DAC718 - Compiler Design

Top-Down Parsing

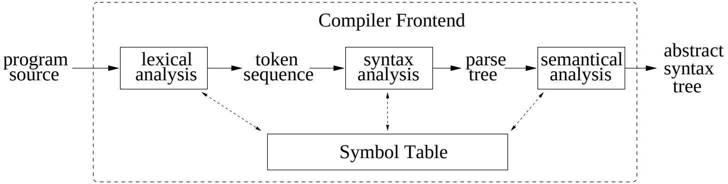
Jonas Lundberg

Jonas.Lundberg@msi.vxu.se

http://w3.msi.vxu.se/users/jonasl/dac718

5 februari 2007

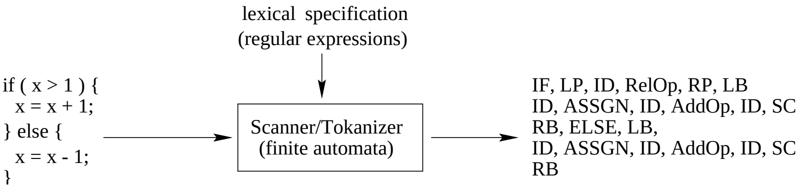
Frontend Overview



- **Lexical Analysis:** Identify atomic language constructs.
  Each type of construct is represented by a token.
  (e.g. `3.14` $\mapsto$ `FLOAT`, `if` $\mapsto$ `IF`, `a` $\mapsto$ `ID`).
- **Syntax Analysis:** Checks if the token sequence is correct with respect to the language specification.
- **Semantical Analysis:** Checks type relations + consistency rules.
  (e.g. if $type(lhs) = type(rhs)$ in an assignment $lhs = rhs$).

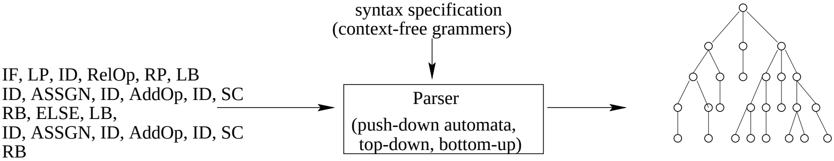
Each step involves a transformation from a program representation to another.

Lexical Analysis Overview



- Input program representation: Character sequence
- Output program representation: Token sequence
- Analysis specification: Regular expressions
- Recognizing (abstract) machine: Finite Automata
- Implementation: Finite Automata

Syntax Analysis Overview



- Input program representation: Token sequence
- Output program representation: Parse (or syntax) tree
- Analysis specification: Context-free grammar
- Recognizing (abstract) machine: Push-down Automata
- Implementation: Top-down or Bottom-up parsers

Top-down Methods

Consider the grammar

(1) $S \rightarrow aABe$ (2) $A \rightarrow b$ (3) $A \rightarrow Abc$ (4) $B \rightarrow d$

- Using a left-most derivation we can show that *abbcde* is in the language

$$S \stackrel{1}{\Longrightarrow} aABe \stackrel{3}{\Longrightarrow} aAbcBe \stackrel{2}{\Longrightarrow} abbcBe \stackrel{4}{\Longrightarrow} abbcde$$

  This is a top-down approach since we start from the start symbol S (the syntax tree root) and work our way down to the tokens *abbcde* (the leaves of the syntax tree).
- Problem: What production to use when facing one (or k) tokens.
- Fast and easy when it works.
- JavaCC uses a top-down parsing method.

Agenda

- Recursive Desecent
- Table-driven Parsing.
- Deriving a LL(1) parse table.

A Simple Method: Recursive Descent (RD)

- We associate one procedure `pA()` with each nonterminal A.
- `lookahed` = next token to process.
- The procedure `pA()` is called whenever we want to resolve $A \rightarrow \alpha$.
- For example, consider $A \rightarrow bCd \mid eF$ where $A, C, F \in N$ and $b, d, e \in T$

```
pA() {
  if lookahead = b then
    eat(b); pC(); eat(d);
  elsif lookahead = e then
    eat(e); pF();
  else
    reportError();
  end if;
}
```

```
eat(Token t) {
  if lookahead = t then
    lookahead = nextToken();
  else
   reportError();
  end if;
}
```

The variable `lookahead` holds the next input token.

Predictive Parsing

- RD in summary:
  - Given a lookahead $a \in T$ ...
  - ... and a non-terminal $A \in N$ ...
  - it should decide which production $A \rightarrow \alpha$ to use.
- The problem with RD (as with any LL(k) method) is that it must be able to decide which branch of a production to use just by looking at one (or k) token(s) ahead.
- These methods are also called **Predictive Parsing Methods** since every production decision implies a prediction of what will follow.

Predictive Parsing Problems

- **Ambiguous Grammar:** Gives non-deterministic left-most derivation.
- **Left-factoring:** $A \rightarrow \alpha\beta \,|\alpha\omega$ makes prediction impossible.
- **Left-recursion:** $A \rightarrow A\alpha$ causes an infinite loop.

Arithmetic Expressions (Grammar 3)

A non-ambiguous grammar for arithmetic expressions with correct operator priorities:

$$\begin{aligned} G &= \{T, N, P, S\} \\ T &= \{id, +, *, (,), \} \\ N &= \{E, E', T, T', F\} \\ S &= E \end{aligned}$$

where P is defined as

$$\begin{array}{llll} (1) & E & \rightarrow & TE', \quad E' \rightarrow +TE' \mid \varepsilon, \\ (2) & T & \rightarrow & F\,T', \quad T' \rightarrow *F\,T' \mid \varepsilon, \\ (3) & F & \rightarrow & id \mid (E) \end{array}$$

Notice: In Grammar 3 is ambiguity, left-factoring, and left-recursion already removed.

Recursive Descent Revisited

RD in summary:

- Given a lookahead $a \in T$ ...
- ... and a non-terminal $A \in N$ ...
- it should decide which production $A \rightarrow \alpha$ to use.

The procedure associated with $T' \rightarrow *F\ T' \mid \varepsilon$

```
Tprime() {
   if lookahead = * then
      eat(*); F(); Tprime();
   elsif lookahead = +,) then
      ;    //Do nothing
   else
      reportError();
   end if;
}
```

The ε-production for T' is the tricky part. Here we must determine on what input T' should do nothing and when to report error. A non-trivial task. Fortunately, we have algorithms that can help us.

Problems with Recursive Descent

- The large number of simultanious recursive calls makes the compiler slow and memory consuming. (calls $\Rightarrow$ new activation records $\Rightarrow$ several object creations)
- Grammar updates are often difficult to handle.
- We have no systematic approach to decide which production branch to chose given some input token t.

A Parse Table Driven Approach

- Recursive calls are replaced by a stack.
- Which production branch to chose is given by a **parse table** $M[A, t]$.
- Given a non-terminal A and lookahead t, $M[A, t]$ returns the appropriate production to use.
- We have algorithms for constructing parse tables

A Parse Table for Grammar 3

	id	$+$	$*$	(	)
E	$E \to TE'$			$E \to TE'$	
E'		$E' \to +TE'$			$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$	

Parse Tables

- Given a non-terminal A and lookahead t, $M[A, t]$ returns the appropriate production to use.
- Using a parse table is easy (next slide)
- Implementing the use of a parse table is a bit more tricky (but not very hard)
- Constructing a parse table is much more difficult
  (but we have algorithms who can help us!)

Using Parse Tables

Parsing $id + id\$$ (where \$ symbolizes end-of-file)

- **Start:**
  - Push start symbol E on stack $\Rightarrow TOP = E$
  - Lookahead is first input token $\Rightarrow LA = id$, Remains $= +id\$$
- **Parse:**
  - Rule: *reduce* iff TOP element equals LA, otherwise *shift*.
  - shift $\Rightarrow$ replace top element with $M[TOP, LA]$ right-hand side
  - reduce $\Rightarrow$ pop element (a terminal) and set lookahead to next input.
- **Success:** When lookahead is end-of-file ($LA = \$$)

Remains	LA	TOP	Stack
$+id\$$	id	E	E
$+id\$$	id	T	TE'
$+id\$$	id	F	$FT'E'$
$+id\$$	id	id	$idT'E'$
$id\$$	$+$	T'	$T'E'$
$id\$$	$+$	E'	E'

Remains	LA	TOP	Stack
$id\$$	$+$	$+$	$+TE'$
\$	id	T	TE'
\$	id	F	$FT'E'$
\$	id	id	$idT'E'$
	\$	T'	$T'E'$

Algorithm for table driven LL-parsing

```
stack.push(StartSymbol)
LA = input.nextToken()
repeat
   X = stack.top()
   if X ∈ T or X = EOF then
      if X = LA then
         stack.pop()
         LA = input.nextToken()
      else
         error(stack,LA,input)          (Token not in agreement with prediction)
      end if
   else
      if M[X, t] = X → Y_1 ... Y_n then
         stack.pop()
         push Y_n ... Y_1 onto stack, with Y_1 on top
         add X → Y_1 ... Y_n to parse tree
      else
         error(stack,LA,input)      (Can't make a prediction, empty slot in M[X, t])
      end if
   end if
until LA = EOF
```

Constructing Parse Tables: Introduction

- Given a grammar G we can construct a parse table $M[X,t]$ systematically.
- Ambiguity, left-recursion, and left-factorization give multiple entries in $M[X,t]$.
- Before constructing $M[X,t]$, try to eliminate all cases of the above problems. (It will save you both time and effort.)
- **Basic idea:** Constructing three methods for each non-terminal $X \in N$.
  - **Nullable(X):** is true if X can derive the empty string ε.
  - **FIRST(X):** the terminals that can **begin** strings derived from X.
  - **FOLLOW(X):** is the set of terminals that can immediately follow X.
- Use Algorithm 4 to construct $M[X,t]$ using these methods.

Notations to be used

$a, b, \ldots \in T, \qquad A, B, \ldots \in N, \qquad \ldots X, Y, Z \in (N \cup T), \qquad \alpha, \beta, \gamma \ldots \in (N \cup T)^*$

Algorithm 1: Nullable(X)

- Nullable(X) is true if X can derive the empty string ε.
- Algorithm for constructing Nullable(X).

  nullable(X) $:=$ false for all $X \in (N \cup T)$
  repeat
    for each production $X \rightarrow Y_1 Y_2 \ldots Y_n$ **do**
      if $Y_1 Y_2 \ldots Y_n$ are all nullable (or if $X \rightarrow \varepsilon$) **then**
        nullable(X) $:=$ true
      end if
    end for
  until nullable not changed in this iteration

- Furthermore, a string $\alpha = X_1 X_2 \ldots X_n$ is nullable if every X_i is nullable.

Algorithm 2: FIRST(α)

- FIRST(X) is the set of terminals that can **begin** strings derived from X.
- Algorithm for FIRST(X)

  FIRST(a) := $\{a\}$ for each $a \in T$
  FIRST(A) := $\{\}$ for each $A \in N$
  repeat
    for each production $X \rightarrow Y_1 Y_2 \ldots Y_n$ **do**
      if Y_1 not nullable **then**
        add FIRST(Y_1) to FIRST(X)
      else if $Y_1 \ldots Y_{i-1}$ are all nullable (or if $i = n$) **then**
        add FIRST(Y_1) $\cup \ldots \cup$ FIRST(Y_i) to FIRST(X)
      end if
    end for
  until FIRST not changed in this iteration

- Given string $\alpha = X_1 X_2 \ldots X_n$ where $X_i \in N \cup T$, we have

$$\begin{array}{lcll} \text{FIRST}(\alpha) & = & \text{FIRST}(X_1), & \text{if not } X_1 \text{ nullable} \\ \text{FIRST}(\alpha) & = & \text{FIRST}(X_1) \cup \ldots \cup \text{FIRST}(X_i)\ , & \text{if } X_1 \ldots X_{i-1} \text{ nullable} \end{array}$$

$\Rightarrow$ given FIRST(X), we can compute FIRST(α) for each string α.

Algorithm 3: FOLLOW(X)

- FOLLOW(X) is the set of terminals that can immediately follow X.
- Example, $t \in$ FOLLOW(X) if there is any derivation containing Xt. This can occur if a derivation contains $XYZt$ where both Y and Z are nullable.
- Algorithm for FOLLOW(X)

repeat
    for each nonterminal Y **do**
        for each production $X \rightarrow \alpha Y \beta$ **do**
            add FIRST(β) to FOLLOW(Y)
            if β is nullable (or ε) **then**
                add FOLLOW(X) to FOLLOW(Y)
            end if
        end for
    end for
until FOLLOW not changed in this iteration

Algorithm 4: Parse Table Construction

- $M[X, t]$ gives the production to use when resolving X given lookahead t.
- Basic idea: $X \rightarrow \alpha \ \in M[X, t]$ iff $t \in \text{FIRST}(\alpha)$
- α is Nullable requires special treatment.
- Algorithm

  for each production $X \rightarrow \alpha$ **do**
    for each terminal $t \in \text{FIRST}(\alpha)$ **do**
      add $X \rightarrow \alpha$ to $M[X, t]$
    end for
    if α is Nullable (or ε) **then**
      for each $t \in \text{FOLLOW}(X)$ **do**
        add $X \rightarrow \alpha$ to $M[X, t]$
      end for
    end if
  end for

Summary: Table Construction for Grammar 3

Auxiliary functions Nullable, FIRST, and FOLLOW

	Nullable	*FIRST*	*FOLLOW*
E	*No*	$id, ($	$)$
E'	*Yes*	$+$	$)$
T	*No*	$id, ($	$+,)$
T'	*Yes*	$*$	$+,)$
F	*No*	$id, ($	$+, *,)$

Corresponding Parse Table

	id	$+$	$*$	$($	$)$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$	
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$	

Multiple Entries

Consider the following "dangling else" grammar:

$$S \to iEtSS'|a, \qquad S' \to eS|\varepsilon, \qquad E \to b$$

where E = expression, S = statement, S' = elsePart, i = if, t = then, e = else, a = OtherStatement, and b = someExpression. It has the following parse table

	a	b	e	i	t
S	$S \to a$			$E \to iEtSS'$	
S'			$S' \to eS, S' \to \varepsilon$		
E		$E \to b$			

- The ambiguous grammar is manifested as a duplicate entry when e (else) is seen. We can resolve the ambiguity by always chosing $S' \to eS$ (That is, remove $S' \to \varepsilon$ from that entry.)
- Removing $S' \to \varepsilon$ from that entry is not the same as removing $S' \to \varepsilon$ from the grammar.
- In general, the parse table is a good place to do some minor adjustments of the parser.

LL(1)

- LL(1) stands for *Left-to-right parse, Leftmost-derivation, 1-symbol lookahead*.
- Left-to-right parse means that we are scanning the input left-to-right.
- A grammar generating a table with no multiple entries is a LL(1) grammar. (multiple entry $\Rightarrow$ not deterministic $\Rightarrow$ ambiguous grammar)
- An LL(1) table is of size $O(|N| * |T|)$ where $|N|$ and $|T|$ are the numbers of non-terminals and terminals.

LL(k)

- LL(k) stands for *Left-to-right parse, Leftmost-derivation, k-symbol lookahead*.
- Grammars parsable with LL(k) parsers are called *LL(k) grammars*.
- An LL(3) grammar might require 3 token to chose the correct branch.
- An LL(3) table has an entry for every possible triple of tokens $\Rightarrow O(|N| * |T|^3)$
- No ambiguous grammar is LL(k) for any k.
- LL(k) parsers can be constructed systematically, FIRST(X) gives all k-tuples that can begin a string derived from X, FOLLOW(X) is all k-tuples that can immediately follow X. It is straight forward but not so fun ....

Written Assignment 2: LL(1) Parsing Tables

Consider the following grammar

$$S \rightarrow uBDz \quad B \rightarrow w \mid Bv \quad D \rightarrow EF \quad E \rightarrow y \mid \varepsilon \quad F \rightarrow x \mid \varepsilon$$

where S is the start symbol and u, v, w, x, y, z are terminals.

1. Compute *Nullable*, *FIRST*, and *FOLLOWS* for the non-terminals in above grammar using the algorithms presented in the lecture slides.
2. Construct the LL(1) parsing table.
3. Give evidence that this grammar is not LL(1).
4. Modify the grammar **as little as possible** to make it an LL(1) grammar that accepts the same language.
5. Recompute the results in 1) and 2) using the modified grammar.
6. Simulate the parsing of the string *uwvvyz* using the newly constructed parsing table.

Deadline: 2007-02-18 (One week before PA step 1)