1 Tabela de Símbolos

A função básica da tabela de símbolos é associar informações aos símbolos criados pelo usuário — basicamente, identificadores. Também, a tabela de símbolos deve permitir controlar a visibilidade e tempo de vida dos identificadores como especificado pela linguagem.

Embora nossa linguagem MiniPascal não permita o aninhamento de sub-rotinas, a estrutura para a tabela de símbolos descrita a seguir vai permitir. Faremos assim porque, além de não comprometer a funcionalidade limitada do MiniPascal, o código será simplificado e, numa linguagem "mais" real, a possibilidade de aninhamento existe.

Nossa implementação da tabela de símbolos consiste de uma estrutura em árvore. Os nós desta árvore são denominados *ambientes*. Na raiz da árvore temos o ambiente do programa principal. Em qualquer momento do processo de compilação, apenas um ambiente é selecionado como *corrente*. Novas declarações são inseridas sempre no ambiente corrente.

Para cada sub-rotina, um novo ambiente é criado e inserido como filho do ambiente corrente. Quando a compilação entra na sub-rotina, o ambiente corrente deve ser mudado para o ambiente da sub-rotina e quando a compilação da sub-rotina termina, o ambiente corrente deve retornar ao anterior (pai).

Como um exemplo, considere o programa MiniPascal a seguir. Após a compilação do código, a tabela de símbolos terá três ambientes: o ambiente do programa Teste, o ambiente do procedimento inicializa e o ambiente da função max. Note que os nomes do procedimento e da função são mantidos no ambiente do programa Teste.

```
Program Teste(input, output);
var
   x, y: integer;
var
   a: array [1..10] of integer;

procedure inicializa;
var
   i : integer;
begin
   i := 1;
   while i <= 10 do
       begin
           a[i] := a[0];
           i := i + 1
       end
end;

function max(a, b: integer; i, j: real) : integer;
begin
   if a > b then max := a else max := b
end;

begin
   x := 10;
   y := 20;
   if max(x, y) = x then
       x := y
   else
       y := x
end.
```

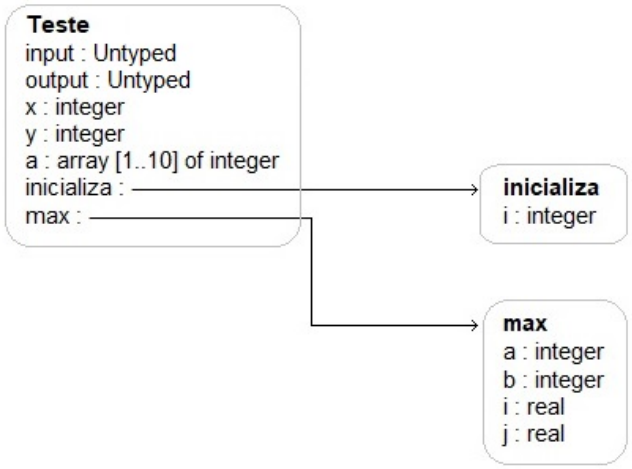

Figure 1: Tabela de símbolos do programa exemplo.

Novos símbolos são inseridos no ambiente através de sentenças declarativas. Em muitas linguagens semelhantes à MiniPascal, no mesmo escopo não podem ser atribuídos diferentes significados para um mesmo símbolo. Nesses casos, o símbolo a ser inserido no ambiente deve ser um novo símbolo, isto é, não deve ter sido previamente inserido no mesmo ambiente.

Nas sentenças imperativas (aquelas que geram código), a tabela de símbolos é **consultada** sempre que um identificador é requerido. Essa consulta é realizada em toda a tabela de símbolos começando pelo ambiente corrente (busca global); caso o símbolo não seja encontrado no ambiente corrente, a busca prossegue no ambiente pai e assim sucessivamente até a raiz. A linguagem MiniPascal não permite que durante as sentenças imperativas sejam criados novos símbolos. Portanto, se a consulta não for exitosa, um erro de "identificador não encontrado" deve ser emitido.

```
package Compiladores.MiniPascal;

import java.util.HashMap;
import java.util.Map;

import Compiladores.CompilerBase.AbstractSymbolTable;

public class SymbolTable extends AbstractSymbolTable {

    private static SymbolTable global = new SymbolTable(null);
    public static SymbolTable actual = global;

    private boolean _incode; // Parte das sentencas Imperativas?

    public SymbolTable _parent;

    // Declaracao dos parametros
    public Map<String, IdType> _parameters;
    // Declaracoes de variaveis, funcoes e procedimentos
    public Map<String, IdType> _locals;

    public SymbolTable(SymbolTable parent) {

        // Uma tabela de simbolos eh construida apenas no momento das
        // declaracoes
        _incode = false;

        _parameters = new HashMap<String, IdType>();
        _locals = new HashMap<String, IdType>();

        _parent = parent;
    }

    public boolean getInCode() {
        return _incode;
    }

    public void setInCode(boolean flag) {
        _incode = flag;
    }

    public void InsertLocal(String id, IdType type) {
        _locals.put(id, type);
    }

    public void InsertParameter(String id, IdType type) {
        _parameters.put(id, type);
    }

    // Procura um simbolo no escopo local
    public IdType FindLocal(String key) {
        // Procura no escopo local
        if (_parameters.containsKey(key)) {
            return _parameters.get(key);
        }
        if (_locals.containsKey(key)) {
            return _locals.get(key);
        }
        // Nao encontrado
        return null;
    }

    // Procura um simbolo no escopo global
    public IdType FindGlobal(String key) {

        IdType ret = FindLocal(key);
```

```
        if (ret != null) {
            return ret;
        }

        // Procura no escopo pai
        if (_parent != null) {
            return _parent.FindGlobal(key);
        }

        // Nao encontrado
        return null;
    }
}
```

Na listagem anterior, note como o aninhamento de ambientes é manipulado pelo construtor. Note também que a classe diferencia fisicamente símbolos locais e parâmetros, mas semanticamente, não há diferença entre eles. Para completar a descrição da classe, temos o flag _incode que sinaliza em que contexto o processo de compilação está: no contexto das sentenças imperativas delimitado pelas palavras reservadas "begin .. end", ou no contexto das sentenças declarativas. A manutenção deste flag é feita pelas ações semânticas @Begin e @End introduzidas pela regra 22.

22. ⟨*compound_statement*⟩ → @Begin 'begin' ⟨*optional_statements*⟩ @End 'end'

```
        if (action.equals(Tag._Begin)) {
            // Entrando na area de codigo
            table.setInCode(true);
            return;
        }
        if (action.equals(Tag._End)) {
            // Saindo na area de codigo
            table.setInCode(false);
            return;
        }
```

1.1 Lista de Identificadores – $\langle identifier\_list \rangle \uparrow^{lst}$

Uma lista de identificadores do tipo id_1, id_2, ..., id_n é resolvida através das regras 2, 3 e 4. O primeiro identificador é processado pela ação semântica @CreateList. Essa ação cria a lista lst1, insere na lista o identificador proveniente do scanner e envia a lista para o não terminal ⟨*identifier_list′*⟩ no topo da pilha do parser.

2. ⟨*identifier_list*⟩$\uparrow^{lst}$ → 'id'$\uparrow^{"tk''}$ @CreateList$\downarrow_{"tk''}\uparrow^{lst1}$ ⟨*identifier_list′*⟩$\downarrow_{lst1}\uparrow^{lst}$

```
        if (action.equals(Tag._CreateList)) {
            List<String> list = new ArrayList<String>();
            String id = ((IdentifierToken)token).getId();
            list.add(id);
            stk.peek().SetAttribute(0, list);
            return;
        }
```

Os demais identificadores da lista são processados pelas ações @Echo e @InsertList introduzidas nas regras para ⟨*identifier_list′*⟩. Note que este não terminal recebe da ação @CreateList (ou @InsertList) uma lista e, portanto, na substituição de ⟨*identifier_list′*⟩ por um dos seus lados da direita (regras 3 ou 4) essa lista deve ser copiada para a ação @Echo ou @InsertList conforme o caso. Essa cópia é feita no parser:

3. ⟨*identifier_list′*⟩$\downarrow_{lst1}\uparrow^{lst}$ → @Echo$\downarrow_{lst1}\uparrow^{lst}$

4. ⟨*identifier_list′*⟩$\downarrow_{lst1}\uparrow^{lst}$ → ',' 'id'$\uparrow^{"tk''}$ @InsertList$\downarrow_{lst1,"tk''}\uparrow^{lst2}$ ⟨*identifier_list′*⟩$\downarrow_{lst2}\uparrow^{lst}$

```
    @Override
```

```
protected void AttributeAdjust(AbsTag A, int rule, Stack<AbsTag> stk) {

    int tos = stk.size() - 1; // topo da pilha

    switch(rule) {
        // 3. <identifier_list'> ::= @Echo .
        case 3: {
            stk.peek().SetAttribute(0, A.GetAttribute(0));
            break;
        }
        // 4. <identifier_list'> ::= "," "id" @InsertList <identifier_list'> .
        case 4: {
            stk.elementAt(tos - 2).SetAttribute(0, A.GetAttribute(0));
            break;
        }

        default: {
            // Nothing todo
            break;
        }
    }
}
```

A ação semântica @InsertList recebe uma lista e um token que é um identificador. Então, a ação insere o identificador na lista e envia a nova lista para o próximo elemento da pilha do parser que, obrigatoriamente será uma nova instância do não terminal $\langle identifier\_list' \rangle$.

Quando não houver mais um identificador a ser processado, a regra 3 será aplicada e a ação @Echo é executada. O resultado produzido tanto pela ação @InsertList quanto @Echo deve ser o mesmo, isto é, uma lista; assim, a ação @Echo simplesmente repassa adiante a lista por ela recebida.

```
    if (action.equals(Tag._InsertList)) {
        @SuppressWarnings("unchecked")
        List<String> list = (List<String>)action.GetAttribute(0);
        String id = (IdentifierToken)token).getId();
        list.add(id);
        stk.peek().SetAttribute(0, list);
        return;
    }
    if (action.equals(Tag._Echo)) {
        stk.peek().SetAttribute(0, action.GetAttribute(0));
        return;
    }
```

Após o processamento completo de $\langle identifier\_list \rangle \uparrow^{lst}$, a lista lst produzida contém todos os identificadores de uma lista, na ordem em que eles aparecem.

1.2 Tipos – $\langle type \rangle \uparrow^{tp}$

Conforme as regras 8, 9, 10 e 11, os tipos em MiniPascal podem representar arrays, inteiros e reais. Inteiros e reais são tipos padrões ($\langle standard\_type \rangle$); os arrays são coleções homogêneas indexadas e delimitadas por um range ('num' '..' 'num'). As ações semânticas @Integer e @Real criam objetos representando os tipos correspondentes e os envia para o símbolo seguinte àquela ação semântica (topo da pilha de parsing).

8. $\langle type \rangle \uparrow^{tp} \rightarrow \langle standard\_type \rangle \uparrow^{tp}$
9. $\langle type \rangle \uparrow^{tp} \rightarrow$ 'array' '[' 'num' $\uparrow^{"tk''}$ @BeginRange $\downarrow_{"tk''} \uparrow^{s}$ '..' 'num' $\uparrow^{"tk''}$ @EndRange $\downarrow_{"tk''} \uparrow^{e}$ ']' 'of' $\langle standard\_type \rangle \uparrow^{tp1}$ @ArrayDec $\downarrow_{tp1,e,s} \uparrow^{tp}$
10. $\langle standard\_type \rangle \uparrow^{tp} \rightarrow$ 'integer' @Integer $\uparrow^{tp}$
11. $\langle standard\_type \rangle \uparrow^{tp} \rightarrow$ 'real' @Real $\uparrow^{tp}$

```
    if (action.equals(Tag._Integer)) {
        stk.peek().SetAttribute(0, BasicType.INTEGER); // Modificar para uma instancia de IntegerType
        return;
    }
    if (action.equals(Tag._Real)) {
```

```
            stk.peek().SetAttribute(0, BasicType.REAL); // Modificar para uma instancia de RealType
            return;
        }
```

A ação semântica @ArrayDec$\downarrow_{tp1,e,s}\uparrow^{tp}$ que produz uma representação do tipo array (tp) é mais complicada, pois requer três argumentos: o valor inicial do range (s), o valor final do range (e) e o tipo de cada elemento da coleção homogênea (tp1). Os limites do range são produzidos dentro da mesma regra 9 que contém a ação @ArrayDec – ações @BeginRange e @EndRange. O tipo tp1 é um tipo padrão produzido durante o parsing do não terminal ⟨*standard_type*⟩.

```
    if (action == Tag._BeginRange) {
        stk.elementAt(tos - 6).SetAttribute(2, ((Integer)tk).Value);
        return;
    }
    if (action == Tag._EndRange) {
        stk.elementAt(tos - 3).SetAttribute(1, ((Integer)tk).Value);
        return;
    }
    if (action == Tag._ArrayDec) {
        AbsType type = (AbsType)action.GetAttribute(0);
        int rangeEnd = (int)action.GetAttribute(1);
        int rangeBegin = (int)action.GetAttribute(2);

        stk.peek().SetAttribute(0, new ArrayType(type, rangeBegin, rangeEnd));
        return;
    }
```

Note que ambas as definições de ⟨*type*⟩ produzem o mesmo tipo de objeto (tp); portanto, a menos de suas especificidades, todos os tipos devem ser abstraídos para uma mesma classe o que nos conduz à hierarquia de classes da Figura 2. A hierarquia é preliminar e inclui apenas tipos de dados — mais adiante, serão incluídos tipos para representar funções e procedimentos.

Substituir a enumeração BasicType pela hierarquia a seguir.

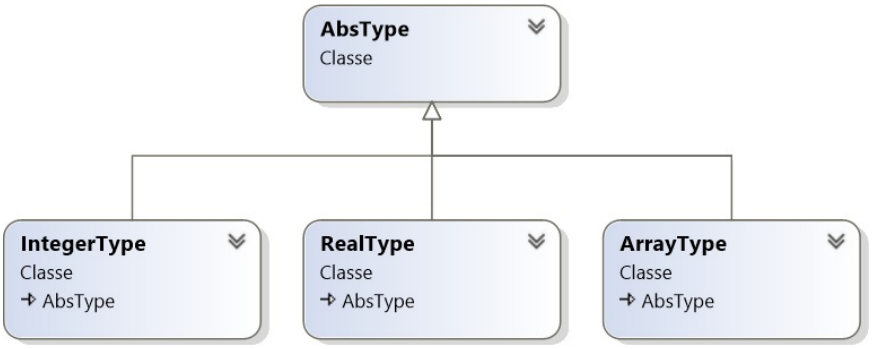

Figure 2: Hierarquia de tipos (preliminar).

1.3 Declarações de variáveis – ⟨*declarations*⟩

Agora que sabemos como as listas de identificadores (⟨*identifier_list*⟩) e tipos de dados (⟨*type*⟩) são processados, podemos descrever o processamento das sentenças declarativas de variáveis. Como vemos na regra 5, o processamento das sentenças declarativas de variáveis é feito com o auxílio das ações @VarIdList e @VarDec.

6. ⟨*declarations'*⟩ → ϵ

7. $\langle declarations' \rangle \rightarrow$ 'var' $\langle identifier\_list \rangle \uparrow^{lst1}$ @VarIdList$\downarrow_{lst1} \uparrow^{lst2}$ ':' $\langle type \rangle \uparrow^{tp}$ @VarDec$\downarrow_{tp,lst2}$ ';' $\langle declarations \rangle$

A ação @VarIdList recebe a lista produzida durante o parsing de $\langle identifier\_list \rangle$ e, sem modificá-la, simplesmente a envia para @VarDec. A ação @VarDec recebe a lista de identificadores e o tipo a ser associado a cada um dos identificadores. Cada associação "identificador + tipo" é inserido na tabela de símbolos env.

```
if (action.equals(Tag._VarIdList)) {
    @SuppressWarnings("unchecked")
    List<String> list = (List<String>)action.GetAttribute(0);
    stk.elementAt(tos-2).SetAttribute(1, list);
    return;
}
if (action.equals(Tag._VarDec)) {
    BasicType type = (BasicType)action.GetAttribute(0);
    @SuppressWarnings("unchecked")
    List<String> ids = (List<String>)action.GetAttribute(1);

    for(String id: ids) {
        table.InsertLocal(id, new VarType(type));
    }
    return;
}
```

1.4 Declaração de sub-rotinas – $\langle subprogram\_declarations \rangle$

A linguagem MiniPascal prevê dois tipos de sub-rotinas: os procedimentos são sub-rotinas que não produzem um resultado e as funções produzem um resultado de um determinado tipo (padrão). As regras relacionadas com as declarações de sub-rotinas são as regras 12 a 21.

As regras 15 e 16 são aquelas que definem as sentenças declarativas de funções e procedimentos, respectivamente. Ambas as declarações associam a um identificador uma lista de argumentos (parâmetros de entrada). No caso da declaração de uma função, também é associado ao identificador o tipo de retorno da função. O identificador sendo declarado deve ser novo no ambiente da sentença.

15. $\langle subprogram\_head \rangle \rightarrow$ 'function' 'id'$\uparrow^{"id''}$ @FuncId$\downarrow_{"id''} \uparrow^{id}$ $\langle arguments \rangle$ ':' $\langle standard\_type \rangle \uparrow^{tp}$ @FuncDec$\downarrow_{tp,id}$ ';'

16. $\langle subprogram\_head \rangle \rightarrow$ 'procedure' 'id'$\uparrow^{"id''}$ @ProcId$\downarrow_{"id''} \uparrow^{id}$ $\langle arguments \rangle$ @ProcType$\downarrow_{id}$ ';'

Ao iniciarmos o parsing das sub-rotinas, um ambiente local deve ser criado para receber todas as informações sobre os símbolos locais à sub-rotina. Deste ambiente local devem fazer parte os parâmetros declarados no cabeçalho da sub-rotina e variáveis locais. Entretanto, o identificador da sub-rotina deve ser inserido no ambiente da sentença declarativa, isto é, no ambiente anterior ao ambiente local criado para a sub-rotina. Observe como isso

é feito no código das ações @ProcId e @FuncId.

```
if (action.equals(Tag._ProcId)) {
    String id = ((IdentifierToken)token).getId();
    SymbolTable actual = SymbolTable.actual;
    actual.InsertLocal(id, new ProcType());
    return;
}
if (action == Tag._FuncId) {
    String id = ((IdentifierToken)token).getId();
    SymbolTable actual = SymbolTable.actual;
    actual.InsertLocal(id, new FuncType());
    return;
}
```

Ambas as ações inserem no contexto corrente o identificador da sub-rotina, associando-o a um objeto do tipo ProcType ou FuncType dependendo da rotina ser um procedimento ou função, respectivamente (Figura 3). Durante a construção do objeto ProcType ou FuncType um novo ambiente é criado (propriedade Env). Em seguida, o ambiente muda para esse contexto local e o parsing continua a partir daí. No final das ações @ProcId e @FuncId, o nome da sub-rotina é enviado para a ação que completará o processo de declaração da sub-rotina.

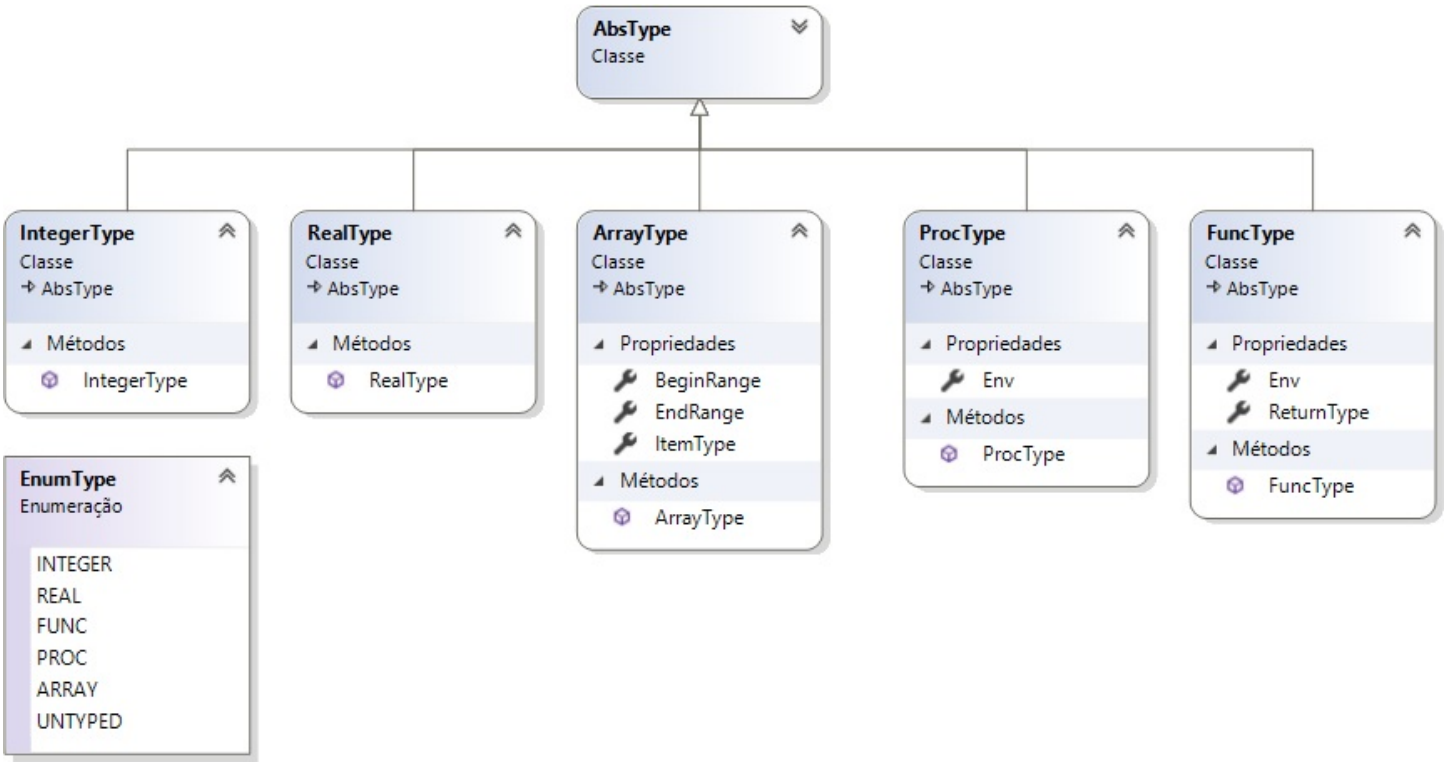

Figure 3: Hierarquia de tipos (final).

O parsing dos cabeçalhos das sub-rotinas termina com a execução das ações semânticas @ProcType ou @FuncDec. Ambas as ações recebem o identificador da sub-rotina a partir do qual o objeto ProcType ou FuncType criado nas ações anteriores é recuperado. Observe

que a busca destes objetos é local e feita no ambiente anterior (pai). Se o objeto for do tipo ProcType, não há nada a fazer; se o objeto for do tipo FuncType, o tipo padrão produzido pelo parsing do não terminal $\langle standard\_type \rangle$ é salvo na propriedade ReturnType.

```
if (action == Tag._ProcType) {
    // Identificador do procedimento
    String lexema = (String)action.Inherited[0];

    ProcType type = (ProcType)Environment.SearchLocal(env._parent, lexema);
    return;
}
if (action == Tag._FuncDec) {
    // Identificador da funcao
    String lexema = (String)action.GetAttribute(1);

    FuncType type = (FuncType)Environment.SearchLocal(env._parent, lexema);

    type.ReturnType = (AbsType)action.GetAttribute(0);
    return;
}
```

15. $\langle subprogram\_head \rangle \rightarrow$ 'function' 'id'$\uparrow^{"id"}$ @FuncId$\downarrow_{"id"}\uparrow^{id}$ $\langle arguments \rangle$ ':' $\langle standard\_type \rangle\uparrow^{tp}$ @FuncDec$\downarrow_{tp,id}$ ';'

16. $\langle subprogram\_head \rangle \rightarrow$ 'procedure' 'id'$\uparrow^{"id"}$ @ProcId$\downarrow_{"id"}\uparrow^{id}$ $\langle arguments \rangle$ @ProcType$\downarrow_{id}$ ';'

Note nas regras 15 e 16 reproduzidas acima que os argumentos das sub-rotinas, não terminal $\langle arguments \rangle$, são compilados no ambiente da sub-rotina. Durante o parsing de $\langle arguments \rangle$, a declaração de cada parâmetro de entrada na lista $\langle arguments \rangle$ é inserida no ambiente local – mais precisamente, na tabela parameters declarada dentro da classeEnvironment. As regras de 17 a 21 definem a sintaxe da lista de parâmetros e introduzem as ações que mantém a lista de parâmetros, @CreateList e @EndParList, e a ação @ParDec.

17. $\langle arguments \rangle \rightarrow \epsilon$

18. $\langle arguments \rangle \rightarrow$ '(' $\langle parameter\_list \rangle$ ')'

19. $\langle parameter\_list \rangle \rightarrow$ 'id'$\uparrow^{"tk"}$ @CreateList$\downarrow_{"tk"}\uparrow^{lst1}$ $\langle identifier\_list' \rangle\downarrow_{lst1}\uparrow^{lst2}$ @EndParList$\downarrow_{lst2}\uparrow^{lst3}$
':' $\langle type \rangle\uparrow^{tp}$ @ParDec$\downarrow_{tp,lst3}$ $\langle parameter\_list' \rangle$

20. $\langle parameter\_list' \rangle \rightarrow \epsilon$

21. $\langle parameter\_list' \rangle \rightarrow$ ';' $\langle identifier\_list \rangle\uparrow^{lst}$ @EndParList$\downarrow_{lst}\uparrow^{lst1}$ ':' $\langle type \rangle\uparrow^{tp}$ @ParDec$\downarrow_{tp,lst1}$ $\langle parameter\_list' \rangle$

```
if (action == Tag._EndParList) {
    stk.elementAt(tos - 2).Inherited[1] = action.Inherited[0];
    return;
}
if (action == Tag._ParDec) {
    foreach (String arg in (List<object>)action.Inherited[1])
        env._parameters.Add(arg, (AbsType)action.Inherited[0]);
    return;
}
```

Note que o parsing de $\langle arguments \rangle$ adiciona as declarações dos argumentos no ambiente corrente e não produz um resultado.

Para completar a discussão do parsing das sentenças declarativas, temos que comentar a ação semântica @EnvRestore inserida na regra 14. Embora um pouco fora do contexto das sentenças declarativas, esta ação semântica é fundamental para o entendimento da dinâmica da troca de ambientes devida à introdução de sub-rotinas. Vimos que após o parsing do nome da sub-rotina inserido no cabeçalho da sub-rotina, o ambiente corrente é o ambiente local da sub-rotina. Após a compilação total da sub-rotina, o ambiente corrente deve retornar para o anterior – este é o papel da ação @EnvRestore.

14. ⟨*subprogram_declaration′*⟩ → ⟨*subprogram_head*⟩ ⟨*declarations*⟩ ⟨*compound_statement*⟩ @EnvRestore ';' ⟨$subprogram_{d}eclarations'$⟩

```
if (action == Tag._EnvRestore) {
    corrente = env._parent;
    return;
}
```

1.5 Argumentos do programa principal

Para completar a descrição da tabela de símbolos, note na listagem do programa MiniPascal exemplo, no início deste texto, que o ambiente principal contém os símbolos input e output, ambos associados a um objeto Untyped. Fizemos assim porque não nos preocuparemos com estes argumentos, uma vez que estamos interessados nos principais elementos que compõem uma tabela de símbolos. Então, sem maiores explicações, esses argumentos são manipulados pela ação semântica @ProgramArguments.

1. ⟨*program*⟩ → 'program' 'id' '(' ⟨$identifier\_list\rangle\uparrow^{lst}$ @ProgramArguments$\downarrow_{lst}$ ')' ';' ⟨*declarations*⟩ ⟨*subprogram_declarations*⟩ ⟨*compound_statement*⟩ '.'

```
if (action.equals(Tag._ProgramArguments)) {
    @SuppressWarnings("unchecked")
    List<String> list = (List<String>)action.GetAttribute(0);

    // Argumentos do programa serao declarados como "undefined"
    for(String id: list) {
        table.InsertParameter(id, new UndefinedType());
    }
    return;
}
```

2 Geração de Código Intermediário

A geração de um código intermediário, normalmente adotado pelos compiladores, torna o processo o processo de tradução um pouco mais lento. Embora um programa fonte possa ser traduzido diretamente para a linguagem objeto, o uso de uma representação intermediária, independente de máquina, tem as seguintes vantagens:

1. Reaproveitamento de código, facilitando o transporte de um compilador para diversas plataformas de hardware; somente os módulos finais precisam ser refeitos a cada transporte.
2. Um otimizador de código independente de máquina pode ser usado no código intermediário.

Pode-se pensar nessa representação intermediária como um programa para uma *máquina abstrata*. A representação intermediária deve possuir duas propriedades importantes: ser fácil de produzir e fácil de traduzir no programa alvo.

Compiladores sofisticados tipicamente realizam a tradução do código fonte para algum código de máquina através de múltiplos passos definidos sobre formas intermediárias de código. Este processo de múltiplos estágios é usado porque muitos algoritmos de otimização de código são mais facilmente implementados em uma ou outra representação. Esta organização também facilita a criação de um único *front-end* que pode ser combinado com diferentes *back-end's*, cada qual, endereçando uma arquitetura de máquina diferente.

De acordo com o nível de abstração do código intermediário ele pode ser classificado como:

- ***HIR*** – *High Intermediate Representation*
  - Usada nos primeiros estágios do compilador.
  - Simplificação de construções gramaticais para somente o essencial para otimização/geração de código.

  Algumas representações deste tipo incluem "Árvore e Grafo de Sintaxe", "Notações Pós-fixada e Pré-fixada" e "Representações Linearizadas".
- ***MIR*** – *Medium Intermediate Representation*
  - Boa base para geração de código eficiente.
  - Pode expressar todas características de linguagens de programação de forma independente da linguagem.

– Representação de variáveis, temporários, registradores.

Além das mesmas representações utilizadas no tipo **HIR**, o tipo **MIR** inclue a representação "Código de Três Endereços" (TAC – Three Address Code) a qual pode ser implementada como "Quádruplas", "Triplas" ou "Grafos Acíclicos Dirigidos (DAG)".

- ***LIR*** – *Low Intermediate Representation*
  – Quase 1-1 para linguagem de máquina.
  – Dependente da arquitetura.

  Neste tipo de representação, as instruções são implementadas em linguagem Assembly.

A tradução do código de alto nível para o código objeto do processador está associada a traduzir para a linguagem-alvo a representação da árvore gramatical obtida para as diversas expressões do programa. Embora tal atividade possa ser realizada para a árvore completa após a conclusão da análise sintática, em geral ela é efetivada através das ações semânticas associadas à aplicação das regras de reconhecimento do analisador sintático. Este procedimento é denominado tradução dirigida pela sintaxe.

Por conveniência, o analisador sintático gera código para uma máquina abstrata, com uma linguagem próxima ao *assembly*, porém independente de processadores específicos. Em uma segunda etapa da geração de código, esse código intermediário é traduzido para a linguagem *assembly* desejada. Dessa forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores.

2.1 Código de três endereços

O código de três endereços é composto por uma sequência de instruções envolvendo operações binárias ou unárias e uma atribuição. O nome *três endereços* está associado à especificação, em uma instrução, de no máximo três variáveis: duas para os operadores binários e uma para o resultado. Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução. Dessa forma, obtém-se um código mais próximo da estrutura da linguagem *assembly* e, consequentemente, de mais fácil conversão para a linguagem-alvo.

Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções: expressões com atribuição, desvios, invocação de rotinas e acesso indexado e indireto.

Instruções de atribuição são aquelas nas quais o resultado de uma operação é armazenado na variável especificada à esquerda do operador de atribuição, aqui denotado por :=. Há três formas para esse tipo de instrução. Na primeira, a variável recebe o resultado de uma operação binária:

x := y op z

O resultado pode ser também obtido a partir da aplicação de um operador unário:

x := op y

Na terceira forma, pode ocorrer uma simples cópia de valores de uma variável para outra:

x := y

Por exemplo, a expressão em MiniPascal $a := b + c * d$; seria traduzida nesse formato para as instruções:

```
tmp1 := c * d
a := b + tmpt1
```

As instruções de desvio podem assumir duas formas básicas. Uma instrução de desvio incondicional tem o formato:

goto L

onde L é um label simbólico que identifica uma linha do código. A outra forma de desvio é o desvio condicional, com o formato:

ifExp x opr y goto L

onde opr é um operador relacional de comparação e L é o label da linha que deve ser executada se o resultado da aplicação do operador relacional for verdadeiro; caso contrário, a linha seguinte é executada.

Derivados do ifExp temos também o ifTrue que testa o resultado da última operação e realiza o desvio se este for verdade e o ifFalse que realiza o desvio se este for verdade caso o resultado da última operação for falsidade:

```
tmp := ...
ifTrue tmp goto L

tmp := ...
ifFalse tmp goto L
```

Por exemplo, a seguinte iteração em MiniPascal

```
while i < k do
    begin
        i := i+1;
        x[i] := 0
    end
x[0] := 0;
```

poderia ser traduzida para

```
$1:
    tmp1 := i < k
    ifFalse goto $2        ; Se for falso sai do while
    tmp1 := i + 1
    i := tmp1
    x[i] := 0
    goto $1
$2:
    x[0] := 0
```

A invocação de rotinas ocorre em duas etapas. Inicialmente, os argumentos do procedimento são registrados com a instrução param; após a definição dos argumentos, a instrução call completa a invocação da rotina. A instrução return indica o fim de execução de uma rotina. Opcionalmente, esta instrução pode especificar um valor de retorno, que pode ser atribuído na linguagem intermediária a uma variável como resultado de call.

Por exemplo, considere a chamada de uma função $x := f(a, b, c)$; que recebe três argumentos e retorna um valor – essa expressão seria traduzida para:

```
param a
param b
param c
call f, 3
x := tmp
```

onde o número após a vírgula indica o número de argumentos utilizados pelo procedimento f. Com o uso desse argumento adicional é possível expressar sem dificuldades as chamadas aninhadas de procedimentos.

O último tipo de instrução para códigos de três endereços refere-se aos modos de endereçamento indexado e indireto. Para atribuições indexadas, as duas formas básicas são

```
x := y[i]
x[i] := y
```

As atribuições associadas ao modo indireto permitem a manipulação de endereços e seus conteúdos. As instruções em formato intermediário também utilizam um formato próximo àquele da linguagem C:

```
x := &y      ; x recebe o endereço de y
w := *x      ; w recebe o que tem no endereço x
*x := z      ; z é copiado para a memória cujo endereço é x
```

A representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em tabelas com quatro ou três colunas. Na abordagem que utiliza quádruplas (as tabelas com quatro colunas), cada instrução é representada por uma linha na tabela com a especificação do operador, do primeiro argumento, do segundo argumento e do resultado

Para algumas instruções, como aquelas envolvendo operadores unários ou desvio incondicional, algumas das colunas estariam vazias.

Na outra forma de representação, por triplas, evita a necessidade de manter nomes de variáveis temporárias ao fazer referência às linhas da própria tabela no lugar dos argumentos. Nesse caso, apenas três colunas são necessárias, uma vez que o resultado está sempre implicitamente associado à linha da tabela.

Como pode ser visto no texto anterior, as instruções de código intermediário manipulam labels, nomes, temporários e constantes. Outros tipos de objetos também são comuns, porém não introduzidos aqui. De uma forma geral, todos esses tipos abstraídos e denominados endereços. Resumindo, um endereço podem ser um dos seguintes:

- Nomes aparecendo no programa fonte podem aparecer como endereços no código de três endereços. Em uma implementação, um nome é substituído por um ponteiro para uma entrada na tabela de símbolos, onde todas as informações associadas àquele nome são armazenadas.

- Temporários gerados pelo compilador.
- Constantes.
- Labels (endereços).
- Outros (Undef, NAC etc.)

2.2 Implementação dos códigos de três endereços (TAC) como triplas

Para a geração de código intermediário da linguagem MiniPascal, os endereços serão representados pela hierarquia de classes da Figura 4.

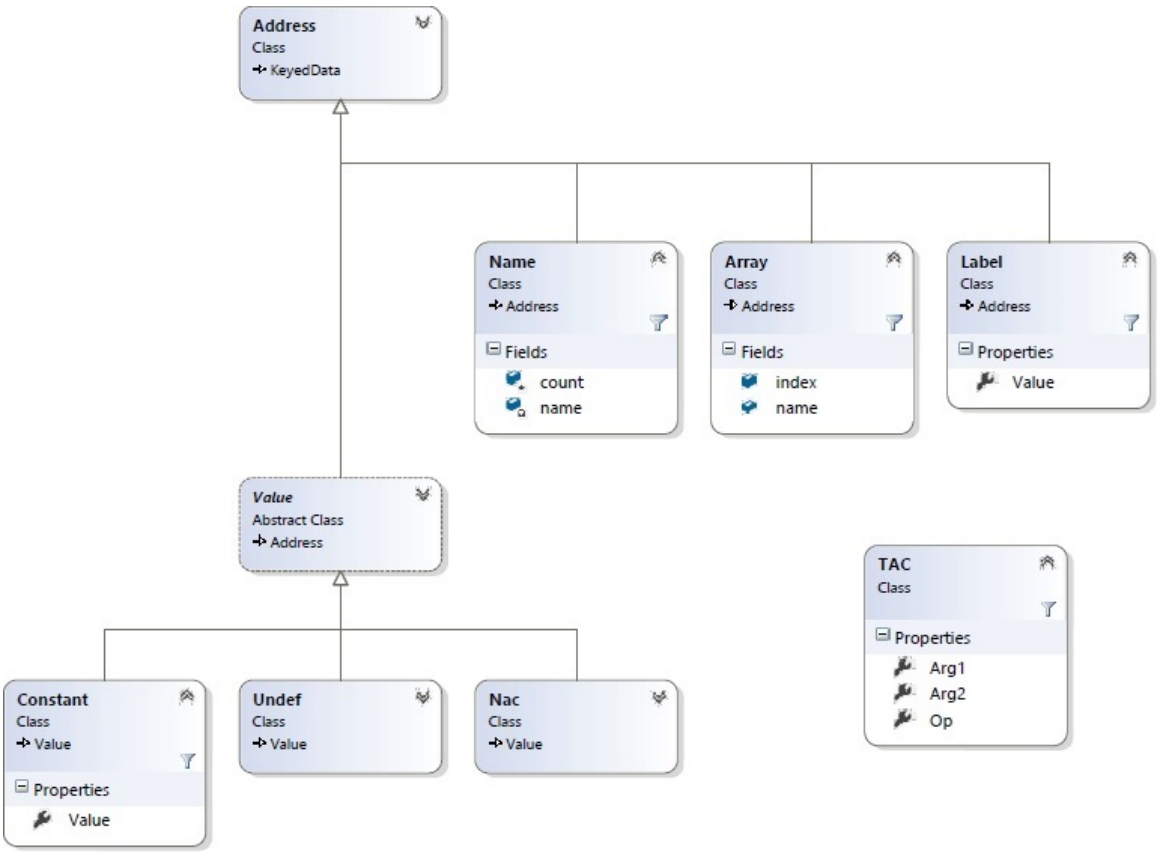

Figure 4: Hierarquia de classes de endereços

Na mesma Figura 4 aparece os detalhes da classe TAC a qual será utilizada para representar as instruções do código intermediário. O atribulo Op representa uma espécie de *mnemônico* da instrução e pode assumir um dos seguintes valores:

```
public enum Operator
{
    COPY,                       // x := y
    MUL, DIV, ADD, SUB,         // x := y op z
    INC, DEC, NEG, NOT,         // x := op y
    GOTO,                       // goto L
    IFTRUE, IFFALSE, IFEXP,     // it ... goto L
    PARAM,                      // param x
    CALL,                       // call L
    RETURN,                     // return
    RETVAL,                     // return x
    ADDRESS,                    // &x
    LT, LE, GT, GE,             // x := y opr y
    EQ, NEQ,                    // x := y opr y
    FROMMEMORY, TOMEMORY,       // *ptr
    FROMARRAY, TOARRAY,         // a[i]
    CONTINUE,                   // continuacao da instrucao
};
```

A hierarquia de classes que implementa as instruções acima aparece na Figura 5 e os detalhes das tuplas de cada instrução segue.

- Atribuições da forma x := y op z e x := op y.

x := y op z		
Operator	Arg1	Arg2
op	y	z
CONTINUE	—	x

```
public Binary(Operator op, Address x, Address y, Address z)
// x := y op z
// op = MUL, DIV, ADD, SUB, LT, LE, GT, GE, EQ, NEQ
{
    TAC ic;

    ic = new TAC(op, y, z);
    pos = Add(ic);
    ic = new TAC(Operator.CONTINUE, null, x);
    Add(ic);

    target = ic.Arg2;
}
```

x := y		
Operator	Arg1	Arg2
op	x	y

```
public Unary(Operator op, Address x, Address y)
// x := op y
// op = INC, DEC, NEG, NOT
{
    TAC ic;

    ic = new TAC(op, x, y);
    pos = Add(ic);

    target = ic.Arg1;
}
```

- Cópia x := y.

x := y		
Operator	Arg1	Arg2
COPY	x	y

```
public Copy(Address x, Address y)
// x := y
{
    TAC ic;

    ic = new TAC(Operator.COPY, x, y);
    pos = Add(ic);

    target = ic.Arg1;
}
```

- Desvios incondicionais da forma goto L.

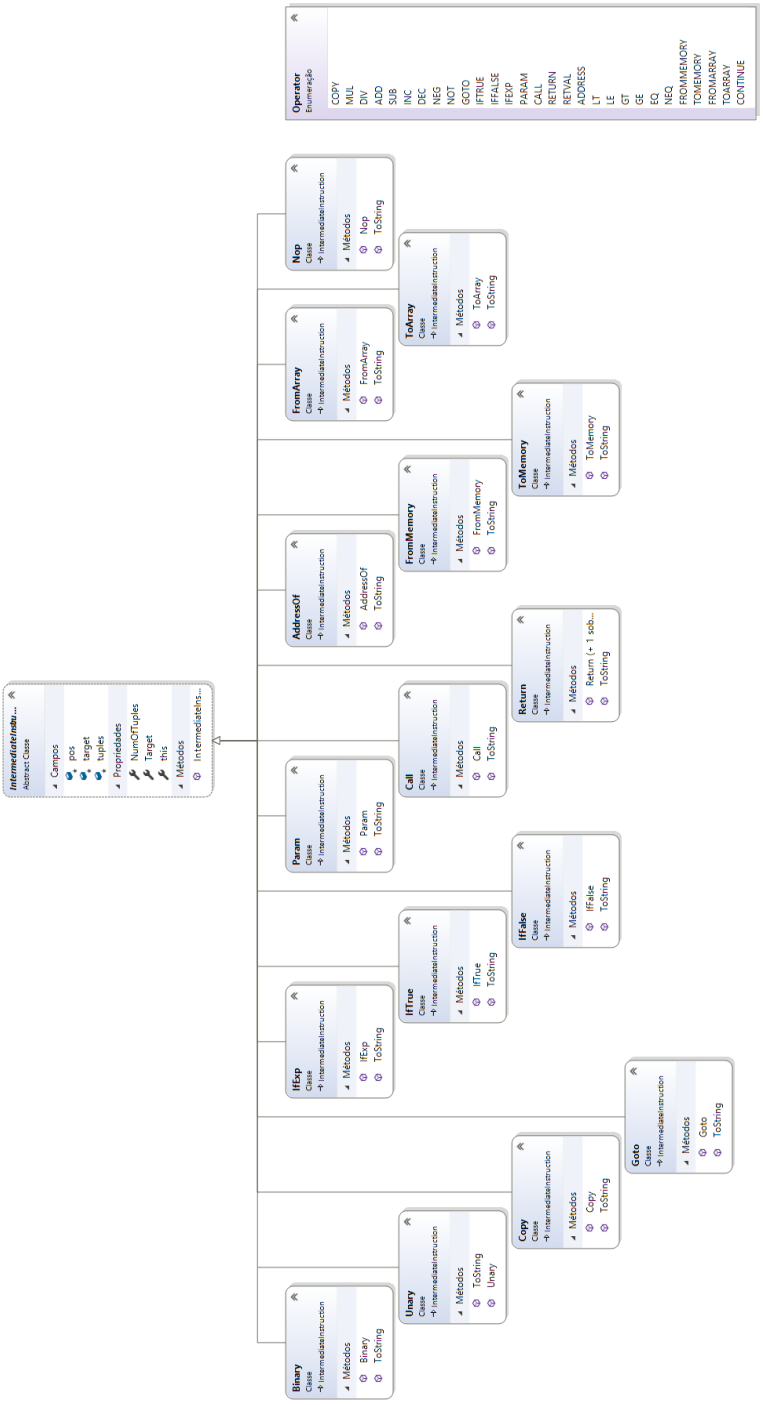

Figure 5: Hierarquia de classes de IC

goto L		
Operator	Arg1	Arg2
GOTO	—	L

```
public Goto(Label L)
// goto L
{
    TAC ic;

    ic = new TAC(Operator.GOTO, null, L);
    pos = Add(ic);
}
```

- Desvios condicionais

ifTrue x goto L		
Operator	Arg1	Arg2
IFTRUE	x	L

```
public IfTrue(Address x, Label L)
// if x goto L
{
    TAC ic;

    ic = new TAC(Operator.IFTRUE, x, L);
    pos = Add(ic);
}
```

ifFalse x goto L		
Operator	Arg1	Arg2
IFFALSE	x	L

```
public IfFalse(Address x, Label L)
// ifFalse x goto L
{
    TAC ic;

    ic = new TAC(Operator.IFFALSE, x, L);
    pos = Add(ic);
}
```

ifExp x oprel y goto L		
Operator	Arg1	Arg2
IFEXP	—	L
oprel	x	y

```
public IfExp(Operator oprel, Address x, Address y, Label L)
// ifExp x oprel y goto L
{
    TAC ic;

    ic = new TAC(Operator.IFFALSE, x, L);
    pos = Add(ic);
    ic = new TAC(oprel, x, y);
    Add(ic);
}
```

- param x e call p, n para invocação de procedimentos e return y – onde y representa um valor de retorno opcional.

param x		
Operator	Arg1	Arg2
PARAM	x	—

```
public Param(Address x)
// param x
{
    TAC ic;

    ic = new TAC(Operator.PARAM, x, null);
    pos = Add(ic);
}
```

call L, n		
Operator	Arg1	Arg2
CALL	n	L

```
public Call(Label L, int n)
// call L, n
{
    TAC ic;

    ic = new TAC(Operator.CALL, Constant.Create(n), L);
    pos = Add(ic);
}
```

return		
Operator	Arg1	Arg2
RETURN	—	—

```
public Return()
// return
{
    TAC ic = new TAC(Operator.RETURN, null, null);
    pos = Add(ic);
}
```

return x		
Operator	Arg1	Arg2
RETVAL	x	—

```
public Return(Address x)
// return x
{
    TAC ic = new TAC(Operator.RETVAL, x, null);
    pos = Add(ic);
}
```

- Indexação da forma x:= y[i] e x[i] := y.

x[i] = y		
Operator	Arg1	Arg2
TOARRAY	x	i
CONTINUE	y	—

```
public ToArray(Address x, Address i, Address y)
// x[i] := y
{
    TAC ic;

    ic = new TAC(Operator.TOARRAY, x, i);
    pos = Add(ic);
    ic = new TAC(Operator.CONTINUE, y, null);
    Add(ic);
}
```

x = y[i]		
Operator	Arg1	Arg2
FROMARRAY	y	i
CONTINUE	x	—

```
public FromArray(Address x, Address i, Address y)
// x := y[i]
{
    TAC ic;

    ic = new TAC(Operator.FROMARRAY, y, i);
    pos = Add(ic);
    ic = new TAC(Operator.CONTINUE, x, null);
    Add(ic);

    target = ic.Arg1;
}
```

- Atribuições de ponteiros e endereços da forma x := &y e x := *y.

x := &y		
Operator	Arg1	Arg2
ADDRESS	x	y

```
public AddressOf(Address x, Address y)
// x := &y
{
    TAC ic;

    ic = new TAC(Operator.ADDRESS, x, y);
    pos = Add(ic);

    target = ic.Arg1;
}
```

x := *y		
Operator	Arg1	Arg2
ADDRESS	x	y

```
public FromMemory(Address x, Address y)
// x := *y
{
    TAC ic;

    ic = new TAC(Operator.FROMMEMORY, x, y);
    pos = Add(ic);

    target = ic.Arg1;
}
```

*x := y		
Operator	Arg1	Arg2
ADDRESS	x	y

```
public ToMemory(Address x, Address y)
// *x := y
{
    TAC ic;

    ic = new TAC(Operator.TOMEMORY, x, y);
    pos = Add(ic);
}
```