Compilers

2018A7PS0193P

April 18, 2021

Chapter 1

Process of Translation

The process of compilers converting from source code (the high level language) to target code (the machine language) is known as translation. It consists of many steps, which are described below.

1.1 Lexical Analysis

Lexical analysis follows the following steps:

1. Identify the valid set of characters in the language
2. Break the sequence of characters into appropriate words or tokens (keywords, numbers, operators, etc.)
3. Find out whether these tokens are valid or not.

The key goal of the lexical analyzer is to break a sentence into a series of words/tokens. These breaks are generally done via certain separators. The tokens are recognized via some rules encoded into a Finite State Machine.

During the lexical phase, we may experience the following lexical errors:

- Occurrence of illegal characters
- Exceeding the length of identifier

The output of the lexical analyzer will be a sequence of tokens and their "type", which signifies whether it is a identifier, operator, etc.

1.2 Syntax Analysis

The syntax analysis takes the sequence of tokens as an input, and generates a parse tree. In case the syntax is not correct according to the grammar rules, it flags a syntactical error. This is modelled using Context Free Grammars that will be recognized using PDAs or Table Driven processes.

1.3 Semantic Analysis

Semantic analysis takes the parse tree as an input, and outputs a disambiguated parse tree. It performs the following:

- Check Semantics
- Error reporting (types, etc.)
- Disambiguate overloaded operators (meaning of operators depends on operands)
- Type coercion (type casting)
- Uniqueness checking (redeclaration of variables)

As such, the disambiguated parse tree gives us an unambiguous representation of the parse tree.

The phases mentioned till now comprise the **front end** of the compiler, where the source code is handled. After this, the compiler works on generating the target code.

1.4 Code optimization

This is an optional phase that modifies the programs to run faster and consume less resources like memory, registers, etc. However, it will not change the representation of the program.

Some examples of machine independent code optimization done is:

- **Common sub-expression elimination:** The compiler searches for instances of identical expressions and analyzes whether it is worthwhile to replace them with a single variable holding the computer value.
- **Copy Propagation:** The compiler replaces the occurrences of targets of direct assignments with their value. For instance, if we had the code `y=x; z = 3+y;`, copy propagation would yield `z=3+x;`

- **Dead code elimination:** The compiler removes code which does not affect the program results, including unreachable code and unused variables.
- **Code Motion:** The compiler moves statements and expressions out of the body of a loop if they are loop-invariant, i.e., doing so does not affect program semantics.
- **Strength Reduction:** The compiler replaces expensive operations with equivalent but less expensive operations.
- **Constant Folding:** The compiler recognizes and evaluates constant expressions at compile time instead of runtime.

1.5 Code Generation

This is the process of mapping from source level abstractions (identifiers, values, etc) to target machine abstractions (registors, memory, etc.). This is a two step process - initially, intermediate code gets generated from the disambiguated parse tree, which is used to generate the final machine code.

During code generation, we have to do the following:

- Map identifiers to locations (memory or registers)
- Map source code operators to opcodes or sequences of opcodes.
- Transform conditionals and iterations to a test/jump or compare instructions
- We use layout parameter passing protocols - the locations for parameters, return values, etc.

1.6 Post Translation Optimizations

Unlike in the code optimization phase where we perform machine independent code optimizations, this does machine-dependent code optimizations. This is an optional phase as well, where we may remove unneeded operations or rearrange to prevent hazards. It is a flexible phase, and may occur at any time in the back-end of the compiler.

1.7 Symbol Table

The symbol table contains information required about the source program identifiers during compilation, including:

- Category of variable
- Data type
- Quantity stored in structure
- Scope information
- Address in Memory

The symbol table must be present in every phase of the compiler, and is used in all the phases to get information about the identifiers.

1.8 Advantages and Disadvantages of Compilers

The advantages of compilers are:

- Highly modular in nature
- It is retargetable. This means that if there is a single language and multiple machines, then we can use the same front end.
- Source code and machine independent optimizations are possible.

The limitations of the compiler are:

- Design of programming languages has a huge effect on the performance of compilers.
- Lots of work is repeatable. For S languages and M machines, $S \cdot M$ compilers are needed. This is known as the $S * M$ problem of compilers.

The $S * M$ problem is generally solved by introducing some common intermediate language, called the **Universal Intermediate Language Generator**. Some common machine independent intermediate code generation techniques are:

- Postfix Notation
- Three Address code
- Syntax tree

- Directed Acyclic Graph

Chapter 2

Lexical Analysis

2.1 Functions of the Lexical Analyzer

The lexical analyzer performs the following functions:

- Take high level language as input and output a sequence of tokens
- It generally cleans the code, by stripping off blanks, tabs newlines and comments.
- Keeps track of the line numbers for associated error messages

The lexical analyzer is modelled using regular expressions. As such, it's implementation is done with a DFA. An example of one rule is $L \cdot (L + D)^*$, where L refers to a letter and D refers to a digit.

Definition 2.1.1. A token is a string of characters which logically belong together, e.g. keywords, number, identifiers, etc.

Definition 2.1.2. A pattern is the set of strings for which the same token is produced.

Definition 2.1.3. A lexeme is a sequence of characters matched by a pattern to form the corresponding token.

Now that we understand the definitions, we can see what the lexical analyzer actually does - it transforms strings to the token and passes the lexeme as it's corresponding attribute. For instance, the integer 43 would become `<num,43>`.

2.2 Working of the Lexical Analyzer

The lexical analyzer reads the character one by one from the source code into the lexeme. When it reaches a separator, it assigns a token to the lexeme based on certain rules, and continues to read the characters once more.

However, reading the lexemes character by character is slow, and involves many IO operations. This is done from a buffer instead of directly from the file. Moreover, the prefix of a lexeme is often not enough to determine the token - think of the lexemes `=` and `==`. We instead use a lookahead pointer to determine the appropriate token for a lexeme, and then push back the characters that we do not need in the current lexeme.

2.3 Symbol Table and the Lexical Analyzer

The lexical analyzer also interfaces with the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme to the symbol table. Sometimes, information regarding the token of a particular lexeme may also be store in the symbol table. As such, the symbol table must implement the following operations:

1. `insert(s,t)` : Save lexeme `s` and token `t` and return pointer.
2. `lookup(s)` : return the index of entry for lexeme `s` or `'0'` if `s` is not found.

To make the symbol table space efficient, we save lexemes in some separate memory, and instead store pointers to the lexemes in the symbol table.

The rule for identifying an identifier and a keyword is generally the same. To be able to tokenize the identifiers and keywords separately, we initialize the symbol table with the list of keywords, say, by calling `insert("if",keyword)`.

2.4 Challenges in Development of Lexical Analyzer

- **Free vs Fixed Lexemes** : A language could specify that lexemes must be in a free or a fixed format. For instance, in a free format, code could look like this.

```
flag = flag
* 6;
```

But in the case of fixed format, this must be entirely in one line. An example of a fixed format language is Python, while a free format language is C.

- **Whitespaces** : How do we deal with whitespaces? Some languages ignore whitespaces until a separator is reached (or interpret contextually), while some languages consider the whitespaces as separators themselves. The former is much more complicated to implement than the latter.
- **Maximal Munch** : The principle of maximal munch directs the lexical analyzer to consume as much available input as possible while creating a construct. This allows us to deal with lexemes like `iff`, and correctly assign it as a identifier rather than the keyword `if`.

2.5 Techniques for specifying tokens

Definition 2.5.1. Consider R_i is a regular expression and N_i is a unique name, then a regular definition is a series of definitions of the following form

$$N_1 \to R_1$$

$$N_2 \to R_2$$

$$...$$

$$N_n \to R_n$$

where each R_i is a regular expression over $\sum \cup \{N_1, N_2, ...N_n\}$.

Hence, by assigning a special name N_i to the regular expression R_i, we are in effect defining macros, that remove redundancy in later parts.

The following is an example regular definition for identifiers:

$$\text{Alphabet} \to A|B|C|...Z|a|b|c|...|z$$

$$\text{Digit} \to 0|1|2|...|9$$

$$\text{Identifier} \to \text{Alphabet}(\text{Alphabet}|\text{Digit})^*$$

This too comes with its own challenges. Regular expressions often fail when identifying the appropriate token, and may pass the invalid tokens to the subsequent translation phases of

the compiler (how?). They are only language specifications. Tokenization is a implementation problem.

Tokenization can be done via the following steps:

1. Construct regular expressions for lexemes of each token
2. Construct R matching all lexemes of tokens, so $R = R_1 + R_2 + ...$, in some well defined precedence order.
3. Consider the input stream to be $S = s_1 s_2 ... s_n$. For $i \in [1, n]$, verify whether $s_1 ... s_i \in L(R)$.
4. If $s_1 ... s_i \in L(R) \implies s_1 .. s_i \in L(R_x)$ for some x. We choose the smallest x to be the class of $s_1 .. s_i$.
5. Discard the tokenized input and go back to step 3.

The procedure gives preference to tokens specified earlier using regular expressions. If $s_1 .. s_i \in L(R)$ and $s_1 ... s_j \in L(R)$, we choose the longest prefix, in accordance with the principle of maximal munch.

To implement our regular definitions and recognize tokens, we use **transition diagrams**. They are shown diagrammatically in the same way as Finite Automata. Transitions can be labelled with a symbol, a group of symbols, or regular definitions. A few states may be **retracting states** that indicates that the lexeme does not include the symbol that can bring us to the accepting state.

Sometimes, we may want to push back extra characters into the token stream (think of `>` and `>=`, we may want to push back the extra character read if the token is `>`). We mark those states with a `*`, to show that we must push back extra characters.

Let us consider the example of hexadecimal and octal constant. The regular definition would be:

$$hex \rightarrow 0|1|2|...|9|A|B|C|...F$$

$$oct \rightarrow 0|1|2|...|7$$

$$Qualifier \rightarrow u|U|l|L$$

$$OctalConstant \rightarrow 0oct^{+}(Qualifier|\epsilon)$$

$$HexadecimalConstant \rightarrow 0(x|X)hex^{+}(Qualifier|\epsilon)$$

The transition diagram for the given regular definition is given in Fig 1.

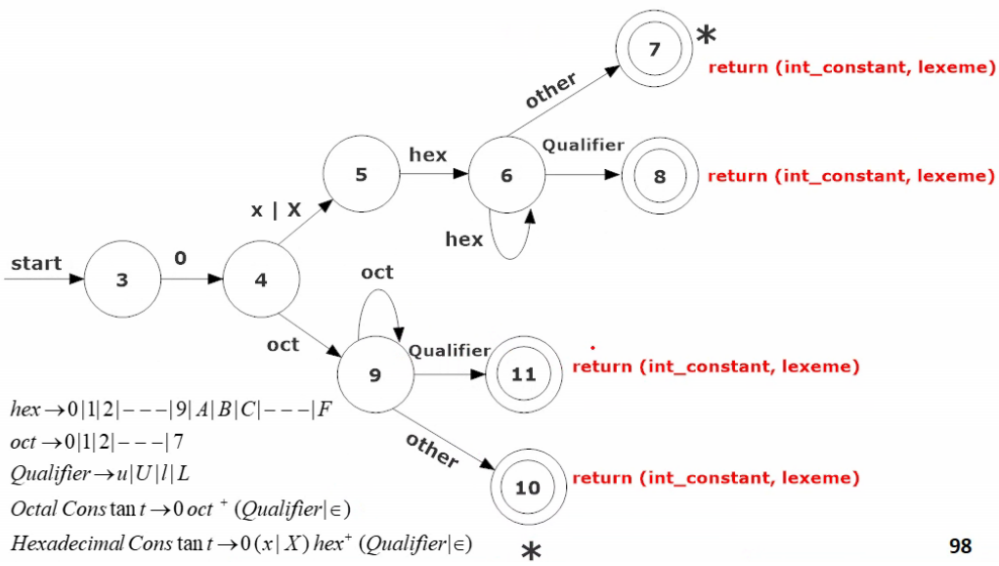

Figure 2.1: Transition diagram for the Hex and Octal constants

The retracting states are not given. If we get a "bad character", we should report it as a lexical error.

Let us consider another example of a generalized expression for unsigned numbers. The regular definition is:

$$Digit \rightarrow 0|1|2|...|9$$

$$Digits \rightarrow Digit^{+}$$

$$Fraction \rightarrow .Digits|\epsilon$$

$$Exponent \rightarrow (E(+|-|\epsilon)Digits)|\epsilon$$

$$Number \rightarrow Digits \cdot Fraction \cdot Exponent$$

The resulting transition diagram from this is in Fig 2.

The issue here is this often introduces a lot of complexity. Instead, we may add split this into multiple transition diagrams, to improve implementation complexity. This also is seen empirically to give better speeds. These multiple transition diagrams can be appropriately combined to generate a lexical analyzer.

The matching process should always start with some transition diagram. If failure occurs in one transition diagram, we retract the forward pointer to the start state, and activate the

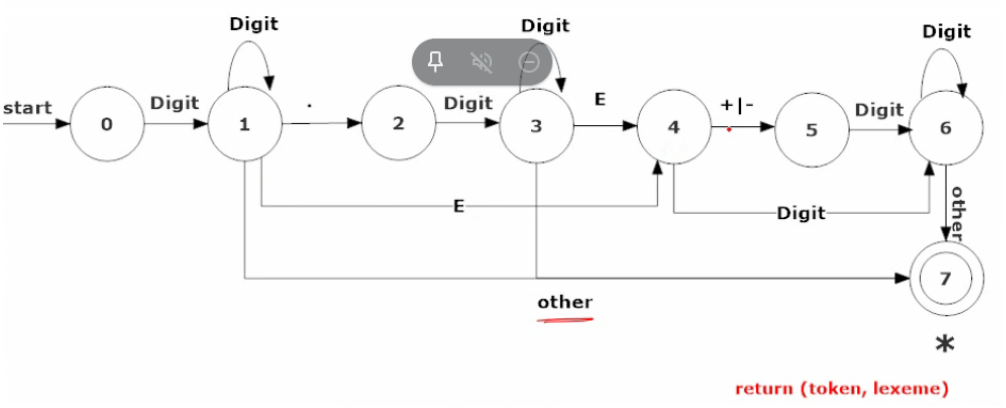

Figure 2.2: Transition Diagram for Unsigned Numbers

next transition diagram. If there is failure in all the transition diagrams, we throw a lexical error.

Chapter 3

Syntax Analysis

3.1 Grammar Rules

Regular definitions are not enough for syntax analysis, since languages are generally not regular. Instead, we use Context Free Grammars to model our language, given by $G = \{T, N, P, S\}$, where

- T is the set of tokens (or terminals)
- N is the set of non-terminals
- P is the set of production rules
- S is the start symbol.

However, just creating the CFG is not the end of our problems - we must choose a method of derivation (left vs right), choose the right non-terminal to expand, choose the right production rule and maintain some precedence order.

When we replace the leftmost non-terminal in any sentential form of the grammar, it is called **leftmost derivation**. Choosing the rightmost non-terminal makes it **rightmost derivation**. If a grammar generates more than one leftmost/rightmost derivation of a string, it is said to be **ambiguous**.

The end goal of our syntax analysis is to create the **parse tree** - where the root is a start symbol, internal nodes are non-terminals and leaf nodes are labelled by tokens. An ambiguous grammar can have more than one parse tree.

Unfortunately, no well defined algorithm exists to remove ambiguity in a grammar. However,

we can manually convert any ambiguous grammar into an unambiguous grammar. We can do this by enforcing associativity and precedence rules. This is pretty well covered in PPL and TOC.

Remember that right associativity is created via right recursion ($N \rightarrow \alpha N$) and left associativity is created via left recursion ($N \rightarrow N\alpha$)

In programming languages, ambiguity can come in many forms. For instance, consider the sentence `if A then if B then C1 else C2`. This could be:

```
if A then
{
  if B then
    C1
  else
    C2
}
```

or:

```
if A then
{
  if B then
    C1
}
else
  C2
```

Since there is ambiguity in its meaning, we must resolve it. One way to do this is to relate an `if` with the closest unrelated `else` (the second interpretation). Another way is to match each `else` with the closest previous `then`(the first interpretation). The unambiguous grammar would then be:

$$stmt \rightarrow matchedstmt|unmatchedstmt$$

$$matchedstmt \rightarrow \text{if } exp \text{ then } matchedstmt \text{ else } matchedstmt|others$$

$$unmatchedstmt \rightarrow \text{if } exp \text{ then } stmt|\text{if } exp \text{ then } matchedstmt \text{ else } unmatchedstmt$$

3.2 Parsing

Resolving ambiguity is the problem of the language specification, not of the syntax analysis. Parsing is the job of implementation. We can do this in two ways:

- **Top down parsing:** Construction of the parse tree starting from the root node (start symbol) and proceeds towards the leaves (terminals).
- **Bottom up parsing:** Construction of the parse tree starts from the terminal nodes and builds up towards the root node(start symbol).

3.3 Top Down Parsing

In top down parsing, we repeat the following two steps:

1. At a node labelled with non-terminal A, select one of the productions of A and construct the children nodes.
2. Find the next node at which the subtree is to be constructed.

To choose the production rule, as in step 1, backtracking must be done. We try every production rule that we can apply and see which one works. The parser must be intelligent enough to choose the production rule to apply based on the token being pointed to by the pointer.

3.4 First Set

If we could determine the first character of the string produced when a production rule is applied, we could compare it to the current token and choose the production rule correctly. For this, we use a concept called the First Set.

The **First set**, denoted by `FIRST(X)` for a grammar symbol X is the set of tokens that begin the strings derivable form X. If there is a production rule:

$$A \to \alpha$$

then First(A) is the set of tokens that appear as the first token in strings generated from α.

To compute FIRST(X), we use the following rules:

1. If X is a terminal, then FIRST(X) = $\{X\}$

2. If X is a non terminal and $X \rightarrow Y_1 Y_2 ... Y_k$ is a production for $k \geq 1$, then place a in First(X) if for some i, a is in First(Y_i) and ϵ is in all of First(Y_j) for $j < i$. If ϵ is in all Y_i, then add ϵ to First(X).

3. If $X \rightarrow \epsilon$, then add ϵ to First(X).

Consider the following example:

$$S \rightarrow ABCDE$$

$$A \rightarrow a|\epsilon$$

$$B \rightarrow b|\epsilon$$

$$C \rightarrow c$$

$$D \rightarrow d|\epsilon$$

$$E \rightarrow e|\epsilon$$

Then, the first sets are as following:

- First(S) = $\{a, b, c\}$
- First(A) = $\{a, \epsilon\}$
- First(B) = $\{b, \epsilon\}$
- First(C) = $\{c\}$
- First(D) = $\{d, \epsilon\}$
- First(E) = $\{e, \epsilon\}$

3.5 Recursive Descent Parser

Let us look at our first parser - the **recursive descent parser**. RDP is a top down method of syntax analysis in which a set of recursive procedures are executed to parse the stream of tokens. A procedure is associated with each non-terminal of the grammar. The procedure generally implements one of the production rules of the grammar.

In each procedure, we perform a match operation on hitting any token on the RHS of the grammar with the current token in the input that needs to be parsed. For example, if we have the string aba and the rule $S \rightarrow abB$, it will match the tokens a in both the RHS and the string, and move on to the next token. If there is no match, we throw a syntax error.

This approach, of course, has it's own limitations. Consider a grammar with two productions:

$$X \to \gamma_1$$

$$X \to \gamma_2$$

Suppose $\text{First}(\gamma_1) \cap \text{First}(\gamma_2) \neq \phi$, and that a is the common terminal symbol. In cases like this, we need to perform backtracking to choose the right production rule. However, RDP does not support backtracking right now. To support it, all productions should be tried in some order. Failure for some productions implies we need to try remaining productions. We would report an error only when there are no other rules.

Moreover, a recursive descent parser may loop forever on productions of the form:

$$A \to A\alpha_1|A\alpha_2|...|A\alpha_n|\beta_1|...|\beta_m$$

This is known as left recursion. We can remove it from the grammar by rewriting the rule as:

$$A \to \beta_1 A'|\beta_2 A'|...|\beta_m A'$$

$$A' \to \alpha_1 A'|\alpha_2 A'|...|\epsilon$$

Left recursion can be hidden as well, where expanding production rules in some order could lead to left recursion. This is called hidden left recursion, and we must handle it in the same way as we did left recursion.

Left factoring is the process of removing the common left factor that appears in two productions of the same non-terminal. An example of a rule to remove is:

$$A \to \alpha\beta_1|\alpha\beta_2|...|\alpha\beta_n$$

We can remove the left factoring and get:

$$A \to \alpha A'$$

$$A' \to \beta_1|\beta_2|...|\beta_n$$

In the initial form, the parser would be confused about which production rule to try. With the second case, we can kick the can down the road, and use a (hopefully) single lookahead pointer to decide what to do with A'.

3.6 Follow Set

Consider the following grammar:

$$A \rightarrow aBb$$

$$B \rightarrow c|\epsilon$$

And suppose we want to parse an input string "ab". How would the parser know to use ϵ for B? To do this we use the **follow set**.

Follow(X) for a non terminal X is the set of symbols that might follow the derivation of X in an input stream. The follow of the start symbol S is {\$}. The follow set can never be ϵ, since if it ends with an epsilon, the follow would be the ending symbol of the string - the \$. The steps to compute the follow set is:

- Always include \$ in Follow(S).
- If there is a production rule $A \rightarrow \alpha B\beta$, where B is a non terminal, then Follow(B) = First(β).
- If there is a production rule $A \rightarrow \alpha B\beta$, where B is a non terminal, and First(β) contains ϵ, then everything in Follow(A) is in Follow(B).
- If there is a production rule $A \rightarrow \alpha B$, then Follow(B) = Follow(A).

3.7 Predictive Parsing

Now that we have talked about all the issues that can face RDP, let us look at a new parser - **predictive parsing**. This is a non recursive top down parsing method, which recognizes LL(1) languages. LL languages are those that can be parsed by an LL parser, which parses the input from left to right, and constructs a leftmost derivation. The 1 means that we use one input symbol of lookahead to make parsing action decisions. The predictive parser makes use of a parse table and a stack to process the input token stream.

The parse table is a two dimensional array $M[X, a]$ where X is a non terminal and a is a terminal. This parse table tells the parser which production rule to use when we have a non-terminal X on top of the stack and the lookahead pointer points to a. This parse table is generated by using the First and Follow set of every non-terminal in the grammar.

To construct the parse table, we do the following steps for each production rule $A \rightarrow \alpha$:

1. For each terminal a in First(α), $M[A, a] = A \rightarrow \alpha$

2. If ϵ is in First (α), $M[A, b] = A \rightarrow \alpha$ for each terminal b in Follow(A).
3. If ϵ is in First(α), and \$ is in Follow(A), $M[A, \$] = A \rightarrow \alpha$.

Now we can finally see the algorithm for predictive parsing. Consider that \$ is a special token at the bottom of the stack and also terminates the input string. Assume that initially, the predictive parser has X symbol on top of the stack and a is the current input symbol. Then, the predictive parser does the following:

- If $X = a = \$$, then stop.
- If $X = a \neq \$$, then pop X and increment the lookahead pointer.
- If X is a non terminal, then
  - If $M[X, a] = X \rightarrow PQR$, then pop X and push R, Q, P.
  - Else, throw an error

Predictive parsing needs a LL(1) Grammar. A grammar is LL(1) if the constructed parse table has no multiple entries. If it does have multiple entries of in the same cell, it cannot be LL(1), and cannot be parsed using predictive parsing.

Another way to tell if a grammar is LL(1) is to consider rules of the type:

$$A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n$$

All rules of this type must be such that $\bigcap_{i=1}^{n} First(\alpha_i) = \phi$ and rules of the type:

$$A \rightarrow \alpha | \epsilon$$

must be such that $First(\alpha) \cap Follow(A) = \phi$.

3.8 Error Recovery

The compiler must recover from errors and identify as many errors as possible. For this, the most frequently used error technique is **panic mode**.

In panic mode, when an error is encountered anywhere in the statement, the rest of the statement is ignored by not processing the input from erroneous input to delimiters. This mode prevents the parser from developing infinite loops and is considered as the easiest way for recovery of the errors.

The error is detected when an entry in the parse table is empty. We then skip over symbols in the input until a token in a selected set of **synchronizing tokens** appears. If the error

occurs for the parse table entry $M[A, a]$, we place the symbols in Follow(A) in a synchronizing set in the parse table. So, we skip tokens until an element in the synchronizing set appears, then pop A and continue parsing from that token.

3.9 Bottom Up Parsing and Shift-Reduce Parsing

In bottom up parsing, we design a parse tree for an input string starting from the leaf nodes and going towards the root. This is equivalent to finding a series of steps to reduce a string w of input to the start symbol of the grammar by tracing out the rightmost derivations of w in reverse.

Bottom up parsing is done via **shift reduce parsing**. Shift reduce parsing splits the string into two parts, separated by a special character '.'. The left part is a string of terminals and non terminals (on the stack), and the right part is a string of terminals (rest of the input string). Initially, we assume the input to be $.w$. The parser does one of two operations:

- **Shift:** It moves terminal symbol from the right part of the string to the left part of the string. If string before the shift if $\alpha.pqr$, then the string after shift is $\alpha p.qr$. This is equivalent to pushing a terminal onto the stack.

- **Reduce:** It occurs immediately on the left of '.' and identifies a string name as RHS of a production and replaces it by the LHS. If string before reduce action is $\alpha\beta.pqr$ and $A \to \beta$ is a production then string after reduction is $\alpha A.pqr$. This is equivalent to popping the RHS of the production, and pushing the LHS onto the stack.

Bottom up parsing is capable of handling left recursive grammars.

A string that matches the RHS of a production and whose replacement gives a step in the reverse of right most derivation is called a **handle**. Our goal is to always reduce the handle and not just any RHS. Hence the shift reduce parser must be capable of detecting these handles.

A shift reduce parser could also find a **conflict**. A shift-reduce conflict is when both a shift and a reduce operation are valid. A reduce-reduce conflict is when reduction can be done by more than one production rule.

3.10 The LR(0) Parser

Now let us look at our first bottom up parser, the LR(0) parser. LR parsers are those that read from left to right, producing a rightmost derivation in reverse. LR(0) uses 0 tokens of lookahead to make it's predictions. Before we discuss the exact algorithm we must understand some concepts.

3.10.1 Augmentation of Grammar

If G is a grammar with a start symbol S, the augmented grammar G' which has a new start symbol S' and a additional production:

$$S' \rightarrow S$$

When the parser reduces by this new rule, it will stop immediately in the accept state.

3.10.2 LR(0) Items

An LR(0) item of a grammar is a production of G with a special symbol '.' at some position on the RHS. So, the production $A \rightarrow XYZ$ gives four LR(0) items:

$$A \rightarrow .XYZ$$

$$A \rightarrow X.YZ$$

$$A \rightarrow XY.Z$$

$$A \rightarrow XYZ.$$

Each item indicates how much of a production has been seen at a point in the process of parsing. For instance, the second rule indicates that we have seen an input string derivable from X and hope to see a string derivable from YZ on the next input.

3.10.3 Closure

Let I be a set of items for a grammar G. Then the Closure(I) is a set constructed as follows:

- Every item I is in closure(I)
- If $A \rightarrow \alpha.B\beta$ is in closure (I) and $B \rightarrow \gamma$ is a production then $B \rightarrow .\gamma$ is in closure(I).

3.10.4 Goto

The Goto operation, defined by Goto(I, X), where I is a set of items and X is a grammar symbol, is closure of set of items $A \to \alpha X.\beta$ such that $A \to \alpha.X\beta$ is in I. If I is a set of items for some valid prefix α, then Goto(I, X) is set of valid items for prefix αX.

3.10.5 LR(0) Automaton

The goal of the LR(0) parser is to create a LR(0) automaton, also called the **Goto graph**. The states of the automaton are the closure sets of items of the LR(0) collection, and the transitions are given by the Goto function. The start state of this automaton is Closure($\{S' \to .S\}$). All the states are accepting states.

This automaton helps us with shift-reduce decisions. Suppose that the string γ of grammar symbols takes the LR(0) automaton from the start state 0 to some state j. Then, shift on the next input symbol a if state j has a transition on a. Otherwise, we reduce; the items in state j tell us which production to use.

3.10.6 Parsing Table

This LR(0) automaton is encoded in the **parsing table**. It consists of two parts - a parsing action function "Action" and a goto function "Goto".

The Action function takes as arguments a state i and a terminal a (or \$, the input end marker). The value of Action(i, a) can have one of four forms:

1. Shift j, where j is a state. The action taken by the parser effectively shifts input a onto the stack, but uses state j to represent a. This is denoted by sj.
2. Reduce $A \to \beta$. The action of the parser effectively reduces β on the top of the stack to head A. This is denoted by rj, where j is the number of the production.
3. Accept the input and finish parsing. This is denoted by acc.
4. Error. This is denoted by a blank cell.

The Goto function maps a state i and a non terminal A to state j if Goto$(I_i, A) = I_j$

3.10.7 Is this Grammar LR(0)?

A grammar is LR(0) if its LR(0) parsing table does not contain multiple defined entries. Another way is to check if it has any shift-reduce/reduce-reduce conflicts.

3.10.8 Contents of LR Parser

The configuration of a LR parser is defined by the 2-tuple (Stack Contents, Remaining Input). Hence, the initial configuration would be $(S_0, a_0a_1a_2...a_n\$)$

3.10.9 Example

Let us consider the grammar

$$S \to AA$$

$$A \to aA|b$$

And parse the input *aabb*. The grammar will be augmented to become:

$$0 : S' \to S$$

$$1 : S \to AA$$

$$2 : A \to aA$$

$$3 : A \to b$$

Now we can generate the Goto graph. This is done by first starting with the start state $S' \to .S$. This is state is populated with other LR(0) items which exist in it's closure. Then, from each state, we make transitions over terminals (whichever are possible), where each transition is a goto (or a shift).

Each edge in the automaton is the goto, and every state is the closure. Using this graph, we can encode it as a parsing table (Fig 2). The goto operations over terminals are equivalent to shift operations. If a state has no outgoing goto operations, we must reduce by the item in that state, over every terminal.

The parsing operation of *aabb* can be seen in Table 1. In the 4th step, we see we pop out $b4$ and push $A6$. How did we decide to move to state 6? This is because of the Goto operation. We checked the preceding state $a3$ and check it's goto over non terminal A. This was a transition to state 6.

3.10.10 Limitations of the LR(0) Parser

The primary limitation of the LR(0) parser is that it does not take the lookeahead into account at all - it reduces indiscriminately, regardless of the input token. As such, it only works when states have a single reduce action!

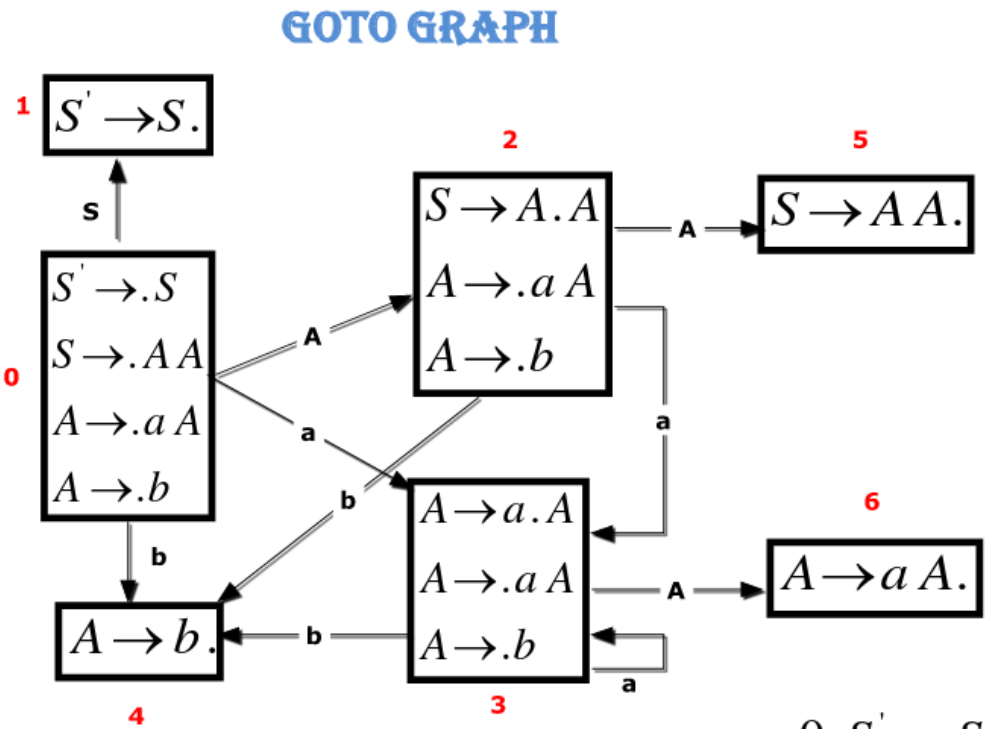

Figure 3.1: Goto graph

LR(0) PARSING TABLE

	Action			Goto	
	a	**b**	**$**	**S**	**A**
0	S3	S4		1	2
1			accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5	R1	R1	R1		
6	R2	R2	R2		

Figure 3.2: Parsing Table

Table 3.1: Parsing Process

Stack	Input	Action
0	*aabb*\$	Shift : Push a and state 3 onto the stack
0a3	*abb*\$	Shift : Push a and state 3 onto the stack
0a3a3	*bb*\$	Shift Push b and state 4 onto the stack
0a3a3b4	*b*\$	Reduce by $A \to b$. Pop 2 symbols from the stack and push A and state 6 onto the stack
0a3a3A6	*b*\$	Reduce by $A \to aA$. Pop 4 symbols from the stack and push A and state 6 onto the stack
0a3A6	*b*\$	Reduce by $A \to aA$. Pop 4 symbols from the stack and push A and state 2 onto the stack.
0A2	*b*\$	Shift : Push b and state 4 on stack.
0A2b4	\$	Reduce by $A \to b$. Pop 2 symbols from the stack and push A and state 5 onto the stack
0A2A5	\$	Reduce by $S \to AA$. Pop 4 symbols from the stack and push state 1 onto the stack.
0S1	\$	Accept.

3.11 The SLR(1) Parser

The SLR(1) parser is much like the LR(0) parser, except that the "reduce" operations are only written for terminals which are in Follow of the variable whose production is reduced. This way, the SLR parser uses 1 token of lookahead to make it's predictions, which is the reason that we include the 1.

3.11.1 Constructing the SLR(1) Parsing Table

The algorithm for constructing the parsing table is as follows:

1. Construct $C = \{I_0, \cdots, I_n\}$ the collection of sets of LR(0) items
2. If $A \to \alpha.a\beta$ is in I_i and goto$(I_i, a) = I_j$, then action$[i, a]$ = shift j.
3. If $A \to \alpha.$ is in I_i then action$[i, a]$ = reduce $A \to \alpha$ for all a in Follow(A).
4. If $S' \to S.$ is in I_i, then action$[i, \$]$ = accept
5. If goto$(I_i, A) = I_j$ then goto$[i, A] = j$ for all non terminals A

6. All undefined entries are errors

As we can see, the only difference between this and the LR(0) parser is that the parser will not only reduce in a given state with $A \to \alpha.$. Instead, it will make smarter decisions based on the follow set.

3.11.2 Is it SLR(1)?

As long as there are no conflicts in the parsing table, the grammar is $SLR(1)$.

3.11.3 Limitations of the SLR(1) Parser

The SLR(1) parser is better, but also does not take into account all the information available. Assume we have a configuration such as $X \to u.$, we know that this corresponds to having u as a handle at the top of the stack that we can reduce. We do this whenever the input symbol is in Follow(X). However this may not always be desirable, since elements below u in the stack may preclude u form being a handle for reduction. In other words, SLR only considers the top elements of the stack! We may need to divide the states into parts so that we could consider the different ways in which certain reductions can be carried out.

3.12 The Canonical LR(1) Parser

3.12.1 The LR(1) Item

An LR(1) item has the form $[I, t]$ where I is in LR(0) item and t is a lookahead token. As the dot moves through the right hand side of I, token t remains attached to it. When I is of the form $A \to \alpha.\beta$ and $\beta \neq \epsilon$, the lookahead token has no meaning. However, the LR(1) item $[A \to \alpha., t]$ calls for a reduce action if the lookahead is t.

3.12.2 The Closure Operation

Closure is done in a different way from LR(0) items. The algorithm is as follows:

Algorithm 1: CLOSURE(I)

while *No more items are added to* I **do**
  for *each item* $[A \to \alpha.B\beta, a]$ *in* I **do**
    for *each production* $B \to \gamma$ *in G'* **do**
      for *each terminal* b *in First*(βa) **do**
        add $[B \to .\gamma, b]$ to set I;
      end
    end
  end
end
return I;

3.12.3 The Goto Operation

For LR(1) items, the algorithm to find the Goto is:

Algorithm 2: GOTO(I)

initialize J to be the empty set;
for *each item* $[A \to \alpha.X\beta, a]$ *in* I **do**
  add item $[A \to \alpha X.\beta, a]$ to set J;
end
return Closure(J);

This is essentially the same as with LR(0) items, but explicitly stating to keep the lookahead token when performing the goto.

3.12.4 The CLR(1) Automaton

Just like before, we can create an automaton representing the operation of the parser. The states are the sets of items and the transitions are the goto over terminals and non-terminals. To get the LR(1) automaton, we can use the algorithm 3. Notice it does exactly the same

thing as in an LR(0) parser, but stated in an algorithmic form.

Algorithm 3: items(G')

Let $C = CLOSURE(\{[S' \rightarrow .S, \$]\})$
while *no new sets of items are added to* C **do**
  for *Each set of items* I *in* C **do**
    for *Each grammar symbol* X **do**
      if *GOTO(I,X) is not empty and not in* C **then**
        add GOTO(I,X) to C;
      end
    end
  end
end

3.12.5 The CLR(1) Parsing Table

Just like before, we have to generate a parsing table to encode the automaton. It is generated as follows:

1. Construct $C' = \{I_0, ..., I_n\}$, the collection of sets of LR(1) items for G'.
2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows:
   (a) If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and GOTO$(I_i, a) = I_j$, then set Action$[i, a]$ to shift j. Here a is a terminal.
   (b) If $[A \rightarrow \alpha., a]$ is in I_i, $A \neq S'$, then set Action$[i, a]$ to reduce $A \rightarrow \alpha$.
   (c) If $[S' \rightarrow S., \$]$ is in I_i, then set Action$[i, \$]$ to accept.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $GOTO(I_i, A) = I_j$, then GOTO$[i, A] = j$.
4. All other entries are error.
5. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S, \$]$

3.12.6 Is a Language CLR(1)?

A language is CLR(1) if and only if the parsing table does not have repeated entries.

3.13 The LALR parser

3.13.1 Why Do We Need It?

Generally, LALR parsers are used over CLR parsers. This is because CLR parsers would have thousands of states for a grammar that an LALR parser could parse with hundreds of states. It is hence more economical to use these in practice. LALR reduces the number of states by merging similiar sets. For instance, if we have two sets:

$$[A \to b., a/b]$$

$$[A \to b., \$]$$

We could merge them to create

$$[A \to b., a/b/\$]$$

Those two sets are said to share a common **core**.

3.13.2 Parse Table Construction

The construction of a parse table for LALR works as follows:

1. Construct $C = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(1) items
2. Find all the sets sharing the same cores, and merge them into their union.
3. Let $C' = \{J_0, J_1, \cdots, J_m\}$ be the resulting sets. The parsing actions for each of these states are the same as in CLR(1).
4. The Goto table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of GOTO$(I_1, X)$, GOTO$(I_2, X)$, ... , GOTO$(I_k, X)$ are the same, since $I_1, I_2, ..., I_k$ all have the same core. Let K be the union of all sets of items having the same core as GOTO(I_1, X). Then GOTO(J, X) $= K$.

The meaning of (4) is that we can just directly merge the rows of the CLR(1) parsing tables of $I_1, ..., I_k$ to get the LALR parsing table. In case there is an overlap, that overlap would be a merged set present in C'.

3.13.3 Relative Power

The relative power of LALR(1) is actually less than CLR(1), since it merges some states that could have earlier contained some contextual information. When talking about parsers,

$$LR(0) \subset SLR \subset LALR(1) \subset CLR(1)$$

The LL(1) parser has no direct relation to them - however, $LL(1) \subset LR(1)$. This relation is in fact true for all k in LL(k) and LR(k).

Chapter 4

Semantic Analysis

4.1 Introduction

A parser cannot catch all the program errors. Some language features cannot be modelled using context free grammars. For instance, a parser cannot find whether an identifier has been declared before use, and cannot check scoping rules.

The semantic analyzer takes the generated parse tree as input and gives an unambiguous program representation in the form of the disambiguated parse tree.

Some semantic errors could be:

- Type errors
- Use of undeclared variables
- An identifier may refer to different variables in different parts of the program
- An identifier may be usable in one part of the program but not another.

4.2 Attribute Grammar Framework

The attribute grammar framework is the generalization of CFGs where each grammar symbol has an associated set of attributes. The semantic attributes are computed according to some semantic rules.

The attribute grammar framework is generally notated by the **syntax directed definition**. Attributes are associated with grammar symbols and rules are associated with productions.

If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse tree node labeled X.

Non terminals can have two types of attributes:

- A **synthesized attribute** for a non terminal A at a parse tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as it's head. This means that the synthesized attribute is defined only by attributes at N and at its children.
- An **inherited attribute** for a non terminal B at a parse tree node N is defined by a semantic rule associated with the production at the parent of N. The production must have B as a symbol in it's body. This means that the inherited attribute is defined only by attributes at N, it's parent and it's siblings.

An SDD that uses only synthesized attributes is said to be an S-attributed definition.

Let us consider the following example grammar for arithmetic expressions:

$$L \to E$$

$$E \to E + T$$

$$E \to T$$

$$T \to T \times F$$

$$T \to F$$

$$F \to (E)$$

$$F \to id$$

The associated S-attributed SDD, in order will be:

$$Print(E.val)$$

$$E.val = E.val + T.val$$

$$E.val = T.val$$

$$T.val = T.val \times F.val$$

$$T.val = F.val$$

$$F \to E.val$$

$$F.val \to id.lexval$$

4.3 Dependency Graphs

To evaluate an SDD, we generally use **dependency graphs**. A dependency graph shows the interdependencies among the attributes of the various nodes of a parse tree. If an attribute b depends on an attribute c, we add an edge $c \to b$ in the dependency graph. If $c \to b$, then we need for use the semantic rule of c first, and then the semantic rule for b.

As an example, let us consider the following grammar:

$$D \to TL$$

$$T \to real$$

$$T \to int$$

$$L \to L_1, id$$

$$L \to id$$

The associated SDD (in order) will be:

$$L.in = T.type$$

$$T.type = real$$

$$T.type = int$$

$$L_1.in = L.in$$

$$addtype(id.entry, L.in)$$

Here, the first rule is an inherited attribute. We can always write a grammar with only synthesized attributes, but including inherited attributes makes it simpler.

Then, the parse tree and associated dependence graph (shown in green) for an input `real x,y,z` will be as seen in Fig 1.

4.4 L Attributed Definitions

L attributed definitions contain both synthesized and inherited attributes. A syntax directed definition is L attributed if each inherited attribute of X_j in a production:

$$A \to X_1 \cdots X_j \cdots X_n$$

depends only on

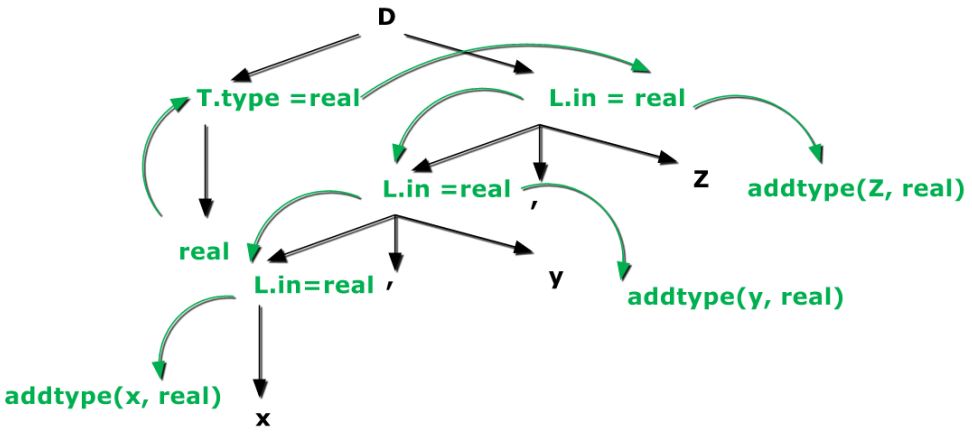

Figure 4.1: Dependency graph for the given grammar

- The attributes of the symbols to the left of X_j
- The inherited attributes of X_j

$A \to BC\{B.S = A.S\}$ is an example of an L-attributed definition.

In an L-attributed definition, attributes can be evaluated in depth first order, which is natural for both top down and bottom up parsing.

The following SDD is a L-attributed definition:

$$A \to LM$$

$$L.i = f_1(A.i)$$

$$M.i = f_2(L.s)$$

$$A.s = f_3(M.s)$$

The following SDD is not a L-attributed definition

$$A \to QR$$

$$R.i = f_4(A.i)$$

$$Q.i = f_5(R.s)$$

$$A.s = f_6(Q.s)$$

4.5 Translation Scheme

A **syntax directed translation scheme** (SDT) is a context free grammar with program fragments called **semantic actions** embedded within production bodies. By convention, curly braces or quotes are placed around actions.

If we have a production $B \to X\{a\}Y$, then action a is done after X is recognized or has been expanded. More precisely:

- If the parse is bottom up, then we perform action a as soon this occurrence of X appears at the top of the stack
- If the parser is top down, we perform a just before we attempt to expand this occurrence of Y or check for Y on the input

To evaluate the SDT, we assume the actions are terminal symbols and create the parse tree. From this, the actions form leaf nodes in the parse tree. Then, we can perform a DFS traversal to evaluate all the actions.

For the SDT to have a defined output, we should ensure that all attributes are computed before they are used. For instance, the following SDT would give an error:

$$S \to A_1 A_2 \{A_1.in = 1, A_2.in = 2\}$$

$$A \to a\{print(A.in)\}$$

Obviously, A_i cannot do it's action until $A_i.in$ is defined.

To be able to use an SDT for bottom up evaluation, we must make it so that actions only occur at the end of their productions. So, we replace each action with a distinct marker non terminal M and attach the action at the end of $M \to \epsilon$.

Chapter 5

Intermediate Representations

5.1 Abstract Syntax Tree

AST is condensed form of parse tree. Chains of single productions can be collapsed, and operators move to the parent nodes. An example can be seen in Fig 2.

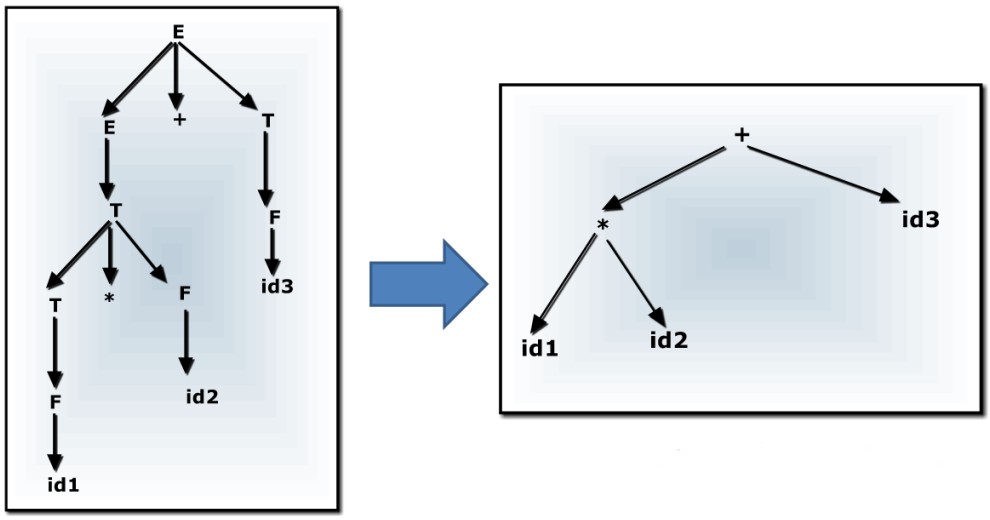

Figure 5.1: Abstract Syntax Tree

Each node can be represented as a record. In the above grammar, the operators can be represented by a record with a field for the operator type and remaining fields as pointers to operands. Every identifier can be a record with the label id and another pointer to the symbol table. Numbers can be records with a field for the label num and another to keep the number's value.

Consider the grammar:

$$E \to E_1 + T$$

$$E \to T$$

$$T \to T_1 * F$$

$$T \to F$$

$$F \to (E)$$

$$F \to id$$

$$F \to num$$

The respective operations to build the AST would be:

$$E.ptr = mknode(+, E_1.ptr, T.ptr)$$

$$E.ptr = T.ptr$$

$$T.ptr = mknode(*, T_1.ptr, F.ptr)$$

$$T.ptr = F.ptr$$

$$F.ptr = E.ptr$$

$$F.ptr = mkleaf(id, entry.id)$$

$$F.ptr = mkleaf(num, val)$$

We use $mkleaf$ to create leaf nodes and $mknode$ to make internal nodes.

5.2 Directed Acyclic Graph

A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. A node N in a DAG has more than one parent if N represents a common subexpression. This allows a DAG to represent expressions more succinctly and give the compiler clues regarding the generation of efficient code to evaluate the expressions

Consider the expression `a+a*(b-c)+(b-c)*d`. The equivalent DAG is in Fig 2.

We can construct DAGs from SDDs. We can do it the same way we created ASTs, but now we check whether an identical node already exists when creating a new node. If it does exist, then the existing node is returned.

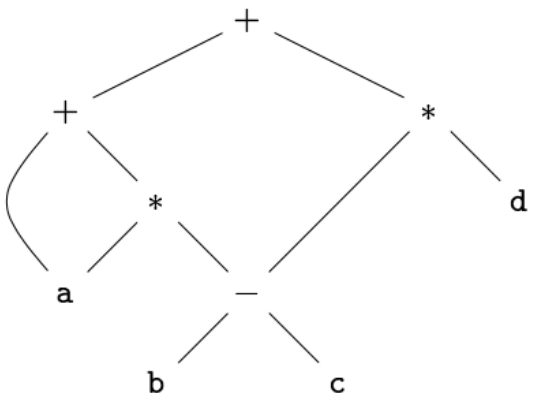

Figure 5.2: Directed Graph

5.3 Three Address Code

5.3.1 Introduction

In three address code, there is at most one operator on the right side of an expression. Hence, something like `x+y*z` would become `t1=y*z;t2=x+t1;`. This scheme allows for flexibility during target code generation and optimization.

Three address codes are built of addresses and instructions. An address can be:

- A name
- A constant
- A compiler generated temporary

5.3.2 Assignment Statements

Assignment statements can be of the following forms:

- A binary statement, like `x = y op z`
- A unary statement, like `x = op z`
- `x=y`

For every operator considered here, there must be a counterpart in the assembly level language.

5.3.3 Index Assignments

Three address code only supports 1 dimensional arrays. Hence, higher dimensional arrays must be converted into 1 dimensional arrays. This could be done by converting `x[i][j]` into `x[n*i+j]`. An example of such a statement is `x=a[i]`.

5.3.4 Jump Statements

Jump statements could be conditional or unconditional. An example of an unconditional statement is `goto L` where L is a label. An example of an conditional jump is `if x >= y goto L`.

5.3.5 Address and Pointer Assignments

Three Address codes allow for address and pointer assignments. For instance, `x = &y` will work like in C, as will `x = *y`. Just like in C, arithmetic cannot be done on pointers.

5.3.6 Procedure Calls and Returns

A call to a procedure $p(x_1, x_2, \cdots, x_n)$ can be written as a sequence of three address instructions:

```
param x1
param x2
...
param xn
enter f
leave f
return x
```

5.3.7 Miscellaneous Statements

More statements may be needed depending upon the requirement of a language. Some such statements are `break, continue`. These can be implemented using jump statements.

5.3.8 Quadruples

Quadruples are a way to represent three address code. A quadruple or a quad has four fields - $op, arg_1, arg_2, result$. For instance, if we have the instruction `x=y+z`, the quad would be $(+, z, y, x)$.

Instructions with unary operators do not use arg_2. For the copy assignment `x=y`, we consider `=` as the *op*. In all the other cases, the assignment operator is implied.

Operators like `param` use neither arg or $result$. Conditional and unconditional jumps put the target label in $result$.

5.3.9 Triples

Another way to implement three address code is using triples. A triple has only three fields - op, arg_1, arg_2. The result of an operation is now referred to by it's position in the instruction list. Hence, instead of referring to some temporary t_1 as arg, we would refer to (0).

Quadruples are still ideal compared to triples, because during optimization we generally rearrange instructions to avoid hazards. If you do this with triples, you also have to changed the number of the referenced instruction.

To fix this issue, **indirect triples** store a list of pointers to triples, rather than a listing of the actual triples. This way, the compiler can reorder the instruction list without affecting the triples themselves.

5.3.10 Three Address Code Generation

Three address code can be generated using SDDs. For this, we need to define three functions:

- `new Temp()` : Returns a unique new temporary variable
- `gen` : Produces a 3 address code for an expression
- `||` : Concatenates two three address codes.

To illustrate the generation, let us use the example from Fig 3.

The attribute *code* will hold the 3 address codes for S and E, and $E.addr$ denotes the address that will hold the value for E. The productions and semantic rules are self explanatory - the parse tree will be evaluated bottom up to generate the final three address code.

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **id** = E ;	$S.code = E.code \;\|\|$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1$ + E_2	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \;\|\| E_2.code \;\|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
| - E_1	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \;\|\|$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
| (E_1)	$E.addr = E_1.addr$ $E.code = E_1.code$
| **id**	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Figure 5.3: An example grammar

5.3.11 Three Address Code Generation for Arrays

When accessing elements, if we have an array of k dimensions, we can calculate the address of `A[i1][i2]..[ik]` with the formula:

$$base + i_1 w_1 + i_2 w_2 + \cdots + i_k w_k$$

To understand the meaning of w_i, consider the case of $k = 2$. Then, w_1 is the width of a row, and w_2 is the width of an element in the row. Another possible formula is:

$$base + ((\cdots((i_1 \times n_2 + i_2) \times n_3 + i3)) \times n_k + i_1) + w$$

So for the case of $k = 2$, the formula would be:

$$base + (i_1 \times n_2 + i_2) \times w$$

In some languages, array elements do not start numbering from 0. In that case for a single dimensional array the address would be:

$$base + (i - low) \times w$$

$$(base - low \times w) + i \times w$$

So now the first term can be a constant, the new base. We can calculate this at compile time if the dimensions are already known.

Arrays can also have two different memory layouts - row major or column major. This whole thing has been covered in PPL as well so I won't be repeating anything more.

The SDD for arrays can be found in Fig. 4.

$$
\begin{array}{lll}
S \rightarrow \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\
\quad | \quad L = E\ ; & \{\ gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\quad | \quad \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} \\
\quad | \quad L & \{\ E.addr = \textbf{new}\ Temp\,();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\
L \rightarrow \textbf{id}\ [\ E\ ] & \{\ L.array = top.get(\textbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \textbf{new}\ Temp\,();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
\quad | \quad L_1\ [\ E\ ] & \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \textbf{new}\ Temp\,();\ L.addr = \textbf{new}\ Temp\,();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

Figure 5.4: SDD for array generation

L has three synthesized attributes:

- $L.addr$ denotes a temporary used to compute the offset of the array reference by summing the terms $i_j w_j$
- $L.array$ is a pointer to the symbol table entry for the array name. $L.array.base$ is used to determine the actual l value of an array reference
- $L.type$ is the type of the subarray generated by L, whose width is given by $L.type.width$.

Chapter 6

Code Optimization

6.1 Introduction

Optimization seeks to improve the program's resource utilization. While it is generally used to reduce the execution time, it can also reduce disk access, code size, or memory usage. Of course, it is vital that the optimization not alter what the program computes.

Optimization can be at varying levels of granularity:

- Local optimization applies to a basic block in isolation. For instance, we could delete useless statements or simplify them.
- Global optimization applies to a control flow graph in isolation
- Inter-procedural optimizations apply across method boundaries

Not all optimizations are generally implemented, because they may be costly or have a low payoff. The goal is to get the maximum benefit for the lowest cost.

We know by now that we can optimize our final executable by the means of code optimization. How do we know when we should perform the optimization step?

We could do it right after the semantic analysis step, when we have an abstract syntax tree. This is machine independent, which is highly desirable when performing optimizations. However, this is very high level, and it is difficult to make optimizations

We could do it after the code generation step, when we have our final assembly language executable. However, this is machine dependant, and the optimizations need to be reimplemented when we retarget the executable.

Hence, the best time to do it is on the intermediate language. It is machine independent, while still being low level enough that we may perform meaningful optimizations.

6.2 Basic Blocks

A **basic block** is a maximal sequence of instructions with no labels (i.e. it cannot be jumped into) and no jumps (it cannot be jumped out of), besides at the starting and ending of the block. This way, the block is a single entry, single exit, straight line code segment.

How do we know when a basic block begins or ends? We need to identify **leaders**, the first instructions of the basic block. We can identify the leaders as one of the following:

- The first instruction of three address code
- Any instruction target of a conditional or unconditional jump
- Any instruction immediately following a conditional or unconditional jump.

The space between any two leaders forms a basic block.

A **control flow graph** is a directed graph where the basic blocks form the nodes, and there is an edge between blocks u and v if execution passes from the last instruction in u to the first instruction in v.

The first and last nodes can be connected to two special nodes - the entry and exit node. In this way, the control flow graph diagrammatically shows the flow of execution in the graph.

6.3 Local Optimizations

Some examples of local optimizations are:

- Some statements can be deleted (e.g. `x = x+0`)
- Some statements can be simplified (e.g. `x = x * 0`)
- Constant folding (e.g. `x = 3*5+2`)

To choose statements to delete, something we could do is eliminate unreachable basic blocks. Think of a conditional jump which always evaluates to true, and hence skips over some code (e.g. `if (1 != 0) x++;`). Another scenario is when using libraries - some methods may never be used. This would make the program smaller and also faster. This would be a global optimization.

Optimizations can be simplified if each register occurs only once on the left hand side of an assignment. For example,

```
x = z+y;
a = x;
x = 2*x;
```

could be rewritten as:

```
b = z+y;
a = b;
x = 2*b;
```

This is known as **single assignment form**. With this, we can do some more optimizations.

For instance, we can now do **common subexpression elimination**. If a definition $x =$ is the first use of x in the block, then when two assignments have the same RHS, they can compute the same value. For instance,

```
x = y+z;
w = y+z
```

can be optimized to give

```
x = y+z;
w = x
```

Another possible optimization is **copy propagation**. If $w = x$ appears in a block, we can replace all subsequent uses of w with uses of x. For instance,

```
b = z+y;
a = b;
x = 2*a;
```

can be optimized into:

```
b = z+y;
a = b;
x = 2*b;
```

Interestingly, we can now perform dead code elimination to remove `a=b`.

How do you perform dead code elimination? If $w = rhs$ appears in a basic block and w does not appear anywhere else in the program, we can remove $w = rhs$ - it is dead.

It must have become clear by now that typically, optimizations interact - one optimization may enable another optimization. Generally, an optimizing compiler repeats optimizations

until no improvements are possible.

6.4 Global Optimizations

Using dataflow analysis, we may be able to extend some optimizations across the entire control flow graph.

In general, a global optimization task will have the following traits:

- The optimization depends on knowing a property x at a particular point in the program.
- Proving x at any point requires knowledge of the entire program
- It is okay to be conservative. If the optimization requires x to be true, we want to be absolutely sure x is true. It is always safer to assume that we do not know for sure whether it is true or not.

To do this, we need to be able to perform **global dataflow analysis**.

For instance, we could propagate constant folding. To be able to replace a variable x with k, we must ensure that on every path to the use of x, the last statement modifying x is $x = k$. To do this, we attach one of the following values with x at every program point:

- $\perp$, which indicates that the statement never executes.
- C, which indicates that x is a constant.
- T, which says that X is not a constant.

At the beginning of execution, we always assume $X = T$, since we do not know yet what the value of X is. If we come across a statement $X = k$, then we will set $X = C$, since we know for sure it will be a constant. If, due to the control flow, we ever become unsure what the value of X is, we will set $X = T$.

To implement this, we need some rules. The idea is to "push" or "transfer" information from one statement to the next. For each statement s, we compute information of x immediately before (denoted by $C(x, s, in)$) or after s (denoted by $C(x, s, out)$). To do this, we define a **transfer function**, which encodes some rules:

1. Let s have predecessors $P_1, P_2, \cdots, P_n$. Then, if $C(P_i, x, out) = T$ for any i, then $C(s, x, in) = T$.
2. Let s have predecessors $P_1, P_2, \cdots, P_n$. Then, if there is no i such that $C(P_i, x, out) =$

T, but there is i, j such that $C(P_i, x, out) = D$ and $C(P_j, x, out) = C$, then $C(s, x, in) = T$.

3. If $C(P_i, x, out) = C | \perp$ for all i, then $C(s, x, in) = C$.
4. If $C(P_i, x, out) = \perp$ for all i, then $C(s, x, in) = \perp$.
5. If $C(s, x, in) = \perp$, then $C(s, x, out) = \perp$.
6. If $C(s, x, in) = T$ and s is $x = k$, then $C(s, x, out) = C$.
7. If s is $x = f(\cdots)$, then $C(s, x, out) = T$.
8. If s is $y = \cdots$, then $C(s, x, out) = k$.

Initially, as mentioned before, the values of the variables are T right before the entry statement. However, everywhere else in the control flow graph, we consider the values of the variables to be $\perp$, indicating that we have not reached there yet. Doing this can be useful, especially in the case of loops. When considering loops or cycles in the graph, we need each basic block to have some initial values. Doing so also allows us to break cycles if possible.

Notice that these values have an ordering - $\perp < C < T$. Values can only change in an increasing order on this ordering. All the constants are incomparable and are merely denoted by C. Let lub be the least upper bound in this ordering. Some examples of the operation to aid in understanding are given here:

- $lub(1, \perp) = 1$
- $lub(C, T) = T$
- $lub(1, 2) = T$

Then, rules 1 through 4 can be written using lub as:

$$C(s, x, in) = lub(\{C(p, x, out) | p \text{ is a predecessor of } s\})$$

This idea of ordering can also tell us something important - the value of a variable can change at most 2 times. This means that the optimization process will surely terminate. This tells us that the constant propagation algorithm is linear in program size.

Once all of the constants have been globally propagated, we can perform dead code elimination. To do this, we need to perform a **liveness analysis**. A variable x is live at statement s if:

- There exists a statement s' that uses x

- There is a path from s to s'
- That path has no intervening assignment to x

Liveness is a boolean property, denoted by $L(s, x, in)$. Just as before, we can write the transfer function as a set of rules:

1. $L(p, x, out)$ will be $\vee\{L(S, x, in)|S$ is a successor of $p\}$.
2. If $L(p, x, in) = 1$ if p is a function on x.
3. If $p = (x = e)$, then $L(p, x, in) = 0$, if e doesn't refer to x.
4. If p does not refer to x, then $L(p, x, in) = L(p, x, out)$.

When we perform liveness analysis, we set all $L(...)$ to false, and then use the rules on it.

6.5 Register Allocation

Intermediate code uses an unlimited number of temporaries, which is not possible in a real system. Hence, we need to map temporaries to registers in a meaningful way while keeping system constraints in mind. To do this, we assign multiple temporaries to each register without changing program behaviour.

Temporaries t_1 and t_2 can share a register if at any point in the program at most one of t_1 and t_2 is live.

To figure out how to assign registers, we can use graph colouring. We create a graph with as many nodes as there are variables, and add an edge between two variables if they are live at the same time. This is called the **Register Interference Graphs**. After we construct the RIG, the register allocation algorithm is dependent on the architecture.

One such register allocation algorithm is by using graph colouring, where each colour is a register. If the RIG is k colourable, then we can use k registers for the program. Coloring a graph is NP-Hard, but we can use some heuristics to find an approximate answer.

One such heuristic is:

1. Pick a node t with fewer than k neighbours in the RIG.
2. put t on a stack.
3. Eliminate t and its edges form the RIG.

4. Take the last node from the stack, and at each step pick a colour different from those assigned to already coloured neighbours.

If this is possible to do, the graph must be k colourable.

Sometimes, the RIG may not be k colourable, i.e., we do not have enough registers to store all the variables. In this case, we need to "spill" the registers into memory. Say we need to spill a variable f. Then, we allocate a memory location for f, at an address say a_f. Before each operation that reads f, we execute `f := load a_f`, and after every write, execute `store f, a_f`.

This is useful, but we can make it even better - we can break f into multiple pseudo-registers - $f_1, f_2, ...$ When we do this, we also have to take care of liveness information. f_i will be live only:

- Between a `f_i = load a_f` and the next instruction
- Between a `store f_i,a_f` and the preceding instruction

Now we can see why we do this - we can reduce the live range of f, and hence have fewer neighbours in the RIG. Once we do this, we can make Fig X colourable (TODO).

It is not always this simple. Additional spills could be needed, and we have to decide what to spill. When we spill, we want to minimize the cost of a spill, where the cost is given by:

$Cost$ = No. of occurrences−No. of conflicts+(5 if node corresponds to variable in loop, 0 otherwise)

Some possible heuristics are:

- Spill the temporaries with the most conflicts
- Spill the temporaries with few definitions and uses
- Avoid spilling in inner loops