Software II: Principles of Programming Languages

Lecture 4 – Language Translation: Lexical and Syntactic Analysis

The Compiling Process

Source Code → Compiler → Object Module → Linker → Executable version

Assembler version
The organization of a compiler

- The various components of a compiler are organized into a **front end** and a **back end**.
- The front end is designed to produce some intermediate representation of a program written in the source language.
- The back end is designed to produce a program for a target computer from the intermediate representation.
Lexical Analysis

- The lexical analyzer (or scanner) breaks up the stream of text into a stream of strings called “lexemes” (or token strings).
- The scanner checks one character at a time until it determines that it has found a character which does not belong in the lexeme.
- The scanner looks it up in the symbol table (inserting it if necessary) and determines the token associated with that lexeme.
Lexical Analysis (continued)

- **Token** - the language component that the character string read represents.
- Scanners usually reads the text of the program either a line or a block at a time. (File I/O is rather inefficient compared to other operations within the compiler.

Syntactic Analysis

- A syntactic analyzer (or **parser**) takes the stream of tokens determines the syntactic structure of the program.
- The parser creates a structure called a **parse tree**. The parser usually does not store the parse in memory or on disk, but it does formally recognize program’s the grammatical structure
Semantic Analysis

- Semantic analysis involves ensuring that the semantics (or meaning) of the program is correct.
- It is quite possible for a program to be correct syntactically and to be correct semantically.
- Semantic analysis usually means making sure that the data types and control structures of a program are used correctly.

Semantic Analysis (continued)

- The various semantic analysis routines are usually incorporated into the parser and do not usually comprise a separate phase of the compiling process.
- The process of generating an intermediate representation (usually an abstract syntax tree) is usually directed by the parsing of the program.
The Symbol Table

• The symbol table tracks all symbols used in a given program.
• This includes:
  – Key words
  – Standard identifiers
  – Numeric, character and other literals
  – User-defined data types
  – User-defined variables

The Symbol Table (continued)

• Symbol tables must contain:
  – Token class
  – Lexemes
  – Scope
  – Types
  – Pointers to other symbol table entries (as necessary)
Transition Diagrams

- Transition diagrams are a special form of finite automaton, incorporating features that belong in a compiler’s scanner:
  - Actions associated with final states.
  - Backup from a state, allowing for a lookahead character being returned to the input stream.
  - Transitions can be labeled as belonging to “other”, indicating any class of character not explicitly accounted for.

Transition Diagrams (continued)

In drawing transition diagrams, it is helpful to use an alternate approach to describing regular expressions:

- $a|b$ denotes a or $b$.
- $ab$ denotes a followed by $b$.
- $(a|b)^*$ denotes a followed by $b$ zero or more times.
- $(a|b)c$ denotes a or $b$ followed by $c$.
The different lexical categories or *classes* can be described in this fashion:

- **letter**: (a | b | c | d | e ... | A | B | C | D | E ... | X | Y | Z)
- **digit**: (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
- **other**: (! | @ | # | $ | % | ^ | & | * | ( | ) | _ | + | = | - | ` | ~ | { | } | | | " | ' | : | ;)
- **identifier**: letter (letter | digit)*
- **integer**: digit digit*
- **real**: (digit digit* . digit digit*)
  | (digit digit* . digit digit* (E | e) (+|- | ) digit digit*)

Transition Diagrams (continued)

The transition diagram for our language shown before becomes:

![Transition Diagram](image-url)
Practical Issues in Lexical Analysis

There are several important practical issues that arise in the design of a scanner:

- Lookahead
- Case sensitivity
- Skipping of lead blanks and comments
- Use of first characters

Lookahead characters

Since you cannot determine if you have read beyond the end of a lexeme until you have done so, you must be prepared to handle the “lookahead” character. There are two approaches available:

- Start with a lookahead character and fetch a new one every time the lookahead character is “consumed” by the lexeme.
- Use two functions to manipulate the input stream, one to “get” the next character and one to “unget” the next character, returning it temporarily to the input stream.
Lookahead characters (continued)

```
// gettc() - Fetches a character from a
// file. It uses get and adjusts
// the line number count when
// necessary.
char scanner::gettc(void)
{
    char c;

    // If we're at the end of file, return a null
    // byte, which serves to mark the end of file.
    if (infile.eof())
        c = '\0';

    // If the next character is a newline,
    // increment the line count
    else if ((c = infile.get()) == '\n')
        linenum++;

    // Return the character converted to lower case
    return(tolower(c));
}
```
Lookahead characters (continued)

```cpp
// ungettc() - Returns a character to the file. Uses ungetc and will adjust line number count.
void scanner::ungettc(char c)
{
    // If it's a newline, decrement the line count; we haven't gone to the next line yet.
    if (c == '\n')
        --linenum;
    // Put it back into the input stream.
    infile.putback(c);
}
```

Case sensitivity

- Although “a” and “A” are regarded as the same character in the English language, they are represented by different ASCII codes. For a compiler to be case insensitive, we need to consider these both as the same letter.
- The easiest way to do this is to convert all letters to the same case.
- Not all languages do this, e.g., C.
Skipping lead blanks and comments

• Before reading the first significant character in a lexeme, it is necessary to skip past both lead blanks as well as comments.

• One must assume that the scanner can encounter either or both repeatedly and interchangeably before reading the first significant character.

// firstchar() - Skips past both white space
//               and comments until it finds
//               the first non-white space
//               character outside a comment.
char scanner::firstchar(void)
{
    char c;
    bool goodchar = false;

    // If we're at the end of the file,
    // return the EOF marker so that we'll
    // return the EOF token
    if (infile.eof())
        return(EndOfFile);
Skipping lead blanks and comments (continued)

    // We're looking for a non-white space
    // character that is outside a comment.
    // Keep scanning until we find one or
    // reach the end of the file.
    while (!goodchar) {
        // Skip the white space in the
        // program
        while (!infile.eof() && isspace(c = getcc())) ;

        // Is it a comment or a real
        // first character?
        if  (c != '{')
            goodchar = true;
    }

    // If we're at the end of file, return
    // the EOF marker. Otherwise, return
    // the character.
    if (infile.eof())
        return(EndOfFile);
    else
        return(c);
Use of first character

- In most programming languages, the first character of a lexeme indicates the nature of the lexeme and token associated with it.
- In most instances, identifiers and reserved words begin with a letter (followed by zero or more letters and digits), numbers begin with a digit and operators begin with other characters.

Use of first character (continued)

```c
// gettoken() - Scan out the token strings of
// the language and return the
// corresponding token class to the
// parser.

tokentype scanner::gettoken(int &tabindex)
{
    char c;

    // If this is the end of the file, send the
    // token that indicates this
    
    if ((c = lookahead) == EndOfFile)
        return(tokeof);
```
Use of first character (continued)

// If it begins with a letter, it is a word.
// If begins with a digit, it is a number.
// Otherwise, it is an error.
lookahead = gettc();
if (isalpha(c))
    return(scanword(c, tabindex));
else if (isdigit(c))
    return(scannum(c, tabindex));
else
    return(scanop(c, tabindex));

Scanning for reserved words and identifiers

• Once the scanner determines that the first character is a letter, it continues to read characters and concatenate them to the lexeme until it encounters a character other than a letter or digit.
• If the resultant lexeme is not in the symbol table, it must be a new identifier.
// scanword() - Scanning reserved words
// something other than a letter.
tokentype scanner::scanword(char c, int &tabindex)
{
    char lexeme[LexemeLen];
    int i = 0;

    // Build the string one character at a time.
    // It keeps scanning until either the end of
    // file or until it encounters a non-letter
    lexeme[i++] = c;
    while ((c = lookahead) != EndOfFile && (isalpha(c) || isdigit(c)))
    {
        lexeme[i++] = c;
        lookahead = getch();
    }

    // Add a null byte to terminate the
    // string and get the lookahead that
    // begins the next lexeme.
    lexeme[i] = '\0';
    ungetch(lookahead);
    lookahead = firstchar();

    // If the lexeme is already in the symbol
    // table, return its tokenclass.  If it
    // isn't, it must be an identifier whose
    // type we do not know yet.
    if (st.installname(lexeme, tabindex))
        return(st.gettok_class(tabindex));
    else
    {
        st.setattrib(tabindex, stunknown, tokidentifier);
        return(tokidentifier);
    }
}
Scanning for numeric literals

- After determining that the lexeme begins with a digit, the scanner reads characters, concatenating them to the lexeme until it encounters a non-digit.
- If it is a period, it will concatenate this to the lexeme and resume reading characters until it encounters another non-digit.
- If it is an “E”, it must then read the exponent.
- The token associated with the lexeme is either number or the number’s type.

```c
// scannum() - Scan for a number.
tokentype scanner::scannum(char c, int &tabindex)
{
    int ival, i = 0;
    bool isitreal = false;
    float rval;
    char lexeme[LexemeLen];

    // Scan until you encounter something that
    // cannot be part of a number or the end of
    // file
    lexeme[i++] = c;
    while ((c = lookahead) != EndOfFile && isdigit(c)) {
        lexeme[i++] = c;
        lookahead = gettc();
    }
```
Scanning for numeric literals (continued)

// Is there a fractional part?
if (c == '.') {
    isitreal = true;
    lexeme[i++] = c;
    while ((c = lookahead) != EndOfFile && isdigit(c)) {
        lexeme[i++] = c;
        lookahead = gettc();
    }
}

// Add a null byte to terminate the
// string and get the lookahead that
// begins the next lexeme.
ungetc(lookahead);
lexeme[i] = '\0';
lookahead = firstchar();

Scanning for numeric literals (continued)

// If there is no fractional part, it is an
// integer literal constant. Otherwise, it
// is a real literal constant. Firstly, is
// it already in the symbol table?
if (st.installname(lexeme, tabindex))
    return(st.gettok_class(tabindex));
// If not, is it real?
else if (isitreal) {
    st.setattrib(tabindex, stunknown, tokconstant);
    st.installdatatype(tabindex, stliteral, dtreal);
    rval = atof(lexeme);
    st.setvalue(tabindex, rval);
    return(st.gettok_class(tabindex));
}
// Must be an integer literal
else {
    st.setattrib(tabindex, stunknown,
                  tokconstant);
    st.installdatatype(tabindex,
                        stliteral, dtinteger);
    ival = atoi(lexeme);
    st.setvalue(tabindex, ival);
    // ungettc(lookahead);
    return(st.gettok_class(tabindex));
}
ungettc(lookahead);
return(st.gettok_class(tabindex));

Scanning for operators and characters literals

• If the first character is neither a letter nor a digit, the
  lexeme must be one of the following:
  – an operator
  – a character literal
  – a string literal
• In scanning an operator:
  – we should be cognizant of how many characters it may
    contain.
  – we may wish to hand-code the token that will be
    returned by the symbol table.
• In scanning a literal, we read characters until encountering
  the appropriate closing quotation mark.
Special problems in lexical analysis

There are a few other problems faced in lexical analysis:

• Token overloading
• Backtracking
• Buffering
• When keywords are not reserved words

Token overloading

• On occasion, there are difficulties presented by a lexeme serving more than one role in a programming language. E.g., = is the test of equality AND the assignment operator.
• This can be handled by using different lexemes – E.g., C uses \( = = \) and \( = \), Pascal uses \( = = \) and \( : = \), FORTRAN uses \( .EQ. \) and \( = \).
• If several lexemes are grouped into one token, it may become necessary to separate one or more of the lexemes out to become a distinctly different token.
Backtracking

• In rare instances, it may become necessary to backtrack and re-scan the text of the program.
  E.g., the *DO* statement in FORTRAN
  \[
  \text{DO 101 I = 1, 50}
  \]
  is initially read as
  \[
  \text{DO101I = 1}
  \]
  until the , is encountered.

Scanner generators

• Scanner generators automatically generate a scanner given the lexical specifications and software routines given by the user.

• Scanner generators take advantage of the fact that a scanner is essentially an implementation of a finite automaton and can thus be created in an automated fashion.

• LEX is an example of such a software tool.
What is top-down parsing?

- Top-down parsing is a parsing-method where a sentence is parsed starting from the root of the parse tree (with the “Start” symbol), working recursively down to the leaves of the tree (with the terminals).
- In practice, top-down parsing algorithms are easier to understand than bottom-up algorithms.
- Not all grammars can be parsed top-down, but most context-free grammars can be parsed bottom-up.

LL(k) grammars

- Top-down grammars are referred to as LL(k) grammars:
  - The first L indicates Left-to-Right scanning.
  - The second L indicates Left-most derivation
  - The k indicates k lookahead characters.
- We will be examining LL(1) grammars, which spot errors at the earliest opportunity but provide strict requirements on our grammars.
LL(1) grammars

- LL(1) grammars determine from a single lookahead token which alternative derivation to use in parsing a sentence.
- This requires that if a nonterminal A has two different productions:
  \[ A ::= \alpha \quad \text{and} \quad A ::= \beta \]
  - that \( \alpha \) and \( \beta \) cannot begin with the same token.
  - \( \alpha \) or \( \beta \) can derive an empty string but not both.
  - if \( \beta \Rightarrow^* \varepsilon \), \( \alpha \) cannot derive any string that begins with a token that could immediately follow A.

LL(1) grammars (continued)

If you look at the first token of expression 3*x + y*z (which is \texttt{const}) and the productions for the start symbol E

\[
E ::= E + T | T
\]

How can you tell whether it derives \( E + T \) or simply \( T \)? This requires information about the subsequent tokens.
LL(1) grammars (continued)

It becomes necessary to convert many grammars into LL(1). The most common conversions involve:

• Removing left-recursion (whether it is direct or indirect)
• Factoring out any terminals found out the beginning of more than one production for a given nonterminal

Removing left-recursion

Aho, Sethi and Ullman show that left recursion of the form:

\[ A ::= A\alpha | \beta \]

can be converted to right-recursion (which is LL(1)) by replacing it with the following productions:

\[ A ::= \beta A' \]
\[ A' ::= \alpha A' | \varepsilon \]
Left-Factoring

Many grammars have the same prefix symbols at the beginning of alternative right sentential forms for a nonterminal:

\[ A ::= \alpha \beta | \alpha \gamma \]

We replace these production with the following:

\[ A ::= \alpha A' \]
\[ A' ::= \beta | \gamma \]

Converting an expression grammar into LL(1) form

- Our expression grammar is:
  \[ E ::= E + T | T \]
  \[ T ::= T \ast F | F \]
  \[ F ::= \text{id} | \text{const} | (E) \]

- Following our rule for removing direct left-recursion, our grammar becomes:
  \[ E ::= T E' \]
  \[ E' ::= + T E' | \epsilon \]
  \[ T ::= F T' \]
  \[ T' ::= \ast F T' | \epsilon \]
  \[ F ::= \text{id} | \text{const} | (E) \]
Once the grammar is in LL(1) form, we create a table showing which production we use in parsing each nonterminal for every possible lookahead token:

```
<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>E'</th>
<th>T</th>
<th>T'</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

E ::= TE'
E' ::= +TE'
E' ::= □
T ::= FT'
T' ::= *FT'
T' ::= □
F ::= id
F ::= const
F ::= ( E )

Parsing an expression using the LL(1) parse table

Let’s take a look at the expression

3\#x + y

Our parse tree is initially just the start symbol E and our lookahead token is **const** (the lexeme is 3)

The production for E and a lookahead token of **const** is #1, making our parse tree:

```
E
  └── T
      └── E'
  └── F
      └── T'
```
Parsing an expression using the LL(1) parse table (continued)

The production for F and a lookahead token of `const` is #8, making our parse tree:

Since we have now matched the token, we get a new lookahead

The production for T' and a lookahead token of `*` is #5, making our parse tree:

We get another lookahead

The production for T' and a lookahead token of `+` is #6, making our parse tree:
Parsing an expression using the LL(1) parse table (continued)

The production for $E'$ and a lookahead token of $+$ is #2, making our parse tree:

We get a new lookahead

The production for $T$ and a lookahead token of id is #4, making our parse tree:

We get a new lookahead

Having reached the EOF (represented by $\$$), the productions for $T'$ and $E'$ are 6 and 3 respectively. Our parse tree is complete.
Bottom-up Parsing

• Bottom-up parsers parse a program from the leaves of a parse tree, collecting the pieces until the entire parse tree is built all the way to the root.

• Bottom-up parsers emulate pushdown automata:
  – requiring both a state machine (to keep track of what you are looking for in the grammar) and a stack (to keep track of what you have already read in the program).
  – making it fairly easy to automate the process of creating the parser
  – ensuring that all context-free grammars can be parsed by this method.

Bottom-up parsers as shift-reduce parsers

• Bottom-up parsers are frequently called shift-reduce parsers because of their two basic operations:
  – A shift involves moving pushing the current input token onto the stack and fetching the next input token.
  – A reduce involves popping all the variables that comprise the right-sentential form for a nonterminal and replacing them on the stack with the equivalent nonterminal that appears on the left-hand side of that production.
  – While shifting involve pushing and reducing involve popping, do not think of them as equivalent: a shift also involve advancing the input token stream and a reduce involves zero or more pops followed by a push.
Bottom-up Parsing as an Emulation of Pushdown Automata

- Most bottom-up parsers are table-driven, with the table encoding the necessary information about the grammar.
- The parser decides what action to perform based on the combination of current state and current input token.
- A state in the machine which the computer is emulating reflects both what the machine has already parsed and that which it is expect to see in the input token stream.
- Several parser generators have been created based on this theoretical machine, the best known of which is YACC (Yet Another Compiler Compiler), is available on many UNIX system and its public domain lookalike Bison.

LR(k) grammars

- Bottom-up grammars are referred to as LR(k) grammars:
  - The first L indicates Left-to-Right scanning.
  - The R that is second indicates Right-most derivation
  - The k indicates k lookahead characters.
- There should be no need for anything more than a single lookahead, i.e, an LR(1) grammar.
An example - a LR(0) grammar

An LR(0) grammar does not use a lookahead character to determine the action that it will take - the current token will be used to determine the state into which it will go.

Consider the following grammar:

\[
E ::= E + T \lor T \\
T ::= + F \lor - F \lor F \\
F ::= \text{id} \lor \text{const}
\]

An example - a LR(0) grammar (continued)

Let’s write out our grammar and add to it a special first production with a special start symbol $S$:

1. $S ::= E \, \$  \quad \text{(indicates that the expression is followed by EOF)}$
2. $E ::= E + T$
3. $E ::= T$
4. $T ::= + F$
5. $T ::= - F$
6. $T ::= F$
7. $F ::= \text{id}$
8. $F ::= \text{const}$
### The LR(0) parse table

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>+</th>
<th>-</th>
<th>id</th>
<th>const</th>
<th>$</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>S</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tracing LR(0) parsing**

There are 3 parsing operations:

- **Shift** - moving a token and state onto the stack (we find the state using the GOTO table).
- **Reduce n** - we pop enough items from the stack to form the right side of production n and then we push the nonterminal on its left side of production n on the stack, together with the state indicated by the GOTO table.
- **Accept** - we accept the program as completely and correctly parsed and terminate execution.
Tracing LR(0) parsing - an example

Example - the expression \(-27 + x\)

We place the state 0 and the EOF marker $ on the stack. The action for state 0 is \textit{shift}. We place the - and GOTO(0, -) = 5 on the stack.

\[
\begin{array}{c|c}
  5 & - \\
  0 & $ \\
\end{array}
\]

The action for state 5 is \textit{shift}. We place the constant on the stack together with GOTO(5, const) = 7.

\[
\begin{array}{c|c}
  7 & \text{const} \\
  5 & - \\
  0 & $ \\
\end{array}
\]

The action for state 7 is reduce by production 8. Pop the const (and state 7). Push F and GOTO(5,F) = 11

\[
\begin{array}{c|c}
  11 & F \\
  5 & - \\
  0 & $ \\
\end{array}
\]

Tracing LR(0) parsing - an example (continued)

The action for state 11 is reduce by production 5. Pop the - and F (along with states 5 and 11) and push the T together with GOTO(0,T) = 2

\[
\begin{array}{c|c}
  2 & T \\
  0 & $ \\
\end{array}
\]

The action for state 2 is reduce by production 3. Pop the T (and state 2). Push the E and GOTO(0,E) = 1.

\[
\begin{array}{c|c}
  1 & E \\
  0 & $ \\
\end{array}
\]

The action for state 1 is shift. We move the + onto the stack together with GOTO(1, +) = 8.

\[
\begin{array}{c|c}
  8 & + \\
  1 & E \\
  0 & $ \\
\end{array}
\]
The action for state 8 is shift. We move the id and GOTO(8, id) = 6 onto the stack.

The action for state 6 is reduce by production 7. We pop the id and state 6. We push F and GOTO(8, F) = 3.

The action for state 3 is reduce by production 6. We pop the F and state 3. We push T and GOTO(8, T) = 10.

The action for state 10 is reduce by production 2. We pop the T (and state10), the + (and state8) and the E (and state1). We push the E and GOTO(0,E) = 1.

The action for state 1 is shift. We push the $ and GOTO (1,E) = 12 onto the stack.

The action for state 12 is accept. The only item on the stack (excluding the $s) is E, which is the start symbol in our expression grammar.
Right sentential forms

• A right sentential form is a partially formed sentence (or program). It can contain the variables on the right-hand side of a production or phrases derived from it.

• Right sentential forms are derived from the rightmost derivation.

• Formally, if S \Rightarrow^* \beta, then \beta is a right sentential form.

Handles

• In performing a reduce operation, we must decide which variables in a right-sentential form will be popped and replaced on the stack by the nonterminal on the production’s left-hand side. These variables are collectively called the handle.

• If A \Rightarrow \beta, then \beta would be handle for the production.
Items

• An item is a production, with a dot added to it indicating how much of the production has been matched up so far.

• Example:
  
  \[ E ::= . \ E + T \]  \textit{nothing in the production has been matched yet.}  
  
  \[ E ::= E + . T \]  \textit{we have matched the E and the +}  

Shift-Reduce Conflicts

Let’s consider our original expression grammar:

\[
E ::= E + T | T \\
T ::= T * F | F \\
F ::= \text{id} | ( \ E )
\]

If we try to build an LR(0) parser, we will have a problem

Example: \( x + y * z \)
Shift-Reduce Conflicts (continued)

We have a conflict - do we shift or reduce?
Looking at the state machine by itself does not tell us.

Answer: We must use a lookahead.

The simplest version of an LR(1) parser is called SLR(1) (The “S” is for “Simplified”)
We will use the follows as a means of resolving the shift-reduce conflict.
The SLR(1) Parse Table

<table>
<thead>
<tr>
<th>state</th>
<th>+</th>
<th>*</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>S</th>
<th>E</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s4</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r3</td>
<td>s9</td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r5</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>r5</td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r6</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>r6</td>
<td>r6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>s4</td>
<td>s6</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
<td>s9</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r4</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r7</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td>r7</td>
<td>r7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>