Summary

Compiler Construction

WBCS039-05

Bjorn Pijnacker
b.pijnacker.1@student.rug.nl

2021–2022, 1b

Contents

About this summary

This summary is based on the lecture slides for Compiler Construction `WBCS039-05` 2021–2022 by Arnold Meijster.

Chapter 1

Introduction to compilers

1.1 What is a compiler?

A compiler converts a program written in some language into an equivalent program in another language. We call these languages the source and destination, respectively. Most commonly, the target language will be machine code for some architecture, but it may also be a higher level language. In the later case we may also call it a source-to-source compiler, or a transpiler.

1.1.1 The parts of a compiler

A compiler consists of a few parts. We begin with source code, then we do (1) lexical analysis; (2) syntax analysis; (3) semantic analysis; (4) intermediate code generation; (5) code optimization; (6) target code generation.

The lexical analyzer recognizes "strings" in our source code and converts these to tokens. It keeps track of line/column position and then passes these tokens to the parser. In this stage we can recognize strings that do not correspond to any token, and as such are invalid. The parser takes these tokens and does syntactical analysis. After this, we know whether our source code is correct—according to the grammar of the language.

The semantic analyzer does multiple things, for example type checking, declaration checking, block structure analysis, and more.

1.1.2 Symbol table

Most compilers have the above elements as a pipeline, where the output of one element the input of the next. Aside from this, a symbol table is stored, which all elements can access. The first time that a scanner recognizes some identifier it is inserted in the symbol table, and it then returns the token `IDENTIFIER`. If ever it finds the same identifier which already exists in the symbol table, it may pass `VARIABLE` or `FUNCNAME`, depending on the information the parser has added to the symbol table.

Chapter 2

Lexical Analysis

2.1 Role of the lexical analyzer

The lexical analyzer reads input characters and groups them into *lexemes*. It then serves the parser with a stream of *tokens*, one token for each lexeme. Optionally, the lexer interacts with the symbol table as discussed in subsection 1.1.2. The lexer is also responsible for e.g. skipping whitespace and comments, and for keeping track of line/column numbers for error message purposes.

The lexer and parser are separated so that complexity is reduced. A parser that is also in charge of skipping whitespace and comments and recognizing illegal string sequences is more complex, while a standalone lexer is not. The parser can also process input at a courser level (tokens v.s. characters) which makes its process and its development more efficient.

2.1.1 Tokens

Consider the statement `y = x >= 0 && x < 42;`. The lexer will read this, and transform it into a list of tokens like IDENTIFIER ASSIGN IDENTIFIER GREQ LOGICAND IDENTIFIER LESSTHAN INTCONST SEMICOLON. We have freedom in choosing granularity of our tokens. For example, we might want to make every comparison operator COMPARE and every logic boolean operator LOGICBINOP, and pass the exact operator used as a attribute of the token to the parser. This makes the grammar a lot simpler, even if the lexer is more complex.

Token attributes

When a token is passed to the parser, some attributes may be included. This might be the lexeme, that is, the string of the read token; the pointer to the symbol table location or a line/column location. In the symbol table we store the lexeme, type, and line of first appearance for error reporting.

2.1.2 Lexer error reporting

The only error that can be reported by a lexer is that of illegal characters. Typos are not detected, since a typo that would produce the IF token will for example yield the IDENTIFIER with lexeme "fi". The parser at its turn will recognize an IDENTIFIER that is not supposed to be there.

The usual solution when an illegal character is read is to abort lexing and inform the user of the error. Some lexers also have a panic mode recovery, where they simply delete the offending characters and continue lexing. Some lexers try to repair input using deletions *and* insertions, but this is complicated and computationally expensive.

2.2 Alphabet, strings, and languages

An alphabet is a finite string of characters. A string over some alphabet is a finite sequence of characters drawn from that alphabet. We denote the empty string by ϵ. The length of a string s is $|s|$. A language is a set of strings over an alphabet.

Let $x =$ 'ba', $y =$ 'nana', then the notation xy or $x \cdot y$ denotes the concatenation of x and y, that is $xy =$ 'banana'. A similar notation holds for exponents, that is, $s^0 = \epsilon$, $s^i = s \cdot s^{i-1}$ for $i > 0$.

2.2.1 Operations on languages

We can take the union of two languages A and B as $A \cup B$. This is defined as $A \cup B = \{x \mid x \in A \vee x \in B\}$. We can similarly define the concatenation of two languages $A \cdot B$ as $A \cdot B = \{x \cdot y \mid x \in A \wedge y \in B\}$, that is, the concatenation of each pair of strings from either language.

The Kleene closure of a language is that language concatenated zero or more times. For some language L, the Kleene closure L^* is $\cup_{i=0}^{\infty} L^i = \{x \mid x \in L^i, i \geq 0\}$. We also have L^+, which is one or more times, as opposed to zero or more times. Note that if $\epsilon \in L$, then $L^+ = L^*$.

2.2.2 Regular expressions

Regular expressions provide a notation that describe all languages that can be built from the Union, Concatenation, and Closure, applied to some alphabet. Regular expressions over the alphabet Σ are defined recursively:

- ϵ is the regular expression that denotes the language $L(\epsilon) = \{\epsilon\}$
- If $a \in \Sigma$ then a is the regular expression for the language $L(a) = \{a\}$

If r and s are regular expressions, then

- $r \mid s$ is the regular expression for the language $L(r) \cup L(s)$
- rs is the regular expression for the language $L(r) \cdot L(s)$

– r^* is the regular expression for the language $L(r)^*$

We have the operator precedence '$*$', then '$\cdot$', then '$|$', so $ab^* \mid cd \equiv (a(b^*)) \mid (cd)$.

In this definition, we also include some extensions. For example, classes. If we have a regular expression $a \mid b \mid \cdots \mid z$, we can also write this as $[a-z]$, meaning a through z; similarly $[abc]$ means $a \mid b \mid c$. We also include the 'optional' operator ?. If we append this to some regular expression, that expression may match zero *or* one instance. An example is matching unsigned numbers:

```
digit        -> [0-9]
digits       -> digit+
unsigned num -> digits(.digits)?([Ee][+-]?digits)?
```

2.2.3 Transition diagrams

Regular expressions can be converted to transition diagrams. These diagrams indicate which character can be accepted at some point, and which characters can follow from some 'state'. Some state are marked as 'accepting states', meaning that we have recognized a lexeme. In some cases, an accepting state is marked with a $*$, meaning that we need to retract the character pointer after accepting the lexeme in question.

Given a small deterministic finite transition diagram, we can 'run' this. Some simple code sets the state and loops through all the characters, changing the state if necessary, and returning the token accompanying the accepted lexeme, when applicable. This is a handcoded lexer. To contrast, we can also automatically generate a lexer given the regular expressions and corresponding tokens for a programming language.

Handcoding vs Generating lexers

Handcoding a lexer is usually done because people claim it is faster than using a generator. A scanner generator, however, yields much faster development, and is less error prone. One argument for handcoding a lexer is that some languages have very strange syntactical rules, which a lexer generator may not be able to handle.

2.3 Scanner generation (`flex`)

A very common scanner generator is `flex`. Flex generates a lexer in C, given an input file. This input file has the structure

```
definitions
%%
rules
%%
user code
```

The definitions section holds definitions that we can use in the rules section, like `DIGIT [0-9]`, or `ID [a-zA-Z][a-zA-Z0-9]*`. This section also contains C code that is copied verbatim into the output file, e.g. variable declarations or `#include` statements. We place this between `%{` and `}%`.

The rules section holds the rules for our lexemes. Each rules has the form `pattern {action }`, where a pattern is the regular expression input pattern to be matched, and the action is performed when the pattern is matched. This action is C code, that may or may not end with a return statement for the token. If no rule matches the read character, then it is copied to `stdout`. When more than one pattern matches the input, the longest match is chosen. If multiple occur, the rule listed first in the input file is chosen.

The user code section holds any other C code that the user wants to have verbatim in the output file, like a `main()` function, or any other accompanying functions.

The tokens that are returned are defined by the parser, which we will see later, is generated by `LLNextgen` or `Bison`. These are placed in an include file specifically made for the lexer, which we can include in the definitions section.

2.3.1 How a lexer is produced

Flex first grabs all the regular expressions named by the user. This is converted into an NFA (non-deterministic state automata), then into a DFA (deterministic state automata), and converted into a minimal DFA. This is finally converted into C-code that can scan our input.

2.4 Automata & Language Theory

An FSA (finite state automaton) is a recognizer that takes an input string and determines whether it is a valid string of some language. An FSA can be both deterministic (DFA) and non-deterministic (NFA). A DFA has in each state at most one action for any given input symbol, while an NFA may have multiple actions for one character in some state. In the case of an NFA, we would need multiple-character lookahead. Luckily, we can convert an NFA to a DFA.

2.4.1 (Non-)Deterministic Finite Automata

A DFA is a mathematical model consisting of S, a set of states; Σ, the input alphabet; a start state $s_0 \in S$; $F \subseteq S$, a set of accepting states; and a move function $move : S \times \Sigma \to S$.

An NFA has all of the above, except its move function is $move : S \times \Sigma \to \mathcal{P}(S)$, which means that one transition may have multiple possible states to end up in.

Simulating an NFA is very inefficient, so we would like to convert our NFA into an equivalent DFA. We have an algorithm for this named the powerset construction algorithm. The downside of this algorithm is that the number of states can explode, from N in the NFA, to 2^N in the DFA. Luckily, 2^N rarely occurs, and a DFA can be minimized later on to reduce the number of states.

Powerset Construction Algorithm

We start with an NFA, then:

(1) We merge any two states that can be reached using ϵ-transitions (we call this the ϵ-closure).

(2) An aggregated state is a final state iff at least one of its original states is a final state.

Written as pseudocode:

Algorithm 1: Powerset Construction Algorithm

```
Input: NFA, where ε-closure(s_0) is only (unmarked) state in D_states
while unmarked state T ∈ D_states do
    mark T ;
    foreach input symbol a do
        U = ε-closure(move(T, a)) ;
        if U ∉ D_states then
            add U to D_states as unmarked state ;
        end
        D_tran[T, a] = U ;
    end
end
```

Constructing an NFA from regular expressions

For the regular expression a, construct a move from starting state to accepting state with transition a. If s and t are regular expressions, then we make their NFA for $s \mid t$ by having a starting state with ϵ-transitions to the s_0 of the respective NFA's for s and t, and an accepting state with ϵ-transitions from the accepting states of the NFA's for s and t. If we have $s \cdot t$, we simply overlap the starting state for t with the accepting state for s.

For the Kleene closure of s, s^*, we have the NFA where we can go from starting state to accepting state, from starting state to the starting state of s, from the accepting state of s to the starting state of s, and from the accepting state of s to the actual accepting state of s^*.

This all is better illustrated then written, so please also see the lecture slides for this section.

Minimal DFA

A DFA is minimal iff no DFA with fewer states does the same task. We can achieve minimization using Brzozowski's algorithm for minimization. Let reverse(N) be the NFA constructed by making the initial state final, and the the initial state that is the union of all final states, and then reversing all transitions. Let dfa(N) be the application of the NFA to DFA algorithm. Then dfa(reverse(dfa(reverse(N)))) is the minimal DFA of N.

Chapter 3

Parsing

3.1 Backus-Naur Form (BNF)

BNF is a metalanguage used to describe the grammar of a programming language. It is formal and precise, and a form of it is used in parser generators to define the programming language to parse.

A grammar in BNF is described in terms of transformation rules, of the form

```
<symbol> ::= <expression> | <expression> | <expression> | ...
```

Expressions can consist of terminals: fundamental symbols of the language, or non-terminals: a higher-level symbol describing language syntax, which are transformed into terminals by the rules of the grammar.

An extension to BNF is eBNF. This adds the two operators `[]` and `{}`. In this case, any expression surrounded by `[]` is optional, and any expression surrounded by `{}` is repeated zero or more times.

3.2 Parsing

Parsing is the process of determining whether a string of tokens can be produced by a grammar. Parsing methods fall into two classes: top-down parsing and bottom-up parsing. Top-down parsing is simple, but works only for LL grammars. Bottom-up parsing is complicated, but much more flexible in terms of which grammars it can parse. For both types of parsing there exist parser generators. We will consider `LLnextgen` and `Bison` for top-down and bottom-up parsing respectively.

Two examples of parsing are shown below, given the grammar `E ::= 0 | E + E`, and the string to parse "`0 + 0`".

```
E  ->  E + E  ->  0 + E  ->  0 + 0 (top-down)
0 + 0  <-  E + 0  <-  E + E  <-  E (bottom-up)
```

In the top-down example, after seeing the first 0, we don't yet know which rule to use. As we read in the +, we can expand `E` to `E + E`. In bottom-up parsing, the `0` can be reduced to `E` right away, without seeing +.

3.2.1 Top-down parsing

In general, top-down parsing is easier to understand and implement directly, though in some cases it may require changes to the grammar. Top-down parsing can be done programmatically (recursive descent parsing), or by table lookup and transitions. A recursive descent parser does not require backtracking to take alternative paths, and is thus very efficient.

Recursive descent parsing

Consider a BNF grammar rule of the form

```
<LHS> ::= <RHS>
```

If there is only one RHS, then for each input token we compare it with the terminal symbol. If they match we read the next token and continue. If they do not, we have an error. For each non-terminal symbol in RHS, call its associated parsing routine.

When we have multiple RHS's, so the form

```
<LHS> ::= <RHS1> | <RHS2>
```

then we choose the correct RHS on the basis of the next input token (lookahead). The next token is compared with the first token that can be generated by each RHS until a match is found. If no match is found, we have a syntax error.

> **The Left Recursion Problem** A grammar rule like $S \rightarrow S\alpha$ is left recursive. A recursive descent parser will loop forever. As such, a left recursive grammar cannot be the basis for a top-down parser.
>
> Introducing notation "$\rightarrow^*$": Let α be any string of grammar symbols, then $\alpha \rightarrow^* \beta$ denotes that we can derive β from α in zero or more applications of grammar rules. Then, a grammar is left recursive if there exists a non-terminal A such that $A \rightarrow^* A\alpha$.
>
> We can often systematically eliminate left recursion by introducing another rule. Consider this example:
>
> ```
> expr ::= expr + term
>     | expr - term
>     | term
> ```
>
> that is clearly left recursive. We can transform it to the following (equivalent) grammar, making it not left recursive:
>
> ```
> expr  ::= term expr'
> expr' ::= + term expr'
> ```

```
    | - term expr'
    | eps
```

First sets Another characteristic of grammars that may disallow top-down parsing is the inability to determine the correct RHS based on one token of lookahead. This is, for example, the case with the grammar
`A ::= a | aB`

We define $First(\alpha)$ to be the set of terminal symbols that may appear at the beginning of a sentence derived from α. It also includes ϵ if $\alpha \rightarrow^* \epsilon$. We have some rules for computing first sets:

- If a is a terminal, then $First(a\alpha) = \{a\}$.
- If $X \rightarrow \epsilon$ then place ϵ in $First(X)$.
- If X is non-terminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a grammar rule, then place s in $First(X)$ if $s \in First(Y_i)$ and $\epsilon \in First(Y_j)$ for all $1 \leq j < i$.

Given this information, we arrive at the grammar restriction that the *First* set for every alternative for a given LHS must be disjoint, as so we can know which path to take.

If we have a grammar with overlapping First-sets, we can solve this by left-factorizing: For each non-terminal A, find the longest prefix α common to two or more of its alternatives. Then replace all the A productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$ by $A \rightarrow \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m$ and $A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$.

Follow sets The last issue we now have to solve is that of ϵ productions, in ase there is no symbol to identify α. If we have $First(\beta) = \{\epsilon\}$, then how do we know we are matching this rule? Solution: we build a Follow set for each non-terminal that can produce ϵ.

$Follow(N)$, where N is a non-terminal is the set of terminal symbols which can follow immediately after any sentence derived from N. In the grammar
`E ::= 0 | E + E`
we have $Follow(E) = \{+, \text{EOF}\}$.

To compute a follow set for a non-terminal T we examine the cases where it appears on the RHS of a grammar rule. We have two cases:

(1) $N \rightarrow \alpha T \beta \quad \vee \quad N \rightarrow \alpha[T]\beta \quad \vee \quad N \rightarrow \alpha\{T\}\beta$

In this case, $First(\beta)\backslash\{\epsilon\} \subset Follow(T)$

If β can produce ϵ then also $Follow(N) \subset Follow(T)$.

(2) $N \rightarrow \alpha T \quad \vee \quad N \rightarrow \alpha[T] \quad \vee \quad N \rightarrow \alpha\{T\}$

$Follow(N) \subset Follow(T)$.

> The follow set of T, $Follow(T)$, is the smallest set that satisfies these rules. As a base case, we state that the Follow set for the start symbol equals $\{\text{EOF}\}$, often denoted as $\{\$\}$.

We now arrive at grammar restriction 2: If a non-terminal is optional, its First and Follow sets must be disjoint (so we know whether to parse it or skip it).

Using all the above for top-down parsing, we arrive at the definition of LL(1) grammars. A grammar is LL(1) iff for all rules $A \to \alpha$ and $A \to \beta$ it must be the case that Lookahead$(A \to \alpha) \cap$ Lookahead$(A \to \beta) = \varnothing$, where Lookahead$(A \to \alpha)$ is defined as $First(\alpha) \cup Follow(A)$ if $\epsilon \in First(\alpha)$, and $First(\alpha)$ otherwise.

A top-down parser can parse LL(k) grammars, where k indicates the number of lookahead characters.

Table-driven parsing

Instead of using recursive descent, we can also encode our grammar in a table. We have a row for each non-terminal, and a column for each terminal. We have a rule in each cell, where we have a non-terminal to be parsed and a terminal symbol that is being read.

Consider a production $X \to \beta$. To construct a parse table, we add $\to \beta$ to the X row for each symbol in $First(\beta)$. If β can derive ϵ, then add $\to \beta$ for each symbol in $Follow(X)$. An example is shown in the slides.

Parsing by using a table is very easy. The pseudocode in algorithm 2 showcases the algorithm.

Algorithm 2: Table driven LL(1) parsing

```
Push EOF onto empty stack ;
Push the start non-terminal onto stack ;
top ← top of stack ;
while true do
    if top = EOF and token = EOF then
        break and report success ;
    end
    if top is a terminal then
        if top matches token then
            pop stack ;
            token ← next_token() ;
        else
            syntax error ;
        end
    else
        if table[top, token] is → B_1 B_2 ... B_k then
            pop stack ;
            push B_k, B_{k-1}, ..., B_1 ;
        else
            syntax error ;
        end
    end
    top ← top of stack ;
end
```

Conclusion

Sadly, not every grammar can be converted to a LL(k) grammar and be parsed top-down. For example, for the grammar

```
S ::= A | B
A ::= a A b | eps
B ::= A B b b | eps
```

there is no k for which it is in the class LL(k). For this, we need bottom-up parsing.

3.2.2 Bottom-up parsing

Bottom-up parsing allows for the parsing of a far greater set of grammars than top-down parsing does. We show a small example for the grammar

```
S ::= A b | B b
A ::= aa
B ::= aaa
```

While this grammar can be converted into an equivalent LL(1) grammar, it cannot be parsed top-down as is. It, however, can be parsed bottom up, in the following steps:

(1) Input = `aaab`. With 1-token look-ahead, using the knowledge of everything we have seen so far.

(2) We read the first token and know the string starts with an `a`. This tells us nothing about which rule to apply.

(3) We skip over `a` and remember it. We read another `a`. We have seen `aa`, but still know nothing about which rule to apply.

(4) We look at the third token and read another `a`. We deduce it cannot have been produced by `A`. We know we are in the rule `S ::= B b`. Our look-ahead becomes `b` which is what we expect. We decide that the `aaa` we have seen so far results from `B`.

(5) We 'forget' `aaa` and remember we have seen `B`. (This is an LR-parser *reduce* action)

(6) We read the `b` and find that we are at the end of the input. We reduce to `S` and accept the input.

Bottom-up LR parsing scans the input Left-to-right, and finds the Rightmost derivation in reverse order. It is often more powerful than top-down parsing, because it is not subject to left recursion problems. The key decision to make in LR parsing is when to reduce and which production rule to use for reduction. These are called the shift and reduce, and the reduce and reduce problems, respectively.

Shift-reduce parsers

In a shift-reduce parser, the parse stack contains symbols already parsed: the history. Tokens are *shifted* onto the stack until the top of the stack contains a *handle*. The handle is then *reduced* by replacing it on the parse stack with the corresponding non-terminal. The decision whether to shift or reduce is based on a parse table. Parsing is successful when the input has been consumed and the stack contains only the goal symbol.

LR(0) Items An item is a production with "•" somewhere on the RHS. The grammar symbols *before* the • are on the stack, the grammar symbols *after* • represent what we expect. An item set is a set of items, corresponding to the state of the parser.

We define the closure of some item: Given an item $A \to \alpha \bullet B\beta$, the closure of this item is the smallest set that contains both the item itself, and every item in $closure(B \to \bullet\gamma)$, for every production $B \to \gamma \in G$. For example, consider the following grammar, and below it, the closure that represents the initial state. This is the closure of the top-most rule, in this case.

```
S' -> S $
S  -> B C
B  -> a
C  -> a
-----------
S' -> • S $
S  -> • B C
B  -> • a
```

In the rest of the table, we specify the transitions. In the initial state, if we reduce the first rule by moving the • over the `S`, we end up with `S' -> S • $` in a new state. We add to this state the transition rule $goto(I_0, S)$, since if we are at state I_0 and we accept an S, we wish to move to the current state.

This gives us a state automaton, which we can write in a transition diagram. In some state, accepting some symbol, we can transition to some other state, until the "•" is at the end of our item.

Resulting from the above process is an action table, containing states as rows, as terminals as columns. When we are in some state and we read a terminal, we look at the accompanying cell to see what to do. This cell can contain "Shift ¡num¿", where we shift the terminal to the stack, then move to the named state, or "Reduce ¡num¿", where we reduce the stack using the indicated grammar rule. Sadly, one cell may contain both shift *and* reduce, or multiple reduce operations, in these cases we have a shift-reduce and reduce-reduce conflict, respectively.

SLR(1): Simple LR Parsing

Let $\{I_0, I_1, \ldots, I_n\}$ be the set of LR(0) items. Then, for each item:

- If $A \to \alpha \bullet a\beta \in I_i$ and $\text{goto}(I_i, a) = I_j$ then $\text{action}[i, a] = \text{shift } j$.
- If $A \to \gamma\bullet \in I_i$ (A is not the start symbol) then for each $a \in \text{Follow}(A)$, $\text{action}[i, a] = \text{reduce } A \to \gamma$
- If $S' \to S \bullet \$ \in I_i$ then $\text{action}[i, \$] = \text{accept}$
- If $\text{goto}(I_i, A) = I_j$ (A is non-terminal) then $\text{goto}[i, A] = j$

If, using the rules above, a conflict arises, then we say the grammar is not SLR(1).

Solving SLR(1) conflicts

> **LR(1) Items** To solve a SLR(1) conflict, the states need to carry more information. We augment our states with extra information by adding a lookahead terminal symbol as the second component of the items.
>
> This gives us an **LR(1) Item:** $[A \to \alpha \bullet \beta, a]$, where a is the lookahead of the item. The lookahead has no effect in an item of the form $[A \to \alpha \bullet \beta, a]$ where $\beta \neq \epsilon$, but an item of the form $[A \to \alpha\bullet, a]$ calls for a reduction by $A \to \alpha$ only if the next input symbol is a.

To build an LR(1) finite state machine, we add the production $S' \to S$. This yields for the initial state the closure of $\{[S' \to \bullet S, \$]\}$. The remaining states are then constructed in the same was as (S)LR states.

The closure of a set of LR(1) items is computed as follows:

Algorithm 3: Computing the closure of a set S of LR(1) items

```
while not stable do
    foreach item [A → α • Bβ, t] ∈ S do
        foreach production B → γ ∈ G do
            foreach token b ∈ First(βt) do
                Add [B → •γ, b] to S ;
            end
        end
    end
end
```

Using LR(1) in this manner solves an SLR(1) shift/reduce conflict, since we may now reduce in fewer situations than before, while shifting remains an option. The number of states in LR(1) is significantly bigger than in LR(0) or SLR(1), however.

LR(1) parsing table and LALR(1)

To construct a parsing table for LR(1) we go through the following steps:

(1) Construct the collection of LR(1) item sets. The initial state of the parser contains $[S' \to \bullet S, \$]$

(2) Create the parsing table:

(1) If a is terminal, $[A \to \alpha \bullet a\beta, b] \in I_i$ and goto$(I_i, a) = I_j$ then action$[i, a] =$ shift j.

(2) If $[A \to \alpha\bullet, a] \in I_i$, then action$[i, a] =$ reduce $A \to \alpha$ where $A \neq S$.

(3) If $[S' \to S\bullet, \$] \in I_i$, then action$[i, \$] =$ accept.

If there are any conflicts generated by these rules, the grammar is not LR(1)

(3) Create the parsing goto table. For all non-terminals A, if goto$(I_i, A) = I_j$ then goto$[i, A] = j$

In LR(1) parsing, the number of states is often much larger than necessary. We can use LALR(1) parsing to reduce the number of states. We merge LR(1) item sets, *then* build the parsing table. These are typically much smaller than an LR(1) parsing table.

> **LALR Parsing: Cores & Construction** A core is a set of LR(0) items that is obtained from a set of LR(1) items by ignoring the lookahead information. Two states can have the same core, but different lookaheads.
>
> We can now construct the new item sets:
>
> (1) Construct the set of LR(1) items;
>
> (2) Merge the sets with common core together as one set, if no conflict arises by doing this. If a conflict arises, the grammar is not LALR(1);
>
> (3) The parsing table is constructed from the collection of merged sets.

While LALR(1) reduces the number of states significantly, it may introduce reduce/reduce conflicts. From this, we can conclude that LALR(1) is less powerful than LR(1). Seeing each as the set of grammars it can parse, we find that

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

Chapter 4

Semantic Analysis

Semantic analysis is the third part of a compiler. We have now translated our input into tokens, and we have used a parser to determine that our input is correct. This parser has, most likely, also given us an Abstract Syntax Tree, which we can use for further analysis.

Semantic analysis catches all the errors that were not caught by lexical and syntactical analysis, like using undeclared variables, or incorrect types for variables. This is obviously very different for each use, programming language, and context, which means there is also no real theory for it.

Some examples of semantic errors are undeclared identifiers / variables not in scope; identifiers declared multiple times; index out of bounds; wrong number/types of arguments in a function call; incompatible types for some operation; break statement outside of switch or loop; goto a non-existing label; etc. . . .

> **Abstract Syntax Trees** Usually, semantic analysis is done on the abstract syntax tree. While we could put it in the parser code, using actions such as in Bison, this quickly becomes hard to read and maintain. We strip the parse tree of unnecessary parts, simplifying it, and encode the syntactic structure of the program such that it can be easily annotated with semantic information. This yields the Abstract Syntax Tree.
>
> We build the AST by augmenting the parser with actions. For any syntactic element such as an assignment, we encode each part of the assignment in a node or subtree, depending on the element. We can then post-order examine the tree to find any relevant information for the current node from the left and right subtree, and in this way, check the entire program.

During semantic analysis, the symbol table is again of great importance. In the symbol table we can store identifier metadata, such as its type and scope. When "getting" the identifier, we can then return the one with the 'closest' enclosing definition. There is multiple ways to efficiently implement this; for that please see the slides.

4.1 Type checking

Type checking is one of the most important checks we can do. Say we have the assignment `x = E`. This assignment is sound iff type(`x`) and type(`E`) are of the same 'class', *or* (size of) type(`E`) $\leq$ type(`x`). If neither is true, we produce a implicit cast warning, or an error.

Type checking in expressions is usually done by recursive descent of the AST. We check the children, and pass their type up to the parent. We can then use these inferences to deduce the type of the parent, recursively. If there is an error in any of the children, we pass this error along. If not, we check node n and return ok/error (and type), that may be applicable.

Chapter 5

Code Generation, Optimization & Activation Records

5.1 Code Generation

Using our AST we can generate code. Shown here are excerpts from an AST, and the associated code to generate the intermediate representation code. The intermediate representation we consider are C-code quadruples.

To achieve this, we use a lot of temporary variables. This is done by a global variable which is increased every time we use it. We can then store the name we give to certain temporary variables to use them further up in the AST. For if and while statements we use labels and goto's in a similar manner.

5.1.1 Memory layout

When we store a 2D array we do so in column-major or row-major storage mode. It is important to know which we use, since we need to calculate the index for the 1D representation of this array for storage in memory.

For a row-major storage mode, the address of `a[i][j]` of `int a[N][M]` is given by `&a + sizeof(int)*(i*M + j)`. This gets more complicated in a language like Pascal, where an array can have an arbitrary (positive) starting index. We need to subtract these indices when doing the calculation above. For column-major storage, we switch the `i` and `j` in the calculation, and replace `M` by `N`.

5.2 Code optimization

We, of course, wish to optimize our code. We might want to optimize a compiler to produce fewer total instructions, minimal memory usage in the resulting code, small execution time, or

even something like power consumption.

We use our AST to discover opportunities through program analysis. Below, we discuss a few local and global optimizations.

5.2.1 Local optimization

Algebraic simplifications

If our code contains structures such as `-(-i)`, this can be replaced by simply `i`. Similarly, constructs such as `b || true` and `(x < 43) && (x > 41)` can be replaced by `true` and `x == 42`, respectively, given that `x` is an integer.

Constant expression evaluation

An expression such as `c = 1 + 3` or `a = not (1 + 3 == 2 * 2)` can be evaluated at compile time. If the compiler recognizes that all children of some root expression are constant, it can simplify this into a single node containing the value. The above statements could become `c = 4` and `a = false`, respectively.

Common sub-expression elimination

If we have an AST where two child nodes encode the same sub-expression, it is a waste of processing power to also calculate this twice. Consider the expression `a = b * c + b * c`. This would normally be encoded to the quadruples `t1 = b * c; t2 = b * c; a = t1 + t2`, while writing `a = t1 * t1` and omitting `t2`, is faster.

This can also be done if the sub-expressions are function calls *iff* the function has no side-effects. For a functional language, this is very easy to optimize, but for a language where a function can have side-effects, it is most often not optimized.

For local common sub-expression elimination, we traverse the building block from top to bottom. We maintain a table of expressions evaluated so far. We remove an expression from the table if any operand is changed. We then modify applicable instructions on-the-fly. We generate a temporary variable, and store the expression in it. We use this variable next time the expression is encountered.

Copy/constant propagation

If we have an assignment `x = y` then we can replace later uses of `x` with `y` given that neither `x` nor `y` is changed in the mean time. This algorithm works the same as CSE, but with copy assignments instead of evaluated expressions.

Constant propagation works the same, except `y` is a constant in this case. A pro of this is that constant-value expressions can be evaluated to decide at compile-time to remove certain code branches that are never executed at all.

Dead store elimination

Assignments that have no use, such as the first assignment in `a = 1; b = 2; a = 42` can be removed by dead store elimination. We store for each variable the location of its most recent definition. At the time this location needs to be updated we determine whether is was used between the previous definition and the current update. If not, we remove the previous definition.

5.2.2 Global optimization

Flow graphs

In global optimization, we consider flow graphs, where each basic block is shown as a block in the graph, and any edges are the possible transitions between these blocks. In Figure 5.1, some quadruples-code and its associates flow graph is shown. Important to note is that from block 3 we may loop back to loop 2, using a so-called back-edge. A back-edge signifies our code has a loop.

```
      c = a + b;
      d = c;
      e = d * d;
lab1: f = a + c;
      g = e;
      a = g + d;
      if (a < c) goto lab2;
      f = d - g;
      goto lab 3;
lab2: h = g + 1;
      b = g * a;
      if (h < f) goto lab1;
lab 3: ;
```

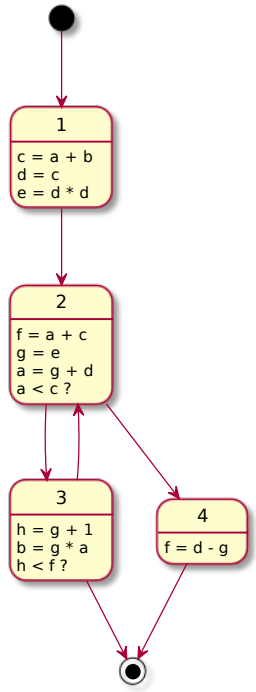

Figure 5.1: Some quadruples-code (left) and its associated flow graph (right).

For the purposes of flow analysis we need to detect basic blocks. A basic block is a single-entry, single-exit code fragment. We follow the following rules to determine how to section basic blocks: the first instruction in a basic block is called the leader of a block; the first instruction of a program is therefore the leader; any instruction that is the target of a jump is a

leader; any instruction that follows a jump is also a leader; every procedure call ends a basic block iff procedures can have side-effects. Using these rules, we determine that a basic block includes the leader and all instructions following it, up to but not including the next leader.

We call a basic block B_i a *dominator* of block B_j if every path to B_j goes through B_i. From this we can build a dominator tree: each node in the tree is the immediate dominator of its children.

Unreachable code elimination

Sometimes, we may have code that is never reached. We can easily find this using our flow graph, e.g. by a flood fill algorithm. These blocks can be entirely removed from our code.

Copy propagation

Global copy propagation is performed on the flow graph. Given a definition statement `x = y` and *use* `w = x`, we can replace `w = x` with `w = y` if `x = y` is the only definition of `x` reaching `w`, and there are no redefinitions of `y` on any path from `x = y` to `w = x`. We use data flow analysis to check these conditions.

Copy propagation may yield dead code. As before, we can recognize dead code, and remove it globally.

Common sub-expression elimination

To do this, we need to know which expressions are available at the endpoints of basic blocks. We also need to know where each of those expressions was most recently evaluated. We can then replace common sub-expressions by replacing parts of expressions with temporary variables, then seeing which of those variables get the same assignment, and replacing the corresponding sections of code.

Code hoisting

Sometimes, we evaluate parts of code in a loop where the RHS of that assignment is loop invariant. If we recognize this, we can move this code to the immediate dominator of the start of the loop. We always want to do this starting with the most-inner loop, such that code is hoisted as far as possible.

> **Recognizing loop-invariant calculations** A computation `i` is constant iff for each operand: that operand is constant; that operand depends solely on other loop-invariant computations, *or* there is exactly one in-loop definition of the operand that reaches the instruction, and *that* instruction is loop-invariant.

5.2.3 Activation Records

When a function is called, we supply a new computing environment, we pass information to the new environment, and we transfer the flow-of-control to the new environment. When the

environment finishes, we return and get info from that environment.

In some programming languages, a function call causes an *activation record* to get allocated in a stack-based fashion. This stack is called the stack of activation records (a.k.a. call stack or runtime stack). A call equals pushing a new activation record, a return equals popping an activation record. Due to this structure, we have only one "active" activation record—the top of stack.

Activation records contain memory allocated for arguments and local variables, possible local temporaries introduced by the compiler (like `_t1` in our compiler), and bookkeeping information. This bookkeeping information consists of the frame pointer (we will see later), the return address (saved program counter), and the dynamic link, which is used to restore the stack after the call.

Storage organization

The stack has the active AR at the top of the stack, growing towards lower addresses (as the stack memory does). We have a stack pointer pointing to the top-of-stack: the first available memory location. We also have a frame pointer, that points to the beginning of the current frame. This means our current AR lives in $[\texttt{frame pointer}, \texttt{stack pointer})$ AR contents are accessed as offsets relative to the frame pointer, which are defined at compile-time.

We store the caller's activation record address in the current activation record, which is called the dynamic link.

Calling sequence

A typical calling sequence looks as follows:

(1) The caller assembles and transfers control:

   (a) evaluate arguments;

   (b) push arguments on stack;

   (c) reserve space for return value callee, if needed;

   (d) push return address;

   (e) jump to callee's first instruction.

(2) Callee saves info on entry:

   (a) allocate memory for stack frame, update stack pointer;

   (b) save registers (to put them back later for caller);

   (c) save old frame pointer;

   (d) update frame pointer.

(3) Callee execute's code.

(4) Callee restores info on exit and returns control:

(a) place return value in appropriate location;

(b) restore frame pointer;

(c) restore registers;

(d) pop the stack frame;

(e) jump to return address.

(5) Caller continues.