Compiler

wugouzi

May 13, 2019

Contents

1 Chap1 introduction

source code -> scanner -> [tokens] -> parser -> [syntax tree] -> semantic analyzer -> [annotated tree] -> source code optimizer -> [intermediate code] -> code generator -> [target code] -> target code optimizer -> [target code]

2 chap2 scanning

source code(**character stream**) -> **token stream**. Use regular expression. a R|S RS R* R+=R(R*) R?=(R|) [abce]=(a|b|c|e) [a-z] [^{az}]=anything but one of the listed chars

comment "*"([*/]|[*]""|""[^/])"*/"

finite automata

Thompson's construction

Minimizing the number of states in a DFA

1. it begins with the most optimistic assumptions possible: it create two sets
   - one consisting of all the accepting states
   - the other consisting of all the nonaccepting states
2. given this partition of the states of the original DFA, consider the transitions on **each character** a of the alphabet
   - if all accepting states have transitions on a to accepting states, defines an a-transition from the new accepting state to itself
   - if all accepting states have transitions on a to nonaccepting states . . .
3. given this partition of the states of the original DFA, consider the transitions on each character a of the alphabet

- if there are two accepting states s and t that have transitions on a that land in different sets, no a-transition can be defined for this grouping of the states. a distinguish the states s and t
- if there are two accpeting states s and t s.t. s has an a-transition to another accepting state, while t has no a-transition at all. a distinguish s and t

3 Chap3 context-free grammars and parsing

3.1 context-free grammars

A context-free grammar involves recursion rules. 4-tuple $(V, \Sigma, S, \rightarrow)$. V nonterminal. terminal. S start symbol. $\rightarrow \subset V \times (V \cup \Sigma)*$

Left recursive: the nonterminal A appears as the first symbol on the right-hand side of the rule defining A

Right recusive:

-production: empty-> A grammar that generates a language containing the empty string must have at least one **-production**

3.2 Parse tree and abstract syntax trees

3.2.1 parse tree

A **parse tree** corresponding to a derivation is a labeled tree

- the interior nodes are labeled by **nonterminals**
- the leaf is **terminals**

left-most derivation

3.2.2 abstract symtax tree

```
  +
 / \
3   4
```

3.3 Ambiguity

3.3.1 ambiguity grammars

ambiguous grammar: a grammar that generates a string with two distinct parse trees

Two basic methods:

1. A rule: that specifies in each ambiguous case which of the parse trees is the correct one. **disambbiguating rule**
   - associativity
2. change the grammar

3.3.2 precedence and associativity

A *left recursive* rule makes its operators associate on the left

3.3.3 the dangling else problem

$\langle$*statement*$\rangle$ ::= $\langle$*if-stmt*$\rangle$
  | ‘`other`’

$\langle$*if-stmt*$\rangle$ ::= ‘`if`’ ‘`(`’ $\langle$*exp*$\rangle$ ‘`)`’ $\langle$*statement*$\rangle$
  | ‘`if`’ ‘`(`’ $\langle$*exp*$\rangle$ ‘`)`’ $\langle$*statement*$\rangle$ ‘`else`’ $\langle$*statement*$\rangle$

disambiguating rule is **most closely nested rule**. grammar is

$\langle$*statement*$\rangle$ -> $\langle$*matched-stmt*$\rangle$
  | $\langle$*unmatched-stmt*$\rangle$

$\langle$*matched-stmt*$\rangle$ -> ‘`if`’ ‘`(`’ $\langle$*exp*$\rangle$ ‘`)`’ $\langle$*matched-stmt*$\rangle$ ‘`else`’ $\langle$*matched-stmt*$\rangle$
  | ‘`other`’

$\langle$*unmatched-stmt*$\rangle$ -> ‘`if`’ ‘`(`’ $\langle$*exp*$\rangle$ ‘`)`’ $\langle$*statement*$\rangle$
  | ‘`if`’ ‘`(`’ $\langle$*exp*$\rangle$ ‘`)`’ $\langle$*matched-stmt*$\rangle$ ‘`else`’ $\langle$*unmatched-stmt*$\rangle$

$\langle$*exp*$\rangle$ -> ‘`0`’
  | ‘`1`’

3.3.4 inessential ambiguity

sometimes a grammar may be ambiguous and yet always produce unique *abstract syntax tree*.

inessential ambiguity: the associated semantics don’t depend on what disambiguating rule is used

3.3.5 extended notations: EBNF and syntax diagrams

$A \to A\ \alpha \mid \beta \Longrightarrow A \to \beta\ \{\alpha\}$. $A \to \alpha\ A \mid \beta \Longrightarrow A \to \{\alpha\}\ \beta$

3.4 Formal properties of context-free language

A context-free grammar consists of the following

1. T terminals
2. N nonterminals
3. P grammar rules
4. S start symbol

sentential form a string a in $(T \cup N)*$

A grammar G is **ambiguous** if there exists a string $w \in L(G)$ s.t. w has two distinct parse trees

4 Chap4 top-down parsing

4.1 Top-down parsing by recursive-descent

not easy and use EBNF

$\langle \mathit{if\text{-}stmt} \rangle$ ::= ‘`if`’ ‘`(`’ $\langle \mathit{exp} \rangle$ ‘`)`’ $\langle \mathit{statement} \rangle$
  | ‘`if`’ ‘`(`’ $\langle \mathit{exp} \rangle$ ‘`)`’ $\langle \mathit{statement} \rangle$ ‘`else`’ $\langle \mathit{statement} \rangle$

to if-stmt -> if (exp) statement [else statement]

4.2 LL(1) parsing

use an explicit stack rather than recursive calls.

$\langle S \rangle$ ::= $\langle E \rangle$ ‘`+`’ $\langle S \rangle$
  | $\langle E \rangle$

$\langle E \rangle$ ::= ‘`num`’
  | ‘`(`’ $\langle S \rangle$ ‘`)`’

.

partly-derived string	lookahead	parsed part	unparsed part
S	(		(1+2+(3+4))+5
E+S	(		(1+2+(3+4))+5
(S)+S	1	(	1+2+(3+4))+5
(E+S)+S	1	(	1+2+(3+4))+5
(1+S)+S	2	(1+	2+(3+4))+5
(1+E+S)+S	2	(1+	2+(3+4))+5
(1+2+S)+S	(	(1+2+(	(3+4))+5

For $S \to (S)\ S \mid \epsilon$

step	parsing	input	action
1	$S	()$	S->(S)S
2	$S)S(	()$	match
3	$S)S	)$	S->e
4	$S)	)$	match
5	$S	$	S->e
6	$	$	match

Two actions:

1. generate
2. match: match a token on top of the stack with the next input token

This corresponds to the leftmost derivation. **characteristic of top-down parsing**

4.2.1 LL(1) parsing table

parsing table

M[N,T]	(	)	$
S	S->(S)S	S->e	S->e

M is the set of non-terminals. T is the set of terminals or tokens including $

Table-constructing rule:

1. if $A \to \alpha$ is a production choice and there is a derivation $\alpha \Rightarrow *a\beta$ where a is a token then add $A \to \alpha$ to M[A,a]
2. if $A \to \alpha$ and $\alpha \Rightarrow *\epsilon, S\$ \Rightarrow^* \beta A a \gamma$, where S is the start symbol and a is a token(or $), then add $A \to \alpha$ to M[A,a]

A grammar is LL(1) if LL(1) parsing table has at most one production in each entry

4.2.2 left recursion removal and left factoring

left recursion removal

- **immediate left recursion**: $exp \to exp\ +\ term|exp\ -\ term|term$

- **indirect left recursion**: $A \to Bb$ and $B \to Aa$

1. Simple immediate left recursion. $A \to A\alpha|\beta$ to $A \to \beta\ A'$ and $A' \to \alpha\ A'|\epsilon$
2. general immediate left recursion. $A \to A\alpha_1|\dots|A\alpha_n|\beta_1|\dots|\beta_m$ to $A \to \beta_1 A'|\dots|\beta_m A'$ and $A' \to \alpha_1 A'|\dots|\alpha_n A'|\epsilon$
3. general left recursion. grammars with no \$$\epsilon$\$-productions and no cycles

doesn't change language, but changes the grammar and parse tree
left factoring. $A \to \alpha\beta|\alpha\gamma$ to $A \to \alpha A'$ and $A' \to \beta \mid \gamma$

4.2.3 Syntax tree construction in LL(1) parsing

4.3 First and follow sets

X a grammar symbol(a terminal or non-terminal) or ϵ. Then First(X) is

1. if X is a terminal or ϵ, then First(X)={X}
2. if X is a non-terminal, for each $X \to X_1X_2\dots X_n$, First(X) contains First(X1) - {e}

A non-terminal A is **nullable** iff there exists $A \Rightarrow^* \epsilon$ iff First(A) contains
ϵ

Follow(A) is

1. if A is start symbol, \$ is in Follow(A)
2. if $B \to \alpha A\gamma$, then First$(\gamma) - \{\epsilon\} \subseteq$ Follow(A)
3. if $B \to \alpha A\gamma, \epsilon \in$ First(γ), then Follow(A) contains Follow(B)

4.4 Error recovery in top-down parsers

5 Chap5 Bottom-up parsing

5.1 Overview of bottom-up parsing

- A bottom-up parser uses an **explicit stack** to perform a parse
- The parsing stack will contain both tokens and nonterminals

\$	inputstring \$
...	...
\$StartSymbol	\$accept

- **right-most** derivation – backward start with the tokens; end with the start symbol

(1+2+(3+4))+5
(E+2+(3+4))+5
(S+2+(3+4))+5
(S+E+(3+4))+5
(S+(3+4))+5
(S+(E+4))+5
(S+(S+4))+5
(S+(S+E))+5
(S+(S))+5
(S+E)+5
(S)+5
E+5
S+5
S+E
S

- **parsing actions**: a sequence of **shift** and **reduce** operations **parser state**: a stack of terminals and non-terminals **current derivation step** = always stack + input

derivation	step stack	unconsumed input
(1+2+(3+4))+5		(1+2+(3+4))+5
	(	1+2+(3+4))+5
(E+2+(3+4))+5	(E	+2+(3+4))+5
(S+2+(3+4))+5	(S	+2+(3+4))+5
	(S+	2+(3+4))+5
	(S+2	+(3+4))+5
(S+E+(3+4))+5	(S+E	+(3+4))+5

- 1. **shift**: shift a terminal from the front of the input to the top of the stack

  1. **reduce**: reduce a string at the top of the stack to a nonterminal A, given the BNF choice A

A bottom-up parser: **shift-reduce parser**

- One further feature of bottom-up parsers grammars are always augmented with a **new start symbol**. if S is the start symbol, a new start symbol S' is added to the grammar : S' S

- example

  S'->S

  S ->(S)S|e

  S'=>S=>(S)S=>(S)=>()

	Parsing stack	Input	Action
1	$	( ) $	Shift
2	$ (	) $	Reduce S ->
3	$ (S	) $	Shift
4	$ (S)	$	Reduce S ->
5	$ (S) S	$	Reduce S –> (S) S
6	$S	$	Reduce S'–> S
7	$S'	$	Accept

- example

  E'->E

  E->E+n|n

  E'=>E=>E+n=>n+n

	Parsing stack	Input	Action
1	$	n+n$	Shift
2	$n	+n$	Reduce E->n
3	$E	+n$	Shift
4	$E+	n$	Shift
5	$E+n	$	Reduce E->E+n
6	$E	$	Reduce E'->E
7	$E'	$	Accept

Right sentential form – A **sentential** form is any string derivable from the start symbol. Note that this includes the forms with non-terminals at intermediate steps as well.

  - A **right-sentential form** is a sentential form that occurs in a step of rightmost derivation (RMD). Each of the intermediate strings of terminals and nonterminals in such a derivation is called a right sentential form Each such sentential form is split between the parsing stack and the input during a shift-reduce parse
  - A **sentence** is a sentential form consisting only of terminals

  E,E+,E+n are **viable prefixes** of the right sentential form E+n. The sequence of symbols on the parsing stack is called **viable prefix** of the right sentential form

- **handle** This string, together with the **position** in the right sentential form where it occurs, and the production used to reduced it, is called the **handle** of the right sentential form

  determining the next handle in a parse is the main task of a shift-reduce parser

5.2 Finite automata of LR(0) items and LR(0) parsing

- An **LR(0) item** of a context-free grammar: a production choice with a distinguished position in its right-hand side
- If **A ->** , = , then **A ->** ů is an LR(0) item
- Example

S' -> S
S -> (S)S \ e
S' -> ůS
S' -> Sů
S -> ů(S)S
S -> (ůS)S
S -> (Sů)S
S -> (S)ůS
S -> (S)Sů
S -> ů

5.2.1 Finite automata of items

- The LR(0) items: as the state of a finite automata
- construct the DFA of sets of LR(0) using the subset construction from NFA

- If X is a token or a nonterminal

$$A \to \alpha \cdot X\eta \xrightarrow{\text{X}} A \to \alpha X \cdot \eta$$

- If X is a token, then this transition corresponds to a shift of X from the input to the top of the stack during a parse
- if X is a nonterminal, X will never appear as an input symbol

$$A \to \alpha \cdot X\eta \xrightarrow{\epsilon} X \to \cdot\beta$$

- The **start state** of the NFA the **initial state** of the parser: the stack is empty
- the solution is to augment the grammar by a single production S' -> S
- **S'->ůS** the **start state** of the NFA

5.2.2 The LR(0) parsing algorithm

- the parsing stack to store: **symbols** and **state numbers**
- pushing the new **state number** onto the parsing stack after each push of **a symbol**
- Let s be the current state. Then actions are
  1. if state s contains any item of the form **A -> ůX** (X is a terminal). Then the action is to shift the current input token onto the stack
  2. If state s contains any **complete item** (an item of the form **A->ů**), then the action is to reduce by the rule **A->ů**
     - A **reduction** by the rule **S'->S** where S' is the start state
     - **acceptance** if the input is empty
     - **Error** if the input is not empty

- A grammar is **LR(0)** grammar if the above rules are unambiguous
- A grammar is **LR(0)** iff
  - Each state is a shift state
  - A reduce state containing a single complete item
- table

state	action	rule	input	input	input	goto
			(	a	)	A
0	shift		3	2		1
1	reduce	A'->A				
2	reduce	A->(A)				
3	shift		3	2		4
4	shift				5	
5	reduce	A->a				

5.3 SLR(1) Parsing (simple LR(1))

- **definition**
  1. if state s contains any item of form $A \to \alpha \cdot X\beta$, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item $A \to \alpha \cdot X\beta$
  2. if state s contains the complete item $A \to \gamma\cdot$, and the next token in the input string is in Follow(A), then the action is to reduce by the rule $A \to \gamma$
     - A reduction by the rule **S'->S** where S' is the start state, this will happen only if the next input token is $
     - remove the string and all of its corresponding states from the parsing stack
     - back up in the DFA to the state from which the construction of begin
     - this state must contain an item of the form $B \to \alpha \cdot A\beta$. Push A to the stack, and push the state containing the item $B \to \alpha \cdot A\beta$
  3. if the next input token is s.t. neither of the above two cases applies, an error is declared

- A grammar is **SLR(1)** iff for any state s
  1. for any item $A \rightarrow \alpha \cdot X\beta$ in s with X a terminal, there is no complete item $B \rightarrow \gamma\cdot$ in s with X Follow(B)
  2. For any two complete item $A \rightarrow \alpha\cdot$ and $B \rightarrow \beta\cdot$ in s, Follow$(A) \cap$ Follow(B) $= \varnothing$
- right recursion can cause stack overflow

5.3.1 disambiguating rules for parsing conflicts

- two kinds of parsing conflicts in SLR(1) parsing **shift-reduce** conflicts **reduce-reduce** conflicts
- in the case of shift-reduce conflicts, there is a natural **disambiguaiting rule**: always prefer shift over the reduce
-

5.3.2 limits of SLR(1) parsing power

5.4 General LR(1) and LALR(1) parsing

- the difficulty with the SLR(1) method: applies lookaheads after the construction of the DFA of LR(0) items
- An **LR(1)** item is a pair consisting of an **LR(0)** item and a **lookahead** token
- **LR(1)** item as [**A->ů, a**] A->ů is LR(0) item, a is a token
- **definition of LR(1) transitions** main difference of LR(0) and LR(1) [**A->ůX, a**], X is any symbol, there is a transition on X to [**A->Xů,a**] [**A->ůB,a**], B nonterminal, there are -transitions to items [**B->ů,b**] for every **B->** and for every token b in **First(a)**

5.4.1 Finite automata of LR(1) items

- **start** state S'->S
- start item

  [**S'->ůS, $**]

5.4.2 The LR(1) parsing algorithm

- the general LR(1) parsing algorithm Let s be the current state.

    1. s:[A->ůX,a], X terminal, X is the next token in the input string **shift**
    2. s: [A->ů,a], the next token in the input string is a **reduce**
    3. otherwise error

- A grammar is **LR(1)** iff for any state s

    1. for any item [**A->ů,a**] in s with X a terminal, there is no item in s of the form [**B->ů,X**] (otherwise there is a shift-reduce conflict
    2. there are no two item in s of the form [**A->ů,a**] and [**B->ů,a**]

5.5 LALR(1) parsing

- the size of the DFA of sets of LR(1) items is too large
- first principle of LAIR(1) parsing the core of a state of DFA of LR(1) is a state of the DFA of LR(0) items
- second principle of LAIR(1) parsing s,s of DFA of LR(1) that have the same core, suppose there is a transition on the symbol X from s to a state t, then there is also a transition on X from state s to a state t, and the states t and t have the same core
- if a grammar is LR(1) then the LALR(1) parsing table cannot have any shift-reduce conflicts, there may be reduce-reduce conflicts
- if a grammar is SLR(1), then it's LALR(1)
- compute the DFA of LALR(1) items directly from the DFA of LR(0) items through a process of **propagating lookaheads**

5.6 Error recovery in Bottom-up parsers

A bottom-up parser will detect an error when a blank entry is detected

6 chap6 semantics analysis

6.1 Attributes and attribute grammars

attribute: any property of a programming language constructs. May be fixed prior to the compilation process or be only determinable during program execution

binding of the attribute: the process of computing an attribute and associating its computed value with the language construct in question

binding time: the time during the compilation/execution process when the binding of an attribute occurs

static attributes/dynamic attributes: based on the difference of the binding time

type checker: an analyzer. computes the data type attribute of all language entities for which data types are defined. And verifies that these types conform to the type rules of the language

type checking: set of rules that ensure the type consistency of different constructs in the program. e.g. operands types and so on

6.1.1 attribute grammars

- $X.a$: the value of a associated to X

  X is a grammar symbol and a is an attribute associated to X

- **syntax-directed semantics**: attributes are associated directly with the grammar symbols of the language

- given attributes $a_1, a_2, ..., a_k$ for each grammar rule $X_0 \to X_1 \ldots X_n$, the values of the attributes $X_i.a_j$ of each grammar symbol X_i are related to the values of the attributes of the other symbols in the rule

- an **attribute grammar**

  $X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$

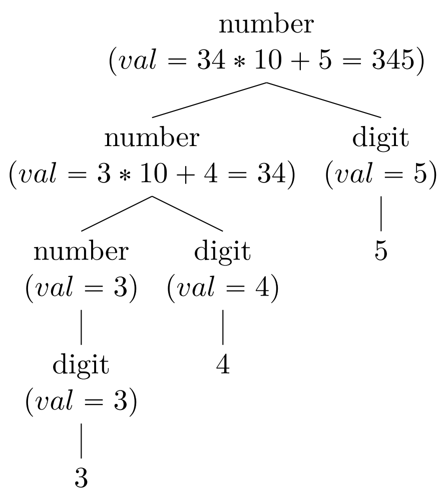
- example

  For

  $\langle number \rangle$ ::= $\langle number \rangle$ $\langle digit \rangle$
  | $\langle digit \rangle$

  $\langle digit \rangle$ ::= ‘[0123456789]’

grammar rule	semantic rules
$number1 \rightarrow number2\ digit$	$number1.val = number2.val \times 10 + digit.val$
$number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow 0$	$digit.val = 0$



6.1.2 simplifications and extensions to attribute grammars

- **metalanguage** for the attribute grammar: the collection of expressions allowable in an attribute equation
- **functions** can be added to the metalanguage whose definitions may be given elsewhere
- **simplifications**
  1. using ambiguous grammar
  2. using abstract syntax tree instead of parse tree

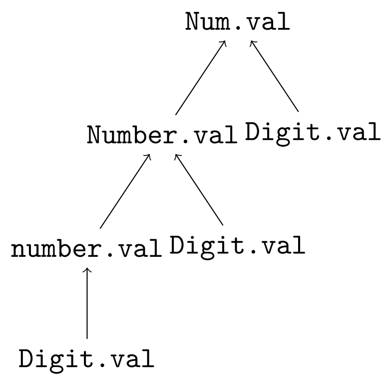
6.2 Algorithms for attribute computation

- an edge from X.a to X.a expressing the dependency of X.a on X.a

6.2.1 dependency graphs and evaluation order

- each grammar rule choice has an **associated dependency graph**

- $X_i.a_j = f_{ij}(\ldots, X_m.a_k, \ldots)$
  an edge from each $X_m.a_k$ to $X_i.a_j$



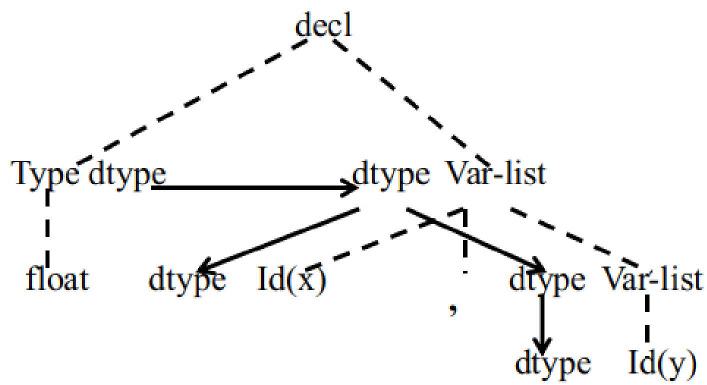
- another example

  $\langle \textit{decl} \rangle$ ::= $\langle \textit{type} \rangle$ $\langle \textit{var-list} \rangle$

  $\langle \textit{type} \rangle$ ::= '`int`'
  | '`float`'

  $\langle \textit{var-list} \rangle$ ::= '`id`' '`,`' $\langle \textit{var-list} \rangle$
  | '`id`'

grammar Rule	semantic Rules
$decl \rightarrow type\ var - list$	$var - list.dtype = type.dtype$
$type \rightarrow int$	$type.dtype = integer$
$type \rightarrow float$	$type.dtype = real$
$var - list1 \rightarrow id,\ var - list2$	$id.dtype = var - list1.dtype$
	$var - list2.dtype = var - list1.dtype$
$var - list \rightarrow id$	$id.dtype = var - list.dtype$



- **directed acyclic graphs** DAG topological sort

How attribute values are found at the roots of the graph

- **Parse tree method**: construction of the dependency graph is based on the specific parse tree at compile time, add complexity and need circularity detective
- **Rule based method**: fix an order for attribute evaluation at compiler construction time. It depends on an analysis of the attribute equations, or semantic rules

6.2.2 synthesized and inherited attributes

- **synthesized attributes**
  - an attribute is synthesized if all its dependencies point from child to parent in the parse tree
  - **S-attributed grammar**
    an attribute grammar where all the attributes are synthesized
- **inherited attributes**

  inheritance from parent to siblings, from siblings to siblings.

6.2.3 attributes as parameters and returned values

6.2.4 The use of external data structures to store attributes values

- Applicability

- Not suitable to the method of **parameters** and **returned values**
- particularly when the attribute values have significant structure and may be needed at arbitrary points during translation
- Not reasonable to be stored in the syntax tree nodes

- Ways:
  - external data structures: table, graphs and other data structures. One of the prime examples is the symbol table
  - replace attribute equations by calls to procedures representing operations on the appropriate data structure used to maintain the attribute values

6.2.5 The computation of attributes during parsing

- **L-attributed**
  - An attribute grammar of $a_1, \ldots, a_k$ is **L-attributed** if for each inherited attribute a_j and each grammar rule $X_0 \rightarrow X_1 \ldots X_n$ the associated equations for $\mathrm{a_j}$ are
  $X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_{i-1}.a_1, \ldots, X_{i-1}.a_k)$

- **S-attributed grammar** is L-attributed

- given an *L-attributed* grammar where the *inherited* attributes don't depend on the *synthesized* attributes
  1. **Top-down parser**: a recursive-descent parser can evaluate all the attributes by turning the inherited attributes into parameters and synthesized attributes into returned values.
  2. **Bottom-up parser**: LR parsers are suited to handling primarily synthesized attributes, but are difficult for inherited attributes

- $A \rightarrow B\ C \quad C.i = f(B.s)$ s is a *synthesized* attribute

Grammar Rule	Semantic Rules
$A \rightarrow BDC$	
$B \rightarrow \ldots$	compute $B.s$
$D \rightarrow \epsilon$	$saved_i = f(valstack[top])$
$C \rightarrow \ldots$	$saved_i$ is available

6.2.6 The dependence of attributes computation on the syntax

Theorem. Given an attribute grammar , all inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without changing the language of the grammar. (Knuth[1968])

6.3 The Symbol Table

semantic checks refer to properties of identifiers in the program - their scope or type

NAME	KIND	TYPE	ATTRIBUTES
foo	fun	int * int -> bool	extern

6.3.1 The structure of the symbol table

1. Linear list
2. Various search tree structures

   AVL, B tree
3. hash tables

   best choice

   Collision resolution

   (a) open addressing
   (b) separate chaining

   The process of the hash function $f : \Sigma^* \to \mathbb{N}/(size - 1)\mathbb{N}$

   Good solution: repeatedly use a constant α as multiplying factor

   $h_{i+1} = \alpha h_i + c_i, \quad h_0 = 0$

   Final hash value $h = h_n \mod size$. Typically α is a power of 2

6.3.2 Declarations

- constant declarations
- type declarations
- variable declarations
- procedure/function declarations

6.3.3 Scope rules and block structure

two rules

- Declaration before use
- the most closely nested rule for block structure

6.3.4 interaction of same-level declarations

6.3.5 an extended example of an attribute grammar using a symbol table

$\langle S \rangle$ ::= $\langle exp \rangle$

$\langle exp \rangle$ ::= '`(`' $\langle exp \rangle$ '`)`'
  | $\langle exp \rangle$ '`+`' $\langle exp \rangle$
  | '`id`'
  | '`num`'
  | '`let`' $\langle dec\text{-}list \rangle$ '`in`' $\langle exp \rangle$

$\langle dec\text{-}list \rangle$ ::= $\langle dec\text{-}list \rangle$ '`,`' $\langle decl \rangle$
  | $\langle decl \rangle$

$\langle decl \rangle$ ::= '`id`' '`=`' $\langle exp \rangle$

Three attributes

- `err`: synthesize attribute. represent error
- `symbol`: inherited attribute. represent the symbol table
- `nestlevel`: inherited attribute, nonnegtive integer. represent the current nesting level of the let blocks

Grammar Rule	Semantic Rules

S → exp	`exp.symtab = emptytable`
	`exp.nestlevel = 0`
	`S.err = exp.err`

exp1 → exp2+exp3	`exp2.symtab=exp1.symtab`
	`exp3.symtab=exp1.symtab`
	`exp2.nestlevel=exp1.nestlevel`
	`exp3.nestlevel=exp1.nestlevel`
	`exp1.err = exp2.err or exp3.err`

exp1 → (exp2)	`exp2.symtab =exp1.symtab`
	`exp2.nestlevel =exp1.nestlevel`
	`exp1.err = exp2.err`

exp → id	`exp.err = not isin(exp.symtab, id.name)`
exp → num	`exp.err = false`

exp1 → let dec-list in exp2	`dec-list.intab=exp1.symtab`
	`dec-list.nestlevel=exp1.nestlevel+1`
	`exp2.symtab=dec-list.outtab`
	`exp2.nestlevel=dec-list.nestlevel`
	`exp1.err = (dec-list.outtab=errtab) or exp2.err`

dec-list1 → dec-list2,decl	`dec-list2.intab= dec-list1.intab`
	`dec-list2.nestlevel=dec-list1.nestlevel`
	`decl.intab=dec-list2.outtab`
	`decl.nestlevel=dec-list2.nestlevel`
	`decl-list1.outtab=decl.outtab`

dec-list → decl	`decl.intab = dec-list.intab`
	`decl.nestlevel=dec-list.nestlevel`
	`dec-list.outtab=decl.outtab`

decl → id = exp	`exp.symtab = decl.intab`
	`exp.nestlevel=decl.nestlevel`
	`decl.outtab =`
	`if(decl.intab = errtab)or exp.err`
	`then errtab`
	`else`
	`if (lookup(decl.intab, id.name)= decl.nestlevel)`
	`then errtab`
	`else`
	`insert(decl.intab,id.name,decl.nestlevel)`

6.4 Data types and type checking

Type inference. Type checking

6.4.1 type names, type declarations and recursive type

6.4.2 type equivalence

two type expression represent the same type

structural equivalence: two types are the same if and only if they have the same structure

name equivalence: two type expressions are equivalent if and only if they are either the the same simple type or are the same type name

declaration equivalence: weaker version of name equivalence. $t2 = t1$ are interpreted as establishing type aliases rather than new types

6.4.3 type inference and type checking

6.4.4 additional topics in type checking

- **overloading**
- **type conversion and coercion**

7 Chap7 runtime environments

7.1 memory organization during program execution

procedure activation record

space for arguments(parameters)
space for bookkeeping information, including return address
space for local data
space for local temporaries

processor registers

- part of the structure of the runtime environment
- special-purpose registers

  PC program counter

  SP stack pointer

  FP frame pointer

AP argument pointer

calling sequence

1. the allocation of memory for the activation record
2. the computation and storing of the arguments
3. the storing and setting of necessary registers to affect the call

returning sequence

1. the placing of the return value where it can be accessed by the caller
2. the readjustment of registers
3. the possible releasing for activation record memory

7.2 fully static runtime environment

all data are static, remaining fixed in memory for the duration of program execution

No pointer or dynamic allocation. no recursive procedure calling
entire program memory

- the global variables and all variables are allocated statically
- each procedure has only a single activation record
- all variables can be accessed directly via fixed address
- no extra information about the environment needs to be kept in an activation record

the calling sequence(simple)

1. each argument is computed and stored into its appropriate parameter location in the activation of the procedure being called
2. the **return address** of the caller is saved
3. a jump is made to the begining of the code of the called procedure
4. on return, a simple jump is made to the return address

7.3 stack-based runtime environments

the stack of **activation records** grows and shrinks with the main of calls in the executing program

Each procedure may have several **different activation records** on the call stack at one time

In a language where all procedures are global, the stack-based environment requires two things

1. frame point, `fp`, a pointer to the current activation record to allow access to local variable.

   control link or **dynamic link**, a point to a record of the immediately preceding activation

2. stack pointer, `sp`, a pointer to the last location allocated on the call stack

7.3.1 stack-based environments without local procedures

the calling sequence

1. compute the *arguments* and store them in their correct positions in the new activation record of the procedure.

   because C parameters' order is reverse because of an indefinite number of arguments

2. store the *fp* as the control link in the new activation record

3. change the *fp* s.t. it points to the beginning of the new activation record

4. store the *return address* in the new activation record

5. perform a *jump* to the code of the procedure to be called

when a procedure exits

1. copy the *fp* to the *sp*

2. load the control link into the *fp*

3. perform a *jump* to the return address

4. change the *sp* to pop the arguments