notation, replace the productions $A \to A\alpha|\beta|\gamma$ with:
    - $A \to NA'$
    - $N \to \beta|\gamma$
    - $A' \to \alpha A'|\varepsilon$

Example of fixing left-recursion (for `<expr>, <term>`) in our grammar:

```
<expr> ::= <term> <expr'>
<expr'> ::= + <term> <expr'> | - <term> <expr'> | E // epsilon

<term> ::= <factor> <term'>
<term'> ::= * <factor> <term'> | / <factor> <term'> | E
```

- to perform **left-factoring** on a grammar, we want to do repeated prefix factoring until no two alternaties for a single non-terminal have a common prefix:
  - an important property for LL(1) grammars
  - eg. in grammar notation, replace the productions $A \to \alpha\beta|\alpha\gamma$ with:
    - $A \to \alpha A'$
    - $A' \to \beta|\gamma$
  - note that our example grammar after removing left-recursion is now properly left-factored

Achieving LL(1) Parsing

Predictive Parsing

- for multiple productions, we would like a *distinct* way of choosing the *correct* production to expand:
  - for some RHS $\alpha \in G$, define *FIRST*(α) as the set of tokens that can appear *first* in some string derived from α
  - key property: whenever two productions $A \to \alpha$ and $A \to \beta$ both appear in the grammar, we would like:
    - *FIRST*$(\alpha) \cap$ *FIRST*$(\beta) = \emptyset$, ie. the two token sets are disjoint
  - this property of **left-factoring** would allow the parser to make a correct choice with a lookahead of only *one* symbol
  - if the grammar does not have this property, we can hack the grammar
- by left factoring and eliminating left-recursion can we transform an *arbitrary* CFG to a form where it can be predictively parsed with a single token lookahead?
  - no, it is undecidabe whether an arbitrary equivalent grammar exists that satisfies the conditions
  - eg. the grammar $\{a^n 0 b^n\} \cup \{a^n 1 b^{2n}\}$ does not have a satisfying form, since would have to look past an arbitrary number of a to discover the terminal
- idea to translate parsing logic to code:
  1. for all terminal symbols, call an `eat` function that *consumes* the next char in the input stream
  2. for all nonterminal symbols, call the corresponding function corresponding to the production of that nonterminal
     - perform predictive parsing by looking *ahead* to the next character and handling it accordingly
       - there is only one valid way to handle the character in this step due to the left-factoring property
     - how do we handle epsilon?
       - just do nothing ie. consume nothing, and let recursion handle the rest
  - creates a mutually recursive set of functions for each production
    - the name is the LHS of production, and body corresponds to RHS of production

Example simple recursive descent parser:

```
Token token;
void eat(char a) {
```

```
  if (token == a) token = next_token();
  else error();
}

void goal() { token = next_token(); expr(); eat(EOF); }
void expr() { term(); expr_prime(); }
void expr_prime() {
  if (token == PLUS) { eat(PLUS); expr(); }
  else if (token == MINUS) { eat(MINUS); expr(); }
  else { /* noop for epsilon */ }
}
void term() { factor(); term_prime(); }
void term_prime() {
  if (token == MULT) { eat(MULT); term(); }
  else if (token == DIV) { eat(DIV); term(); }
  else { }
}
void factor() {
  if (token == NUM) eat(NUM);
  else if (token == ID) eat(ID);
  else error(); // not epslion here
}
```

Handling Epsilon

- handling epsilon is not as simple as just ignoring it in the descent parser
- for a string of grammar symbols α, *NULLABLE*(α) means α can go to ε:
  - ie. $\mathit{NULLABLE}(\alpha) \iff \alpha \Rightarrow^* \varepsilon$
- to compute *NULLABLE*:
  1. if a symbol a is terminal, it cannot be nullable
  2. otherwise if $a \to Y_1 ... Y_a$ is a production:
     - $\mathit{NULLABLE}(Y_1) \wedge ... \wedge \mathit{NULLABLE}(Y_k) \Rightarrow \mathit{NULLABLE}(A)$
  3. solve the constraints
- again, for a string of grammar symbols α, *FIRST*(α) is the set of terminal symbols that begin strings derived from α:
  - ie. $\mathit{FIRST}(\alpha) = \{a \in V_t | \alpha \Rightarrow^* a\mathrm{B}\}$
- to compute *FIRST*:
  1. if a symbol a is a nonterminal, $\mathit{FIRST}(a) = \{a\}$

2. otherwise if $a \to Y_1 ... Y_a$ is a production:
   - *FIRST*$(Y_1) \subseteq$ *FIRST*(A)
   - $\forall i \in 2...k$, if *NULLABLE*$(Y_1 ... Y_{i-1})$:
     * *FIRST*$(Y_i) \subseteq$ *FIRST*(A)
3. solve the constraints, going for the $\subseteq$-least solution

- for a nonterminal B, *FOLLOW*(B) is the set of terminals that can appear immediately to the *right* of B in some sentential form:
  - ie. *FOLLOW*$(B) = \{a \in V_t | G \Rightarrow^* \alpha B \beta \wedge a \in$ *FIRST*$(\beta \$)\}$
- to compute *FOLLOW*:
  1. $\{\$\} \subseteq$ *FOLLOW*(G) where G is the goal
  2. if $A \to \alpha B \beta$ is a production:
     - *FIRST*$(\beta) \subseteq$ *FOLLOW*(B)
     - if *NULLABLE*(β), then *FOLLOW*$(A) \subseteq$ *FOLLOW*(B)
  3. solve the constraints, going for the $\subseteq$-least solution

Formal Definition

- a grammar G is **LL(1)** iff. for each production $A \to \alpha_1 | \alpha_2 | ... | \alpha_n$:
  1. *FIRST*(α_1), ..., *FIRST*(α_n) are pairwise *disjoint*
  2. if *NULLABLE*(α_i), then for all $j \in 1...n \wedge j \neq i$:
     - *FIRST*$(\alpha_j) \cap$ *FOLLOW*$(A) = \emptyset$
  - if G is ε-free, the first condition is sufficient
  - eg. $S \to aS | a$ is not LL(1)
    * while $S \to aS', S' \to aS' | \varepsilon$ accepts the same language and is LL(1)
- provable facts about LL(1) grammars:
  1. no left-recursive grammar is LL(1)
  2. no ambiguous grammar is LL(1)
  3. some languages have no LL(1) grammar
  4. an ε-free grammar where each alternative expansion for A begins with a distinct terminal is a simple LL(1) grammar
- an LL(1) **parse table** M can be constructed from a grammar G as follows:
  1. $\forall$ productions $A \to \alpha$:
     - $\forall a \in$ *FIRST*(α), add $A \to \alpha$ to $M[A, a]$
     - if $\varepsilon \in$ *FIRST*(α):
       * $\forall b \in$ *FOLLOW*(A), add $A \to \alpha$ to $M[A, b]$ (inlcuding EOF)
  2. set each undefined entry of M to an error state
  - if $\exists M[A, a]$ with multiple entries, then the grammar is *not* LL(1)

JavaCC

- the **Java Compiler Compiler (JCC)** generates a parser automatically for a given grammar:
  - based on LL(k) vs. LL(1)
  - transforms an EGBNF grammar into a parser
  - can have embedded (additional) action code written in Java
  - `javacc fortran.jj` ⟶ `javac Main.java` ⟶ `java Main < prog.f`

JavaCC input format:

```
TOKEN :
{
  < INTEGER_LITERAL: ( ["1"-"9"] (["0"-"9"])* | "0" ) >
}

void StatementListReturn() :
{}
{
  ( Statement() )* "return" Expression() ";"
}
```

Handling Syntax Trees

Visitor Pattern

- parsers generate a syntax tree from an input file:
  - this is an aside on design patterns in order to facilitate using the generated tree
  - see Gamma's *Design Patterns* from 1995
- for OOP, the **visitor pattern** enables the definition of a *new* operation of an object structure *without* changing the classes of the objects:
  - ie. new operation *without* recompiling
  - set of classes must be fixed in advance, and each class must have a hook called the `accept` method

Consider the problem of summing up lists using the following list implementation:

```
interface List {}

class Nil implements List {}

class Cons implements List {
  int head;
  List tail;
}
```

First approach using type casts:

```
List l;
int sum = 0;
while (true) {
  if (l instanceof Nil)
    break;
  else if (l instanceof Cons) {
    sum += ((Cons) l).head;
    l = ((Cons) l).tail;
  }
}
```

- *pros*:
  - code is written without touching the classes
- *cons*:

    - code constantly uses type casts and `instanceof` to determine classes

Second approach using dedicated methods (OO version):

```
interface List { int sum(); }

class Nil implements List {
  public int sum() { return 0; }
}

class Cons implements List {
  int head;
  List tail;
  public int sum() { return head + tail.sum(); }
}
```

- *pros*:
    - code can be written more systematically, without casts
- *cons*:
    - for each new operation, need to write new dedicated methods and re-compile
- **visitor pattern** approach:
    - divide the code into an object structure and a **visitor** (akin to functional programming)
    - insert an `accept` method in each class, which takes a Visitor as an argument
    - a visitor contains a `visit` method for each class (using *overloading*)
        * defines both actions and access of *subobjects*
    - *pros*:
        * new methods without recompilation
        * no frequent type casts
    - *cons*:
        * all classes need a hook in the `accept` method
    - used by tools such as JJTree, Java Tree Builder, JCC
    - summary, visitors:
        * make adding new operations easily
        * gather *related* operations
        * can accumulate state
        * can break encapsulation, since it needs access to internal operations

Third approach with visitor pattern:

```
interface List {
  // door open to let in a visitor into class internals
  void accept(Visitor v);
```

```java
}
interface Visitor {
  void visit(Nil x); // code is packaged into a visitor
  void visit(Cons x);
}

class Nil implements List {
  // `this` is statically defined by the *enclosing* class
  public void accept(Visitor v) { v.visit(this); }
}
class Cons implements List {
  int head;
  List tail;
  public void accept(Visitor v) { v.visit(this); }
}

class SumVisitor implements Visitor {
  int sum = 0;
  public void visit(Nil x) {}
  public void visit(Cons x) {
    // take an action:
    sum += x.head;
    // handle subojects:
    x.tail.accept(this); // process tail *indirectly* recursively

    // The accept call will in turn call visit...
    // This pattern is called *double dispatching*.
    // Why not just visit(x.tail) ?
    // This *fails*, since x.tail is type List.
  }
}
```

Using `SumVisitor` :

```java
SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```

Java Tree Builder

- the produced JavaCC grammar can be processed by the JCC to give a parser

that produces syntax trees:
  - the produced syntax trees can be traversed by a Java program by writing subclasses of the default visitor
  - JavaCC grammar feeds into the **Java Tree Builder (JTB)**
  - JTB creates JavaCC grammar with embedded Java code, syntax-tree-node classes, and a default visitor
  - the new JavaCC grammar feeds into the JCC, which creates a parser
  - `jtb fortran.jj` ⟶ `javacc jtb.out.jj` ⟶ `javac Main.java` ⟶ `java Main < prog.f`

Translating a grammar production with JTB:

```
// .jj grammar
void assignment() :
{}
{ PrimaryExpression() AssignmentOperator() Expression() }

// jtb.out.jj with embedded java code that builds syntax tree
Assignment Assignment () :
{
  PrimaryExpression n0;
  AssignmentOperator n1;
  Expression n2; {}
}
{
  n0 = PrimaryExpression()
  n1 = AssignmentOperator()
  n2 = Expression()
  { return new Assignment(n0, n1, n2); }
}
```

JTB creates this syntax-tree-node class representing `Assignment` :

```
public class Assignment implements Node {
  PrimaryExpression f0;
  AssignmentOperator f1;
  Expression f2;

  public Assignment(PrimaryExpression n0,
    AssignmentOperator n1, Expression n2) {
    f0 = n0; f1 = n1; f2 = n2;
  }

  public void accept(visitor.Visitor v) {
```

```
    v.visit(this)
  }
}
```

Default DFS visitor:

```
public class DepthFirstVisitor implements Visitor {
  ...
  // f0 -> PrimaryExpression()
  // f1 -> AssignmentExpression()
  // f2 => Expression()
  public void visit(Assignment n) {
    // no action taken on current node,
    // then recurse on subobjects
    n.f0.accept(this);
    n.f1.accept(this);
    n.f2.accept(this);
  }
}
```

Example visitor to print LHS of assignments:

```
public class PrinterVisitor extends DepthFirstVisitor {
  public void visit(Assignment n) {
    // printing identifer on LHS
    System.out.println(n.f0.f0.toString());
    // no need to recurse into subobjects since assignments cannot be nested
  }
}
```

Appendix

LL(1) Practice Examples

---

1. given the following grammar:
    - $A ::= \varepsilon | zCw$
    - $B ::= Ayx$
    - $C ::= ywz | \varepsilon | BAx$
    - then:
        - $\mathit{FIRST}(A) = \{z\}$
        - $\mathit{FIRST}(B) = \{y, z\}$
        - $\mathit{FIRST}(C) = \{y, z\}$
        - $\mathit{NULLABLE}(A) = \mathit{true}$
        - $\mathit{NULLABLE}(B) = \mathit{false}$
        - $\mathit{NULLABLE}(C) = \mathit{true}$
    - we can make the following observations for each nonterminal on the RHS:
        - $w \in \mathit{FOLLOW}(C)$
        - $y \in \mathit{FOLLOW}(A)$
        - $\mathit{FIRST}(A) \subseteq \mathit{FOLLOW}(B)$
        - $x \in \mathit{FOLLOW}(B)$
        - $x \in \mathit{FOLLOW}(A)$
    - thus:
        - $\mathit{FOLLOW}(A) = \{x, y\}$
        - $\mathit{FOLLOW}(B) = \{x, z\}$
        - $\mathit{FOLLOW}(C) = \{w\}$
    - therefore the grammar is *not* LL(1), since for C:
        - $\mathit{FIRST}(ywz) \cap \mathit{FIRST}(BAx) \neq \emptyset$