# Compilers*

October 12, 2014

## Strongly suggested book

Before you even start reading this notes, do yourself a favor and go buy the *Dragon Book*. (http://dragonbook.stanford.edu/) It is the "bible" of compilation and covers a lot of topics very deeply and extensively.

## Contents

## 1 Overall compiler structure

See diagrams on p. 2-7, p.106 of the dragon book.

Note how an *interpreter*, instead of producing a target program as a translation, directly executes the operations specified in the source program on inputs supplied by the user. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. (This is why, for instance, Java uses Just In Time compilation improve its execution speed) An interpreter, however can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

***Disclaimer**: These notes have been prepared with the **only** purpose to help me pass the Computer Science qualifiying exam at the University of Illinois at Chicago. They are distributed as they are (including errors, typos, omissions, etc.) to help other students pass this exam (and possibly relieving them from part of the pain associated with such a process). I take **no responsibility** for the material contained in these notes (which means that you can't sue me if you don't pass the qual!) Moreover, this pdf version is distributed together with the original LaTeX (and LyX) sources hoping that someone else will improve and correct them. I mean in absolute no way to violate copyrights and/or take credit stealing the work of others. The ideas contained in these pages are **not mine** but I've just aggregated information scattered all over the internet. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

# 2 Lexical analysis (Scanning)

The main task of a lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of token is then sent to the parser for syntax analysis.

Commonly the parser calls the lexical analyzer and such a call causes the lexer to read characters from its input until it can identify the next lexeme and produce for it the next token, which is then returned to the parser. (see Fig. 3.1, p. 110)

Since the lexer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out whitespaces (blank, tab and newline) and comments. Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. If the source program uses a macro-processor, the expansion of macros may also be performed by the lexical analyzer.

## 2.1 Tokens

In many programming languages, the following classes cover most or all of the tokens:

- One token for each keyword (if, else, . . .)
- Tokens for the operators, either individually or in a class (e.g. comparison vs $=, <, \geq, \ldots$)
- One token representing all the identifiers (id)
- One or more token representing constants, such as numbers and literal strings (number, literal, . . .)
- Tokens for each punctuation symbol, such as left and right parentheses, comma and semicolon (lp, rp, . . .)

When more than one lexeme can mach a pattern, the lexical analyzer must provide subsequent phases of the compiler additional information about the particular lexeme that matched. (e.g. any number matches the number token but any number has a different value attribute) Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token. The token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the token id, where we need to associate with the token a great deal of information which is normally stored in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol table entry for that identifier.

## 2.2 Lexical errors

It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source code error. However, suppose a situation arises where the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.

The simplest recovery strategy is "*panic mode*" recovery. We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. Other possible error recovery actions are: delete one character from the remaining input; insert a missing character into the remaining input; replace a character with another character; transpose two adjacent characters. Transformations like these can be tried in an attempt to repair the input.

## 2.3 Input buffering

Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two (contiguous) buffers that are alternatively reloaded. Each buffer is of the same size $N$, where $N$ is usually the size of a disk block. Using a system read command we can read $N$characters into the buffer, rather than using one system call per character. If fewer than $N$ characters remain in the input file, then a special character, represented by eof and different from any possible character in the source program, marks the end of the source file. Two pointers to the input buffer are maintained:

- pointer `lexemeBegin`, marks the beginning of the current lexeme, whose extent we are trying to determine;
- pointer `forward` scans ahead until a pattern match is found.

When parsing starts both pointers are set to point to the first character in the input buffer. Once the next lexeme is determined, `forward` is set to the character at the lexeme's right end and, after the lexeme is recorded as an attribute value of a token returned to the parser, `lexemeBegin` is set to the same value of `forward`.

Advancing `forward` requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move `forward` to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than $N$ we should never overwrite the lexeme in its buffer before determining it.

### 2.4 DFAs

See DFAs section in "Theory of Computation" notes!

To build a lexical analyzer we first express patterns using *regular expressions*, we then convert regular expressions to *transition diagrams* (DFAs) and finally implement a program that simulates input on the DFA and decides to accept/reject the string.

## 3 Symbol tables management

*Symbol tables* are data structures that are used by the compilers to store information about source-program constructs. Such data structure should be designed to allow a compiler to find the record for each construct quickly and to store or retrieve data from that record quickly. The information is collected incrementally by the analysis phases of a compiler (front-end) and used by the synthesis phase (back-end) to generate the target code.

Entries in the symbol table contain information about an identifier such as its lexeme, its type, its position in storage, and any other relevant information. In some cases, lexical analyzers can create symbol table entries as soon as they see the characters that make up a lexeme. More often, the lexical analyzer can only return to the parser a token along with a pointer to the lexeme. only the parser, however, can decide whether to use a previously created symbol table entry or create a new one for the identifiers.

Since symbol tables typically need to support multiple declarations of the same identifier within a program we shall implement scopes by setting up separate symbol tables for each scope (*nested symbol tables*). A program block with declarations will have its own symbol table and with an entry for each declaration in the block. This approach also works for other constructs that set up scopes: for example, a class would have its own table, with an entry for each field and method.

## 4 CFGs

- See CFGs section in "Theory of Computation" notes!
- If $S \overset{+}{\Rightarrow} \alpha$, where $S$ is the start symbol of the grammar $G$, we say that $\alpha$ is a *sentential form.*
- There is a many-to-one relationship between derivations and parse trees. Every parse trees has associated with it a unique leftmost and rightmost derivation.

### 4.1 Eliminating ambiguity

Ambiguous grammars can be used sometimes together with a set of disambiguating rules that "throw away" undesired parse trees leaving only one. (e.g. the "dangling else" ambiguity can be solved with the rule "match each else with the closest unmatched then"[1])

### 4.2 Eliminating Left Recursion

A grammar is *left-recursive* if it has a nonterminal $A$ such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$.

Top down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. The following procedure takes as input a grammar $G$ with *no* cycles or $\epsilon$-productions and returns an equivalent grammar with no left recursion.

```
ELIMINATE-LEFT-RECURSION($G$)
1  arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$
2  for $j = 1$ to $i - 1$
3      replace each production of the form $A_i \to A_j\gamma$ by the
       productions $A_i \to \delta_1\gamma|\delta_2\gamma|\ldots|\delta_k\gamma$, where
       $A_j \to \delta_1|\delta_2|\ldots|\delta_k$ are all current $A_j$-productions
4  eliminate the immediate left recursion among the $A_j$-productions
```

### 4.3 Left factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive (or top-down) parsing. The following procedure takes as input a grammar $G$ and returns an equivalent left-factored grammar.

```
LEFT-FACTORING($G$)
1  for each nonterminal $A$
2      find the longest common prefix $\alpha$ common to two (or more) of its alternatives
3      if $\alpha \neq \epsilon$
4          replace all $A \to \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma$ with
5          $A \to \alpha A'|\gamma$ and $A' \to \beta_1|\beta_2|\ldots|\beta_n$
```

### 4.4 Non-CFG constructs

A few syntactic constructs found in typical programming languages cannot be specified using grammars alone. Typical examples are:

- the language that abstracts the problem of checking that identifiers are declared before they are used in a program;
- the language that abstracts the problem of checking that the number of formal parameters in the declaration of a function agrees with the number of actual parameters in a use of the function.

These construct are checked in later phases of compilation and in particular using semantic analyzer.

## 5 Parsing (Syntax analysis)

### 5.1 Recursive-Descent Parsing

A *recursive-decent* parsing program consists of a series of routines, one for each non terminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. General recursive-descent parsing may require backtracking; that is, it may require repeated scans over the input.

[1]Another important example of ambiguity elimination through rules is the use of associativity and precedence to resolve conflicts in arithmetic expressions.

## 5.2 First and Follow

### 5.2.1 First

To compute First($X$) for all grammar symbols $X$, apply the following rules until no more terminals or $\epsilon$ can be added to any First set.

1. If $X$ is a terminal, then the First($X$)= $X$.
2. If $X$ is a nonterminal and $X \to Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place $a$ in First($X$) if for some $i$, $a$ is in First($Y_i$) and $\epsilon$ is in all of First($Y_1$),...,First($Y_{i-1}$); that is, $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. If $\epsilon$ is in First($Y_j$) for all $j = 1, 2, \ldots, k$, then add $\epsilon$ to First($X$).
3. If $X \to \epsilon$ is a production, then add $\epsilon$ to First($X$).

### 5.2.2 Follow

To compute Follow($A$) for all nonterminals $A$, apply the following rules until nothing can be added to any Follow set.

1. Place \$ in Follow($S$), where $S$ is the start symbol, and \$ is the input right end marker.
2. If there is a production $A \to \alpha B \beta$, then everything in First($\beta$) except $\epsilon$ is in Follow($B$).
3. If there is a production $A \to \alpha B$ (or a production $A \to \alpha B \beta$ where First($\beta$) contains $\epsilon$) then everything is Follow($A$) is in Follow($B$)

## 5.3 Predictive Parsing - LL(1)

*Predictive parsers*, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first "L" stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

A grammar $G$ is LL(1) if and only if whenever $A \to \alpha | \beta$ are two distinct productions of $G$, the following conditions hold:

1. For no terminal $a$ do both $\alpha$ and $\beta$ derive strings beginning with $a$.
2. At most one of a $\alpha$ or $\beta$ can derive the empty string.
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then $\alpha$ does not derive any string beginning with terminal in Follow($A$). Likewise, if $\alpha \overset{*}{\Rightarrow} \epsilon$, then $\beta$ does not derive any string beginning with terminal in Follow($A$).

The first two conditions are equivalent to the statement that First($\alpha$) and First($\beta$) are disjoint sets. The third condition is equivalent to stating that, if $\epsilon$ is in First($\beta$) , then First($\alpha$) and Follow($A$) are disjoint sets, and likewise if $\epsilon$ is in First($\alpha$).

### 5.3.1 Parsing table construction

The following algorithm takes a LL(1) grammar $G$ and returns the parsing table, $M[N, t]$ where $N$ is a nonterminal and $t$ a terminal, necessary to build the parser.

1. For each production $A \to \alpha$ of the grammar, do the following:
   (a) For each terminal $a$ in First($A$), add $A \to \alpha$ to $M[A, a]$.
   (b) If $\epsilon$ is in First($\alpha$), then for each terminal $b$ in Follow($A$), add $A \to \alpha$ to $M[A, b]$. If $\epsilon$ is in First($\alpha$) and \$ is in Follow($A$), add $A \to \alpha$ to $M[A, \$]$ as well.

### 5.3.2 Parser

The following algorithm takes as input a string $w$ and a parsing table $M$ for grammar $G$ and returns a leftmost derivation of $w$, if $w \in L(G)$, or an error, otherwise.

Initially the parser is in a configuration with $w\$$ in the input buffer and $S\$$ on the stack.

```
PREDICTIVE-PARSING($M, w$)
ip = w[1]
X = S
while $X \neq \$$
    if $X == ip$
        pop the stack and advance $ip$
    elseif $X$ is a terminal
        error
    elseif $M[X, a]$ is an error entry
        error
    elseif $M[X, a] == X \to Y_1 Y_2 \dots Y_k$
        output the production
        pop the stack
        push $Y_k, Y_{k-1}, \dots, Y_1$ onto the stack
```

## 5.4 Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which, during left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string $\beta$ to the head of the appropriate production.

The following algorithm, takes as input a string $w$ and a grammar $G$ and returns accept if $w \in L(G)$, and an error, otherwise. Initially the parser is in a configuration with $w\$$ in the input buffer and an empty stack (\$ is also used to mark the bottom of the bottom of the stack)

```
SHITF-REDUCE-PARSING($G, w$)
ip = w[1]
while $stack.top \neq S$ or $ip \neq \$$
    if $stack.top == \alpha$ such that $A \to \alpha$ is in $G$
        REDUCE
    else SHIFT
if $stack.top == S$
    ACCEPT
else ERROR
```

The algorithms relies on four procedures whose behavior is intuitively described below.

**Shift:** Shifts the next symbol onto the top of the stack.

**Reduce:** The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.

**Accept:** Announce successful completion of parsing.

**Error:** Discover a syntax error and call an error recovery routine.

### 5.4.1 Conflicts

**Shift-reduce conflict:** knowing the entire stack content and the next input symbol, the parser cannot decide whether to shift or to reduce. (e.g. $G : S \to 1S, S \to 1$, $w =$ "111", after shifting the first "1" the parser doesn't know if it has to reduce with $S \to 1$ or shift again)

**Reduce-reduce conflict:** knowing the entire stack content and the next input symbol, the parser cannot decide which of several reductions to make. (e.g. $G : S \to A1, S \to B2, A \to 1, B \to 1$, $w =$ "111", after shifting the first "1" the parser doesn't know if it has to reduce with $A \to 1$ or $B \to 1$)

It can be shown that when these conflict happen the grammar is not an LR(0) grammar. Both conflicts can be solved by Simple LR parsers.

## 5.5 Simple LR (SLR) Parsing

The most prevalent type of bottom-up parsers is based on a concept called LR(k) parsing: the "L" is for left-to-right scanning of the input, the "R" for constructing the rightmost derivation in reverse, and the "k" for the number of input symbols of lookahead that are used in making parsing decisions. The SLR or *LR(0)* parser is the simplest of this family.

### 5.5.1 LR(0) automaton

The first step to build a LR(0) parser is to build the so called *LR(0) automaton*, which is simply a DFA whose set of states is called canonical LR(0) collection. In order to so we first compute the *augmented grammar* $G' = (S' \to S) \cup G$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.

The following algorithm takes as input $G'$ and returns the *canonical LR(0) collection* $C$ using the procedures Closure and Goto described below.

```
LR0-Automata-Build($G'$)
1  $C$ = Closure($\{S' \to \cdot S\}$)
2  repeat
3      for each set of items $I \in C$
4          for each grammar symbol $X$ in $I$
5              if Goto($I, X$) is not empty and not in $C$
6                  $C = C \cup$ Goto($I, X$)
7  until no new sets of items are added to $C$
8  return C
```

The following algorithm takes as input $G'$ and returns the set of items $A$ which represents the outcome of the transition function Goto for the LR(0) automata from state $I$ with input $X$.

```
Goto($G', I, X$)
1  if $\{A \to \alpha \cdot X\beta\} \in I$
2      return Closure($G', \{A \to \alpha X \cdot \beta\}$)
```

The following procedure takes as input $G'$ and returns the closure of the items set $I$.

```
Closure($G', I$)
1  $J = I$
2  repeat
3      for each item $A \to \alpha \cdot B\beta$ in $J$
4          for each production $B \to \gamma$ in $G$
5              if $B \to \cdot\gamma$ is not in $J$
6                  add $B \to \cdot\gamma$ to $J$
7  until no more items are added to $J$
8  return $J$
```

Note how, by construction, each state of the LR(0) automaton has a corresponding grammar symbol. States correspond to sets of items, and *all transitions to a particular state must be for the same grammar symbol.* Thus each state, except the start state 0, has a unique grammar symbol associated with it.

### 5.5.2 Parsing tables

The next step necessary to obtain a SLR parser for a grammar $G'$ is to build *action* and *goto* tables. If, after creating the tables, any conflicting action result, we say that the grammar is not SLR(1).

**Action table** The value of $Action[i, a]$ where $i$ is a state of the LR(0) automaton and $a$ is a terminal (or $) are determined as follows:

1. If $\{A \rightarrow \alpha \cdot a\beta\} \in I_i$ and $\text{GOTO}(I_i, a) = I_j$ then $Action[i, a] =$ "shift($j$)".
2. If $\{A \rightarrow \alpha\cdot\} \in I_i$ and $A \neq S'$ then $\forall a \in \text{FOLLOW}(A), Action[i, a] =$ "reduce($\{A \rightarrow \alpha\}$)".
3. If $\{S \rightarrow S'\cdot\} \in I_i$ then $Action[i, \$] =$ "accept".
4. Otherwise $Action[i, a] =$ "error".

**Goto table** Set $Goto[i, A] = j$, where $i$ is a state of the LR(0) automaton, $A$ is a nonterminal and $j$ is the state corresponding to $\text{GOTO}(I_i, A) = I_j$.

### 5.5.3 LR-Parsing algorithm

The following algorithms takes as input an input string $w\$$ and the *Action* and *Goto* parsing tables for a grammar $G$ and returns the reduction steps of a bottom-up parse for $w$, if $w \in L(G)$, an error, otherwise.

Note how the initial state of the parser is the one constructed from the set of items containing $\{S \rightarrow \cdot S'\}$ and labeled as $s_0$. We assume $s$ to be the state on top of the stack.

```
LR-PARSING($G'$, $I$)
 1  $s = s_0$
 2  $a = w[1]$
 3  while 1
 4      if $Action[s, a] == \text{shift}(t)$
 5          push $t$ onto the stack
 6          let $a$ be the next input symbol
 7      elseif $Action[s, a] == \text{reduce}(\{A \rightarrow \beta\})$
 8          pop $|\beta|$ symbols off the stack, which implies $s = r$
 9          push $\text{GOTO}(s, A)$ onto the stack
10          print($A \rightarrow \beta$)
11      elseif $Action[s, a] == \text{accept}$
12          return
13      else error
```

Note that *all* LR parsers behave in this fashion; the only difference between one LR parser and another is the information stored in the *Action* and *Goto* parsing tables.

## 5.6 LR(1) Parsing

This method is called the *canonical-LR* or just *LR* method and makes full use of the lookahead symbol(s). The set of items for this method is large and is called the *LR(1) items*.

### 5.6.1 LR(1) automaton

The general form of an *LR(1) item* becomes $[A \rightarrow \alpha \cdot \beta, a]$ where $A \rightarrow \alpha\beta$ is a production and $a$ is a terminal or the right end marker $. The second element is called the *lookahead*. The lookahead affects only the items where $\beta$ is $\epsilon$ or have the form $[A \rightarrow \alpha\cdot, a]$. In this case such an item calls for a reduction only if the next input symbol is $a$.

The method for building the collection of sets of a valid LR(1) items is essentially the same as the one for building the canonical collection of sets of LR(0) items. We need only to modify the following procedures.

```
LR0-AUTOMATA-BUILD($G'$)
1  $C = $ CLOSURE($\{[S' \to \cdot S, \$]\}$)
2  repeat
3      for each set of items $I \in C$
4          for each grammar symbol $X$ in $I$
5              if GOTO($I, X$) is not empty and not in $C$
6                  $C = C \cup$ GOTO($I, X$)
7  until no new sets of items are added to $C$
8  return C
```

```
GOTO($G', I, X$)
1  $J = \emptyset$
2  for each item $[A \to \alpha \cdot X\beta, a] \in I$
3      add item $[A \to \alpha X \cdot \beta, a]$ to set $J$
4  return CLOSURE($J$)
```

```
CLOSURE($G', I$)
1  $J = I$
2  repeat
3      for each item $[A \to \alpha \cdot B\beta, a]$ in $J$
4          for each production $B \to \gamma$ in $G'$
5              for each terminal $b$ in FIRST($\beta a$)
6                  add $[B \to \cdot\gamma, b]$ to $J$
7  until no more items are added to $J$
8  return $J$
```

### 5.6.2 Parsing tables

Again, we just modify the procedure to build the *Action* table to adapt to LR(1) parsing.

**Action table** The value of $Action[i, a]$ where $i$ is a state of the LR(1) automaton and $a$ and $b$ are terminals (or \$) are determined as follows:

1. If $[A \to \alpha \cdot a\beta, b]$ is in $I_i$ and GOTO$(I_i, a) = I_j$ then set $Action[i, a]$ to "shift $j$". Here $a$ must be a terminal.
2. If $[A \to \alpha\cdot, a]$ is in $I_i$, $A \neq S'$, then set $Action[i, a]$ to "reduce $A \to \alpha$".
3. If $[S' \to S\cdot, \$]$ is in $I_i$ then set $Action[i, a]$ to "accept".
4. Otherwise $Action[i, a] =$ "error".

If any conflicting actions result from the above rules, we say the grammar is not LR(1) and we can't produce a parser.

**Goto table** Same as LR(0) automaton.

### 5.6.3 LR-Parsing algorithm

As said before the parsing algorithm is the same one used in SLR parsers.

## 5.7 LALR(1) Parsing

The *lookahead-LR* or *LALR* method is based on the LR(0) set of items and has many fewer states than typical parsers based on LR(1) items. By carefully introduction lookaheads into the LR(0) items, we can handle many more grammars than with SLR, and build parsing tables that are no bigger than SLR tables. LALR is the method of choice in most situations.

## 5.8 Error handling

It is desirable for a parser to report the presence of errors clearly and accurately, recover from each error quickly enough to detect subsequent errors and add a minimal overhead to the processing of correct programs.

The following are four of the most useful error recovery strategies.

**Panic-Mode Recovery** The parser discards input symbols one at the time until one of a designated set of s*ynchronizing tokens* is found. These are usually delimiters, such as ';' and '}'.

**Phrase-Level Recovery** On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. (e.g. replace ',' with ';') The major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

**Error Productions** By anticipating common errors that might be encountered, we can augment the grammar for the language with productions that generate the erroneous constructs.

**Global Correction** Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

# 6 Syntax-directed Translation

Syntax-directed Translation is done by attaching rules or program fragments to productions in a grammar. There are two different strategies to accomplish this.

## 6.1 Attributes and Syntax Directed Definitions

An *attribute* is any quantity associated with a programming construct. Since we use grammar symbols to represent programming constructs we extend the notion of attributes from constructs to the symbols that represent them.

A *syntax-directed definition* (*SDD*) associates with each grammar symbol, a set of attributes, and with each production, a set of *semantic rules* for computing the values of the attributes associated with the symbols appearing in the production.

An attribute is said to be *synthesized* if its value at a parse-tree node $N$ is determined from attribute values at the children of $N$ and $N$ itself. Synthesized attributes have the desirable property that they can be evaluated during a single bottom-up traversal of the *annotated parse tree.*

An attribute is said to be *inherited* if its value at a parse-tree node $N$ is defined only in terms of attribute values at $N$'s parent, $N$ itself and $N$'s siblings.

Terminals can have synthesized attributes, but not inherited attributes. Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no rules in the SDD itself for computing the value of an attribute for a terminal.

### 6.1.1 Evaluation orders for SDDs

In order to decide in which order to evaluate the attributes we need to build a *dependency grap*h. If the dependency graph has an edge from node $M$ to node $N$, then the attributes corresponding to $M$ must be evaluated before the attributes of $N$. The *topological sort* of the dependency graph leads to a sequential ordering of evaluation for the attributes. Clearly, if there is any cycle in the dependency graph it is impossible to find such a topological order. (see p. 338)

**S-attributed SDDs** An SDD that involves only synthesized attributes is called *S-attributed.* In such SDD, each rule computes an attribute for the nonterminal at the head of the production from attributes taken from the body of the production. An S-attributed SDD can be implemented naturally in conjunction with an LR parser, since a bottom-up parser corresponds to a postorder traversal. This strategy can be used to store the evaluated values for synthesized attributes on the stack during LR parsing, without creating the tree nodes explicitly.

**L-attributed SDDs** An SDD is called *L-attributed* if every attribute is either synthesized or, if it is inherited, it depends only on inherited attributes of its parent and on (any) attributes of siblings to its left. More precisely, suppose that there is a production $A \rightarrow X_1 X_2 \ldots X_n$, and that there is an inherited attribute $X_i.\,a$ computed by a rule associated with this production. Then the rule may only use:

1. Inherited attributes associated with $A$
2. Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of $X_i$.
3. Inherited or synthesized attributes associated with this occurrence of $X_i$ itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this $X_i$.

### 6.1.2 Semantic rules and side effects

Side effects are other actions, like printing the value of an attribute, that are executed during Syntax Directed Translations.

An SDD without side effects is sometimes called *attribute grammar.* The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

Usually we allow controlled side effects; that is, we permit side effects that do not constrain attribute evaluation.

### 6.1.3 Abstract syntax tree construction using SDDs

A typical application of SDDs is the construction of syntax trees, which can be then used as an internal representation usually thought the *.ast* attribute. Clearly, to complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree.

## 6.2 Semantic actions and Syntax Directed Translation Schemes (SDT)

A *translation scheme* is a notation for attaching program fragments to the production of a grammar. The program fragments, called *semantic actions*, are executed when the production is used during syntax analysis.

The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body. When drawing a parse tree for a translation scheme, we indicate an action by constructing an extra child for it, connected by a dashed line in the node that corresponds to the head of the production. Any STD can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order; that is, a *preorder traversal.*

Typically, SDTs are implemented during parsing, without building a parse tree. SDT that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals* in place of each embedded action; each marker $M$ has only one production, $M \rightarrow \epsilon$. If the grammar with the marker nonterminals can be parsed by a given method (LL, LR,...), then the SDT can be implemented during parsing.

## 6.3 Implementing SDDs using SDTs

SDTs are often used to implement SDDs in real compilers.

#### 6.3.1 SDTs for LR-parsable grammars and S-attributed SDD

We can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDTs with all actions at the ends of the production body are called *postfix SDTs*.

#### 6.3.2 SDTs for LL-parsable grammars and L-attributed SDD

The rules for turning an L-attributed SDD, whose underlying grammar is LL-parsable, into an SDT are as follow:

1. Embed the action that computes the inherited attributes for a nonterminal $A$ immediately before that occurrence of $A$ in the body of the production. If several inherited attributes for $A$ depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.
2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production

#### 6.3.3 L-attributed SDDs and recursive-descent parsing

TODO p.338

#### 6.3.4 L-attributed SDDs and LL parsing

TODO p.343

#### 6.3.5 L-attributed SDDs and LR parsing

TODO p.348

# 7 Semantic Analysis and Intermediate representation

## 7.1 Type checking

TODO

## 7.2 Intermediate representation

The purpose of a compiler frontend is to construct an intermediate representation of the source program from which the back end generates the target program. The two most important intermediate representations are:

**Trees:** Including *parse trees* and *Abstract Syntax Trees* ($AST$)[2]. During parsing, syntax-tree nodes are created to represent significant programming constructs. As analysis proceeds, information is added to the nodes in the form of attributes associated with nodes.

**Linear representations:** especially *three-address code.* This is just a sequence of elementary program instructions, such as the addition of two values.

It is possible that a compiler will construct a syntax tree at the same time it emits three-address codes. However, it is common for a compiler to emit three-address code while the parser "goes through the motions" of constructing a syntax tree" without actually constructing the complete tree data structure. Rather, the compiler stores nodes and their attributes needed for semantic analysis or other purposes, along with the data structure used for parsing (i.e. the parse tree). By doing so, those parts of the syntax tree that are needed to construct the three-address code are available when needed, but disappear when no longer needed.

[2]Syntax trees resemble parse trees to an extent; however, in the syntax tree the internal node represent programming constructs while in the parse tree, the interior nodes represent nonterminals. Many nonterminals of a grammar represent programming constructs, but others are "helpers" of some sort. (e.g. see Fig. 2.5, p. 47 and Fig.2.22, p. 70) In the syntax tree, these helpers typically are not needed and hence dropped.

# 8 Code generation

The code generator takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent (and correct) target program. The problem of generating an optimal target program for a given source program is undecidable, and many of the subproblems encountered during code generation such as register allocation are computationally intractable. The three main tasks of a code generator are:

- *Instruction selection* The nature of the instruction set of the target architecture has a strong effect on the difficulty of this task. Also, if we don't care about the efficiency of the target program instruction selection is straightforward: on most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations.
- *Register allocation and assignment* Register allocation is the process of selecting the set of variables that will reside in registers at each point in the problem. Register assignment is the process of picking the specific register that a variable will reside in. This task is NP-complete.
- *Instruction ordering* The order in which computations are performed can affect the efficiency of the target code. Picking an optimal order in the general case is a difficult NP-complete problem.

## 8.1 Basic blocks and flow graph

In order to represent three-address instructions as a flow graph we need to first partition them into basic blocks. Basic blocks are maximal sequences of consecutive three-address instructions with the property that:

- The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
- Control will leave the block without halting or branching, except possibly at the last instruction in the block.

The following algorithm accomplishes such a task:

1. Find all *leaders*, that is, the first instructions in some basic block. The rules for finding the leaders are:
   (a) The first three-address instruction in the intermediate code is a leader.
   (b) Any instruction that is the target of a conditional or unconditional jump is a leader.
   (c) Any instruction that immediately follows a conditional or unconditional jump is a leader.
   (d) The instruction just pass the end of the intermediate program is not included as a leader.
2. Find the basic block: for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

Then the basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which others. There is an edge from block $B$ to block $C$ if and only if it is possible for the first instruction in block $C$ to immediately follow the last instruction in block $B$. There are two ways that such and edge could be justified:

- There is a conditional or unconditional jump from the end of $B$ to the beginning of $C$.
- $C$ immediately follows $B$ in the original order of the three-address instructions, and $B$ does not end in an unconditional jump.

Often we add two nodes, called *entry* and *exit*, that do not correspond to executable intermediate instructions.

## 8.2 Liveliness and next-use

The following algorithm determines the liveliness and next-use for each statement in a basic block $B$.

We start at the last statement in $B$ and scan backwards to the beginning of $B$. At each statement $i : x = y\texttt{op}z$, we do the following:

1. Attach to statement $i$ the information currently found in the symbol table regarding the next use and liveness of $x$, $y$ and $z$.
2. In the symbol table, set $x$ to "not live" and "no next use"
3. In the symbol table, set $y$ and $z$ to live and the next uses of $y$ and $z$ to $i$.

Here we have used `op` as a symbol representing an operator. If the three-address statement $i$ is o the form $x = \texttt{op}y$ or $x = y$, the steps are the same as above, ignoring $z$.