

Análise Semântica e Tratamento de Erros Dependentes de Contexto

O componente Semântico de uma LP

Tarefas da Análise Semântica

Implementação da Tabelas de Símbolos

**Ações Semânticas em Compiladores
Dirigidos por Sintaxe e Erros da Análise
Semântica**

TS implementada estaticamente como uma “pilha”

Type categoria = (constante, tipo, variavel, procedimento, funcao, parametro);

classet = (valor, referencia, procedimento, funcao);

dim = record

inf, sup: integer

end;

item = record

ident: string[tam_max];

nivel: integer;

case categ: categoria of

constante: (case tipoc: integer of

1: (valori: integer);

2: (valorc: char);

3: (valorr: real);

4: (valors: string);

5: (valorb: boolean););

tipo: (nbytes: integer; dimensao: dim; tipo_elem: integer);

procedimento: (npar1: integer; end1: integer);

funcao: (npar2: integer; end2: integer; tipo_f: integer);

parametro: (classe: classet; end3: integer; tipo_p: integer);

variavel: (end4: integer; tipo_v: integer)

end;

TS: record

pilha: array [1..max] of item;

topo: integer

end;

Simplificação: arrays de
inteiros de 1 dimensão,
índices inteiros

Os 11 campos da TS para o PS com algumas extensões

Grupos que tratam de TIPOS estenderão ela com os descritores

TS: ident, nível, categoria, tipo, ender, nbytes, valor, classetransf, npar, inf, sup

Ações Semânticas em Compiladores Dirigidos por Sintaxe

Programa e Bloco

<programa> ::=

program <identificador> ;
<bloco>.

<bloco> ::=

[<parte de definições de constantes>]

[<parte de definições de tipos>]

[<parte de declarações de variáveis>]

[<parte de declarações de sub-rotinas>]

<comando composto>

<parte das definições de tipo> ::=
type <definição de tipo> {; <definição de tipo> }

<definição de tipo> ::= <identificador> ↓ = <tipo> ↓
Rs(1) Rs(2)

<tipo> ::=
<identificador> ↓ |
Rs(9)
array ↓ [<índice>] **of** <tipo> {observem que o tipo dos índices é inteiro no PS}
Rs(10)

<índice> ::=
<número> ↓ .. <número> ↓
Rs(11) Rs(12)

<parte de declarações de variáveis> ::=
var <declaração de variáveis>
{; <declaração de variáveis>;}

<declaração de variáveis> ::=
<lista de identificadores 1> : <tipo> ↓
Rs(4)

<lista de identificadores 1> ::=
<identificador> ↓ {, <identificador> ↓}
Rs(3) Rs(3)

<parte de declarações de subrotinas> ::=
{<declaração de procedimento> ; | <declaração de função> ;}

<declaração de procedimento> ::=

procedure <identificador> ↓ ↓
Rs(5) Rs(13)
[<parâmetros formais>] ; <bloco> ↓
Rs(8)

<declaração de procedimento> ::=

function <identificador> ↓ ↓
Rs(6) Rs(13)
[<parâmetros formais>] : <identificador> ↓ ; <bloco> ↓
Rs(7) Rs(8)

<parâmetros formais> ::=

(<seção de parâmetros formais>
{ ; <seção de parâmetros formais> })

Rs(20)

<seção de parâmetros formais> ::=

[var] <lista de identificadores 2> : <identificador>

Rs(14) ou Rs(15)

Rs(19)

<lista de identificadores 2> ::=

<identificador> { , <identificador> }

Rs(18)

Rs(18)

<comando composto ::=
begin <comando> { ; <comando> } **end**

<comado> ::=

<atribuição>

| <chamada de procedimento>

Se simbolo = “read” then

<lista de identificadores 3> ::=
<identificador> {, <identificador>}

senão se símbolo = “write” then Rs(21) Rs(21)

<lista de expressões 1> ::=
<expressão> {, <expressão>} Não faz nada

senão <chamada de procedimento def usuário>

| <comando composto>

| <comando condicional 1>

| <comando repetitivo 1>

<atribuição> ::=

<variável> \downarrow **:=** <expressão> \downarrow
 $R_s(25')$ $R_s(22')$

<chamada de procedimento def usuário> ::=

<identificador> \downarrow \downarrow [(\downarrow <lista de expressões>)] \downarrow
 $R_s(21)$ $R_s(25')$ $\overset{0}{u}$ $R_s(25)$ $R_s(24)$

<comando condicional 1> ::=

if <expressão> **then** <comando>

[**else** <comando>]

<comando repetitivo 1> ::=

while <expressão> **do** <comando>

Expressões

<expressão> ::=

<expressão simples> [<relação>
 <expressão simples>]

<relação> ::=

= | <> | < | <= | >= | >

<expressão simples> ::=

[+ | -] <termo> {(+ | - | **or**) <termo>}

<termo> ::=

<fator> {(* | **div** | **and** | /) <fator> }

<fator> ::=

<variavel>

| <número_int> ↓
Rs(26)

| <número_real> ↓
Rs(26)

| <chamada de função>

| (<expressão>)

| **not** <fator>

<variável> ::=

<identificador>

Rs(21)

| <identificador> [<expressão>]

Rs(21)

<lista de expressões> ::=

<expressão> { , <expressão> }

Rs(23)

Rs(23)

<chamada de função> ::=

<identificador> [(<lista de expressões>)]

Rs(21)

Rs(22')

Rs(22)

Rs(24)

Observações

- A rotina semântica 8 é usada em procedimentos e funções no final de <bloco> e é muito importante!
- Nela todos os identificadores declarados no nível corrente são apagados da TS,
 - MENOS as informações sobre o **tipo e passagem** de cada parâmetro da função ou procedimento. Devemos encadear estas informações (só estas 2, na ordem de declaração dos parâmetros) na entrada do procedimento (ou função) para ser usada posteriormente para checar o uso dos parâmetros

Erros da Análise Semântica

- Os erros gerados pelas rotinas semânticas vistas em classe são simples, pois a linguagem é um Pascal Simplificado com tipos inteiros e booleano, construtor array e com comandos de atribuição, while-do, if-then-else, chamada de procedimento e E/S.
- Portanto, os erros levantados pelas rotinas semânticas são:
 - identificador (de todas as categorias) já declarado
 - tipo não definido
 - índice superior menor que índice inferior
 - identificador não declarado
 - incompatibilidade no número de parâmetros
 - erros relativos à categoria:
 - “função não definida”,
 - “função, variável, parâmetro, ou constante não definidos”;
 - “procedimento não definido”;
 - “função, procedimento, variável ou parâmetro não definidos”

Tratamento de Algumas Extensões

- Case
- Break
- Módulos
- Record
- Enumerado

Comando CASE

26. <comando condicional 2> ::= **case** <expressão> **of** <elemento do case> { ; <elemento do case> } **end**

27. <elemento do case> ::= <constante> { , <constante> } : <comando>

5. <constante> ::= [+|-] (<identificador> | <numero_inteiro>)

- Comando **Case** do Pascal: as constantes devem ser únicas.
- Checar na análise da declaração se houve duplicação.

Case i of

1,2: ...

2,3: ...

^ erro

Comando Break

- Comandos que permitem desviar o fluxo de controle de comandos de repetição ou de seleção, por exemplo, o break da linguagem C:
 - desviam o controle para o fim de tais comandos mais aninhados.
 - Não deve haver break sem um desses comandos de seleção (case) ou repetição senão um erro será gerado.

Módulos

- Em Módulo-2, os procedimentos e funções tem nome no cabeçalho e no end.
- Eles devem ser idênticos (“casados”).
-
- O compilador deve checar se o mesmo nome foi usado nos dois lugares senão um erro será gerado.

Tipo Record

11. <record> ::= record <lista de campos> end

12. <lista de campos> ::= <lista de identificadores> : <tipo>
{ ; <lista de identificadores> : <tipo> }

15. <lista de identificadores> ::= <identificador> {, <identificador>}

- As variáveis do tipo **Record** (ou Struct) devem ser usadas com os seus campos.
- Na declaração do tipo os campos devem ser únicos.
- Checar na análise da declaração se houve duplicação, indicando erro

Tipo Enumerado

10. <enumerado> ::= (<lista de identificadores>)

15. <lista de identificadores> ::= <identificador> {,
<identificador>}

- O tipo **enumerado** deve ter constantes distintas.
- Checar na análise da declaração se houve duplicação, indicando erro

Exercício (vou fazer no começo da aula da prova)

- Dado o programa em Pascal Simplificado abaixo, pede-se:
 - Mostre na Tabela de Símbolos (PS) os identificadores e atributos que estão disponíveis a cada início de comando composto.
 - Use os nomes dados abaixo para os 11 atributos que podem aparecer na TS para o Pascal simplificado.

```

PROGRAM prova2;
  VAR i: interger;
  PROCEDURE p1 (k:integer);
    VAR i:integer;
    PROCEDURE p2;
      BEGIN
        i := i + 2;
      END;
    BEGIN
      i := k;
      p2;
      write(i)
    END; {p1}
  PROCEDURE p3;
    PROCEDURE p4;
      VAR i:integer;
      BEGIN
        i := 4;
      END;
    BEGIN
      i := i + 3;
      write(i);
      p4;
      write(i)
    END; {p3}
  BEGIN {p.p.}
    i := 10;
    p1(i);
    write(i);
    p3;
    write(i)
  END.

```

TS: ident, nível, categoria, tipo, ender, nbytes, valor, classtranf, npar, inf, sup